A study of the Weight Perturbation algorithm used in neural networks

ten Kate, R.E.

Award date:
1993

A study of the Weight Perturbation algorithm used in neural networks

Master thesis of R.E. ten Kate

August 1993

Supervisor	:	Prof.dr.ir. W.M.G van Bokhoven
Coach	:	Dr.ir. J.A. Hegt
Period	:	Nov. 1992 - Aug. 1993
At	:	Electronic circuit design group (EEB)
		Faculty of Electrical Engineering
		Eindhoven University of Technology

Abstract

This thesis studies different aspects of the Weight Perturbation (WP) algorithm used to train neural networks.

After a general introduction of neural networks and their algorithms, the WP algorithm is described. A theoretical study is done describing the effects of the applied perturbation on the error performance of the algorithm. Also a theoretical study is done describing the influence of the learning rate on the convergence speed. The effects of these two algorithm parameters have been simulated in an ideal Multilayer Perceptron using various benchmark problems.

When the WP algorithm is implemented on an analog neural network chip, several hardware limitations are present which influence the performance of the WP algorithm ; weight quantization, weight decay, non-ideal multipliers and neurons. The influence of these non-idealities on the algorithm performance has been studied theoretically. Simulations of these effects have been done using predicted parameters by SPICE simulations of the hardware. Several proposals are made to reduce the effects of the hardware non-idealities.

Two proposals have been studied to increase the speed of the WP algorithm.

Contents

1 Introduction

Neural networks and algorithms for these networks have been studied for many years. The networks consist of a number of non-linear neurons interconnected by synapses. The neural network is trained by algorithms to respond adequately to its inputs. Learning is achieved by adapting the values of the synapses. When the response is correct the algorithm trained the network to solve a certain problem.

The last years much effort has been made to realize analog neural network chips. As opposed to digital neural network chips, the analog chips offer advantages in terms of neurons-per-chip density. Larger networks can be built consisting of many simple analog elements.

One of the disadvantages of analog CMOS chips is that the implementation of the existing algorithms is difficult. Most of the algorithms are very sensitive to the hardware limitations caused by the chip and network interface. Although their behavior may work properly in computer simulations, the performance of an algorithm can become unacceptable in hardware implementations.

Jabri and Flower [11] developed a new and simple algorithm called 'Weight Perturbation'. This algorithm is suited for neural networks designed in analog CMOS hardware.

Currently, the EEB division of the Eindhoven University of Technology (TUE) is working at the implementation of this proposed weight perturbation algorithm on an analog neural network chip with its interface to a Personal Computer.

The purpose of the neural network chip is that it serves as a demonstration model where further research can be made with. The chip with its interface is designed completely at the EEB division. The implementation of the Weight Perturbation algorithm is bounded to their specifications.

In this thesis a study is made of the behavior of the Weight Perturbation algorithm in an ideal neural network using computer simulations. Next a study is made introducing several hardware limitations in the simulations. Their influence on the performance of the

algorithms is analyzed. The limitations are models which are based on the specifications of the neural network chip and the neural network interface. Because the algorithm is very slow, several proposals are made to accelerate the convergence.

2 Neural Networks

2.1 Introduction

Artificial Neural Networks (ANN) have been studied for many years in the hope of achieving human brain like performances. They are based upon human brains and have also a parallel character similar to the brain structure. Because of this parallelism the network is suitable for certain problems like speech and image recognition. The parallel processing of these problems using a neural network could be faster than when a sequential von Neumann machine would be used.

Neural networks explore many competing hypotheses simultaneously, using, massive parallel nets composed of many computational elements connected by links with variable weights. The possible advantages of a neural network over traditional computers are numerous :

- ☐ It is robust and fault tolerant. Elements in a large neural network could fail without affecting its performance significantly.
- ☐ It is flexible. It can easily adjust to a new environment by 'learning'.
- ☐ It deals with information that is fuzzy, probabilistic, noisy or inconsistent.
- ☐ It is highly parallel.

A neural network is not suitable for numerical problems, where massive controllable and exact arithmetic computations have to be made. The structure of sequential computers is made for these kind of calculations.

The benefits obtained with neural networks in certain tasks give reason to study neural networks extensively. Because the development in speed of sequential computers is limited, a neural network is a parallel alternative which gives new possibilities in dealing with certain problems.

2.2 Network Structure

A neural network consists of nodes and links between the nodes. The nodes are also called

neurons and the links are called synapses according to our brain structure. The neurons are interconnected to each other via synapses. A neuron is a non-linear element which has multiple inputs and a single output as can be seen in figure 2.1.

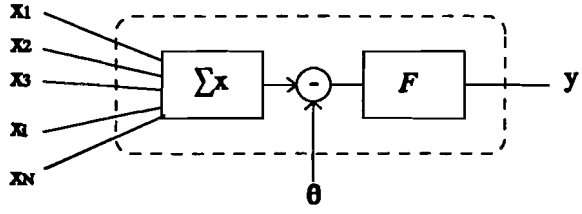


figure 2.1 Model of a neuron.

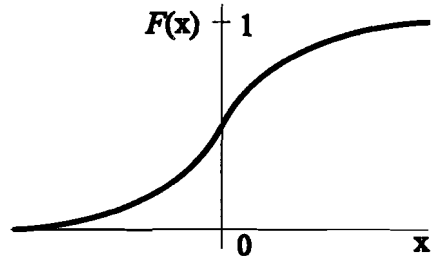
The neuron has multiple inputs x_i and a single output y. The inputvalues are all added and their sum is transferred through a non-linear squashing function *F*. A bias θ is subtracted from the sum of the inputs. By introducing this bias θ, it rises the possibility of shifting *F* over the x range. Else this would be impossible. The relationship between the output y of the neuron and its inputs x_i can be described by :

$$y = F(\sum_{i=1}^{N} x_i - \theta) \tag{2.1}$$

In neural networks, different functions *F* could be applied. Some of these functions are described below :

- *Sigmoid function*

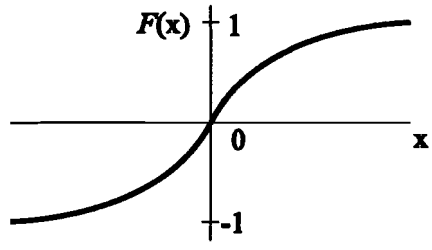
$$F(x) = \frac{1}{1 + e^{-\beta x}} \tag{2.2}$$



The "temperature" β defines the slope of the sigmoid function

- *Hyperbolic Tangent function (Tanh)*

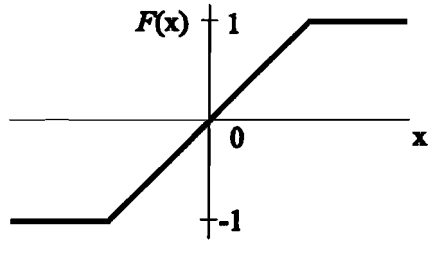
$$F(x) = \tanh(\beta x) = \frac{e^{\beta x} - e^{-\beta x}}{e^{\beta x} + e^{-\beta x}} \tag{2.3}$$



Relating to the sigmoid function (2.2), the temperature β defines the slope of the Tanh function.

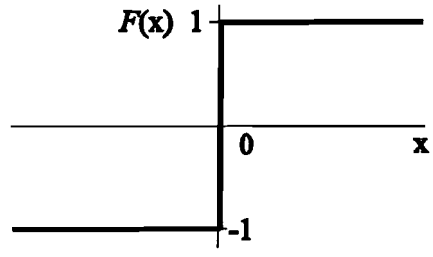
- *Ramp threshold function*

$$\begin{aligned} F(x) &= x , & -1 < x < 1 \\ F(x) &= 1 , & x \geq 1 \\ F(x) &= -1 , & x \leq -1 \end{aligned} \tag{2.4}$$



- *Hard Limiter function*

$$\begin{aligned} F(x) &= 1 , & x \geq 0 \\ F(x) &= -1 , & x < 0 \end{aligned} \tag{2.5}$$



The sigmoid and Tanh functions are the most common in neural networks. These functions are easy to implement electronically using a differential pair [3]. The output of all the non-linear functions $F(x)$ is bounded to the range [-1,1] or [0,1] using a sigmoid function. The input range is unlimited, so F squashes the input into a certain range.

Figure 2.1 shows one neuron and its inputs. A neural network consists of many of these neurons, all interconnected via synapses. A synaps is a weight factor which scales the connection between two neurons with a certain factor. The most common way of connecting neurons is in layers as can be seen in figure 2.2. This layered feedforward network structure is called *Multi Layer Perceptron* (MLP).

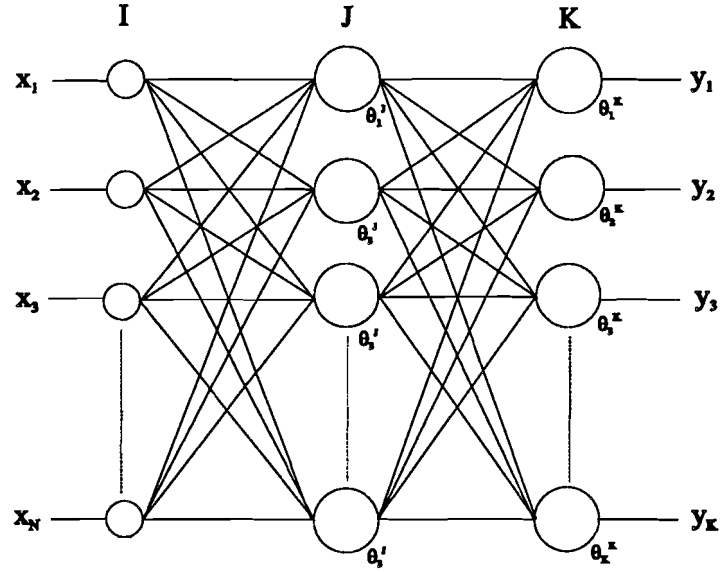


figure 2.2 : Multi Layer Perceptron

As can be seen in this figure, there exists a full interconnection between the neurons in the different layers. This means that all the outputs of the neurons in layer l are connected with all the inputs of the neurons in the next layer l+1. In this way, a whole layer of neurons is connected with its precessor and its successor. The signal flow, however, is only in one direction. The outputs of the neurons in layer l are not connected to the inputs of the neurons in the same layer l, the previous layers and the layers further than the next layer l+1. Three kinds of layers are distinguished in a MLP :

- *Input layer* The input layer receives the input patterns X. It consists of nodes which connect the input signals with all the neurons in the next layer. These nodes are not neurons. A MLP contains one input layer.

- *Output layer* The last layer of the network is called output layer. The outputs Y of these neurons are connected with the outside world. A MLP contains one output layer.

- *Hidden layer* All the layers between the input and the output layer are called hidden layers. The hidden neurons in these layers have no direct connection with the outside world. A MLP contains one or more hidden layers.

The links between the neurons have different weight factors. The connection between the output of neuron i in layer l ($N_{i,l}$) to neuron j in layer l+1 ($N_{j,l+1}$) is multiplied with a factor $w_{ji,l}$. In this thesis $w_{ji,l}$ is also called w_{ji}. The relation between an output y_k and the inputs X of a 3-layer MLP is described as :

$$y_k = F\left(\sum_{j=1}^{J} w_{kj} \cdot F\left(\sum_{i=1}^{I} w_{ji} \cdot x_i - \theta_j^J \right) - \theta_k^K \right) \tag{2.6}$$

Relation (2.6) could be very complex depending on the used squashing function F, the number of neurons and layers in the network. Depending on the problem to be solved, the number of neurons differs from layer to layer. Also the number of layers differ from problem to problem. It has been proved [6] that a three layer MLP suffices to project any continuous representation $\mathbb{R}^A$ on any continuous representation $\mathbb{R}^B$.

As can be seen from the relation (2.6), the parameters in a neural network are the weights w_{ji} and θ for a given network size and squashing function F. A problem can be solved by generating proper values of the outputs Y when a certain pattern X is the input of the network. This is done by finding optimum values of the parameters w_{ji} and θ. For convenience, θ can be seen as a constant input of value unity of the neuron multiplied with a weight w_{j0}.

In order to perform a certain task properly, the values of the weights w_{ji} in the neural network should be optimized. This optimization is carried out by a learning algorithm. Some of these algorithms will be described in chapter 3.

3 Algorithms

3.1 Introduction

Coherent with the research done in neural network structures and hardware, algorithms for these neural networks have been developed. Already in 1943 a first general theory was proposed regarding the information processing based on networks with binary decision elements. Since then, the research into this area has never stopped and is still studied extensively by many researchers [16,28].

The objective of neural network algorithms is to find an optimum set of weights in a neural network which results in the solution of the problem presented to the network. An algorithm adapts the weights iteratively to an optimum and it is therefore used in the learning process of a neural network. Because so many algorithms have been developed which all have different properties, it is useful to distinguish them in certain classes. We can divide algorithms into two main groups :

I *Supervised learning*

- Learning with a teacher :
  In supervised learning, the network's outputs Y are compared with desired patterns D. The comparison results are used by the algorithm to adapt the weights w_{ji} in the network, so that the difference between the outputs Y and the desired patterns D is minimized.

- Reinforced learning :
  A variant of 'learning with a teacher' is 'learning with a critic' or called reinforced learning. As opposed to learning with a teacher, the reinforced learning algorithms have less detailed information available to train a network. The algorithm has no access to the outputs Y and the correct target values D. Only a quantitative measure of the comparison between Y and D is used to train the network. This means that only a scalar is used in the feedback to train the network.

II *Unsupervised learning*

In contradiction to supervised learning, with unsupervised learning algorithms there are no teachers which say what is exactly right and wrong or critics which say how wrong the outputs of the network are. This means that solely inputs X are offered to the network and that these are not compared with desired patterns. The algorithm must discover for itself patterns, features, regularities, correlations or categories in the input data.

The division between supervised and unsupervised learning divides the algorithms into two groups. Because Weight Perturbation belongs to the supervised learning algorithms and this thesis does not intend to give a full report on all algorithms, the rest of this chapter deals only with supervised learning algorithms.

3.2 Supervised learning in neural networks

In supervised learning we use networks with separate inputs and outputs, and it is assumed to have a list or *training set* of correct input-output pairs as examples. When one of the training inputs is applied to the network, the network's output Y is compared with the correct desired patterns D.

Teacher algorithms use all the outputs of the network and target values to change the weights resulting in smaller differences between Y and D. Reinforced algorithms use a difference function of the outputs and target values. Depending on the difference function, reinforced learning algorithms change the connection strengths w_{ji} to minimize this difference. This is typically done incrementally, making small adjustments in response to the difference function, so that the weights w_{ji} converge, if possible, to a solution where for every input pattern the outputs of the network equal the target pattern.

After training the network with a set of training patterns, it could be possible to present input patterns which are not present in the training set which was used to train the problem. If the network responds adequately to these unknown patterns, the network has generalized based on the training input patterns.

Supervised learning algorithms present an input training set to the network. This input training set consists of P patterns, each consisting of a N-dimensional vector. N equals the number of nodes in a MLP input layer. The input X^P of the network consists of the following patterns :

$$X^P = (x_1, x_2, .., x_i, ..., x_N)^P \tag{3.1}$$

The desired output training set D^P consists of K-dimensional vectors. There are also P vectors in the output to match the P vectors in the input.

$$D^P = (d_1, d_2, .., d_k, ..., d_K)^P \tag{3.2}$$

When the patterns X^P are presented to the network, the outputs Y^P consist of K-dimensional vectors. The dimension of the output vectors Y^P corresponds with the dimension of the desired vectors D^P. The output layer of a MLP contains the number of neurons necessary to match the desired pattern.

$$Y^P = (y_1, y_2, .., y_k, ..., y_K)^P \tag{3.3}$$

Given a certain network configuration with fixed parameters such as the number of neurons in each layer, the number of layers and the squashing function F in the neurons, the response Y^P of a neural network on the input patterns X^P can be seen in the next formula :

$$Y^P = \Omega(X^P, W) \tag{3.4}$$

For given input patterns X^P the network depends on the weight set W. The weight set W contains all the weights in a network. The function Ω describes the total network transfer with a static size. It can be seen as a generalized transfer function (2.6). It has to be said that the biases θ are seen as a constant input to a neuron multiplied by a factor w_{j0} and could therefore be treated as weights.

Algorithms using supervised learning, compare the actual output Y^P with the desired patterns D^P. The optimum values for the weight set are found when $Y^P = D^P$. This means that a supervised algorithm has the following objective :

$$\forall_{k \in K,\, p \in P} \quad y_k^p = d_k^p \tag{3.5}$$

Every output neuron y_k of the neural network should be equal to the desired value d_k. This

relationship must be valid for all training patterns. However, condition (3.5) can not always be fulfilled. If condition (3.5) is fulfilled, it is said that the neural network has learned a certain problem perfectly. The objective of all the supervised learning algorithms is to find the optimum weights w_{ji} so that (3.5) is valid or comes as close to it as possible.

Various algorithms have been developed using supervised learning to train the network. The most important group of algorithms using supervised learning are called gradient descent algorithms. Weight perturbation is a gradient descent algorithm and therefore these algorithms are explained in more detail.

3.3 Gradient Descent Learning

Gradient descent learning algorithms have the objective to decrease a difference function to its minimum. This difference function or error function E is a measure of how the outputs Y^p differ from the desired patterns D^p. In this way a quantitative measure is given of how well (3.5) has been approximated. The algorithm adapts the weights so, that steps are taken towards its minimum E_0 in the direction opposite to the gradient of the error function.

Gradient descent algorithms using reinforced learning only use the error function E to train the network. This scalar is used to adapt the weights iteratively with the objective of decreasing E until its minimum. Important examples are the MRIII algorithm [1] and the Weight Perturbation algorithm [11].

Gradient descent algorithms using teachers implement the calculation of the error function E itself in the algorithm. Therefore the algorithms require all the outputs Y^p and target patterns D^p to train the network to a minimum error E_0. An important example is the Backpropagation algorithm.

Several error functions can be used in order to calculate a difference function. The most important is described in the next paragraph.

3.3.1 Error function

The most common error function is the Total Square Error (E_{TSE}) function as given in the next equation :

$$E_{TSE}[W] = \frac{1}{2} \cdot \sum_{p=1}^{P} \sum_{k=1}^{K} \left(y_k^p - d_k^p \right)^2 \tag{3.6}$$

A variant on this which is also used to give a normalized error function is Mean Square Error (E_{MSE}) function. E_{TSE} in (3.6) depends on the number of presented patterns P and the number of output neurons K. To calculate an error function which is normalized to these parameters, (3.6) is divided by P and K as shown in the next equation :

$$E_{MSE}[W] = \frac{1}{2PK} \cdot \sum_{p=1}^{P} \sum_{k=1}^{K} \left(y_k^p - d_k^p \right)^2 \tag{3.7}$$

(3.7) could be used when comparing different problems with different network sizes. To reduce the number of calculations however, it is more sensible to calculate the E_{TSE} (3.6).

Both error functions give a scalar of how well condition (3.5) is fulfilled. When (3.5) is true and the network is trained perfectly, the error functions (3.6) and (3.7) equal zero.

Many more error functions have been developed. This thesis only uses the error functions described in this paragraph because they are the most used and suffice for the WP algorithm.

In the next paragraph the theory of gradient descent is described. This is used to find the minimum error E_0.

3.3.2 Steepest Descent

A gradient descent algorithm updates a weight w_{ji} in the direction of the steepest descent of the error function E along that weight dimension. This is done for each weight in the network. Finally the updates result in a minimum of the error function at a point in the weight space where all error derivatives are zero.

Before the algorithm starts, a set of initial weights w_{ji}^{init} has to be chosen at random. The gradient descent algorithm uses these initial weights as starting points, and the algorithm updates the weights so that the weights converge to an optimum weight set. The weights are updated with steps Δw_{ji} as can be seen in the next equation:

$$w_{ji}^{new} = w_{ji}^{old} + \Delta w_{ji} \tag{3.8}$$

Each weight update Δw_{ji} is made opposite to the error derivative. This insures that the error $E[w_{ji}^{new}]$ is smaller than $E[w_{ji}^{old}]$ when the step Δw_{ji} is taken small enough. Formula (3.9) shows the gradient descent weight update:

$$\Delta w_{ji} = -\eta \cdot \frac{\partial E}{\partial w_{ji}} \tag{3.9}$$

According to (3.9), the gradient descent algorithm changes each weight w_{ji} by a step Δw_{ji} proportional to the derivative of E at the present location. (3.9) Insures that the weight values are updated towards their optimum values. The magnitude of the updates Δw_{ji} does not only depend on the error derivative, but also the learning rate η which is described in the next paragraph.

3.3.3 Learning Rate

The updates Δw_{ji} depend on the learning rate η, which thus determines the speed of the algorithm. A small η results in a small weight update Δw_{ji}, which results in a slow convergence speed. A larger learning rate η will speed up the algorithm. However there is an upperlimit η_{max} for η, which depends on the error function E and is thus different for each problem presented to the neural network.

This upperlimit is illustrated in a simple example when the gradient descent algorithm is studied, applied in a single layer of linear neurons. In this case the error (3.6) transforms into:

$$E[w] = \frac{1}{2} \cdot \sum_{p=1}^{P} \sum_{k=1}^{K} \left(y_k^p - d_k^p \right)^2 = \frac{1}{2} \cdot \sum_{p=1}^{P} \sum_{k=1}^{K} \left(\sum_{i=1}^{I} w_i \cdot x_i - d_k^p \right)^2 \tag{3.10}$$

If (3.10) is diagonalized and the training patterns are consistent pattern pairs, $E[w]$ can be

rewritten as :

$$E[w] = \sum_{\lambda=1}^{M} a_\lambda \left(w_\lambda - w_\lambda^o\right)^2 \tag{3.11}$$

M is the total number of weights and the w_λ's are linear combinations of the weights w_{ji}. The eigenvalues a_λ and the optimimum weight values w_λ^o are constants, depending only on the pattern vectors. The eigenvalues a_λ are necessarily positive or zero because the sum-of-squares in (3.11) can not be negative. If the weight updates Δw_λ are calculated according to a gradient descent algorithm (3.9), this results in :

$$\Delta w_\lambda = -\eta \frac{\partial E}{\partial w_\lambda} = -2\eta a_\lambda \left(w_\lambda - w_\lambda^o\right) \quad , \forall_\lambda \tag{3.12}$$

Thus the distance $\delta w_\lambda = w_\lambda - w_\lambda^o$ from the optimum in the λ-direction is transformed according to:

$$\delta w_\lambda^{new} = \delta w_\lambda^{old} + \Delta w_\lambda = (1 - 2\eta a_\lambda)\delta w_\lambda^{old} \tag{3.13}$$

In directions of a weight w_λ for which $a_\lambda > 0$, the distance δw_λ to the optimum w_λ^o gets closer to the optimum as long as $|1 - 2\eta a_\lambda| < 1$. Thus convergence is achieved when all weights converge, and thus when $\eta < 1/a_\lambda^{max}$. If η exceeds this boundary oscillations around the minimum will occur and the algorithm becomes unstable. This is shown in the next figure.

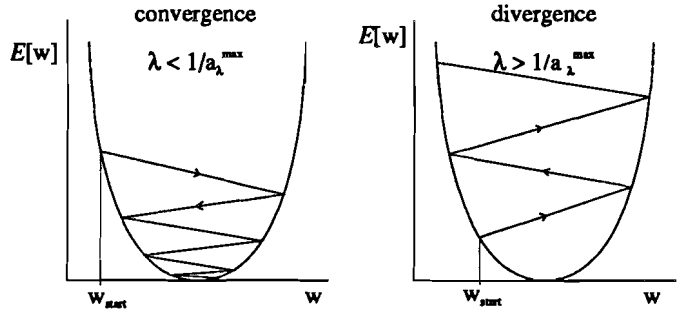


figure 3.1 *An example of learning with a small learning rate and a too large learning rate*

If the training patterns are not consistent, (3.11) transforms into :

$$E[w] = E_0 + \sum_{\lambda=1}^{M} a_\lambda \left(w_\lambda - w_\lambda^o\right)^2 \tag{3.14}$$

with $E_0 > 0$. There is still a single minimum, but $E = E_0 > 0$ shows that the desired condition (3.5) between the output of the network y_k^p and wanted pattern d_k^p is not fulfilled. With a proper choice of η the minimum E_0 of the problem could be achieved and the problem is considered to be learned optimal.

This simple case demonstrates that is important to choose a learning rate η in the right range depending on following arguments:

- Not too small. This will cause an unnecessary slow convergence which can easily be avoided by choosing a larger η.
- Not too large. If the learning rate η is chosen too large, the algorithm becomes instable.

It is hard to set up equations when non-linear neurons are used applied in a multi-layered network. However, the above rules of thumb should be applied for any neural network.

3.3.4 Global/Local Minimum

The last paragraphs have shown that when the learning rate η is chosen smaller than a certain maximum η_{max}, a gradient descent algorithm converges to a minimum. In this minimum the gradient of the error function is zero. It is, however, not said that this minimum is a global minimum or a local minimum.

An error function $E[W]$ depends on many weights w_{ji} and is therefore a multi dimensional function. This can be considered as an error landscape. Because there are many dimensions, the landscape could be very complex depending on the number of weights M and squashing functions F. To get a better idea of how the landscape of an error function looks like, only 2 dimensions are shown, eg the relation between 1 weight and the output error. An example of this relationship could be similar to the one in the next figure.

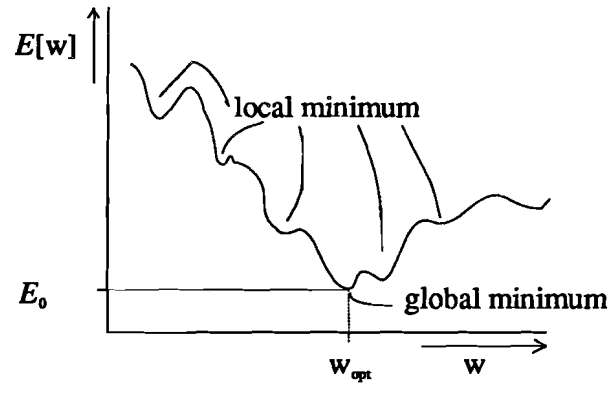


figure 3.2 Example of Error function with only 1 weight as a variable

Because of the non-linear properties of neurons an error landscape could contain multiple local minima as shown in the figure 3.2. Besides these local minima there is always one or more global minima. In a global minimum the lowest possible error E_0 is attained. Depending on the values of the local minima, these minima can sometimes also result in acceptable values for the error when the network is trained to solve a certain problem.

Depending on the initial weight choice a gradient descent algorithm could converge into a local minimum. This is because these kind of algorithms are local search algorithms and only take a small area of the error landscape into account to calculate the direction of the steepest descent, without knowing whether this directions leads to a global minimum or a local minimum. This is the major disadvantage of gradient descent algorithms. When a trial resulted in convergence into a local minimum, the training needs to be restarted with different initial weight values.

Other algorithms, like random search [2,23], alter the weights randomly which results in possibilities to escape out of local minima. These algorithms have not been studied much until now. Our attention goes to the class of gradient descent algorithms where the WP algorithms belongs to. Existing gradient descent algorithms are described in the next paragraph.

3.4 Existing Gradient Descent Algorithms

Many gradient descent algorithms have been developed during the last years. Most of the algorithms are variants on others where small adaptions have been made. If we study the literature, two basic algorithms remain which implement gradient descent learning in a static MLP. The Backpropagation and MRIII algorithm are described in the next paragraphs.

3.4.1 Backpropagation

One of the most widely used and studied gradient descent algorithm is the backpropagation algorithm. A full description of this 'learning with a teacher' algorithm is found in [22]. The weight updates Δw_{ji} are made according to the steepest descent principle (3.9). The weight updates Δw_{ji} are made according to :

$$w_{ji}^{new} = w_{ji}^{old} + \eta \cdot \delta_j \cdot x_i \tag{3.15}$$

Where x_i is an input of the layer. The input x_i is part of the network's input pattern when (3.16) is applied in the first layer of the MLP. The error term δ_j used is calculated in the last layer as :

$$\delta_k^K = F'(s_k) \cdot (d_k - y_k) \tag{3.16}$$

and in all the previous layers as :

$$\delta_j^l = F'(s_j) \cdot \sum_{i=1}^{I} \delta_i^{l+1} \cdot w_{ji}^{l+1} \tag{3.17}$$

where s_j equals the sum of the inputs of neuron j according to :

$$s_j = \sum_{i=1}^{I} w_{ji} \cdot x_i \tag{3.18}$$

where I : number of neurons in layer l

w_{ji}^{l} : connection between neuron i in layer l and neuron j in layer l+1

$F'()$: Derivative of squashing function F

η : learning rate

From (3.17) it can be seen that in order to calculate δ^l the information of the error terms δ^{l+1} in the next layer is necessary. The information is propagated backwards in the network. Therefore this algorithm is called error backwards propagation or backpropagation algorithm.

The advantage of the backpropagation is :

- ☐ All the weights are updated at once. The entire training set (batch) needs to be presented to the network only one time in order to update all the weights in the network once.

The disadvantages are caused by hardware limitations :

- ☐ The error term δ_j propagates backwards from output to input, which requires multiplications. Imperfect electronic implementations of these multipliers introduce errors in the error terms δ_j.
- ☐ Backpropagation uses the derivative of F to calculate δ_j as seen in (3.17) and (3.18). However, when a Tanh function is used as in (2.3) it can be shown that this derivative equals $F'(x) = \beta[1 - F(x)^2]$.
  This eliminates the use of the circuit which implements the derivative $F'(x)$, but still needs a multiplication and the presumption that F equals (2.3) exactly.

These disadvantages cause serious problems in the electronic implementation of the backpropagation algorithm [19]. In computer simulations it works adequately, but implemented in hardware, the algorithm tends to be very sensitive to imperfect multipliers and neurons. Besides the sensitivity, a lot of elements are needed to implement the backward path.

To eliminate these problems of sensitivity and complex backward path a new algorithm was invented which will be described in the next paragraph.

3.4.2 Madeline Rule III

Another algorithm which is based on gradient descent search is the so called Madeline Rule III (MRIII) or Node Perturbation. This algorithm is derived from the MRI and MRII rules and is described in [1].

Combining (3.9) and (3.18) the error derivative can be calculated as :

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial s_j} \cdot \frac{\partial s_j}{\partial w_{ji}} = \frac{\partial E}{\partial s_j} \cdot x_i \tag{3.19}$$

s_j Is the signal in the neuron just before it is applied at the squashing function F of a neuron. (3.19) Results in a weight update :

$$\Delta w_{ji} = -\eta \cdot \frac{\partial E}{\partial s_j} \cdot x_i \tag{3.20}$$

A perturbation δs_j added to s_j is used to approximate the error derivative. The error derivative is approximated by :

$$\frac{\partial E}{\partial s_j} \approx \frac{\Delta E}{\Delta s_j} = \frac{E(s_j + \delta s_j) - E(s_j)}{\delta s_j} \tag{3.21}$$

This means that the weights are updated as follows :

$$w_{ji}^{new} = w_{ji}^{old} - \eta \cdot \frac{\Delta E}{\Delta s_j} \cdot x_i \tag{3.22}$$

The advantages of MRIII are :

- ☐ Only a feed forward path is necessary. Unlike backpropagation, the backward path is not used to calculate the weight updates.
- ☐ Less hardware is necessary to implement MRIII electronicly.
- ☐ The neurons are not bounded to exact and well defined squashing functions *F*, but could use any non-linear squashing function. This means as well that the MRIII algorithm will be less dependent on the non-idealities of the neurons.

The disadvantages of MRIII are :

- ☐ To calculate all the weight updates in a network, it is necessary that all the neurons are perturbed sequentially with a signal δs_j. To approximate the error derivative (3.21), the error is calculated twice. Once before the perturbation $E[s_j]$ and once after the perturbation $E[s_j+\delta s_j]$. Each node disturbance requires the full training set to calculate the error. To approximate the derivative of the error, this needs to be done twice. If the number of neurons in the network is N, then this algorithm will require in theory 2N times more pattern presentations than backpropagation.
- ☐ A multiplication with x_i is necessary to calculate the weight update as can be seen

in (3.22). In hardware this may result in an error in the weight update due to the non-idealities of the multiplier.

Although MRIII is already much better useful for implementation of the algorithm in hardware than backpropagation, it still has disadvantages which could be eliminated.

3.4.3 Implementation

Both algorithms described in the previous paragraphs implement the gradient descent algorithm in its own way. As shown in these paragraphs, they are both sensitive to hardware limitations which are inevitable. The Weight Perturbation described in this thesis will be described in the next chapters and is derived from the MRIII algorithm.

The Weight Perturbation algorithm requires even more presentations of training patterns than the backpropagation and the MRIII algorithm. However, it is expected that the WP algorithm is the easiest to implement and is more tolerant to non-idealities in the chip. Therefore the WP algorithm is suited for an implementation on a neural network chip. The algorithm behavior and the influence of hardware limitations on this behavior is described extensively in the next chapters.

4 Weight Perturbation

4.1 Introduction

As shown in the last chapter, many algorithms have been developed to train a neural network. These algorithms may work properly in computer simulations. But when the algorithm is implemented in analog hardware on a neural network chip, the physical hardware limitations cause a deterioration in the performance of the algorithm which could be unacceptable. Currently used hardware still suffers from limitations like offset, non-linear multipliers and non-ideal neurons. After minimizing all these errors until a minimum, they still cause a certain deterioration in the performance of an algorithm.

To improve the performance of training in neural network chips, new algorithms have been developed. The backpropagation algorithm is a very complex algorithm to implement. It needs a feedforward path as well as a backward path which is even more sensitive to non-idealities. An improvement is the MRIII algorithm which only requires a forward path and multipliers to calculate the error.

Jabri and Flower [11] developed a variant on the MRIII algorithm called *Weight Perturbation* (WP). This algorithm should be even less sensitive to the hardware limitations than MRIII. In their proposed WP algorithm, the derivative of the error function E is approximated for each weight separately by a forward difference method according to :

$$\frac{\partial E}{\partial w_{ji}} \approx \frac{E(w_{ji}+\delta w_{ji}) - E(w_{ji})}{\delta w_{ji}} \tag{4.1}$$

where :
- E : an error function, for example (3.6) or (3.7), but not necessarily
- w_{ji} : weight selected by weight router
- δw_{ji} : perturbation added to the weight w_{ji}

Use of the forward difference method results in a good approximation of the derivative of the error function E when the perturbation δw_{ji} is chosen small enough. The derivative (4.1) is required to calculate the weight updates Δw_{ji} according to :

$$w_{ji}^{new} = w_{ji}^{old} - \eta \cdot \frac{E(w_{ji}+\delta w_{ji}) - E(w_{ji})}{\delta w_{ji}} \qquad (4.2)$$

In this way, the gradient descent algorithm of (3.9) is approximated. Every weight in the network is updated sequentially using (4.2). The WP algorithm is implemented according to the following actions :

1. After presenting all the input training patterns X, the error $E(w_{ji})$ is calculated using the measured outputs Y and the target patterns D according to (3.6).
2. A perturbation δw_{ji} is added to the selected weight w_{ji}.
3. The new error $E(w_{ji} + \delta w_{ji})$ is calculated.
4. The weight update Δw_{ji} is calculated according to (4.2).
5. Stop when an acceptable error ε is reached. If not then select the next weight in the neural network and go to action 1.

The weights are sequentially updated. If all the weights in the network are updated, the first weight in the network is selected again etc. The process stops when an acceptable error ε is reached, for which the problem is considered to be learned enough for a useful application of the neural network chip after the training phase.

Compared with the backpropagation algorithm, where only one time the entire training set is presented to the network to update all the weights once, the WP algorithm needs to present all the training patterns twice (action 1 and 3) to calculate the error derivative along a weight dimension. If M is the number of weights in a neural network, the WP algorithm requires approximately 2M times more presentations of the training patterns than the backpropagation algorithm to update the weights. Larger networks will thus increase the speed difference between both algorithms. The WP algorithm is a very slow algorithm which is its major drawback. Speed is traded in for better practical implementation compared with the backpropagation algorithm.

4.2 Analysis of the Weight Perturbation algorithm

The WP algorithm is based upon the gradient descent weight update rule (3.9). The error derivative needs to be calculated to perform the weight updates. To understand the algorithm better, we study how the derivative of a function f(x) can be approximated by the following two methods :

- *Forward Difference Method (FDM)*

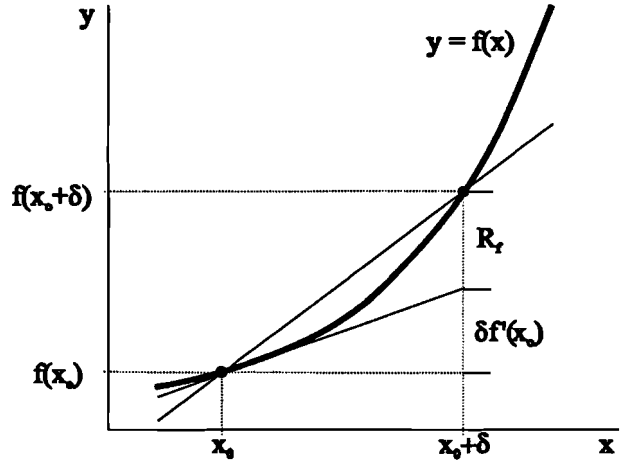


figure 4.1 : Approximation of function derivative using a forward difference method

Every continuous function f(x) at a point x_0 can be approximated around this point by means of a Taylor-series according to :

$$f(x_0+\delta) = \sum_{n=0}^{\infty} \frac{\delta^n f^{(n)}(x_0)}{n!} = f(x_0) + \delta f'(x_0) + \frac{\delta^2}{2} f''(x_0) + .. \tag{4.3}$$

To calculate the derivative $f'(x_0)$, two points of the function f(x) need to be calculated and the derivative can be approximated as shown in the next equation :

$$\frac{f(x_0+\delta)-f(x_0)}{\delta} = \sum_{n=1}^{\infty} \frac{\delta^{n-1} f^{(n)}(x_0)}{n!}$$
$$= f'(x_0) + \frac{\delta}{2} f''(x_0) + ..$$
$$= f'(x_0) + O(\delta) \tag{4.4}$$

The derivation of the function at a point x_0 is approximated with an error remainder in the order of δ. If δ is chosen small enough, (4.4) results in a good approximation of the derivative $f'(x_0)$.

□ *Central Difference Method*

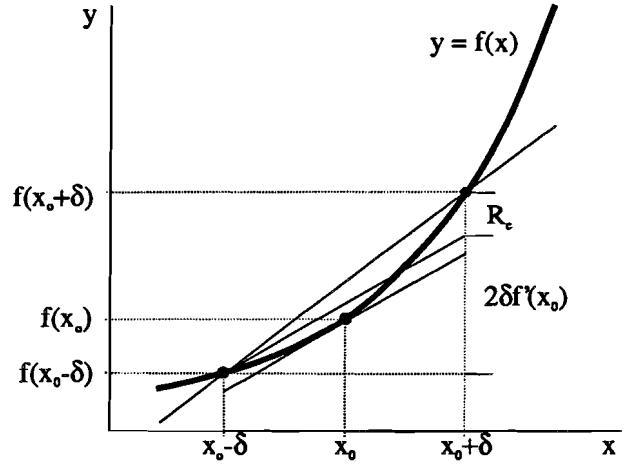


figure 4.2: Approximation of function derivative using a central difference method

As opposed to the forward difference method used in the original WP algorithm of Jabri and Flower, the central difference method does not take into account the value of $f(x_0)$ to approximate its derivative at that point. Two perturbations are added. One with a value of $+\delta$ and one of $-\delta$. The derivative approximation is calculated according to the next equations using the Taylor-series of $f(x_0+\delta)$ and $f(x_0-\delta)$:

$$\frac{f(x_0+\delta) - f(x_0-\delta)}{2\delta} = \sum_{n=0}^{\infty} \frac{\left(1-(-1)^n\right)\delta^{n-1} f^{(n)}(x_0)}{2n!}$$
$$= f'(x_0) + \frac{\delta^2}{6} f'''(x_0) + ..$$
$$= f'(x_0) + O(\delta^2) \tag{4.5}$$

The central difference method approximates the derivative of a continuous function with an error rest-term in the order of δ^2. Small perturbations δ result in a good approximation of the derivative $f'(x_0)$.

When a comparison is made between both perturbation methods, it is necessary to calculate the error remainder exactly. These error remainders, $R_f(x_0,\delta)$ for the forward difference method and $R_c(x_0,\delta)$ for the central difference method, are given in the next equations :

$$R_f = \sum_{n=2}^{\infty} \frac{\delta^{n-1} f^{(n)}(x_0)}{n!} \tag{4.6}$$

$$R_c = \sum_{n=3}^{\infty} \frac{\left(1-(-1)^n\right)\delta^{n-1} f^{(n)}(x_0)}{2n!} \tag{4.7}$$

Depending on these error remainders, a method has to be chosen which results in a minimum error term R in the derivative approximation. This is extensively described in paragraph 5.2 where the influence of the perturbation on the WP algorithm in an ideal network is studied.

The original weight perturbation described by Jabri and Flower uses the forward difference method. The weights w_{ji} are stored on capacitors C_{ji} in the network. Normally the capacitors will discharge due to leakages and the weight will decay. The Forward Difference method is chosen, because they suppose there is no leak at the capacitors C_{ji} occurs and it is therefore not necessary to refresh the weights. The forward difference method has two advantages over the central difference method. The access to a weight is only required once during one perturbation as opposed to the central difference method which always requires two weight accesses for two perturbations. However, as will be described in paragraph 6.3 where the weights leak and their values w_{ji} alter unwanted, the weights need

to be refreshed with their original value before an error E can be measured. The second advantage is that the central difference method requires an extra error measurement to calculate the error $E(w_{ji})$ to check whether the stop criterion has been reached. The disadvantage of the feedforward difference method, is that in most cases, it approximates the error derivative less accurate.

Based on the two advantages Jabri and Flower chose the forward difference method in their implementation of the WP algorithm as will be described in the next paragraph.

4.3 Implementation of the WP algorithm

As opposed to the backpropagation algorithm, the WP algorithm only uses a feedforward path in the neural network and does not require extra multipliers like the MRIII algorithm. Therefore the hardware requirements for implementing the WP algorithm are reduced to a minimum. The minimum hardware requirements which are always necessary are shown in the next figure.

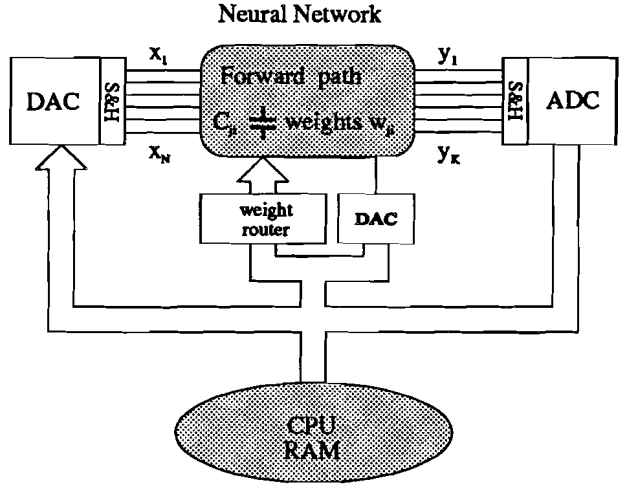


figure 4.3 : Minimum hardware requirements to realize a feedforward path in a neural network

A Central Processing Unit (CPU) controls the data flow on the databus and is the controller of the total system. In our case, when the WP algorithm itself is not implemented on the neural network chip, the CPU is used to calculate the weight updates Δw_{ji}. After selection criteria like speed and price, a PC using a 80486 processor is chosen for this

purpose [25].

The input patterns are stored in an external database RAM or are presented to the network on-line after the network has learned a certain problem. These digital patterns are converted by a Digital Analog Convertor (DAC) and the analog inputs X are presented to the input layer of the MLP. In our case, Sample and Hold (S&H) circuits are necessary to present the input patterns to the network until the responding outputs Y are stabilized. To present more input patterns to the network, pipelining of the input patterns could be used. I.e., a new pattern is already presented to the network, while the signals from the old pattern are still transferred through the network. Because this results in a very critical timing, pipelining is not used in our circuits. The input S&H circuits make it also possible to use a single DAC and multiplex its output to multiple input lines X. The outputs Y from the output layer of the MLP are sampled by the S&H circuits and converted to a digital value by one or more ADCs. These digital values can be stored in the RAM and are used by the CPU to calculate the error function (3.6).

The feedforward path of a neural network is realized on the chip using CMOS techniques. The values of the weights w_{ji} are stored on capacitors C_{ji} by an internal S&H. Any algorithm needs access to these capacitors C_{ji} to alter the weights w_{ji}. A selection mechanism is required which provides analog access to each of the capacitors individually. This is done by a weight router in combination with a DAC. The hardware of figure 4.1 suffices for individual weight access and thus suffices for the Weight Perturbation algorithm. Values of the weights w_{ji} stored in the RAM can be transferred to analog values on the capacitors C_{ji}.

Because the values of the voltages on the capacitors are stored only temporarily on the chip, these values always have to be stored in the RAM or for long term storage on a data storage device. Due to the leakage in the internal S&H circuits, the voltages V_{ji} on the capacitors C_{ji} need to be refreshed periodically to guarantee proper weight values. Refreshing is always necessary, as well during the training period as during the actual use of the neural network chip after the network learned a certain problem.

During the training period, the weight refreshes are combined with the updates of the Weight Perturbation algorithm. In this way, two necessary capacitor accesses are combined. This is an essential property of the WP algorithm which other algorithms do not

have. Besides the insensitivity to hardware limitations, this is the main reason why is chosen for the WP algorithm to implement in an analog CMOS neural network chip.

4.4 Benchmarks

In order to test the weight perturbation algorithm, benchmarks have to be presented to the neural network. A benchmark is a problem used in the training of a neural network with which the performance of an algorithm can be studied and compared with other algorithms and situations. Performance properties of an algorithm are for example the convergence speed and the minimum error which can be attained. Because the neural network chip is not designed to be used for a special purpose, but will be a general demonstration model, various benchmarks can be chosen.

Several benchmarks are used in the literature. However, the problem is that every researcher uses a different benchmark. Some of the benchmarks used are the same, but certain aspects are interpreted differently, e.g. the error function E, the separation in converged trials and non-converged trials or averaging of errors. That is why is it hard to say something about the comparison between different studies done into the performance of neural network algorithms. However, several benchmarks are used quite often in the literature and can be seen as standards. A list of some of these benchmarks [7,17] is given below :

- ☐ *XOR* — The task is to train a network to produce the boolean function Exclusive OR (XOR) of two variables.
- ☐ *Parity* — The parity benchmark is essentially an extension of the XOR problem and the task is to train a network to produce the sum, mod 2, of N binary inputs.
- ☐ *Encoder/Decoder* — The task is to find an efficient set of hidden unit patterns to encode/decode a large number of input/output patterns. The number of hidden units is intentionally made small to enforce an efficient encoding.
- ☐ *Curve Fitting* — The task is to train a network to produce a certain function y = F(x).
- ☐ *Signal prediction* — The task is to train the network so, that it produces a sample y(k+n), given a certain relationship between y(k+1) and y(k).

- ☐ *Sonar* The task is to train a network to discriminate between sonar signals bounced off a metal cylinder and those bounced off a roughly cylindrical rock.
- ☐ *Vowel recognition* Speaker independent recognition of the eleven steady state vowels of British English.
- ☐ *NETtalk* The task is to train a network to produce proper phonemes, given a string of letters as input.

Most of these benchmarks use a large number of training patterns. Complex problems like Sonar, Vowel recognition and NETtalk require networks with a large number of neurons. These benchmarks are too big to simulate with the WP algorithm within a reasonable time. As is shown in paragraph 4.1, 2M batches are necessary to update all the weights (M) once. If the number of weights M and the number of training patterns P are large, long computer simulation time is necessary until the network converges to a certain minimum error ε.

Long simulation times restrict the number of benchmarks which are suitable to study the performance of the WP algorithm. Three benchmarks have been selected which represent three different kind of problems.

* *XOR-problem (Digital)*

The XOR benchmark is probably one of the most used and studied benchmark. The task is to train the network the boolean function XOR. The realized function XOR function uses two inputs X of the network and one output Y and the network is trained to realize the function according to :

x_1	x_2	y
1	1	+0.9
1	-1	-0.9
-1	1	-0.9
-1	-1	+0.9

To train the XOR function, 4 training patterns (P = 4) are necessary. The XOR function

can be realized with a 2-2-1 MLP. The networks contains 3 neurons and 9 weights. The neurons use a Tanh squashing function (2.3) with a temperature β=1. The target values are not +1 or -1 because otherwise the output neuron should be learned completely in the tail of the squashing function and a target value will never be obtained exactly.

* *SINE-problem (Analog)*

The task is to train the network the function y = sin(x) between $-\pi$ and π. Because the problem is analog, the training set consists of representative samples of the function values. Chosen is for 37 sample points (P =37) and the training set can be described by the following function :

$$x^p = \left(-1 + \frac{(p-1)}{18}\right)\cdot\pi \;,\; p = 1..37$$
$$d^p = \sin(x^p) \tag{4.8}$$

The network which is suitable to train this function is a 1-3-1 MLP containing 4 neurons and 10 weights. The three hidden neurons use Tanh squashing functions (2.3) with a temperature β=1. Because the target values become close to one, a linear output neuron is used which characteristic function is $F(x)=x$. In this way output values of 1 and -1 can be obtained.

* *LINES-problem (Classification)*

Both the XOR- and the SINE-problem are very small benchmarks. To give a more complete representation of the problems which could be used to train a neural network, a larger benchmark is chosen. The LINES benchmark classifies its input patterns into three classes. The input of the neural network consist of 16 input nodes representing a 4x4 matrix. This is illustrated in the figure below :

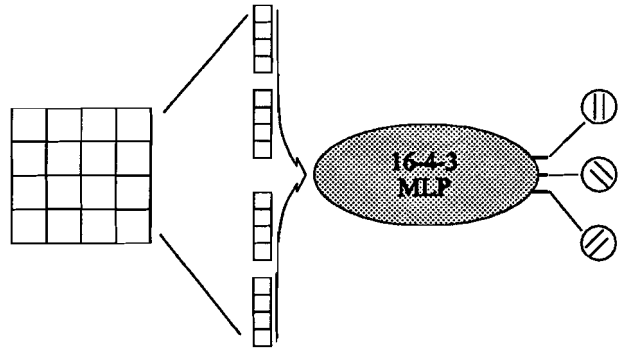


figure 4.4 : Implementation of the LINES-problem

The input array consist of lines which are represented by a value of 1 in the matrix. Empty elements are represented by a value of -1. The lines can be divided in three classes, vertical,left diagonal and right diagonal lines. A 16-4-3 MLP with 7 neurons and 83 weights is used in the training. All the 7 neurons implement the Tanh squashing function (2.3) with β=1. The network is trained until target values of +0.9 and -0.9 are reached according to :

CLASS	y_1	y_2	y_3
Vertical	+0.9	-0.9	-0.9
Left diagonal	-0.9	+0.9	-0.9
Right diagonal	-0.9	-0.9	+0.9

The three classes are separated in the outputs Y of the network. For every class one output neuron gets *high* while the other two stay *low*. Five training patterns each consisting of one ore more lines are chosen out of every class and presented to the network. The total number of training patterns P is 15.

In the used networks for these benchmarks, the bias value θ is simulated by a constant 1 multiplied with a weight w_{j0}. Most of the simulations are done with different initial weight values. The initial weight values are random chosen in the range [-½,+½]. In most of the simulations the total square error E_{TSE} is used. Except in paragraph 6.3 where the mean square error E_{MSE} is used.

The three benchmark problems described in this paragraph, are used in this thesis to test the WP perturbation algorithm. A complete test of the WP algorithm is impossible because then a large number of benchmarks is needed. With these three problems it is tried to present the network different kind of problems using different network sizes. It can be said that these problems represent a large amount of problems used to train neural networks. Of course more benchmarks could have been chosen, but due to time limitations the choice is made for a more extensive study of different kind of aspects of the WP algorithm, than a thoroughly study of a few aspects using many benchmarks.

The networks are tested with computer simulations as will be described in the next paragraph.

4.5 Simulations

In order to test the WP algorithm, computer simulations are necessary to study how the algorithm behaves in an ideal MLP and the response to influences of non-idealities of the analog neural network chip. First a study is done into the behavior of the WP algorithm in ideal networks as will be described in chapter 5. These networks introduce no errors and can be seen as ideal MLPs. These perfect networks are only possible in computer simulations, but are impossible to realize on an analog neural network chip. Therefore, in chapter 6 hardware limitations will be introduced which occur in the chip.

To study the WP algorithm, simulations have to be done before the actual neural network chip is made. A neural network simulator *ANANAS* has been developed which provides a basis on which various simulations of the WP algorithm can be done. The simulator is written in C++ and can therefore easily be adapted to introduce certain effects. The simulations presented in this thesis have been done using various SUN Sparc workstations (models SLC, 2 and 10). The workstations use 32 bit words for the storage of floating point variables. Due to this large wordlength, the signals used in the network can be considered as analog values.

In this chapter, the theoretical foundations and the practical implementation were presented to understand the WP algorithm. In the next chapter the WP algorithm is tested in an ideal MLP network.

5 Behavior of the WP algorithm in an ideal MLP

5.1 Introduction

Because very few research has been done into the WP algorithm [11,13], in this thesis simulations are made in order to establish what the behavior and performances of the algorithm are.

The algorithm needs to be simulated in an ideal MLP as described in paragraph 2.2, in order to yield information about the algorithm itself. Unwanted influences caused by the hardware limitations are eliminated and the parameters of the algorithm can be studied. Later on, in chapter 6, the influence of non-idealities caused by the hardware is studied.

The learning rate η and the perturbation δw_{ji} are parameters which influence the behavior and performance of the WP algorithm. These parameters are studied in the next paragraphs.

5.2 Influence of the perturbation

When the algorithm is implemented in hardware, the perturbation step is preferred to be small compared to the weight range. A perturbation can be implemented in two ways :

- Add a digital value to the weight which is stored in the RAM before it is converted into an analog value is made by the DAC (fig 4.3). The perturbation is restricted by the wordlength and the linearity of the DAC.
- Add an analog value at the analog output of the DAC. The perturbation can not be chosen so that it disappears in the noise on the analog weight value.

Both methods have their limitations of the weight perturbation size. It has not been decided which method will be used. However, it is useful to study the error performance depending on the perturbation step so that later can be decided which perturbation value is acceptable and the appropriate hardware can be realized.
As shown in paragraph 4.2, the accuracy of the approximation of the error derivation

depends on the perturbation δw_{ji}. To calculate the error derivative exactly, δw_{ji} should be infinitely small. Thus when $\delta w_{ji} \rightarrow 0$, the error remainders R_f (4.6) and R_c (4.7) will approach zero, $R \rightarrow 0$. However, this can not be realized when the weight perturbation is implemented on a chip. Even in the computer simulations δw_{ji} has a minimum finite value equal to the Least Significant Bit (LSB) of the computer where the neural network simulator is compiled on. Therefore the perturbation in computer simulations should be chosen larger than the LSB in order to prevent round off errors.

Finite values of δw_{ji} have to be taken into account. A finite value of δw_{ji} will have its influence on the minimum error E_{min} which can be attained by the WP algorithm during the training process of a problem. To study this relationship we first examine the simplified theory of the learning process of the WP algorithm. This model can be extended to real neural networks.

To study the relationship between the E_{min} and δw_{ji}, it is necessary to know how the error landscape behaves when the weight set W in a neural network is trained to an optimum by the algorithm. The weights w_{ji} are all close to their optimum values. According to (3.4), an output y_k of the network can be described by :

$$Y = \Omega(X,W) \tag{5.1}$$

where :
- Y = outputs in the output layer of a MLP.
- Ω = total network transfer.
- W = all the weights in a MLP.
- X = all the presented input patterns.

This formula relates the weights and inputs of the network via a certain network transfer to outputs Y in the output layer of the MLP. From (5.1) it can be seen that there exists a certain relationship between the outputs Y and a weight w_{ji} when all the other weights and the inputs X are seen as constants. If a small perturbation $\delta w_{ji}=\hat{w}_{ji}-w_{ji}$ is added to a weight w_{ji}, the output response can be approximated as :

$$Y + \delta Y \approx \Omega(X,W) + \delta w_{ji} \cdot \frac{\partial \Omega(X,W)}{\partial w_{ji}} \tag{5.2}$$

When the neural network is trained until the entire weight set is optimized and the outputs Y reach their optimum values Y_0, the optimum error E_0 is reached :

$$E = \frac{1}{2}\sum_{p=1}^{P}\sum_{k=1}^{K}\left(\left(y_k^p\right)_0 - d_k^p\right)^2 = E_0 \tag{5.3}$$

The distance between Y_0 and D can not be reduced anymore. Preferably E^0 equals zero but this is not always the case. When a perturbation δY is added to the optimum value the error changes to a value which is always larger than the optimum E_0. This illustrated in the next figure where the points are situated in a PxK dimensional space :

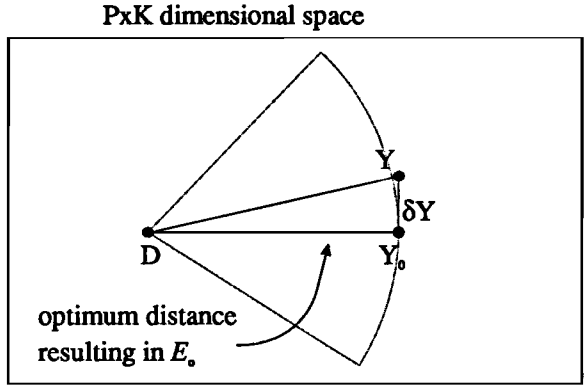


figure 5.1 : Distance between desired values D and optimum outputs Y when a weight perturbation δw_{ji} is made

A small perturbation should be applied in two direction and a perturbation can not be made in the circle, otherwise Y_0 would not have been the optimum. Therefore it is expected that small perturbation is perpendicular on the line between Y_0 and D. When the perturbation δw_{ji} causes a disturbance δY in the outputs, the error function can be described as :

$$\begin{aligned} E &= E_0 + \frac{1}{2}\sum_{p=1}^{P}\sum_{k=1}^{K}\left(\hat{y}_k^p - \left(y_k^p\right)_0\right)^2 \\ &= E_0 + \frac{1}{2}\sum_{p=1}^{P}\sum_{k=1}^{K}\left(\delta y_k^p\right)^2 \end{aligned} \tag{5.4}$$

where δy_k^p is determined by (5.3) and can be described as :

$$\begin{aligned} \delta y_k^p &= \delta w_{ji} \cdot \frac{\Omega_k^p(X,W)}{\partial w_{ji}} \\ &= \delta w_{ji} \cdot \psi_k^p \end{aligned} \tag{5.5}$$

Combining (5.4) and (5.5) results in :

$$\begin{aligned} E &= E_0 + \delta w_{ji}^2 \cdot \frac{1}{2} \sum_{p=1}^{P} \sum_{k=1}^{K} \left(\psi_k^p\right)^2 \\ &= E_0 + \Psi \cdot \delta w_{ji}^2 \\ &= E_0 + \Psi \cdot \left(\hat{w}_{ji} - (w_{ji})_0\right)^2 \end{aligned} \tag{5.6}$$

Formula (5.6) shows a quadratic dependency of the error E to the perturbation δw_{ji}, multiplied with a constant Ψ. If the network is trained to an optimum weight set, the error behavior in one weight direction equals (5.6). This assumption is tested in the simulations of the benchmarks. An example of this square relationship is shown in the figure below :

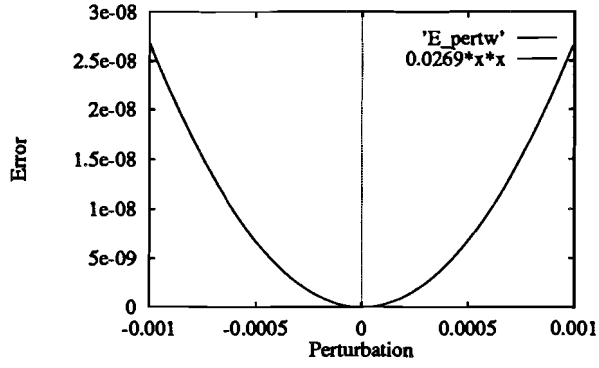


figure 5.2 : An example with the XOR-problem of the relationship between the error E and a perturbation δw_{ji} around an optimum weight value $(w_{ji})_0$. This simulation is compared with the model $\Psi \delta w_{ji}^2$ for Ψ=0.0269. The model and simulation are similar.

It can thus be assumed that near the optimum value of the weights, the error landscape looks like a parabola. Using a simple demonstration model of the error landscape, the two error derivative methods are implemented in the WP algorithm in order to converge to a minimum error E_{min}. The model consist of only a single weight w and the relationship between the error E(w) and w can be described by $E(w)=E_0+\Psi(w-w_0)^2$.

The forward difference method (FDM) implemented in the WP algorithm results in the following weight update rule according to (4.2) :

$$w^{new} = w^{old} - \eta \cdot \frac{E(w+\delta w) - E(w)}{\delta w} \tag{5.7}$$

The FDM converges to a minimum error E_{min} when there are no weight updates Δw anymore, and thus $w^{new} = w^{old}$. If Δw is zero, E_{min} is calculated by solving (5.7) with this condition :

$$\begin{aligned} & -\eta \cdot \frac{E(w_{min}+\delta w) - E(w_{min})}{\delta w} = 0 \\ & \Psi \cdot \left((w_{min}-w_0)+\delta w\right)^2 - \Psi \cdot (w_{min}-w_0)^2 = 0 \\ & \Rightarrow w_{min} = -\frac{1}{2}\delta w + w_0 \\ & \Rightarrow E_{min} = E_0 + \frac{1}{4}\Psi \cdot \delta w^2 \end{aligned} \tag{5.8}$$

A square relationship is found between the minimum error E_{min} and the perturbation δw. The FDM WP algorithm can not converge to an error E_0 when a finite perturbation is used to approximate the error derivative.

The central difference method (CDM) results in the following weight update rule according to (3.9) and (4.5) :

$$w^{new} = w^{old} - \eta \cdot \frac{E(w+\delta w) - E(w-\delta w)}{2\delta w} \tag{5.9}$$

The algorithm is converged when $\Delta w = 0$. This means that $w^{new} = w^{old}$ and a minimum error can be found by solving :

$$\begin{aligned} & E(w_{min}+\delta w) - E(w_{min}-\delta w) = 0 \\ & \Psi \cdot \left((w_{min}-w_0)+\delta w\right)^2 - \Psi \cdot \left((w_{min}-w_0)-\delta w\right)^2 = 0 \\ & \Rightarrow w_{min} = w_0 \\ & \Rightarrow E_{min} = E_0 \end{aligned} \tag{5.10}$$

The CDM results in an optimal $E_{min} = E_0$ and is thus preferred over the FDM. Both error derivative approximations are shown in the next figure where for the optimum weight value $w_0=0$ and the optimum error $E_0=0$:

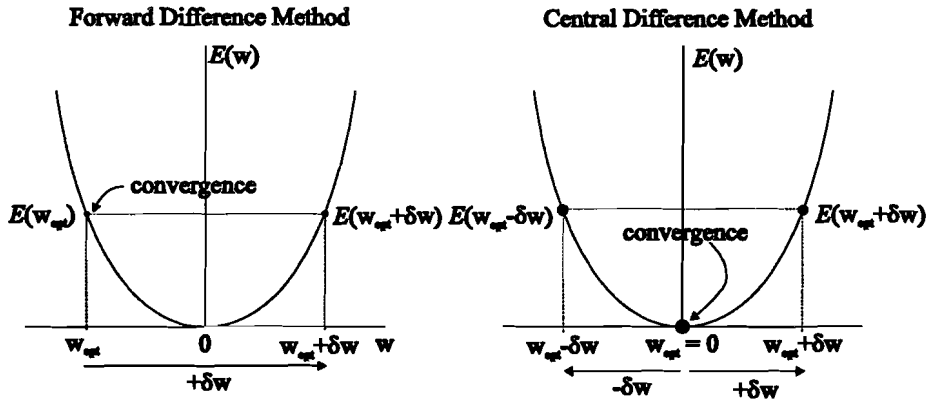


figure 5.3 : Two error derivative approximations used in the WP algorithm to locate an optimum weight w_{ji} corresponding with a minimum error E_{min}.

In the case of the CDM, it is shown that an E_{min} of zero can be found independent of the perturbation δw_{ji}. However, when the WP algorithm is implemented using a CDM an extra action is required to calculate the real error $E(w_{ji})$. Two perturbations are added and $E(w_{ji}+\delta w_{ji})$ and $E(w_{ji}-\delta w_{ji})$ are used to calculate the weight update Δw_{ji} according to :

$$w_{ji}^{new} = w_{ji}^{old} - \eta \cdot \frac{E(w_{ji}+\delta w_{ji}) - E(w_{ji}-\delta w_{ji})}{2\delta w_{ji}} \qquad (5.11)$$

Instead of two times, three times the entire training set needs to be presented. Twice to update the weight and once to calculate afterwards the real error $E(w_{ji})$. Of course this should not be done after each weight update, but e.g. after all the weights in the network are updated once.

If the perturbed error $E(w_{ji}+\delta w_{ji})$ is used as an approximation of the real error $E(w_{ji})$ of the neural network, it is not necessary to calculate $E(w_{ji})$ and the CDM only requires two times the entire training set in stead of three. However, the minimum error will be $E_{min} = \Psi \delta w_{ji}^2$. In this case E_{min} is four times larger than found with the FDM (5.7).

When the theory above is extended to networks with multiple weights, the perturbed error $E(w_{ji}+\delta w_{ji})$ is a function of all the weights in the network, but only one weight w_{ji} is perturbed. This means that for large networks the measured perturbed error approximates the real error $E(w_{ji})$ where all the weight have their unperturbed values.

This can be shown using the extended model of the error function to multiple weight dimensions, eg :

$$E = E_0 + \sum_{m=1}^{M} \Psi_m \cdot \delta w^2 \tag{5.12}$$

Using the FDM, this results in a minimum error E_{min} of :

$$E_{min} = E_0 + \frac{1}{4} \cdot \delta w^2 \cdot \sum_{m=1}^{M} \Psi_m \tag{5.13}$$

The central difference method is independent of the number of weights M in the neural network and $E_{min} = E_0 + \Psi_m \delta w^2$. This results in a different error for each weight direction w_m because the values of Ψ_m are different in each direction. However, when Ψ_m is the same for all directions and equals Ψ, the CDM results in a smaller minimum error when the number of weights $M > 4$. This can not be expected, but in general it could be said that the difference in error performance between both methods becomes larger when the network contains more weights.

The above theory is a simplified model of the behavior of the error landscape around the optimum values. However, with this theory the simulations of the influence of the perturbation on the WP algorithm will be understood better.

The two error derivative methods have been simulated with the three benchmarks. The results of these simulations are summarized in the following figures 5.4, 5.4 and 5.6. The algorithm stops when $0.999*E(t-1)<E(t)<E(t-1)$ for 100 consecutive steps.

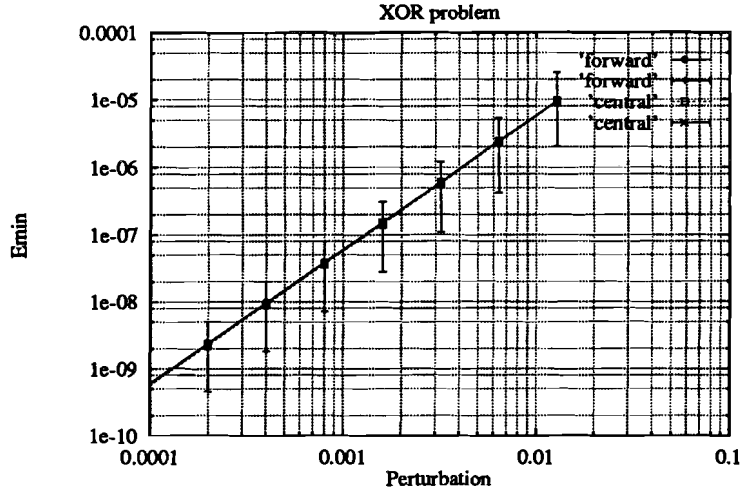


figure 5.4 : Training the network with the XOR problem using the FDM and CDM WP algorithm. η=0.1, #trials=11

The bars in this figure show the maximum and minimum obtained error E_{min}. This is caused by the implemented stopcriterion. Training the XOR problem using the CDM results in hardly any improvement. The achieved minimum error E_{min} decreases with 4% compared with the FDM. An exact quadratic relation between E_{min} and δw_{ji} can be seen in this figure.

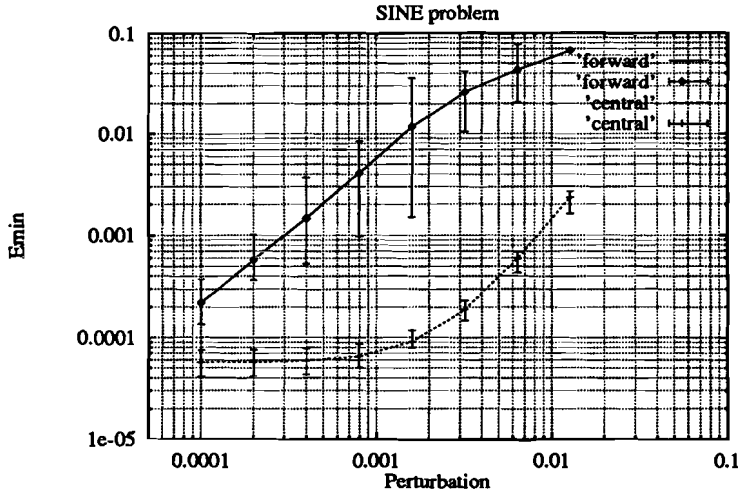


figure 5.5 : Training the network with the SINE problem using the FDM and CDM WP algorithm. η=0.01, #trials=11.

The SINE problem results in an improvement of the minimum error E_{min} using the CDM. The FDM is not completely a straight line. Near E_{min}≈0.1 the line bends because the convergence criterium used in the simulation was 0.1 and some trials have not converged using this criterium.

Due to the limited network size it is not possible to train the network to an error smaller than approximately 6e-5. This limit is independent of the used approximation method or algorithm. In figure 5.5 it can be seen that the CDM reaches this limit when δw≈1e-3. The FDM requires perturbations δw smaller than 1e-4. To converge to the same minimum error E_{min}, the CDM allows ≈30 times larger perturbations than the FDM. This means that the demands posed upon the magnitude of the perturbation step size are weakened.

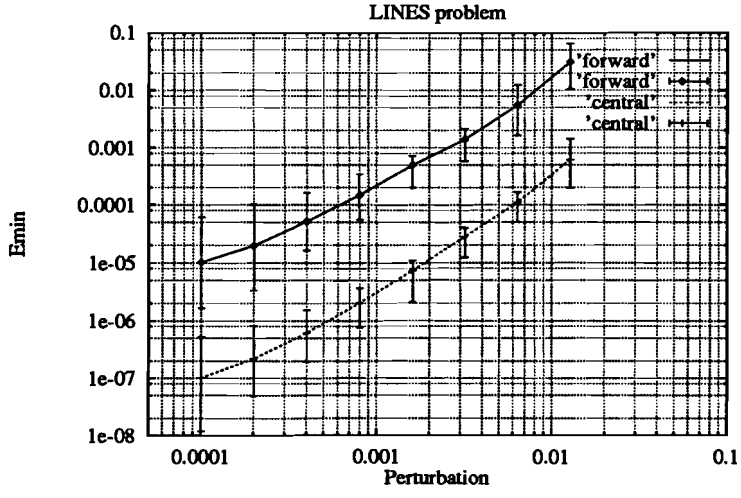


figure 5.6 : Training the network with the LINES problem using the FDM and the CDM WP algorithm. η=0.1, #trials=11

Again the CDM results in a better error performance than the FDM. As seen in the figure, the perturbation step size can be about ≈11 times larger to result in the same E_{min}.

The simulation results have shown that the minimum error E_{min} which can be achieved depends on the magnitude of the weight perturbation δw_{ji}. A smaller perturbation results in a smaller E_{min}. This approximately quadratic relationship is also derived from the theory. Preferred is the smallest physical possible weight perturbation δw. The simulations have shown that the CDM is an improvement in error performance over the FDM.

As stated at the begin of this paragraph, the perturbation can not be made too small. Therefore it can be concluded that the use of the CDM is preferred because larger perturbations can be used while the error performance stays the same.

5.3 Influence of the learning rate

As has been shown in paragraph 3.3.3, the learning rate η is an important factor in the speed of any gradient descent algorithm. The learning rate η should be chosen such, that a maximum convergence speed is achieved. There is an upper limit η_{max} for the learning rate η depending on the error landscape of every problem.

As shown in equation 3.9 and its derivatives, the weight updates Δw_{ji} in any gradient descent algorithm are proportional to the learning rate η. The magnitude of the weight updates Δw_{ji} determines the convergence speed of the algorithm. The optimum weight set is found faster when the weight updates are larger. The convergence time is thus inverse proportional to the learning rate η under the condition that $\eta < \eta_{max}$.
Simulations have been carried out to verify the relation between the convergence time and η. These simulations were done using the three benchmarks and the results are summarized in the next figure.

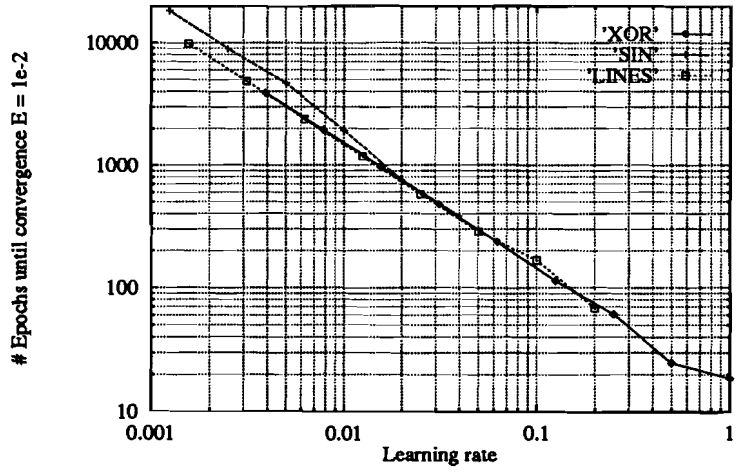


figure 5.7 : The number of epochs necessary to converge to an error E_{TSE} = 1e-2 as a function of the learning rate η. FDM δw=1e-4, XOR #trials 21, LINES, SIN #trials 11.

From the simulations it can be verified that there exists an inverse proportional relation between the convergence time and η. The learning rate can not be chosen too large because it results in no converged trials. The maximum learning rate η_{max} in this simulation is a learning rate where there is still no significant decrease in the number of converged trials. The maximum learning rates η_{max} are approximately : $\eta_{max}^{XOR} \approx 0.4$, $\eta_{max}^{LINES} \approx 0.2$ and $\eta_{max}^{SINE} \approx 0.04$. When the E_{MSE} error (3.7) is used in stead of the normal used E_{TSE} error (3.6), the values of η could be multiplied with the number of output neurons (K) and the number of presented patterns (P). The E_{MSE} error is normalized and (3.9) shows that when the same weight updates are made, η should compensate the normalization of the error and can be multiplied with KP :

$$\eta_{MSE} = KP \cdot \eta_{TSE} \tag{5.14}$$

In this chapter we have studied the behavior of the WP algorithm in an ideal MLP network. It is demonstrated that the WP algorithm according to Jabri and Flower works, but is a slow algorithm according to what was expected. The simulations using an ideal network are only possible in computer simulations. When the WP algorithm is implemented in hardware various non-idealities will occur. The influence of these non-idealities is studied in the next chapter.

6. Behavior of the WP algorithm in a non-ideal MLP

6.1 Introduction

In chapter 5 we studied the behavior of the WP algorithm in an ideal MLP. The goal is to design a neural network chip where the WP algorithm is implemented on.

Implementation of the algorithm in a neural network with non-idealities causes changes in the behavior of the convergence speed and minimum error E_{min}. To study the change in behavior due to hardware limitations, simulations have to be made to say something about how large the unwanted influences can be that still result in a proper behavior.

The neural network chip has several non-idealities. In this chapter the most important influences are studied. Four influences are expected to result in a change of the algorithm behavior :

- ☐ Quantization of the weights caused by the DAC.
- ☐ Decay of the weights caused by current leaks in the weight capacitors.
- ☐ Non-idealities in the multipliers.
- ☐ Non-idealities in the neurons.

These most important hardware limitations are described in the next paragraphs.

6.2 Weight quantization

As described in chapter 4, the weights in a MLP neural network need to be stored in a RAM memory for storage during the training and long-term storage after training. The values of the internal weights stored on capacitors alter continuously due to leaks in the internal S&H circuits which result in the discharge of the capacitors C_{ji}. Within a certain time the weights need to be refreshed with their original values stored in RAM.

The DAC (see figure 4.1) translates the dimensionless weight values w_{ji} in the RAM to analog voltages on the capacitors C_{ji}. Due to the finite word-length (N) of the DAC, the weights w_{ji} are quantized values. It is therefore that only N bits need to be stored in the RAM. The word-length of the DAC is the bottleneck.

The analog voltages on the capacitors will have discrete levels V_{ji}^Q corresponding with discrete weight values w_{ji}^Q. Due to the finite word-length N, the minimum quantization step Δ of the discrete weights w_{ji}^Q is :

$$\Delta = \frac{R}{2^N - 1} \tag{6.1}$$

where R = range of the weights
N = number of bits used to quantize the weights

The total range of the discrete weights is thus defined as :

$$w_{ji}^Q \in \left[-\frac{1}{2}\Delta \cdot (2^N - 1), \frac{1}{2}\Delta \cdot (2^N - 1) \right] \tag{6.2}$$

(6.2) only applies when the weights values are symmetrical around zero. An offset can be added to adapt the weights to certain voltages where they are more practical implemented. When no offset is added $w_{ji}^Q \in [-½R,½R]$. This situation is used in all the simulations where the range R = 5. This range was chosen to eliminate the problem of weight clipping during the training of a problem. This effect is related to the weight quantization. The weights can grow out of the range (6.2) during the training period. When a weight clips against a limit, this will have effects on the behavior of any algorithm. The clipping of weights can be avoided by scaling of the weights and neurons. A static or adaptive scaling can be used to avoid weight clipping. The weights in the three benchmarks stayed within the range of [-2.5,2.5] and this effect did not occur. However, further literature can been found about the subject in [8,29,31].

In the next paragraphs, the effects of weight quantization will be studied and solutions will be proposed to diminish the effects on the WP algorithm.

6.2.1 Influence of quantization

The weight quantization has a strong effect on the behavior of the error performance of the WP algorithm. Quantization of the weights quantizes and causes plateaus in the error function E which otherwise is a continuous function of the weights when they are continuous values. In order to study the number of bits which can be used in the DAC, a theoretical study and simulations have been done of the relation between the weight wordlength and the minimum error E_{min} which can be obtained by the WP algorithm.

Due to the quantization the weight update rule of (3.9) changes into :

$$w_{ji}^{new,Q} = w_{ji}^{old,Q} - Q\left(\eta \cdot \frac{\partial E}{\partial w_{ji}}\right) \tag{6.3}$$

The effect of quantization caused by the DAC is modeled by a roundoff quantization function Q(x) according to :

$$\begin{aligned} Q(x) &= \frac{1}{2}\Delta\cdot\left(2^N - 1\right) \text{ , } if\ x \geq \Delta\cdot\left(2^{N-1} - 1\right) \\ &= \Delta\cdot k \text{ , } if\ \Delta\cdot(k - \tfrac{1}{2}) \leq x < \Delta\cdot(k + \tfrac{1}{2}) \\ &= -\frac{1}{2}\Delta\cdot\left(2^N - 1\right) \text{ , } if\ x < -\Delta\cdot\left(2^{N-1} - 1\right) \end{aligned} \tag{6.4}$$

Q(x) transforms a continuous signal x into discrete values Δ and limits the range of x to $[-\frac{1}{2}\Delta(2^N-1), \frac{1}{2}\Delta(2^N-1)]$. Equations (6.3) and (6.4) show that there are no weight updates Δw_{ji}^Q made when all the weight updates $|\Delta w_{ji}|$ are smaller than $\frac{1}{2}\Delta$. The algorithm stops when the following condition is reached :

$$\left| \eta \cdot \frac{\partial E}{\partial w_{ji}} \right| < \tfrac{1}{2}\Delta \text{ , } \forall w_{ji} \tag{6.5}$$

This is illustrated in the next figure :

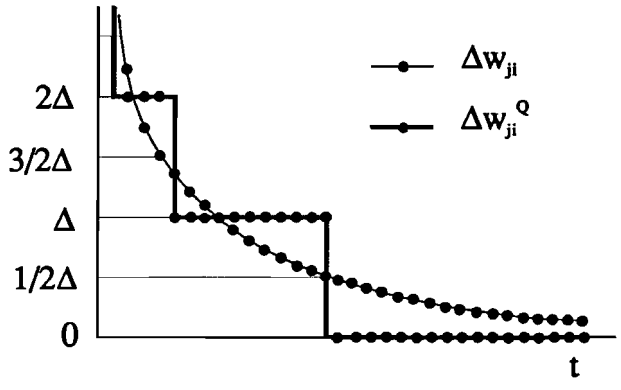


figure 6.1 : Effect of quantization on the weight update Δw_{ji}^{Q} compared with the continuous weight updates Δw_{ji}

It has to be said that figure 6.1 is a simplified perception of the reality. The weight updates Δw_{ji} is not a continuous line but is evaluated every time a weight is selected and will therefore not be a continuous function as shown in the figure. After a weight update Δw_{ji}^{Q} is made the new weight value $(w_{ji}^{new})^{Q}$ is different from the value which was wanted by a weight update Δw_{ji}. This means that the next update Δw_{ji} will be different because the derivative is determined from a point $(w_{ji}^{new})^{Q}$ in stead of w_{ji}^{new}. However, the simplified model suffices to understand the theory of weight quantization.

The minimum error E_{min} to which can be converged with the WP algorithm depends on condition (6.5). When all the weight updates $|\Delta w_{ji}^{Q}|$ are smaller than ½Δ, no further convergence is possible and the algorithm is stuck in the minimum error E^{N}_{min} belonging to that amount of bits (N) used in the quantization.

Given a wordlength N, it is possible to determine the relation between E^{N}_{min} and η. This can be done under the condition that the error function E is close to its optimum E_0. In this case the error function E can be modeled as $E=E_0+\Psi(w-w_0)^2$ according to (5.6) in the case the network contains only one weight. The algorithm stops when $|\Delta w|<½\Delta$. This can be rewritten as :

$$\left|-\eta \cdot \frac{\partial E}{\partial w}\right| = 2\eta\Psi \cdot |w-w_0| < \tfrac{1}{2}\Delta$$
$$\Rightarrow 2\eta\sqrt{\frac{(E-E_0)}{\Psi}} < \tfrac{1}{2}\Delta \qquad (6.6)$$

The minimum error E_{min} is a function of the learning rate :

$$E_{min} = E_0 + \frac{\Psi\Delta^2}{16\eta^2} = E_0 + \frac{C}{\eta^2} \qquad (6.7)$$

where $C=\Psi\Delta^2/16$.

This means that doubling the learning rate η with a factor two, the minimum error E_{min} decreases with factor four when $E_0=0$. This only applies when the wordlength N is large enough for the algorithm to converge the weights near their optimum values. When the wordlength is smaller and E_{min} is not near the optimum, the error landscape does not resemble a parabola anymore and (6.7) does not apply anymore.

According to the theory above, there exists a certain relation between the minimum error E^N_{min} and the number of bits N. When N is large enough the relation is modeled using (6.7). If not, then another relation occurs depending on what the error landscape looks like. If we consider a certain quantization step size Δ_N for N bits weights, (6.1) shows that a quantization step size Δ_M for M bits weights relates to Δ_N according to :

$$\Delta_M \approx 2^{N-M} \cdot \Delta_N \qquad (6.8)$$

When a certain minimum error E^N_{min} is obtained using N-bits weights, the same error can be attained when the weight updates Δw_{ji}^Q are scaled with a factor 2^{N-M}. To fulfill condition (6.5) again, the learning rate η should be scaled with 2^{N-M}. To achieve the same minimum error using different word-lengths for the weights, the learning rate η should thus be adapted to :

$$E^M_{min} = E^N_{min} \text{ , } when\ \eta_M = 2^{N-M} \cdot \eta_N \qquad (6.9)$$

When a small learning rate is applied, this means that when a DAC is used with n bits and the problem can be trained to a certain minimum error E_{min}, the same minimum E_{min} is

achieved with n-1 bits and a learning rate of 2η. he use of larger learning rates can cause that the algorithm follows a different path to the optimum E_0 and (6.9) may not be true.

Large learning rates η have two advantages :

- ☐ As described in paragraph 5.3, a larger learning rate η speeds up the convergence speed of the algorithm.
- ☐ A larger learning rate η decreases the minimum error E_{min}, given a certain word-length of the weights in the neural network.

To demonstrate this theory and to show the effects of quantization, simulations have been done studying the relation between the minimum error E_{min}, the learning rate η and the wordlength N of the weights. These results are combined in the next figures 6.2, 6.3 and 6.4. The algorithm stops when $0.999*E(t-1)<E(t)<E(t-1)$ for 100 consecutive steps.

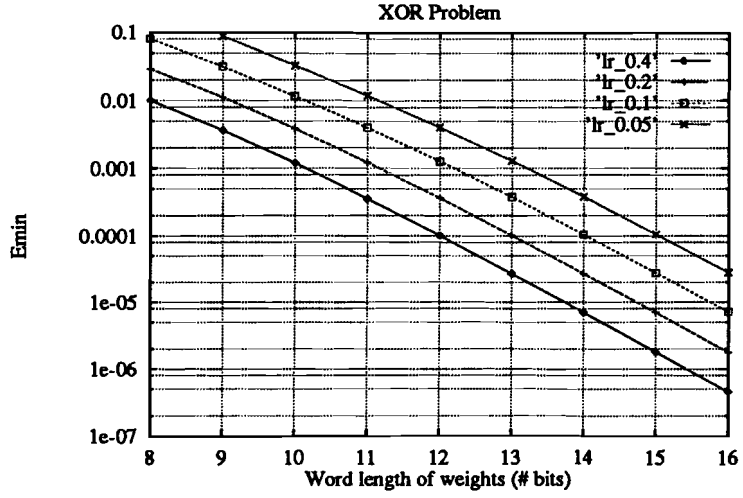


figure 6.2 : Influence of the wordlength N of the weights and the learning rate η on the minimum achievable error E_{min} during the training of the XOR problem with the WP algorithm. FDM δw = 1e-4, #trials 11.

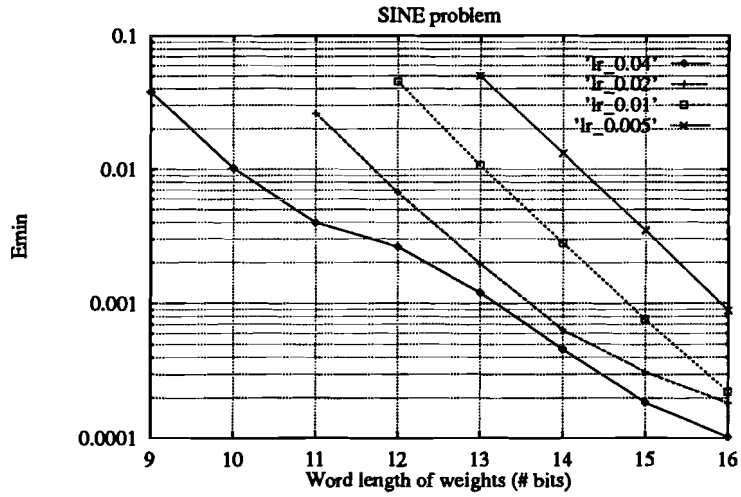


figure 6.3 : Influence of the wordlength N of the weights and the learning rate η on the minimum achievable error E_{min} during the training of the SINE problem with the WP algorithm. FDM δw = 1e-4, #trials 11.

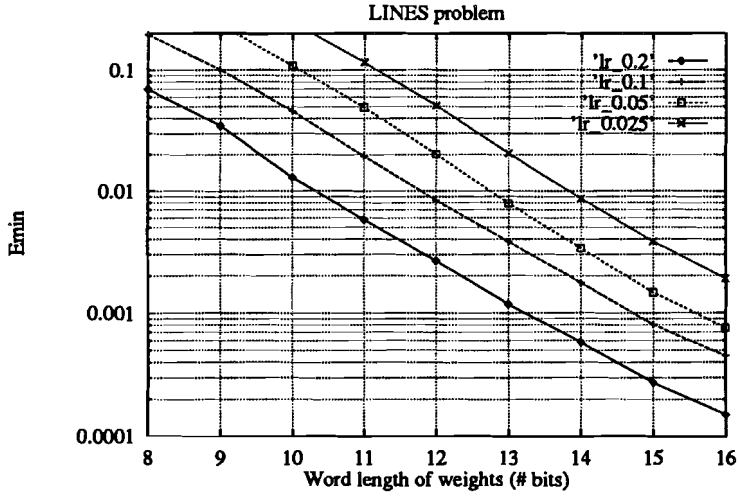


figure 6.4 : Influence of the wordlength N of the weights and the learning rate η on the minimum achievable error E_{min} during the training of the LINES problem with the WP algorithm. FDM δw = 1e-4, #trials 11.

The three simulations have been made to demonstrate the theory of the influence of the weight quantization. The simulations show a relation between E_{min}, η and N approximately according to the theory above. Relation (6.7) is more accurate when larger wordlengths are used. The use of large learning rates η result in side effects and (6.7) and (6.9) are not

correct anymore. These effects are hard to explain and occur when the learning rate approximates its maximum value η_{max}. These distortions are clearly shown in the simulation of the SINE problem with η=0.04.

It can be concluded that in order to train a network with quantized weights, the learning rate should be chosen as large as possible to obtain the lowest possible error and fastest convergence speed. Using a 12 bits DAC, the minimum errors which can be attained are : $E_{min}^{XOR} \approx$2e-4, $E_{min}^{LINES} \approx$2e-3 and $E_{min}^{SINE} \approx$2e-3. These results will be in general sufficient for these problems to train the network accurately. However, E_{min} depends on the maximum learning rate η_{max}. This limits a further decrease of the error.

In the next paragraph we will study what the real minimum error in a network is, given a certain wordlength. This is independent of the algorithm and therefore also independent of the learning rate.

6.2.2 Lower limit of error quantization

In the previous paragraph it is shown that due to the weight quantization, certain minimum errors E_{min} are achieved by the WP algorithm. These errors E_{min} are different for every benchmark and depend on the wordlength of the weights and the learning rate η. This limitation in error performance is caused by the WP algorithm itself. Because the weight updates $|\Delta w_{ji}|$ become smaller than $\frac{1}{2}\Delta$, the algorithm stops converging.

To study if improvements can be obtained by change of the WP algorithm, it is essential to determine the relation between the error E^N_{math} which is mathematically possible in a network with wordlength N of the weights. The goal is that E^N_{min} equals E^N_{math}. In this way the best possible value for error is achieved with the WP algorithm given a wordlength N.

Let us study mathematically the influence of the weight quantization at the minimum error E^N_{math}. The weight quantization can be modeled in many ways. For complex systems like neural networks the statistical model is the most appropriate. The idea is that a quantized signal can be represented by the original signal plus a quantization error (noise) ζ. The noise ζ requires the following conditions :

- ☐ ζ is a stationary random process
- ☐ ζ is white noise
- ☐ Δ is the quantization width of the weights and ζ is uniformly distributed in the range $[-\frac{1}{2}\Delta,\frac{1}{2}\Delta]$. The mean of the noise ζ is zero and has a variance $\sigma_\zeta^2 = \Delta^2/12$ which is uncorrelated with the variance of the weights.

The quantization model implies that on all the weights noise ζ is added and the weight values will be $w_{ji} + \zeta_{ji}$. When noise is added to the weights, the output y_k in the output layer of a MLP is affected by the distortion and results in a disturbed output value $y_k + \hat{y}_k$. According to (3.4) the network transfer Ω_k becomes :

$$y_k + \hat{y}_k = \Omega_k(W + \zeta, X^P) \tag{6.10}$$

The noise ζ_{ji} added at the weights w_{ji} results in a noise $\hat{y}_k$ at the outputs y_k. Provided that $\hat{y}_k$ is small, the output noise $\hat{y}_k$ has a mean of zero and a variance $\sigma_\hat{y}^2$ which depends on σ_ζ^2. The variance $\sigma_\hat{y}^2$ in the outputs depends on σ_ζ^2 and is calculated using statistical mathematics. These rather complex mathematical statistical derivations have been made by several researchers. Three approaches have been found which give an expression for the relation between $\sigma_\hat{y}^2$ and σ_ζ^2. A detailed theoretically study can be found in literature [10,20,31]. Although every approach is different they all result in a same sort of expression for the relationship. If tanh, sigmoids or linear neurons are used, there exists a linear relation between the two variances according to :

$$\sigma_\hat{y}^2 = C \cdot \sigma_\zeta^2 \tag{6.11}$$

The constant C depends on the size of the network, the range of inputs X^P, weight and neuron outputs and the used squashing functions. The constant C is derived with statistics and becomes more in harmony with the measured values when the network sizes are larger.

Now when the relation between $\sigma_\hat{y}^2$ and σ_ζ^2 is known, the relation between E^N_{math} and N can be derived. As shown in equation (3.6), the total square error E_{TSE} is described by :

$$E_{TSE} = \sum_{p=1}^{P} \sum_{k=1}^{K} (y_k + \hat{y}_k - d_k)^2 \tag{6.12}$$

E_{TSE} can be rewritten in three terms as :

$$E_{TSE} = \sum_{p=1}^{P}\sum_{k=1}^{K}\left((y_k-d_k)^2 + 2\cdot(y_k-d_k)\hat{y}_k + \hat{y}_k^2\right) \tag{6.13}$$

Because a noise component $\hat{y}_k$ is present in the expression of E_{TSE} (6.13), it is useful to calculate the mean of the E_{TSE} :

$$\overline{E_{TSE}} = \sum_{p=1}^{P}\sum_{k=1}^{K}(y_k-d_k)^2 + KP\cdot\left(2\cdot\overline{(y_k-d_k)\hat{y}_k} + \overline{\hat{y}_k^2}\right) \tag{6.14}$$

This expression can be simplified as :

$$\overline{E_{TSE}} = E_0 + KP\cdot\overline{\hat{y}_k^2} \tag{6.15}$$

The first term of (6.14), the minimum value E_0 is always present in (6.15), even when there is no weight quantization. E_0 is partly caused by the fact that the perturbation step δw is finite (see paragraph 4.2 and 5.2). E_0 could also partly be caused by the fact that the training patterns are inconsistent or the network is just too small as described in paragraph 3.3.3.

The second term in (6.14) is zero because the mean of $\hat{y}_k$ is zero and thus the mean of $(y_k - d_k)\hat{y}_k$ will be zero as well. This leaves the third term which depends on the word-length N of the weights. Due to the fact that the mean of $\hat{y}_k$ is zero, this term can be rewritten as :

$$\begin{aligned}\overline{\hat{y}_k^2} &= \sigma_{\hat{y}}^2 \\ &= C\cdot\sigma_{\zeta}^2 \\ &= \frac{C}{12}\Delta^2 \\ &= \frac{C}{12}\left(\frac{R}{2^N-1}\right)^2 \\ &\approx \frac{R^2C}{12}2^{-2N}\end{aligned} \tag{6.16}$$

The last approximation can be made when the number of bits N used is large enough. Combining (6.14) and (6.15) results in an expression between the mean of the error E_{TSE} and the wordlength N :

$$\overline{E_{TSE}} = E_0 + \frac{KPR^2C}{12}\cdot 2^{-2N} \tag{6.17}$$

The expression (6.17) for the mean of E_{TSE} is also called E^N_{math} and describes which error

minimal is possible in a network with quantized weights N.

To verify equation (6.17) and to get an idea of the values of E^N_{math}, simulations have been done. Using the three benchmark problems, the networks were trained with the WP algorithm until no further convergence was possible and the E_0 was approximately achieved. The training was simulated without weight quantization. After the weights were trained to their optimum values, noise was added to the weights. Its magnitude corresponded with the model of the quantization of the weights. For each wordlength the error was measured. 1000 Times a noise value was added to the weights and the mean of the measured error was calculated. In this way the mean value of E_{TSE} could be determined. The simulations have been made with the described noise model in stead of real quantization in order to get a more accurate average of the error. The noise model allows multiple random noise values to be averaged while the deterministic quantization function requires a lot of trials and simulation time to calculate the average error. The results of these simulations with the noise model are summarized in the next figure.

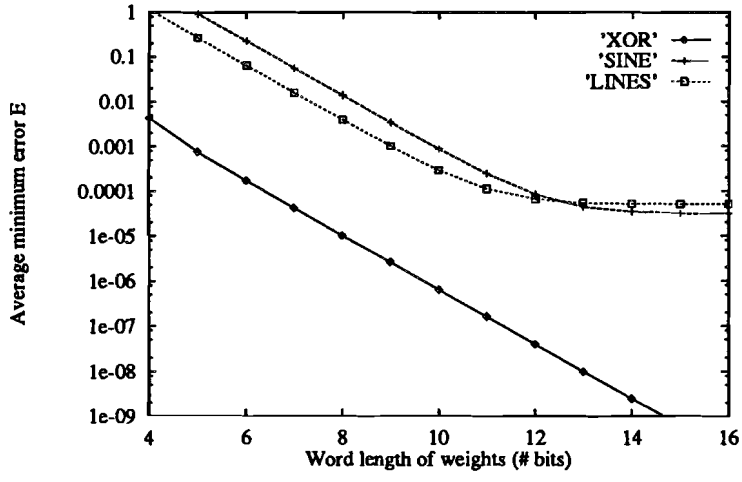


figure 6.5 : The mean of the lowest possible error attained in the three benchmarks when the weights are quantized with a certain wordlength N. FDM δw = 1e-4, # 1000 noise added, XOR #trials=18, SINE #trials =14, LINES #trials=15

The LINES problem attains its minimum error E_0, using 12 bit weights. The SINE problem uses 14 bits and the XOR problem uses 21 bits to attain its minimum. However, the minimum E_0 depends on the perturbation δw as was shown in paragraph 5.2.

It can be seen that all three problems can be described by (6.17). When figure 6.5 is compared with the three simulation figures 6.2, 6.3 and 6.4, it can be seen that the mathematical lowest possible error E^N_{math} is smaller than the minimum errors attained when the algorithm trains the network with quantized weights. Besides, the training with quantized weights result in an error dependency of E_{min} on η.

The WP algorithm should be adapted so, that lower error values can be attained. Probabilistic techniques have been developed to improve the performance of quantized training. This is described in the next paragraphs.

6.2.3 Attaining lower limit of error in training

As shown in paragraph 6.2.1 the WP algorithm stops converging when all the weight updates are smaller than ½Δ (6.5). In order to attain further convergence, the learning rate η can be increased. There is always a maximum learning rate η_{max}, which bounds this method. Other techniques give a probabilistic character to the weight update Δw_{ji}^Q. In the literature 2 possibilities have been found which are described below :

□ *Probabilistic rounding* [8]

Whenever the weight update Δw_{ji} falls below the quantization threshold ½Δ, the minimal step size Δ is taken with a probability p that is proportional to the size of the update Δwji. Because it was observed that many wasted epochs occurred where no weight was updated, a constant 1/fan-in was added to the probability to make it likely that at least one weight in each unit would be changed. The probabilistic rounding technique is defined by :

$$\begin{aligned} \Delta w_{ji}^Q &= Q(\Delta w_{ji}) , && \text{if } |\Delta w_{ji}^Q| \geq \tfrac{1}{2}\Delta \\ &= sign(\Delta w_{ji})\cdot\Delta , && \text{with prob. } p , \quad \text{if } |\Delta w_{ji}| < \tfrac{1}{2}\Delta \\ &= 0 , && \text{with prob. } 1-p , \quad \text{if } |\Delta w_{ji}| < \tfrac{1}{2}\Delta \end{aligned} \qquad (6.18)$$

The probability p in (6.18) is defined as :

$$p = \frac{1}{FANin} + \frac{|\Delta w_{ji}|}{\Delta} \qquad (6.19)$$

As can be seen from the last equations (6.18) and (6.19), the probabilistic rounding is rather difficult to implement. Every time a weight update is made it has to be checked whether it is smaller or larger than the quantization threshold ½Δ. When $|\Delta w_{ji}|$ is smaller than ½Δ, the update should be rounded to the minimum step size Δ with a probability p. The probability p depends on the fan-in and the size of idealized update Δw_{ji}. To implement probabilistic rounding, a lot of extra steps have to be taken to calculate the weight updates Δw_{ji}^{Q}. This is rather hard to implement in hardware.

The second method found in the literature to improve algorithm performance during training with quantized weights is called weight dithering :

□ *Weight dithering* [27]

Just as in the probabilistic rounding, weight dithering avoids that the weight updates Δw_{ji}^{Q} become zero by giving the weight update mechanism a probabilistic character. The theory below is based on the work done in [27] :

Weight dithering allows the weight resolution to be reduced by adding dither just before updating the weight values w_{ji}. A noise signal γ is added which is required to have a mean of zero and an uniform distribution between [-½Δ,½Δ]. This introduces a probabilistic element into the weight updating. This is illustrated in the following figure 6.6 :

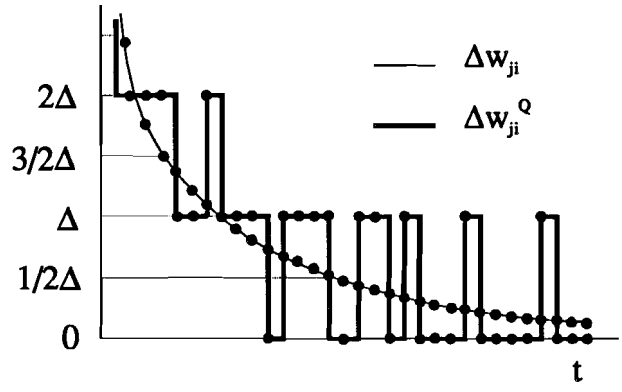


figure 6.6 : Idealized weight update value Δw_{ji} and quantized value Δw_{ji}^{Q} after adding noise just prior to quantization.

Although the idealized weight updates Δw_{ji} decrease to values below ½Δ, quantized weight updates Δw_{ji}^{Q} are still made. Because noise is added the updates can have values larger

than ½Δ. When a weight update is smaller than ½Δ, the dithering occasionally causes a jump in the right direction thereby preventing that the algorithm does not converge anymore because there are no weight updates Δw_{ji}^{Q} made anymore.

The advantage of dither over purely deterministic rounding is further demonstrated by the following analysis :

$$\Delta w_{ji}^{Q} = Q\left(\Delta w_{ji} + \gamma_{ji}\right) \tag{6.20}$$

where :
- Δw_{ji}^{Q} = quantized weight update
- Δw_{ji} = idealized weight update
- γ_{ji} = dither with mean zero and uniform distribution between [-½Δ,½Δ]
- Q() = roundoff truncation function (6.4)

Although the individual quantized weight update Δw_{ji}^{Q} still differs from the ideal update Δw_{ji}, it is important to determine what the mean of Δw_{ji}^{Q} is when dither γ_{ji} is added to the updates Δw_{ji}. If we define $\varepsilon_{ji} = \Delta w_{ji}^{Q} - \Delta w_{ji}$, the average of (6.18) is shown in the next equation :

$$\begin{aligned} \overline{\Delta w_{ji}^{Q}} &= \overline{Q\left(\Delta w_{ji}^{Q} + \varepsilon_{ji} + \gamma_{ji}\right)} \\ &= \Delta w_{ji}^{Q} + \overline{Q\left(\varepsilon_{ji} + \gamma_{ji}\right)} \end{aligned} \tag{6.21}$$

where :

$$\begin{aligned} \overline{Q\left(\varepsilon_{ji} + \gamma_{ji}\right)} &= \Delta \cdot P\left(\varepsilon_{ji} + \gamma_{ji} > \tfrac{1}{2}\Delta\right) - \Delta \cdot P\left(\varepsilon_{ji} + \gamma_{ji} < -\tfrac{1}{2}\Delta\right) \\ &= \Delta \cdot \frac{1}{2\Delta} \int_{\frac{1}{2}\Delta - \varepsilon}^{\frac{1}{2}\Delta} d\gamma - \Delta \cdot \frac{1}{2\Delta} \int_{-\frac{1}{2}\Delta}^{-\frac{1}{2}\Delta - \varepsilon} d\gamma \\ &= \varepsilon_{ji} \end{aligned} \tag{6.22}$$

Thus in combining (6.21) and (6.22) the mean of Δw_{ji}^{Q} can be reduced to :

$$\overline{\Delta w_{ji}^{Q}} = \Delta w_{ji}^{Q} + \varepsilon_{ji} = \Delta w_{ji} \tag{6.23}$$

This means that by adding dither before quantization, the mean value of the quantized weight update Δw_{ji}^{Q} becomes equal to the idealized weight update Δw_{ji}. Because the dither

has a range between ±½Δ, the quantized weight updates Δw_{ji}^{Q} can never be in the opposite direction of Δw_{ji}. It is thus expected that the influence of quantization on the final error during the training of a network with the WP algorithm is canceled and the limiting condition (6.5) does not apply anymore.

To demonstrate the theory above and to study the effect of dither on the minimum error E_{min}, simulations on the three problems have been done. The objective of the simulations is to train the network with uniform dither until the minimum possible error is attained. This error should be equal to E^{N}_{math}. The results of these simulations are shown in the next figures :

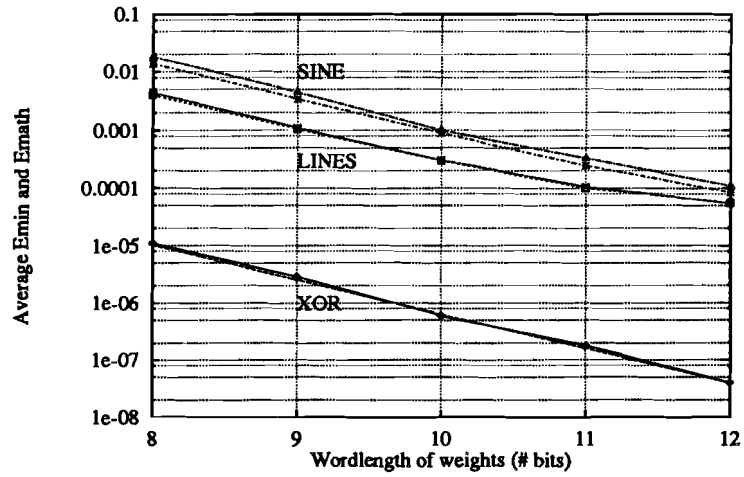


figure 6.7 : Training with WP and uniform dither (traced lines) compared with E_{math} (dashed lines), #trials=11, FDM δw=1e-4, η^{XOR}=0.1, η^{SINE}=0.01, η^{LINES}=0.1, training used 10000 epochs, average taken over epochs 9000-10000

From this figure it can be concluded that adding uniform noise just prior to quantization results in an error performance improvement of the WP algorithm. The lowest possible error E^{N}_{math} can be attained given a wordlength N.

An analog implementation of uniform noise is difficult to realize. It would be easier to use an ordinary normal distributed noise source. However, simulations showed that with the use of a normal distributed noise source, these minimum errors can not be attained anymore. The larger the variance of the noise, the higher the probability is that a weight

update is made in the wrong direction. It is not properly understood why the algorithm does not converge to E^N_{math} due to this opposite weights taken. Further research has to be done to get a more clear theory about this effect and to say something about the magnitude of the noise source which can be used.

6.3 Weight Decay

The values of the weights are stored on the capacitors C_{ji} for a short term storage of these values. The principle of this weight storage circuit uses a S&H circuit which is shown in the next figure :

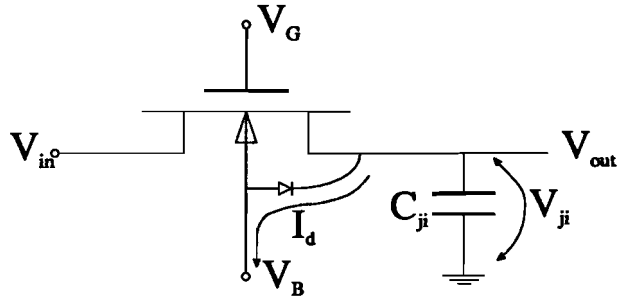


figure 6.8 : Elementary S&H circuit used for short term storage of the weights during the training of the network

The figure shows the most simple way to store a weight w_{ji} in a network. The synaptic weight value w_{ji} is stored as a voltage V_{ji} on the capacitor C_{ji}. The value V_{in} to be stored is sampled by means of the transistor operated as a switch, controlled by the gate voltage V_g. The storage time is limited by the leakage current I_d of the reverse biased diffusion-to-bulk junction which discharges the capacitor C_{ji}. The leakage current I_d does not depend on the weight value w_{ji} (V_{ji}), but is strongly temperature dependent as can be seen in the next equation :

$$I_d = \gamma T^3 e^{-\frac{qE_g}{kT}} \tag{6.24}$$

where I_d : Leakage current [A]
γ : constant [A/K^3] depending on the physics of the used transistor
T : Temperature [K]
q : Charge of an electron [C]

E_g : Bandgap Energy [V]
k : Boltzmann constant [J/K]

The capacitor C_{ji} is constantly discharged with a current source I_d. The voltage $V_{ji}(t)$ becomes time dependent and is calculated according to :

$$V_{ji}(t) = V_{ji}(0) - \frac{I_d \cdot t}{C_{ji}} \tag{6.25}$$

where $V_{ji}(0)$ is the value of the voltage just after the refresh of capacitor C_{ji}. (6.25) Shows that the values of the weights w_{ji} are linearly discharged. To use the weight values w_{ji} during the training and the actual use of the network, the weights need to be refreshed with their original values which are stored in the RAM. When weight refreshing is applied to diminish the effects of weight decay, the function of a weight value looks like a sawtooth curve :

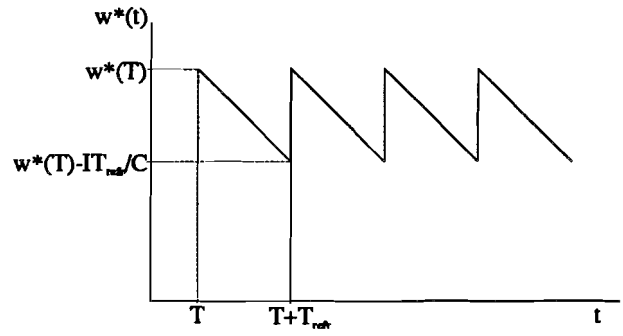


figure 6.9 : Function of the weights due to weight decay and weight refreshment

Each time the multiplexer from fig. 4.3 selects a weight w_{ji}, its original value is stored on the capacitor again. The weight value which decreases linearly and after a time T_{refr} returns to its original value. This causes the typical triangular shape of $w_{ji}(t)$.

Because all the training patterns need to be presented to the network twice for every weight update Δw_{ji}, it could take a long time before a weight is refreshed again. The refresh time T_{refr} increases when the network is larger and more training patterns are presented. T_{refr} can be approximated by :

$$T_{refr} = \frac{2MP}{A} \tag{6.26}$$

where M : Number of weights in a network.
P : Number of training patterns used for the training of a problem.
A : Number of training patterns which can be presented per second to the network [1/s].

The number of patterns/sec (A) which can be presented to the network depends on different factors like the conversion times of the DAC and ADC, the network transfer time and the calculation time needed in the CPU.

Combining (6.25) and (6.26) the decrease in the voltage V_{ji} during T_{refr} can be calculated :

$$\Delta V_{ji} = V_{ji}(0) - V_{ji}(T_{refr}) = \frac{I_d}{C_{ji}} \cdot T_{refr}$$

Now we know what the influence of the current leaks are on the weight values, a study can be made of the influence of the weight decay on the error performance of the WP algorithm.

6.3.1 Influence of weight decay

Weight decay is modeled by decrementing all the weights with a constant value d just before an update is made. The decay is subtracted every period T_{refr}/M. Just before a weight update Δw_{ji} is made, that specific weight is refreshed with its original value stored in the RAM. However, all the other weights have values which differ from their stored value in the RAM. They are affected by the leak current I_d. To model this, the weights in a neural network can be described with weights w_m. The weights are updated sequentially and $1 \leq m \leq M$ where M is the total number of weights in a network. The gradient descent algorithm (3.9) transforms into :

$$w_m^{new} = w_m^{old} - \eta \cdot \frac{\partial E(W^*)}{\partial w_m} \tag{6.28}$$

where W^* is the weight set in the network transformed by the weight decay. When a

weight w_m is updated, the other weights w_k are modeled as :

$$\begin{aligned} w_k^* &= w_k - d\cdot(m-k) \quad , when\ 1\le k<m \\ &= w_k - d\cdot(M+m-k) \ , when\ m<k\le M \end{aligned} \tag{6.29}$$

When the derivative $\partial E(W^*)/\partial w_m$ is only a function of w_m, the leak will be of no influence because a gradient decent algorithm makes a step Δw_m in the error landscape which is in the direction of the lowest error. However, it is more likely that the error derivative $\partial E(W^*)/\partial w_m$ is a function of all the weights. This means that the derivative in the direction of one weight is influenced by the decay of the other weights.

It is not easy to say something about the dependency of the error derivative on the weights. If the weight set W forms a multidimensional space $\mathbf{R}^M$, the error function E depends on how the weight axes are chosen in this landscape. In large networks it can be assumed that the chance that the error derivative $\partial E(W^*)/\partial w_m$ only depends on the weight w_m is very small and it can be expected that there is a dependency on the other weights.

To demonstrate this, we closely look at a two dimensional weight space $\mathbf{R}^2$. The example uses thus an error function $E(w_1,w_2)$ which is a function of the weights w_1 and w_2. Close to the optimum weight values the error E is curved like a parabola and can be modeled as reported in paragraph 5.2 :

$$E(w_1,w_2) = \Psi_1\cdot w_1^2 + \Psi_2\cdot w_2^2 \tag{6.30}$$

The derivative is only dependent on one weight : $\partial E/\partial w_1 = 2\Psi_1 w_1$ and $\partial E/\partial w_2 = 2\Psi_2 w_2$. When the same error function is described with weight axis rotated $\pi/4$ in the weight space, the error function transforms into :

$$E(w_1,w_2) = \Psi_1\cdot w_1^2 + \Psi_2\cdot w_2^2 + (\Psi_2-\Psi_1)\cdot w_1 w_2 \tag{6.31}$$

When a weight is updated, the other weight is influenced by the weight decay and a value d is subtracted from its original value (6.29). The error derivatives are shown in the next equation :

$$\frac{\partial E}{\partial w_1} = 2\Psi_1 w_1 + (\Psi_2 - \Psi_1)\cdot(w_2 - d)$$
$$\frac{\partial E}{\partial w_2} = 2\Psi_2 w_2 + (\Psi_2 - \Psi_1)\cdot(w_1 - d) \qquad (6.32)$$

An optimum E_{min} is found when the derivatives for all weights are equal to zero. In the optimum, the following condition is obtained :

$$E_{min} \; when : \frac{\partial E(W^*)}{\partial w_m} = 0 \quad \forall_m \qquad (6.33)$$

Combining condition (6.33) with (6.32) results in the following optimum weights :

$$w_1 = \frac{\Psi_1 + \Psi_2}{3\Psi_1 - \Psi_2}\cdot w_2$$
$$w_2 = \frac{3\Psi_1^2 + \Psi_2^2 - 4\Psi_1\Psi_2}{\Psi_1^2 + \Psi_2^2 - 6\Psi_1\Psi_2}\cdot d \qquad (6.34)$$

Combining the optimum weights (6.34) and the equation for the error function (6.31) results in a minimum error which depends on d^2. This can be generalized to error functions where the rotation of the axes is φ in stead of $\pi/4$. The minimum error depends on d^2 according to :

$$E_{min} = d^2 \cdot M(\Psi_1, \Psi_2, \varphi) \qquad (6.35)$$

where M() is a function determined by condition (6.33).

The simple two dimensional problem of weight decay already shows that it is a complex problem to solve. However, the example above indicates that there is a relation between E_{min} and d^2 when the curves of the error landscape are parabolas near the optimum solution. This can be generalized to networks with multiple weights.

To study the influence of the weight decay on the error performance, simulations have been done using parameters which resemble the ones of the WP chip which is being designed [3,25].

These reports show that the following parameters are used :

I_d $\approx$ 1e-14 A

C $\approx$ 3e-13 F

A $\approx$ 1e4 s^{-1}

V_{ji} $\in$ [1,2] V

The actual range of the weights is 1 Volt which should correspond with our simulated dimensionless weights in the range of [-2.5,2.5]. This means that d needs to be scaled with a factor 5. After scaling the real weight range to a dimensionless weight range, the decay step d used in the simulations becomes according to (6.27) :

$$d = 5 \cdot \frac{2P}{A} \cdot \left(\frac{I_d}{C_{ji}} \right) \tag{6.36}$$

If the leakage current I_d and the capacitors C_{ji} are equal for all the weights, all the decay steps are proportional to I/C. The ratio I/C is approximately 0.033. But as shown in (6.24) the leakage current is strongly temperature dependent. The values of the capacitors are not exactly known. Therefore the ratio of I/C is taken as a measure of the decay step d. In the figure below the results of the simulations without quantization are summarized :

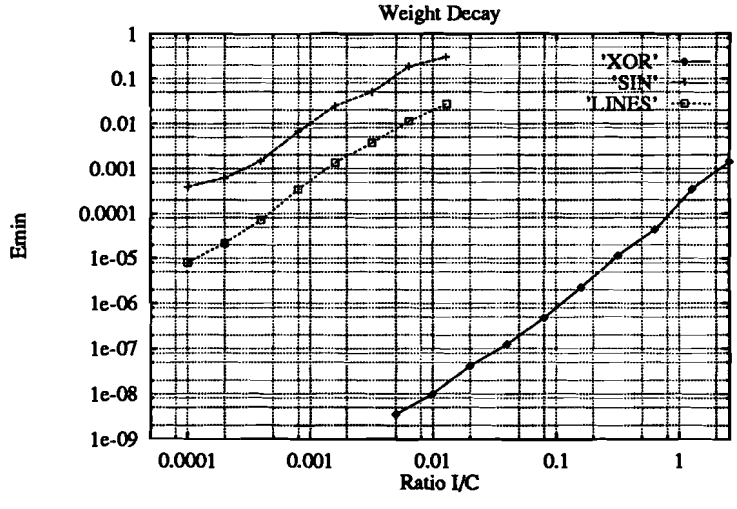


figure 6.10 : Minimum error E_{min} as a function of the ratio I/C in the three benchmark problems. #trials=11, FDM δw=11, η^{XOR}=0.1, η^{SINE}=0.01, η^{LINES}=0.1

The algorithm stops when $0.999*E(t-1)<E(t)<E(t-1)$ for 100 consecutive steps. In this figure it can be seen that the error E_{min} depends on the ratio I/C approximately according to the theory as a relation $E_{min} \approx Cd^2$. The influence of the weight decay is different for each network. Comparing this figure with figure 6.7 it can be seen that the influence of the weight decay dominates the influence of the quantization in the SINE and LINES problem. As expected these influences are more dominant in larger networks and when a larger training set is used.

The simulations indicate that for the used leak currents $I_d \approx 10fA$ and the capacitor values $C \approx 300fF$, the influence on the error performance becomes unacceptable. Some techniques can be used to diminish the influence of weight decay. These are described in the next paragraph.

6.3.2 Methods for reducing weight decay

In order to improve the error performance of the WP algorithm, the weights need to be refreshed with shorter a interval T_{refr}. This can be done by separating the weight update circuit from the weight refresh circuit. This is shown in the next figure :

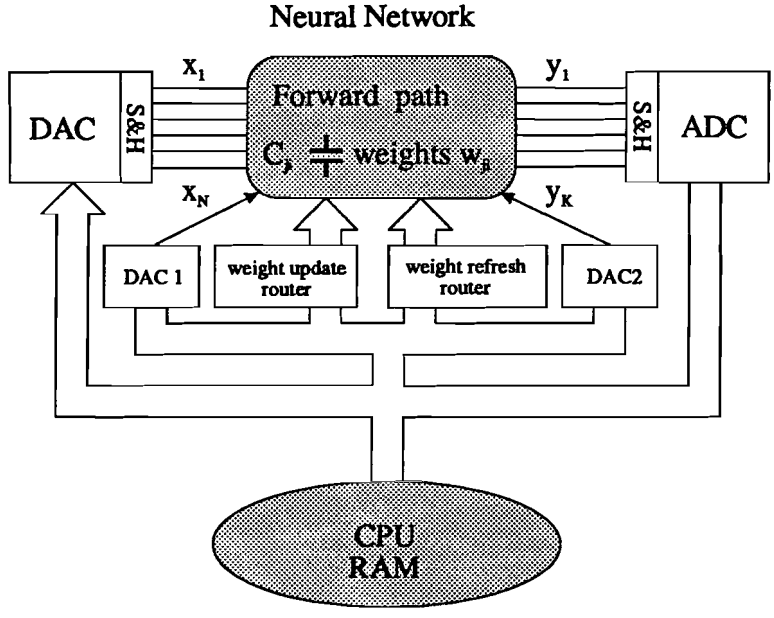


figure 6.11 : Implementation of the WP algorithm on a chip using a separate weight update and weight refresh circuit.

In this way the weight refreshes are made independent of the weight updates and $T_{refr} < T_{upd}$. The refresh time (6.26) becomes independent of the number of presented patterns P which allows more complex problems to be trained. However, there is always a minimum refresh time. The weights are sequentially refreshed which means that it takes a certain time to refresh all the weights before returning to the first weight again. The separation of the circuits requires extra hardware as can be seen from the comparison between figures 6.11 and 4.3. Other less complex solutions should be studied as well.

As shown in the last paragraph the effect of the weight decay diminishes when the weight decay step d (6.35) decreases. This can be accomplished by :

- ☐ Decreasing the leakage current I_d. Two possibilities could be used in order to obtain a smaller leakage current I_d :
  - Decrease the temperature of the WP chip (6.24).
  - Alter the S&H circuit to a more complex one with two capacitors where the weight information is stored as the difference of the voltages of the both capacitors C_{ji} [3].
- ☐ Increasing the storage capacitor C_{ji}.
- ☐ Presenting the patterns faster to the neural network.
- ☐ Decreasing the number of presented patterns.

The first three adaptions already have been optimized and with the current used CMOS chip technology, no improvement is to be expected from the first two solutions. The third solution is bounded by the speed and cost of the interface between the computer and the neural network chip [25]. A decrease in the number of presented patterns can be accomplished without any extra hardware and will be described in the next paragraph.

6.3.3 Subsets of the training patterns

Instead of presenting the whole training set P where the patterns are presented sequentially, a subset D is presented. The subset contains patterns which are in every epoch randomly chosen from the pattern set P with a probability 1/P. This means that from all the training patterns D patterns are selected each chosen with a probability 1/P. After selection of a pattern, this pattern can still be selected again and the same pattern could

occur more than once in the subset D. This selection mechanism is easy to implement.

Because the error function E_{TSE} depends on the number of training patterns, the mean square error E_{MSE} (3.7) is applied when the network is trained with this variant of the WP algorithm. E_{MSE} allows it to make a comparison between error functions independent of the used number of training patterns. The error function can be written as :

$$E_{MSE} = \frac{1}{P}\sum_{p=1}^{P} \mathcal{E}^p \tag{6.37}$$

where :

$$\mathcal{E}^p = \frac{1}{K}\sum_{k=1}^{K}\left(y_k^p - d_k^p\right)^2 \tag{6.38}$$

When no leakage of the weights is applied, the gradient descent weight update (3.9) rule can be rewritten as :

$$\begin{aligned}\Delta w_{ji} &= -\eta \cdot \frac{\partial E_{MSE}}{\partial w_{ji}} \\ &= -\eta \cdot \frac{\partial\left(\frac{1}{P}\sum_{p=1}^{P}\mathcal{E}^p\right)}{\partial w_{ji}} = -\frac{\eta}{P}\cdot\sum_{p=1}^{P}\frac{\partial \mathcal{E}^p}{\partial w_{ji}}\end{aligned} \tag{6.39}$$

When a subset D of the entire training set P is presented to the network and the individual patterns are selected with a probability 1/P, the weight update rule (3.9) becomes :

$$\Delta w_{ji}^{D} = -\eta \cdot \frac{1}{D}\sum_{d=1}^{D}\frac{\partial \mathcal{E}^d}{\partial w_{ji}} \tag{6.40}$$

Obviously Δw_{ji}^{D} differs from Δw_{ji}. However, the mean of Δw_{ji}^{D} equals Δw_{ji} as is proved in the next equation :

$$\begin{aligned}\overline{\Delta w_{ji}^{D}} &= -\eta \cdot \frac{1}{D}\sum_{d=1}^{D}\overline{\frac{\partial \mathcal{E}^d}{\partial w_{ji}}} \\ &= -\eta \cdot \frac{1}{D}\sum_{d=1}^{D}\left(\frac{1}{P}\sum_{p=1}^{P}\frac{\partial \mathcal{E}^p}{\partial w_{ji}}\right) = -\frac{\eta}{P}\cdot\sum_{p=1}^{P}\frac{\partial \mathcal{E}^p}{\partial w_{ji}}\end{aligned} \tag{6.41}$$

(6.41) Equals (6.39) which means that the average of the subset pattern weight update

Δw_{ji}^{D} equals the weight update Δw_{ji} when the entire training set is presented.

Because the E_{MSE} error is used, the error measurement of a subset training E_{MSE}^{D} is a measure for the real error E_{MSE}. There is noise on the error E_{MSE}^{D} introduced due to the fact that the individual weight updates Δw_{ji}^{D} differ from the perfect weight update Δw_{ji}. However, the mean of the weight updates Δw_{ji}^{D} equals the perfect weight update Δw_{ji} which implies that the mean of the error E_{MSE}^{D} is equal to E_{MSE}. In this case it is not necessary to calculate the error E_{MSE}, but the calculation of the mean of E_{MSE}^{D} suffices. A practical and easy implemented measure of the mean of E_{MSE}^{D} can be calculated according to :

$$\overline{E_{MSE}^{D}(t)} = (1-\rho)\cdot\overline{E_{MSE}^{D}(t-1)} + \rho\cdot E_{MSE}^{D}(t) \tag{6.42}$$

The mean of E_{MSE}^{D} can be calculated using its old value and its current value. A exponential average filter is used which filters out the unwanted noise in the error. ρ Determines how much the noise should be filtered and has a value between 0 and 1.

To demonstrate this, the network is trained using the three benchmark problems. During the simulations the network is trained using the WP algorithm with the subset pattern presentation. The mean of the error E_{MSE}^{D} is calculated. These simulations are shown in the next figures 6.12, 6.13 and 6.14 :

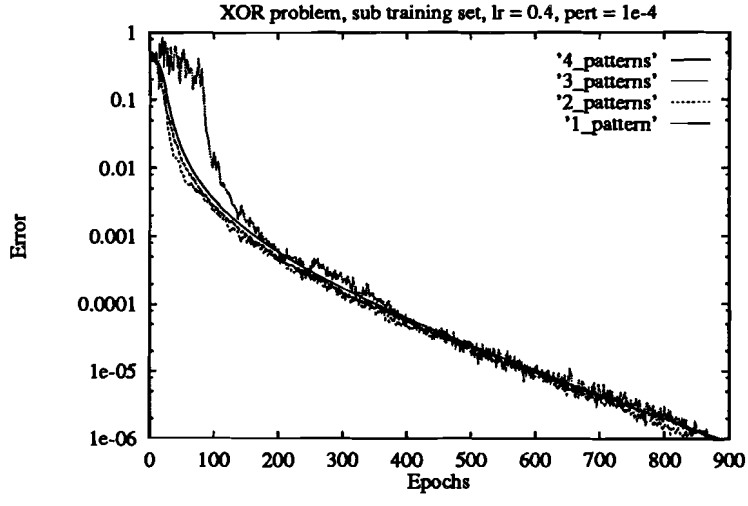


figure 6.12 : The mean of E_{MSE}^{D} in the XOR benchmark. ρ=0.01, η=0.4, FDM δw=1e-4

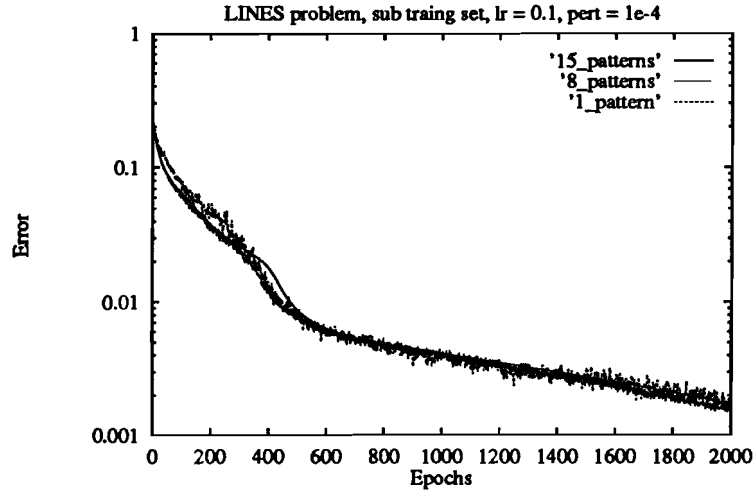


figure 6.13 : The mean of E_{MSE}^{D} in the LINES benchmark. ρ=0.01, η=0.1, FDM δw=1e-4

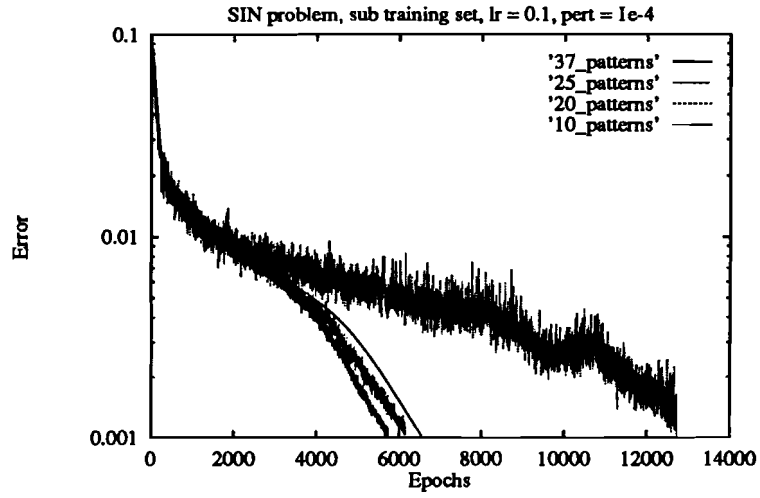


figure 6.14 : The mean of E_{MSE}^{D} in the SINE benchmark. ρ=0.01, η=0.01, FDM δw=1e-4, 10 patterns converge in 13000 epochs

These simulations indicate that the mean of E_{MSE}^{D} can be used as a measure of E_{MSE}. In these problems it can be seen that when the learning rate is chosen small enough, subsets of the total training set P can be selected without effecting the error performance. This results in two advantages :

- Because the subset D is smaller than P, the refresh time T_{refr} (6.26) is shorter and the weights are refreshed P/D times more often.
- When small learning rates are applied, the error behavior stays the same. However, the epochs contain less patterns using a subset. Therefore the speed of the algorithm is P/D times increased.

As shown in (6.39) and (6.40) the subset weight updates Δw_{ji}^{D} differ from the wanted updates Δw_{ji}. When this difference is too large, the training of the network becomes impossible because too many wrong weight updates are made and the algorithm converges in a wrong direction. When a smaller learning rate is applied, the magnitude of a weight step becomes smaller. This means that taking a step which is in the wrong direction isn't as dramatic as it would be when a large learning rate would be applied. When the number of presented patterns is decreased the learning rate should be decreased too. The sensitivity of a problem to taking wrong weight updates depends on the problem. This can be seen from the next simulations which are shown in the figures 6.15, 6.16 and 6.17.

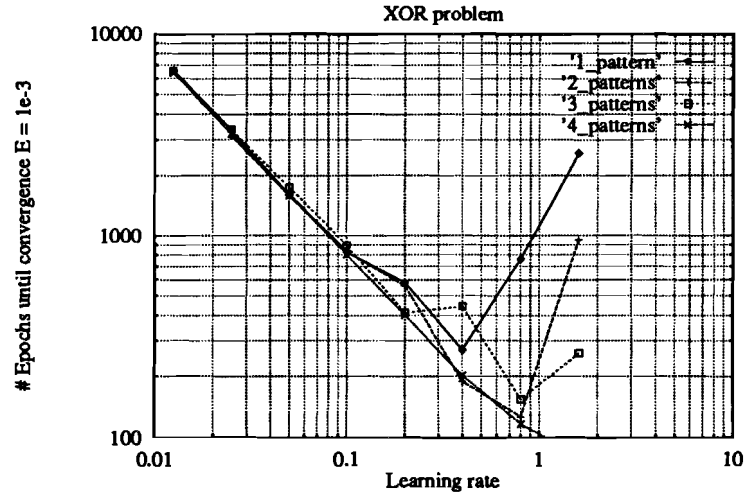


figure 6.15 : Training the network with a subset of the total number of patterns, ε_{MSE}=1e-3,FDM δw=1e-4, #trials=11

From this figure it can be seen that at a learning rate smaller than $\eta_{MSE} \approx 0.1$ no distortions of the convergences occur and that under this limits the weight updates Δw_{ji}^{D} are taken small enough to insure proper convergence behavior.

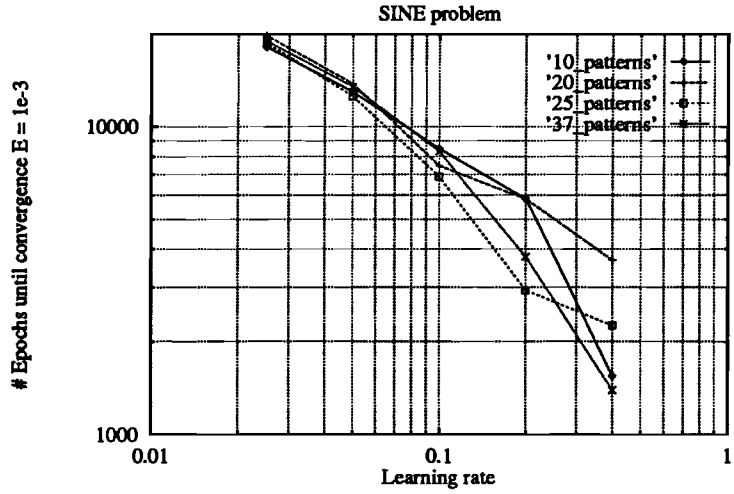


figure 6.16 : Training the network with a subset of the total number of patterns, ε_{MSE}=1e-3, FDM δw=1e-4, #trials=11

The sine problem is very sensitive to decreasing the number of presented patterns. It appears to be that the algorithm gets rather easily stuck in a local minimum of approximately E_{MSE}≈1e-2. To decrease the error, very small learning rates should be taken, which result in very long simulation times. In figure 6.16, simulations have been done up to 10 subset patterns. It can be seen that applying learning rates smaller than η_{MSE}≈0.05 result in approximately the same convergence behavior. The lines are not exactly straight as would be expected from paragraph 5.3. The inverse relation between the number of epochs and η is a bit disturbed due to the fact that the training was stopped at 20000 epochs.

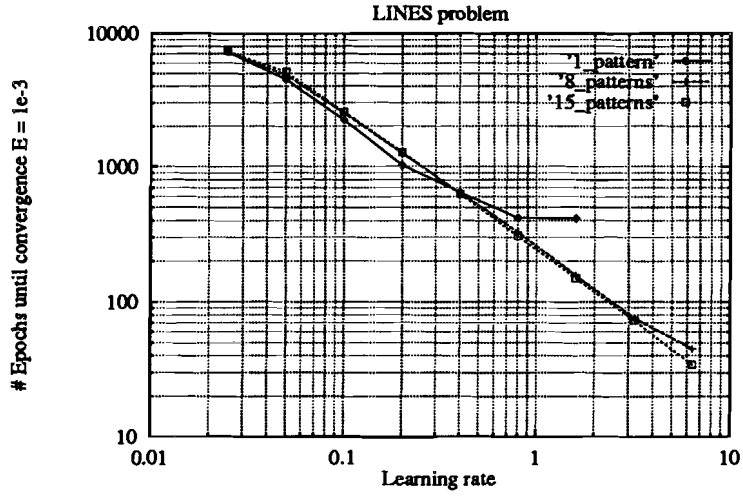


figure 6.17 : Training the network with a subset of the total number of patterns, ε_{MSE}=1e-3, FDM δw=1e-4, #trials=11

The lines problems shows correct training for one pattern when the network is trained with learning rates smaller than $\eta_{MSE} \approx 0.8$.

These simulations indicate that the subsets of the total training set can be applied and the same error behavior occurs when the learning rate is chosen smaller than a maximum value. This maximum value is problem dependent. Applying a small learning rate is not of practical use. It is interesting to know what is the total number of presented patterns needed to train a certain problem, when the number of patterns in the subset D is varied and the learning rate η is optimized for that specific D. These results are not available yet and it is recommended to carry out these simulations.

6.4 Non-idealities in multipliers

Up till now all the multipliers used in the simulations were ideal multipliers and can be described as :

$$y = x \cdot w \tag{6.43}$$

The multiplier has two input signals x and w and an output y. The weight value w is multiplied with the signal x coming from a neuron in the previous layer. A four quadrant

multiplier is chosen using two transistors [3]. The multiplier is shown in the following figure :

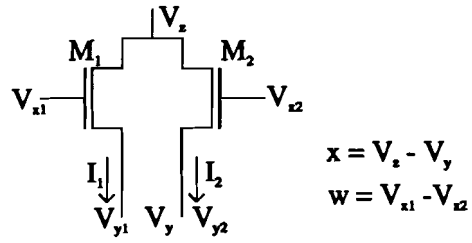


figure 6.18 : Four quadrant multiplier using two transistors.

When both M_1 and M_2 are working in the linear range, the output I_1-I_2 depends on the inputs x and w according to :

$$\begin{aligned} I_1 - I_2 &= \beta \cdot \{(V_{x1} - V_{x2})(V_z - V_y)\} \\ &= \beta \cdot x \cdot w \end{aligned} \tag{6.44}$$

Offsets occur due to several hardware limitations of the multiplier. The four most important offsets are described below [3] :

- ☐ The voltage V_y may not match the intended value which results in an offset (Off_1) at the input x. The offset has a value of about ±1% of the input range.
- ☐ Due to differences in the threshold voltage of the transistors M_1 and M_2 an offset (Off_2) is added at the weight value w. This offset has a maximum of about ±2% of the weight range.
- ☐ Besides the different threshold values, transistors M_1 and M_2 have different values β_1 and β_2 and the common β (6.44) does not apply anymore. This difference results in a rather complex offset behavior which is modeled by adding a term $Off_3 \bullet (w + C_w)$ at the original multiplication (6.43). Off_3 has a maximum of about ±5% of the weight range and the constant C_w is approximately $\approx 1.25 \bullet R_w$ when R_w is the weight range mentioned in paragraph 6.2.
- ☐ The last error occurs when the V_{y1} does not equal V_{y2}. Using a current conveyor circuit, it is tried to keep these voltages at the same value V_y. Offsets in this circuits result in an offset (Off_4) of about ±10% in the output y of the multiplier. Because this circuit is situated just before the input of a neuron and all (32) its input multipliers are connected to this circuit, the offsets have the same value for

these multipliers. Multipliers connected to other neurons have a different offset depending on the current conveyor belonging to that neuron.

In this thesis fixed ranges of the weights and inputs are chosen, the weight range R_w=5 ([-2.5,2.5]) and the x range equals the output of the neurons, therefore R_x=2 ([-1,1]). This results in a range of the multipliers output R_y=5 ([-2.5,2.5]). Given these ranges the following model is made for the multiplier when the four offsets are introduced :

$$y = (x+Off_1)(w+Off_2) + Off_3(w+6.25) + Off_4 \qquad (6.45)$$

where : $Off_1 = \pm 1\%$ of R_x
$Off_2 = \pm 2\%$ of R_w
$Off_3 = \pm 5\%$ of R_w
$Off_4 = \pm 10\%$ of R_y

It is not known how large these offset exactly are. They depend on many factors and, except for Off_4, differ for each multiplier used in the neural network. The model implements these random offset values. Before the training starts, each multiplier is given its own random chosen offsets. After this initialization the algorithm trains the network with these imperfect multipliers. This has to be done in multiple trials. In every trial the offsets are chosen with different random values. In this way the average influence of the offsets can be studied.

In the simulation every offset is introduced separately, in order to study its effect on the convergence speed. Also all the offsets at the same time are added to study whether the multiplier behavior of the convergence speed when a more accurate model of the hardware multiplier. The results of these simulations are shown in the next figures :

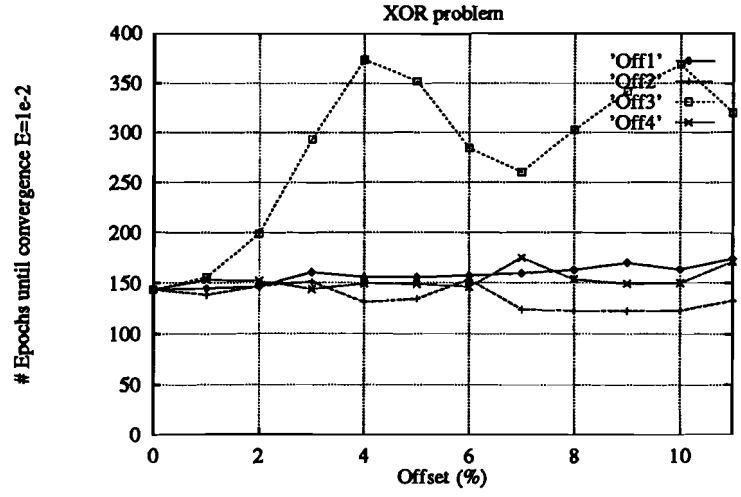


figure 6.19 : Training the network with the XOR problem using non-ideal multipliers. FDM δw=1e-4, #trials=31

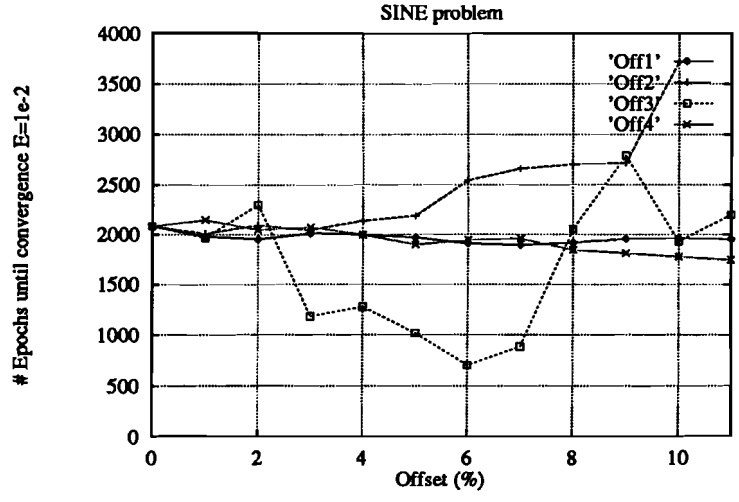


figure 6.20 : Training the network with the SINE problem using non-ideal multipliers. FDM δw=1e-4, #trials=31

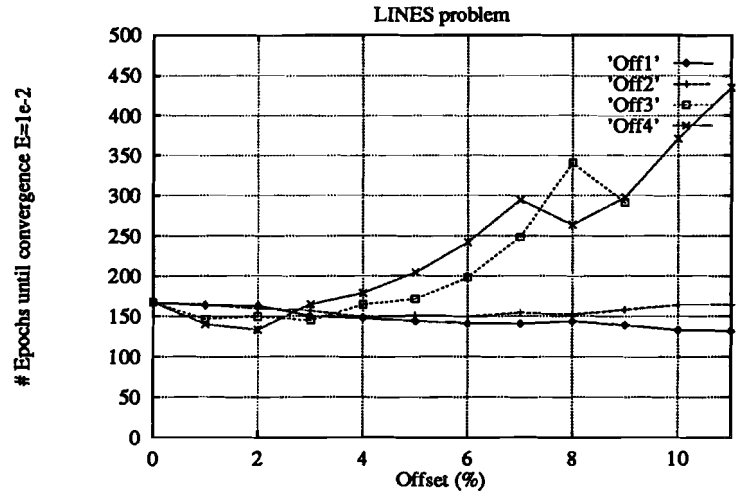


figure 6.21 : Training the network with the LINES problem using non-ideal multipliers. FDM δw=1e-4, #trials=31

It can be seen that the offsets influence the convergence time. Off_1 and Off_2 are not of any real influence. However, Off_3 and Off_4 influence the convergence behavior. In general, a larger offset increases the convergence time. This can go up to convergence times which are twice as slow.

Although the offsets influence the convergence time, it can be seen that the algorithm still functions properly when offsets are introduced. The number of converged trials, however, depends on the magnitude of the offsets. Larger offsets cause less converged trials. We define an offset as acceptable when at least 50% of the trials is converged compared with the situation when no offset is introduced. The result of this definition is shown in the next table :

table 6.1 : The maximum offset allowed when 50% of the trials should converge compared with the situation where no offset is added. All the simulations have been made until with a maximum offset of 11%. > 11% means that offsets are allowed which are at least 11%. The maximum is not known.

	Off_1	Off_2	Off_3	Off_4
XOR	> 11%	> 11%	> 11%	> 11%
SINE	> 11%	< 6%	< 3%	> 11%
LINES	> 11%	> 11%	< 8%	> 11%

From this table it can be seen that the sensitivity of the offset is different for each problem. When these figures are compared with the predicted SPICE offsets it can be concluded that the used multipliers are suited to train these three problems. Except for the SINE problem where Off_3 should be kept smaller than 3%. The number of converged trials is approximately 30% when a 5% offset Off_3 is added in the multipliers for the SINE problem.

As opposes to the backpropagation algorithm which does not tolerate offsets [19], the WP algorithm accepts imperfection. Even when the offsets are simulated with values larger than they are predicted with the SPICE analyses the WP algorithm keeps working. It can therefore be concluded that when the WP algorithm is implemented on a chip, it does not need ideal multipliers (6.43) in order to function properly. The proposal of the multiplier is acceptable. However, better convergence behavior is obtained when the offsets are kept small.

6.5 Non-idealities in neurons

During all the simulations tanh (2.3) squashing functions have been used in the neurons. The sigmoid function is the most common function used in the design of a neural network. The implementation of a tanh or sigmoid function in hardware always results in slightly different functions than originally wanted.

In the design of the WP chip [3] a differential transconductance amplifier is chosen (M_1,M_2,M_3) with quadratic loads (M_4,M_5). The schematic of this neuron is illustrated in the next figure :

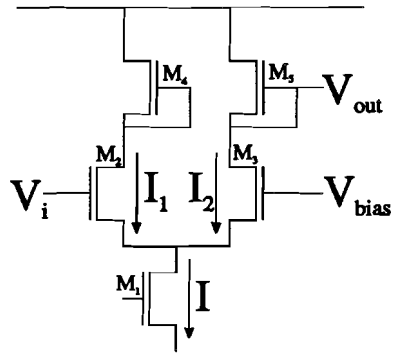


figure 6.22 : Implementation of neuron using a differential transconductance amplifier with quadratic loads

The current I_2 is approximated by :

$$I_2 \approx \frac{C_1}{1+e^{-\beta(V_i-V_{bias})}} \tag{6.46}$$

where V_i : input voltage
V_{bias} : reference voltage
V_{out} : output voltage
C_1 : constant
β : temperature

The equation (6.46) is the approximation of I_2. However, the real current I_2 is not exactly a sigmoid function, but has a different formula which is extensively described in [3]. Because this formula consist of multiple parts depending on the range where the transistors. This is rather hard to implement in the simulations and decided is for (6.46) to model the function. It is expected that the model will never equal the SPICE simulations of I_2.

The neuron output V_{out} depends on the current I_2 (6.46) according to :

$$V_{out} = C_2 + C_3 \cdot \sqrt{I_2} \tag{6.47}$$

where C_2 and C_3 are constants. Quadratic loads are used to insure a large fanout of the neuron. Linear loads cause a smaller fanout. The use of quadratic loads has a side effect

that no longer a tanh function is implemented as can been seen from (6.46) and (6.47). The used neuron is translated into a model for the simulations which is described in the next equation :

$$y = a + b \cdot \sqrt{c \cdot (\tanh(x+d)+1)} \tag{6.48}$$

This model is used to fit the SPICE data of the neuron transfer and is illustrated in the next figure :

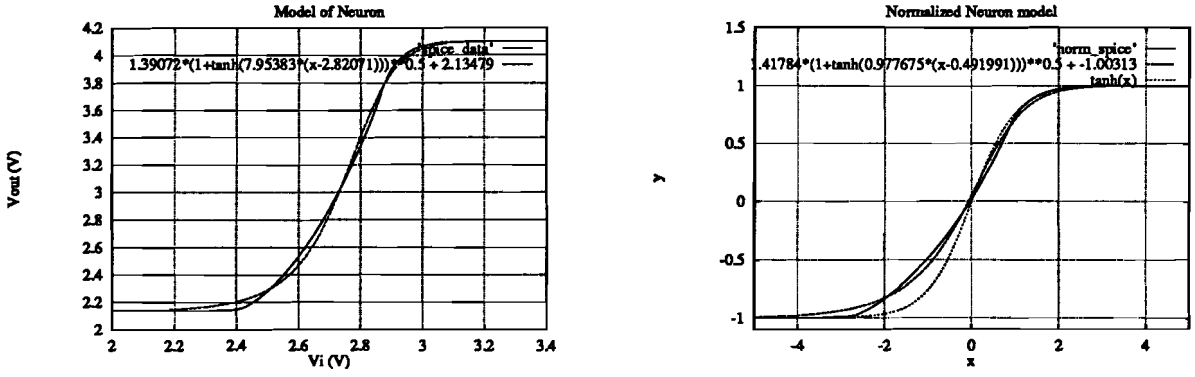


figure 6.23 : On the left, the actual SPICE data of the neuron transfer function is compared with the model which is fitted with a least square algorithm. In the right figure the same data is transformed to dimensionless values used in the simulations. This is compared with the model (6.48) and a tanh (2.3) function.

The figure shows that the used model can not be exactly fitted on the SPICE data. The optimum value for the model (6.48) was found with the values : a=-1.00313, b=1.41784, c=0.977675 and d=-0.491991. This model is used as squashing functions F in the neurons. Simulations have been made with this model and the results are compared with neurons using a tanh function (2.3). The results are shown in the next table :

	η	ε	Model # Epochs until convergence	Tanh # Epochs until convergence
XOR	0.1	1e-3	343	143
SINE	0.01	1e-2	1401	846
LINES	0.1	1e-3	567	308

In this table it can be seen that the convergence is approximately two times slower when the non-ideal neurons are used. The number of converged trials is about the same as when tanh neurons were used. Therefore it can be concluded that the WP algorithm is sensitive to non-ideal neurons but still functions properly. The slower convergence can probably be compensated in the choice of a new learning rate η which is optimal for the new squashing function F.

The convergence becomes slower due to a different squashing function F. The used model resulted in an algorithm which was twice as slow. When the real neurons are implemented according to the SPICE data the convergence can become slower or faster compared to the model. However, it is expected that these neurons will cause no change in the number of converged trials and therefore the proposed neurons [3] can be used in the neural network chip.

7 Accelerating the WP algorithm

7.1 Introduction

The WP algorithm is a very slow algorithm because all the training patterns are presented twice to calculate a weight update Δw_{ji}. As stated in paragraph 4.1, the WP algorithm requires approximately 2M times more training patterns than the backpropagation algorithm. When the speed of the pattern presentation is determing the speed of the algorithm the WP algorithm becomes very slow compared to the backpropagtion algorithm.

Both algorithms are classified as gradient descent algorithms. Finding an optimum weight set using the steepest descent weight update rule (3.9) is a slow method. Therefore the WP algorithm is not only slow because of the large number of presented patterns, but as well due to the fact that it is a gradient descent rule.

There are two reasons why steepest descent algorithms have a slow rate of convergence which are related to the following aspects :

- Magnitude of the components of the gradient.
- The direction of the gradient vector.

The magnitude of a partial derivative of the error with respect to a weight, $|\partial E/\partial w_{ji}|$, may be such, that modifying a weight by a constant proportion of that derivative will yield in a minor reduction in the error E. This occurs in two situations :

- The error surface is fairly flat along a weight dimension w_{ji}. The derivative $\partial E/\partial w_{ji}$ will be small in magnitude which results in a small weight update Δw_{ji} and many steps are required to achieve a significant reduction in the error E.
- The error surface is highly curved along a weight dimension w_{ji} and the derivative $\partial E/\partial w_{ji}$ will be large. The weight update Δw_{ji} will be large and the new weight-value w_{ji}^{new} may overshoot the minimum of the error surface along that weight dimension. An error can be found which is even larger than the error before an update was made.

A second reason for slow convergence of a steepest descent algorithm is that the direction

of the negative gradient descent vector may not point directly towards the minimum of the error surface. This is illustrated in the next figure :

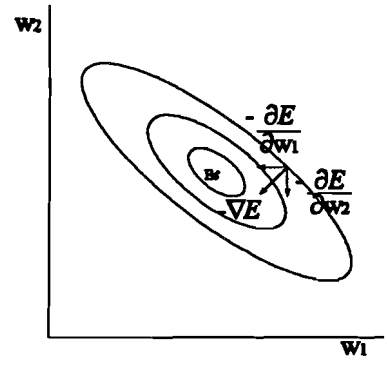


figure 7.1 : Error surface over a two-dimensional weight space.

As can be seen in this figure, the negative gradient $-\nabla E$ does not point directly towards the minimum. Therefore the weight updates are not made in the most optimum direction and the path to reach to minimum error in the weight space will be longer. More weight updates have to be made to reach eventually the minimum which result in a slow convergence speed.

To accelerate any gradient descent algorithm four heuristics have been presented in [14] :

- The learning rate η should be made individual. Thus the calculation of Δw_{ji} uses an individual learning rate η_{ji}. In par. 3.3.3 it is shown that the learning rate η should be smaller than $1/\lambda^{max}$. When individual learning rates η_{ji} are applied, each weight dimension w_{ji} could have its own maximum learning rate depending on the eigenvalue λ_{ji}.
- Every learning rate $\eta_{ji}(n)$ should be allowed to vary over time. It is common for error surfaces E to possess different properties along different regions. In order to get an optimum result, the learning rate $\eta_{ji}(n)$ should vary depending on the properties of the error surface.
- When the derivative of the error function $\partial E/\partial w_{ji}$ possesses the same sign for several consecutive steps, the learning rate η_{ji} should be increased. When the sign of the derivative behaves in this manner, it is frequently the case that the error surface at the current point along that weight dimension possesses a small curvature, and therefore continues to decrease in the same direction for some significant

distance. By increasing the learning rate η_{ji} for this weight dimension w_{ji}, the number of time steps required for the value of the weight w_{ji} to traverse this distance is reduced.

- When the sign of the derivative of the error function $\partial E/\partial w_{ji}$ alternates for several consecutive steps, the learning rate η_{ji} should be decreased. When the sign of the derivative behaves in this manner, it is frequently the case that the error surface at that point along the weight dimension w_{ji} possesses a high curvature, and therefore, the slope of this area of the error surface may quickly change sign. In order to prevent the weight w_{ji} oscillating, the learning rate should be decreased.

Note that by providing different learning rates, the current point in the weight space is not modified in the direction of the negative gradient $-\nabla E$. Thus, such a system is not performing a steepest descent search. Instead, parameters are updated based on both the partial derivatives $\partial E/\partial w_{ji}$ and an estimation of the curvatures of the error surface at a certain point in the weight space along each weight dimension.

The heuristics can be implemented in the WP algorithm using different techniques [14]. Two solutions which are practical to implement without a lot of extra hardware are the Delta-Bar-Delta rule and Momentum. These methods to accelerate the WP algorithm have been studied and are described in the next paragraphs.

7.2 Delta-Bar-Delta rule

The Delta-Bar-Delta (DBD) rule adapts the four heuristics to the following individual learning rate update rule :

$$\eta_{ji}(n) = \eta_{ji}(n-1) + \Delta\eta_{ji}(n) \tag{7.1}$$

where $\Delta\eta_{ji}$ is defined as :

$$\begin{aligned} \Delta\eta_{ji}(n) &= \kappa && , \textit{if } \overline{\delta}_{ji}(n-1)\cdot\delta_{ji}(n) > 0 \quad , \kappa \geq 0 \\ &= -\phi\cdot\eta_{ji}(n-1) && , \textit{if } \overline{\delta}_{ji}(n-1)\cdot\delta_{ji}(n) < 0 \quad , 0 \leq \phi \leq 1 \\ &= 0 && \textit{otherwise} \end{aligned} \tag{7.2}$$

where :

$$\delta_{ji}(n) = \frac{\partial E(n)}{\partial w_{ji}(n)} \tag{7.3}$$

and :

$$\overline{\delta_{ji}}(n) = (1-\theta)\cdot\delta_{ji}(n) + \theta\cdot\overline{\delta_{ji}}(n-1) \quad , 0 \leq \theta < 1 \tag{7.4}$$

(7.4) Calculates the exponential mean of the error derivative along every weight direction. θ Determines how the mean depends on previous values of the error derivative. The DBD rule (7.2) adapts the individual learning rates η_{ji} so, that if the current error derivative of a weight and the exponential average of the weight's previous error derivatives have the same sign, then the learning rate for that weight is incremented by a constant κ. If the current error derivative of a weight and the exponential average of the weight's previous error derivatives possess opposite signs, then the learning rate for that weight is decremented by a portion, ϕ, of its current value. This is illustrated in the next figure :

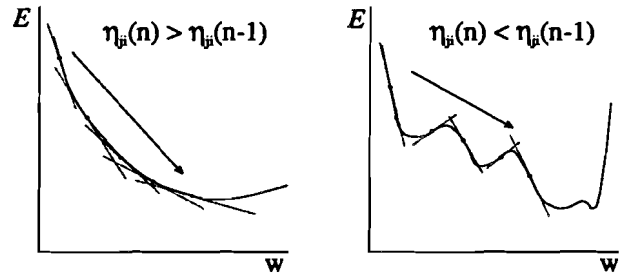


figure 7.2 : Adaption of the learning rate η_{ji} depending on the error behavior

The DBD rule increments learning rates linearly, but decrements them exponentially. Incrementing linearly prevents the learning rates from becoming too large too fast. Decrementing exponentially ensures that the rates are always positive and allows them to be decreased rapidly.

As can be seen from the equations (7.2) and (7.4) the DBD rule uses three parameters, κ, ϕ and θ. Because for every problem which the network is trained with has different error landscapes, the optimum values of these parameters differ for each problem. It is rather hard to find the optimum values because there are three parameters have to be optimized which means that there are a lot of possibilities. First it is tried to find the optimum values for ϕ and θ. The objective is to obtain the fastest convergence. The simulations have been done with the problems choosing only one random initial weight set.

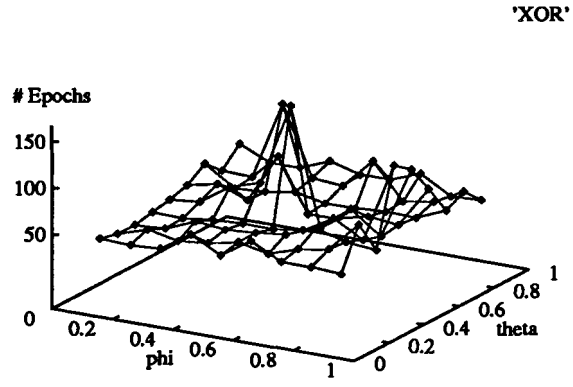


figure 7.3 : Training the XOR problem with the DBD rule. FDM δw=1e-4, η(0)=0.1, κ=0.1, ϕ_{opt}=0.1, θ_{opt}=0.3, #trials=11

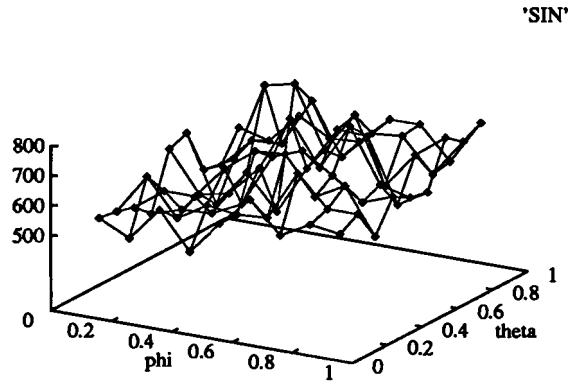


figure 7.4 : Training the SINE problem with the DBD rule. FDM δw=1e-4, η(0)=0.01, κ=2e-4, ϕ_{opt}=0.2, θ_{opt}=0.8, #trials=11

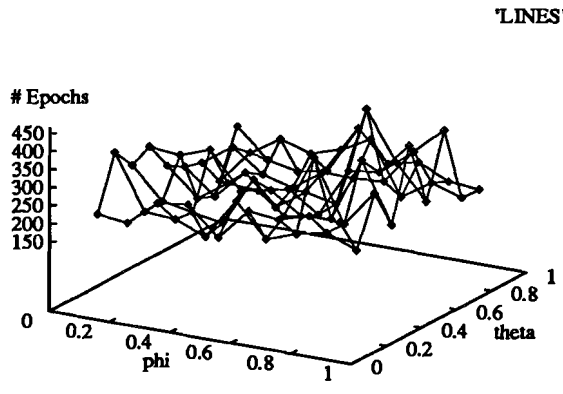


figure 7.5 : Training the LINES problem with the DBD rule. FDM δw*=1e-4,* $\eta(0)$*=0.1,* κ*=1e-2,* ϕ_{opt}*=0.1,* θ_{opt}*=0.6, #trials=11*

These figures show that the convergence speed depends on ϕ and θ. The fastest convergence is achieved with optimum values for ϕ and θ which vary for each problem. In the rest of this paragraph we use these values for ϕ and θ. It has to be said that it can not be concluded that these values are the optimum. The DBD rule implements three parameters, where in the previous simulations ϕ and θ varied but κ was a constant. These parameters depend on each other. Therefore it can only be concluded that the values found for ϕ and θ belong to that specific value of κ. Due to the very long simulation times it would require to find the optimum values for all three parameters, this approach has been chosen.

Now we have found the optimum values for ϕ and θ, the influence of the third parameter κ can be simulated. The result is shown in the next figure :

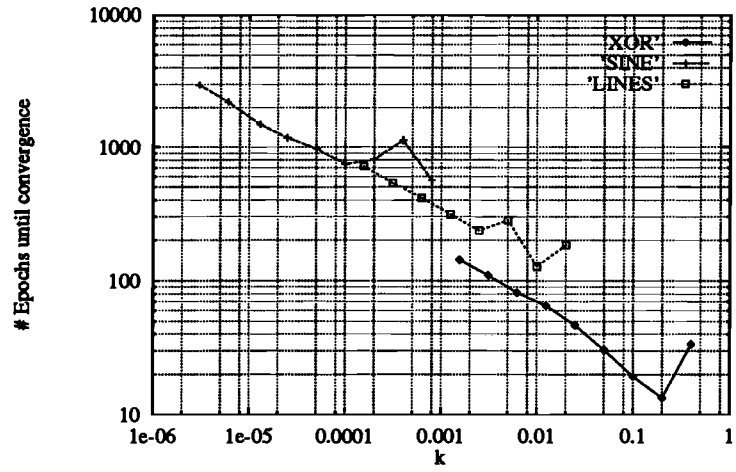


figure 7.6 : Training of the three problems with the DBD rule,
FDM δw=1e-4, #trials=31
XOR : η(0)=0.1, ϕ=0.1, θ=0.3
SINE : η(0)=0.01, ϕ=0.2, θ=0.6
LINES : η(0)=0.1, ϕ=0.1. θ=0.8

In this figure it can be seen that an increase of κ results in a faster convergence. If κ is chosen too large, the DBD rule can not compensate its fast growing learning rate anymore and the algorithm becomes instable. This can be seen by the fact that first slower convergences occur although κ increases. At a certain value κ_{max} there is no convergence anymore. A maximum value is defined when at least 50% of the trials converge compared with the situation where no DBD rule is applied. These maximum values are approximately $\kappa^{XOR} \approx 0.1$, $\kappa^{SINE} \approx 4e\text{-}4$, $\kappa^{LINES} \approx 5e\text{-}3$. These maximum values only apply in this case. It can not be concluded that they can be used in general for these benchmarks. Further simulations have to be done.

To demonstrate the effect of the DBD rule on the convergence speed an example is simulated with a single set of parameters ϕ, θ and κ. In these simulations 31 trials have been made. It is important to simulate with a learning rate which approximates the maximum given in paragraph 5.3. In this way gain using the DBD rule could be shown. Otherwise, when simulations are made with small learning rates the effect of the DBD rule on the convergence speed can always be achieved when the learning rate is increased when no DBD rule is applied. The result is printed in the next table :

				Average # Epochs until convergence ε optimum values ϕ and θ chosen (see figures 7.2, 7.3 and 7.4)	
problem	η(0)	ε	κ	WP with DBD FDM δw=1e-4 #trials 31	WP without DBD FDM δw=1e-4 #trials=31
XOR	0.4	1.0e-3	0.1	21	65
SINE	0.04	1.0e-2	2.0e-4	341	506
LINES	0.2	1.0e-3	1.0e-3	231	308

These results show an accelerating in the convergence for XOR : 3.1, SINE : 1.5, LINES : 1.3. The number of converged trials using the DBD rule is approximately 2/3 compared with the situation without DBD.

From this short study of the DBD rule it can be concluded that there are certain benefits and drawbacks :

Advantage :

- ☐ The DBD rule accelerates the WP algorithm in the three benchmarks. Optimum values of κ, ϕ and θ will further quicken the algorithm.
- ☐ The learning η does not have to be chosen with an optimum value to gain maximum speed. The algorithm can even start with a learning rate of zero.

Disadvantage :

- ☐ In stead of finding only one optimum learning rate η, three parameters are to be optimized for each problem.
- ☐ The implementation of the DBD rule requires extra calculations (time) and RAM to store $\overline{\delta}_{ji}$ and η_{ji}.

Currently, the achieved results indicate that the DBD should not be implemented. However, depending on further research the benefits and the drawbacks have to be compared and then it can be decided whether it is useful to implement the DBD rule.

7.3 Momentum

Momentum implements the heuristics by the addition of a new term to the weight update rule of (3.9). At a time n, each weight $w_{ji}(n)$ is updated according to :

$$\begin{aligned}\Delta w_{ji}(n) &= -\eta \cdot \frac{\partial E(n)}{\partial w_{ji}(n)} + \alpha \Delta w_{ji}(n-1) \\ &= -\eta \cdot \sum_{i=0}^{n} \alpha^i \cdot \frac{\partial E(n-i)}{\partial w_{ji}(n-i)}\end{aligned} \tag{7.5}$$

where α determines the contribution of the past weight update $\Delta w_{ji}(n-1)$ in the present weight update $\Delta w_{ji}(n)$. The momentum α should be chosen $0 \leq \alpha < 1$. Note that when α equals zero, the ordinary gradient descent weight update rule of (3.9) is obtained. α Can not be used with a value larger than one because the weight updates $\Delta w_{ji}(n)$ will diverge as shown in the second part of (7.5).

Momentum is considered an implementation of the heuristics mentioned in paragraph 7.1 for the following reasons :

- ☐ When consecutive derivatives of a weight posses the same sign, the weights sum grows large in magnitude and the weight is adjusted by a larger amount.
- ☐ When consecutive derivatives of a weight posses opposite signs, the sum becomes smaller in magnitude and the weight is adjusted by a small amount.

A drawback of momentum is that an old weight update may have a sign which is opposite to the current derivative and could result in a weight update which sign is opposite to a weight update in the situation where no momentum (α=0) is applied. Thus momentum can cause the weight to be adjusted up the slope of the error surface in stead of down the slope. Depending on the problem and learning rate η, this will impose further restriction of the momentum α which will be shown in the simulations.

This drawback could be turned into an advantage when momentum is not used for accelerating the algorithm, but for the escape out of local minima. This other effect of momentum is not studied in this thesis.

To study the effect of momentum, various simulations have been made with the three benchmark problems. The results are shown in the next figures :

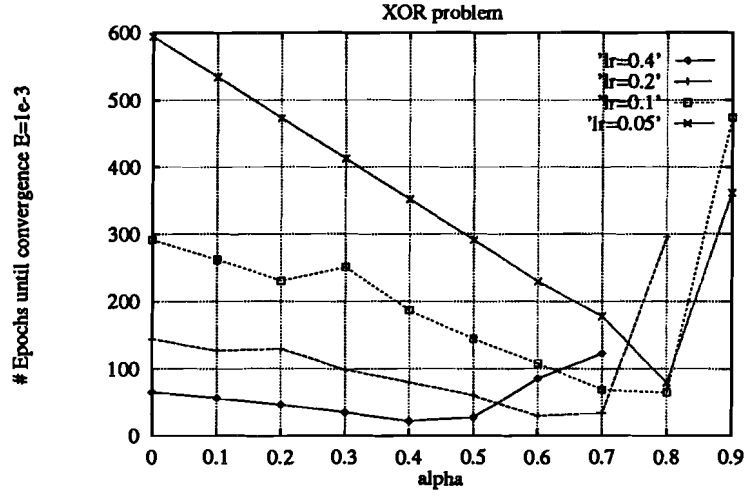


figure 7.7 : training the XOR problem with momentum, FDM δw=1e-4, ε=1e-3, #trials=11

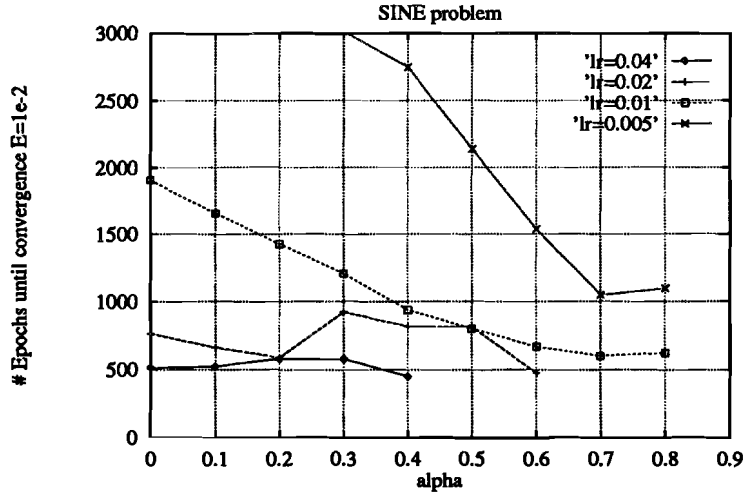


figure 7.8 : training the SINE problem with momentum, FDM δw=1e-4, ε=1e-2, #trials=11

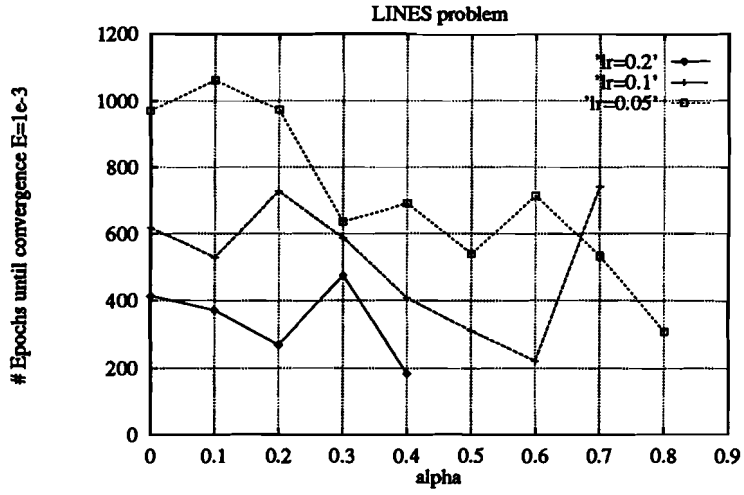


figure 7.7 : training the LINES problem with momentum, FDM δw=1e-4, ε=1e-3, #trials=11

These simulations have been done using different learning rates. From the figures it can be seen that increasing the momentum α results in faster convergence. This applies until a certain value $\alpha=\alpha_{max}$ which depends on the used learning rate η and problem. Using a momentum α which is larger than the maximum results in fewer converged trials and a slower convergence time.

As is shown in these figures, when a learning rate η is chosen so that it approximates the maximum learning rate η_{max} (see par. 5.3), still an improvement in convergence speed is achieved when momentum is applied. Therefore it can be concluded that momentum accelerates the WP algorithm when a learning rate η is used which is smaller than the maximum learning rate η_{max}. This implies that it is more sensible to train the network with the maximum learning rate possible, for that specific problem, then the use of momentum which also requires extra RAM to store the values of the old weight updates $\Delta w_{ji}(n-i)$.

It is interesting to study the effect of momentum when subsets of the total training set are presented to the network which is described in paragraph 6.3.3. The update $\Delta w_{ji}D$ is different from the ideal update Δw_{ji} when the total training set is presenting. However, the mean of Δw_{ji}^{D} equals Δw_{ji}. Momentum is expected to improve the training performance, but this still has to be verified.

8 Conclusions and recommendations

Various effects have been studied in an ideal MLP and a MLP with non-idealities :

- The Central Difference Method is preferred over the Forward Difference Method because it results in a better error performance which means that larger perturbations can be used. This results in weaker hardware ristrictions of the pertubation.
- Every problem has its own maximum learning rate. The number of epochs required to converge is approximately inverse proportional to the learning rate.
- Weight quantization influences the error performance. Better error performance can be obtained using a larger learning rate.
- Adding uniform dither to the weights just prior to the quantization results in the lowest possible error given a certain wordlength.
- Weight decay influences the error performance and should be diminished. Networks using many weights and patterns are more influenced by the weight decay. The expected ratio of I/C≈0.033 is unacceptable.
- A simple technique which reduces the influence of weight decay is the presentation of a subset of patterns.
- The convergence behavior is acceptable when the proposed non-ideal multipliers and neurons are applied in the network.
- The proposed algorithm accelerators accelerate the algorithm, but their implementation requires extra RAM and complex parameter estimation.

During the study of the WP algorithm, several subjects were found which are interesting to study. Due to time limitations, these topics have not been studied properly. The following topics should be studied to gain a better understanding of the implementation of the WP algorithm :

- The influence of normal distributed dither on the error performance.
- The relation between the number of used patterns and the maximum learning rate when subsets of patterns are presented.
- The optimum values of the parameters of the DBD for faster convergence.
- The use of momentum when subsets of the total patterns are presented.

References

[1] Andes, D. et al
MRIII : A robust algorithm for training analog neural networks
Proc. IJCNN, vol. 1 (1990), p. 533-536

[2] Baba, N.
A new approach for finding the global minimum of error function of neural networks
Neural Networks, vol. 2 (1989), p. 367-373

[3] Bruin, P.P.F.M.
A weight perturbation chip set
Master Thesis, TU Eindhoven, Aug. 1993

[4] Caviglia, D.D. et al.
Effects of weight discretization on the back propagation learning method : algorithm design and hardware realization
IJCNN vol. 2 (1990), p. 631-637

[5] Frye, C. et al
Back-propagation learning and nonidealities in analog neural network hardware
IEEE Trans. Neural Networks, vol 2. (1991), no. 1, p. 110-117

[6] Funahashi, K.
On the approximate realization of continuous mappings by neural networks
Neural Networks, vol. 2 (1989), p. 183-192

[7] Hertz, J. et al
Introduction to the theory of neural networks
Redwood City : Addison-Weslley, 1991

[8] Hoehfeld, M. and Fahlman, S.E.
Learning with limited numerical precision using the cascade-correlation algorithm
IEEE Trans. Neural Networks, vol. 3 (1992), no. 4, p. 602-611

[9] Hollis, P.W. et al
The effects of precision constraints in a backpropagation learning network
Neural Computation, vol. 2 (1990), no. 3

[10] Holt, J. and Hwang, J.
Finite precision error analysis of neural network hardware implementations
IEEE Trans. Computers, vol. 42 (1993), no. 3, p. 281-290

[11] Jabri, M.A. and Flower, B.
Weight perturbation : an optimal architecture and learning technique for analog VLSI feedforward and recurrent multi-layer networks
IEEE Trans. Neural Networks, vol. 1 (1992), no. 3, p. 154-157

[12] Jabri, M.A. and Leong, P.H.W.
A VLSI neural network for morphology classification
IJCNN, vol 2. (1992), p. 678-683

[13] Jabri, M.A. and Xie, Y.
On the training of limited precision multi-layer perceptrons
The University of Sydney, SEDAL Technical Report no. 1991-8-3

[14] Jacobs. R.A.
Increased rates of convergence through learning rate adaption
Neural Networks, vol. 1 (1988), p. 295-307

[15] Karaali, O. and Gluch, D.P.
Weight quantization effects in artificial neural networks
Southcon/90 Conference Record, p. 372-377

[16] Lippmann, R.P.
An introduction to computing with neural nets
IEEE Trans. ASSP (1987), p. 4-22

[17] Müller, B. and Reinhardt, J.
Neural Networks
Berlin : Springer, 1990

[18] Mundie, D.B. and Massengill, L.W.
Weight decay and resolution effects in feedforward artificial neural networks
IEEE Trans. Neural Networks, vol. 2 (1991) no. 1, p. 168-170

[19] Petin, Y.A.
Implementation of multi-layer perceptron including backpropagation training algorithm
Master Thesis, TU Eindhoven, Aug. 1993

[20] Piché, S.
Selection of weight accuracies for neural networks
Dissertation, Department of Electrical Engineering, Stanford University, 1992

[21] Ping Wah Wong
Quantization noise, fixed-point multiplicative roundoff noise, and dithering
IEEE Trans. ASSP, vol. 38 (1990), no. 2. p. 286-300

[22] Rumelhart, D.E. et al
Learning internal representations by error propagations
in *Parallel distributed processing : explorations in the microstructure of cognition*
Cambridge, M.I.T Press, 1986, vol. 1

[23] Sun. J. et al.
A fast algorithm for finding global minima of error functions in layered neural networks
IJCNN vol. 1 (1990), p. 715-720

[24] Tarassenko, L. and Tombs, J.
On-chip learning with analogue VLSI neural networks
Micro Neuro (1993) p. 163-174

[25] Teeffelen, J.J.N. van
Interfacing neural network chips with a personal computer
Master Thesis, TU Eindhoven, Aug. 1993

[26] Teulings, P.M.W.
Analog electronic implementation of multi layer neural networks including back-propagation
Master Thesis, TU Eindhoven, Aug. 1991

[27] Vincent, J.M. and Myers, D.J.
Weight dithering and wordlength selection for digital backpropagation networks
British Telecom Technical Journal, vol. 10 (1992), no. 3, p. 124-133

[28] Widrow, B. and Lehr, B.A.
30 Years of Adaptive Neural Networks : Perceptron, Madaline, and Backpropagation
Proc. IEEE, vol. 78 (1990), no. 9, p. 1415-1442

[29] Withagen, H.C.A.M.
Reducing the effect of quantization by weight scaling
internal report, TU Eindhoven, 1993

[30] Woodland, P.C.
Weight limiting, weight quantisation & generalisation in multi-layer perceptons
First IEE International Conf. on Artificial Networks (1989), p. 297-300

[31] Xie, Y. and Jabri, M.
Analysis of the effects of quantization in multilayer neural networks using a statistical model
IEEE Trans. Neural Networks, vol. 3 (1992), no. 2, p. 334-338