Rijksuniversiteit Groningen Faculteit der Wiskunde en Natuurwetenschappen

Vakgroep
Informatica

Parallelism versus Accuracy in Error Back–Propagation Learning

A study towards the possibilities of introducing parallelism that is not restricted to the layer–wise composition of neural networks but is based on a simple architecture and towards the usage of less accuracy for floating–point multiplications in error back–propagation learning.

G. F. Meijering

August 1996

supervisors: Prof. Dr. Ir. L. Spaanenburg
Drs. M. Diepenhorst

RuG

Contents

Abstract 1

Samenvatting 3

Chapter 1: Introduction 5
1.1 Computational Order 5
1.2 Parallelism versus Accuracy 6
1.3 Gradient Descent Method 6

Chapter 2: Traditional Gradient Descent Methods 9
2.1 Newton Raphson 9
2.1.1 The Algorithm 9
2.2 Changing the Computational Order of the Newton Raphson Method 10
2.3 Mathematical Foundation 10
2.4 A First Experiment: The Regula Falsi Method 13
2.4.1 Results 13
2.4.2 Conclusions 15
2.5 A Second Experiment: The Secant Method 16
2.5.1 Results 16
2.5.2 Conclusions 20
2.6 Using Less Accurate Computations 20
2.6.1 Results 20
2.6.2 Conclusions 22

Chapter 3: Introducing Parallelism in Error Back−Propagation 23
3.1 Error Back−Propagation 23
3.1.1 The Algorithm 23
3.2 Introducing Parallelism 27
3.2.1 Motivation for Using Parallelism in the Backward Pass 27
3.2.2 Adapting the backward pass 29
3.2.3 Scheduling 30
3.2.4 Adapting the "original" algorithm 30
3.3 Experiments 33
3.4 Exclusive−or 34
3.4.1 Results 35
3.4.2 Discussion 38
3.5 Sine Function 40
3.5.1 Results 40
3.5.2 Discussion 46
3.6 Three Functions 48
3.6.1 Results 49

3.6.2 Discussion ... 53
3.7 Another Parallel Version ... 53
3.7.1 Using Partially Updated Errors ... 53
3.7.2 The Algorithm ... 54
3.7.3 Results ... 56
3.7.4 Discussion ... 58
3.8 Conclusions ... 58

Chapter 4: Using Less Accuracy in Error Back−Propagation ... 61
4.1 Why Less Accuracy? ... 61
4.2 Floating Point Representation ... 62
4.3 Computations with Less Accuracy ... 62
4.4 Using Less Accuracy ... 63
4.4.1 Results ... 66
4.5 Using Flexible Accuracy Adaptation ... 69
4.5.1 Results ... 71
4.5.2 Discussion ... 77
4.6 Parallelism versus Accuracy ... 78
4.7 Conclusions ... 80

Chapter 5: Conclusions and Recommendations ... 83

Acknowledgement ... 85

Literature ... 87

Appendix A: The Source Code of the Secant Method ... 89
A.1 Description of the Source Code ... 89
A.1.1 Declarations ... 89
A.1.2 Main ... 89
A.1.3 Get_equation ... 89
A.1.4 Equation_solver ... 90
A.1.5 Fill_node ... 90
A.1.6 Get_result ... 90
A.1.7 Shift_on ... 90
A.1.8 SetPrecision ... 90
A.2 Source Code ... 91
A.2.1 Declarations ... 91
A.2.2 Main ... 92
A.2.3 Get_equation ... 93
A.2.4 Equation_solver ... 94
A.2.5 Fill_node ... 96
A.2.6 Get_result ... 97
A.2.7 Shift_on ... 97
A.2.8 SetPrecision ... 98

Appendix B: Parallel Implementations of the Error Back−Propagation Learning in *InterAct* ... 99
B.1 Error Back−Propagation in InterAct ... 99
B.2 Parallel InterAct−Implementation of the "Original" Version ... 101

B.2.1 Description of the Source Code 101
B.2.2 Source Code of the Original Version 101
B.3 Parallel InterAct–Implementation of the "Special" Version 103
B.3.1 Description of the Source Code 103
B.3.2 Source Code of the Special Version 104

Appendix C: Implementation of Using Less Accuracy in Error Back–Propagation 107
C.1 Description of the Source Code 107
C.2 Source Code 107

Abstract

Training neural networks using the error back–propagation algorithm is a time–consuming task. The learn time can be reduced by parallelizing the algorithm. Due to the layer–wise composition, neural networks are well–suited for parallel implementations, at least as long as we stick to this layer–wise composition.

Now an interesting question can be formulated: Is it possible to introduce parallelism in error back–propagation learning that is not restricted to the layer–wise composition but does not damage the convergence properties of the algorithm?

It turned out that that using parallelism over the layers only in some cases results in the convergence of the network. In particular, when we use correlated input data, such as in function approximation, we found that this approach can be successful.

If parallelism is introduced, we need something to realize this with. An obvious solution is the usage of multiple processing units. However, if we are restricted to some kind of hardware realization with a single processor unit, we should cope with this single unit and get the best out of it.

A way to achieve parallelism on a single processor unit is to perform computations with less accuracy. For example on a 64–bit processor we could perform two computations in parallel both with an accuracy of 32 bits. Before this can be realized the next question should be answered: Can computations be performed with less accuracy without damaging the convergence properties?

Introducing less accuracy seems very well possible. What we noticed was that the random initialization of the network becomes more important when we use less accuracy. We also looked at the possibilities of adapting the accuracy in a flexible manner. It turned out that it's very hard to determine at which point we have to raise the accuracy, although some promising results were obtained.

Samenvatting

Het trainen van een neuraal netwerk met behulp van het error back–propagation algoritme is een tijdrovend proces. De tijd die nodig is voor het leren kan teruggebracht worden door het algoritme te parallelliseren. Doordat neurale netwerken opgedeeld zijn in lagen, zijn ze geschikt voor parallelle implementaties, in ieder geval zolang we ons aan deze laagsgewijze indeling houden.

Nu kan er een interessante vraag gesteld worden: Is het mogelijk om parallellisme in het error back–propagation leerproces te introduceren dat niet beperkt is tot de laagsgewijze indeling zonder de convergentie van het algoritme aan te tasten?

Het bleek dat het gebruik van parallellisme over de lagen alleen in sommige gevallen resulteerde in de convergentie van het netwerk. In het bijzonder, wanneer gebruik gemaakt wordt van gecorreleerde invoer data, zoals bij functie benaderingen, blijkt deze vorm van parallellisme mogelijk te zijn.

Als we parallellisme introduceren, hebben we iets nodig om dit mee te realizeren. Een voor de hand liggende oplossing is om meerdere processoren te gebruiken. Echter, als we beperkt worden door een hardware realisatie met slechts één processor, zullen we het hiermee moeten doen en hier zo goed mogelijk gebruik van moeten maken.

Een manier om parallellisme te realiseren op een enkele processor is door de berekeningen met minder nauwkeurigheid uit te voeren. Bijvoorbeeld, op een 64–bit processor kunnen we twee berekeningen parallel uitvoeren, elk met een nauwkeurigheid van 32 bits. Voordat dit gerealiseerd kan worden zal eerst de volgende vraag beantwoord moeten worden: Kunnen berekeningen met minder precisie uitgerekend worden zonder dat dit de convergentie van het netwerk schaadt?

Het introduceren van minder precisie lijkt heel goed mogelijk. Het viel echter wel op dat de random initialisatie van het netwerk belangrijker wordt wanneer we minder precisie gebruiken. We hebben ook gekeken naar de mogelijkheden om de precisie op een flexibele manier aan te passen. Het bleek dat het erg moeilijk is om het punt waarop moet worden overgegaan naar een hogere precisie vast te stellen. Desondanks werden enkele veelbelovende resultaten behaald.

Chapter 1

Introduction

Training a neural network using the well–known error back–propagation procedure [5] [16] is a time consuming task. Over the years various ways to accelerate the learning process by adapting the original algorithm have been researched [4]. Hardware architectures that allow for a certain amount of parallelism may well form an additional means to reduce the learning time. In case a feedforward neural network is considered, it's layer–wise structure provides us with a logical decomposition into parts that can be evaluated in parallel. However, if we restrict ourselves to a layer–wise decomposition, we may not make full use of the available hardware facilities.

1.1 Computational Order

Consider the following example in which the errors of a 1–5–1 feedforward neural network (a network with one input, five hidden and one output neuron) will be propagated through the network. Assume that we can process three neurons in parallel. If we stick to a layer–wise decomposition, we need three computational cycles. During the first cycle, the error of the output neuron will be calculated. In the next two cycles this error will be propagated back to the hidden neurons and their errors are determined. As these errors are needed to compute the weight–updates of the incoming synapses of the neurons and the input neuron has no incoming synapses, we don't need to consider the input neuron.

In the above example we need three computational cycles due to the layer–wise decomposition. Based on the number of neurons, two cycles would be sufficient. With this observation we come to an important issue of this work, namely:

> *How should neural networks be partitioned for an optimal usage of the available hardware facilities?*

The key question that has to be answered here is to which extent learning depends on the computational order of the network. In terms of our example: Could we train the network if the error of two of the hidden neurons and the error of the output neuron are computed in parallel? Obviously, the error computations for these two hidden neurons will then be based on an "old" error. Hence, we can't guarantee on forehand that this parallel implementation of the error back–propagation algorithm will converge. In this report, research is presented towards the possibilities of changing the computational order of the errors in a neural network, in such a way that the error back–propagation algorithm still converges, regardless of the changed computational order.

Not all network architectures may be composed into layers without affecting the data–flow through them. Some recurrent architectures are obvious examples of this. The wish to parallelize these kind of networks too, is another motivation for the research presented here.

1.2 Parallelism versus Accuracy

If we want to introduce parallelism in error back–propagation learning, we need something to realize this with. The ideal situation would be if we would have multiple processing–units to our disposal. But if we're restricted to some hardware realization with for example only one processing–unit, we simply should cope with this single processing–unit and get the best out of it.

To make optimal use of the available hardware we would like to introduce parallelism at the expense of the accuracy of a calculation [7] [10]. Suppose we have a 64–bits processor for our neural calculations. Then we can perform one neural calculation with an accuracy of 64 bits. We could also perform two calculations in parallel with an accuracy of 32 bits, or four with an accuracy of 16 bits, etc..

Furthermore, if the difference between the new and the old error is small, there is a certain resemblance between computations with a limited accuracy and computations with an "old" error. This fact is another reason for choosing this approach to introduce parallelism.

In the error back–propagation learning process the average absolute error of the network should be minimized. So if we have a small error, the learning process is nearly done. Hence the precision of the computation of the error is important, because we want an as exact answer as possible. On the other hand if we have a large learning error, the network is far from the desired network. Thus the precision of the computation of the error is less important, because a less exact answer will do.

Summarizing:

- If we have a small learning error we will use a high accuracy and less parallelism.
- If we have a large learning error we will use a low accuracy and more parallelism.

1.3 Gradient Descent Method

The error back–propagation algorithm (neglecting the momentum term) is an example of a gradient descent method (see section 3.1). The gradient descent method is a method for the minimization of a function of several variables [6]. Starting from an approximation x_n, this method obtaines the next approximation of the minimum of the function f by a shift in the direction of the gradient of the function ∇f.

$$x_{n+1} = x_n - \alpha \nabla f(x_n) \tag{1.1}$$

With α the step–length from x_n to x_{n+1}.

In the error back–propagation algorithm the weights are updated according to this method:

$$w_{ji}(n+1) = w_{ji}(n) + \eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} \tag{1.2}$$

With $\mathcal{E}(n)$ the error function we want to minimize.

Another example of the gradient descent method is the Newton Raphson Method for finding the root of a function. This is a rather simple version of the gradient descent method. Here we want to minimize the function itself. In the next chapter we will take a look at the results of changing the computational order of the Newton Raphson Method. Here, the computation of the derivative is expensive. Therefore we don't want to wait until the derivative is calculated, but rather use "old" derivatives instead. We will describe the effect the changing of the computational order has on the convergence properties of the Newton Raphson method. Furthermore we describe the influence of a changing accuracy on Newton Raphson.

After this, a same approach is followed for the error back–propagation algorithm. In chapter 3 we introduce parallelism, in chapter 4 we describe the effect of using less accuracy. And finally in chapter 5 we will draw some conclusions.

Chapter 2

Traditional Gradient Descent Methods

Before we take a look at error back–propagation learning, we will first examine some traditional gradient descent methods. We do this mainly because these methods are more simple than the error back–propagation algorithm. In this chapter we will describe different kinds of gradient descent methods and the effect changing of the computational order has on them. In particular the effect it has on the convergence of the methods will be discussed. Furthermore the results of a flexible accuracy adaptation [2] [14] [15] used on gradient descent method are presented.

2.1 Newton Raphson

In this section we will describe the Newton Raphson method (or simply Newton method) for solving the equation $f(x) = 0$ [1]. Starting with an approximation of the root (x_0), the Newton Raphson formula will be applied iteratively until the desired root x_N is found. The function $f(x)$ has to be continuous and differentiable on the interval $[x_0, x_N]$.

2.1.1 The Algorithm

1. Initialize x_0 as an approximation of the root.

2. Calculate the new root by taking the tangent line to the function. The intersection of this line with the x–axis gives the new (approximation of the) root. The tangent line at point x_n is defined as:

$$f'(x_n) = \frac{f(x_n) - 0}{x_n - x_{n+1}} \tag{2.1}$$

With this formula we can calculate the new root x_{n+1} by rewriting equation (2.1) to:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2.2}$$

3. Step 2 has to be repeated until a stable result is accomplished.

If we choose $f(x) = g'(x)$ and substitute this in equation (2.2) we get a form of the gradient descent method as described in section 1.3 (compare equation (1.1) with equation (2.4)):

$$x_{n+1} = x_n - \frac{g'(x_n)}{g''(x_n)} \tag{2.3}$$

$$x_{n+1} = x_n - g''(x_n)^{-1}\nabla g(x_n) \tag{2.4}$$

The substitution $f(x) = g'(x)$ is allowed if $f(x)$ is integrable. Because the function is continuous on the interval $[x_o, x_N]$ it is also integrable on this interval. This substitution is generalized for more dimensional space in [11] (pp. 240–241).

2.2 Changing the Computational Order of the Newton Raphson Method

As shown in the previous section the Newton Raphson method resembles the error back–propagation learning algorithm to some extent. In the following we will show that changing the computational order of the Newton method does not change the convergence of the method significantly. In the next chapter we will show this for error back–propagation learning.

If we look at equation (2.2) we see that we need to calculate the function $f(x)$ and the derivative $f'(x)$ of the function. We assume that $f(x)$ can be written as a n–th order polynomial equation:

$$f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 \tag{2.5}$$

Evaluation of this function value takes n multiplications. Computation of the derivative is more complicated, because it involves a division:

$$f'(x_n) = \frac{f(x_n + h_n) - f(x_n)}{h_n} \tag{2.6}$$

Furthermore the main equation (2.2) consist of a division too. This "problem" is solved by multiplying with the inverse of $f'(x)$, instead of division by $f'(x)$.

We will change the Newton Raphson method by separating the task of computing the x–value and the f–value from that of computing the derivative. The idea is that we don't want to wait until the right derivative has been computed, but rather take an "old" derivative.

2.3 Mathematical Foundation

If we change the computation order of the Newton Raphson method we use old derivatives. This means that we use the following equation to find the next approximation of the root:

$$x_{n+1} = x_n - \frac{g'(x_n)}{g''(x_{n-1})} \qquad ,n \geq 1 \tag{2.7}$$

Instead of equation (2.3) we will now use this equation. What we want is that this equation also converges under some restrictive conditions. We will proof this in the following.

Define $y_n \equiv x_{n-1}$, $n \geq 1$ and $y_0 \equiv x_0$. Then $(x_n y_n)$ is generated according to the following algorithm (with x_0 and x_1 chosen arbitrarily):

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = f \begin{pmatrix} x_n \\ y_n \end{pmatrix} = \begin{pmatrix} f_1(x_n y_n) \\ f_2(x_n y_n) \end{pmatrix} \qquad n \geq 1 \tag{2.8}$$

with $f_1(x,y) \equiv x - \dfrac{g'(x)}{g''(y)}$ and with $f_2(x,y) \equiv x$.

Define α according to $g'(\alpha) = 0$. Assume g', g'' and g''' are continuous around α. Then the following holds:

$$\begin{aligned} \alpha - x_{n+1} &= f_1(\alpha,\alpha) - f_1(x_n,x_n) \\ &= \frac{\partial f_1(\bar{x}_n,\bar{y}_n)}{\partial x}\cdot(\alpha - x_n) + \frac{\partial f_1(\bar{x}_n,\bar{y}_n)}{\partial y}\cdot(\alpha - y_n) \end{aligned} \tag{2.9}$$

with $(\bar{x}_n, \bar{y}_n)$ lying on the line segment between (x_n, y_n) and (α,α). To obtain the second equality in equation (2.9) we have used the Mean Value Theorem for a function of two variables.

Analogous holds:

$$\alpha - y_n + 1 = \frac{\partial f_2(\hat{x}_n,\hat{y}_n)}{\partial x}\cdot(\alpha - x_n) + \frac{\partial f_2(\hat{x}_n,\hat{y}_n)}{\partial y}\cdot(\alpha - y_n) \tag{2.10}$$

with $(\hat{x}_n, \hat{y}_n)$ lying on the line segment between (x_n, y_n) and (α,α). In matrix–notations we obtain:

$$\begin{bmatrix} \alpha - x_{n+1} \\ \alpha - y_{n+1} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x}(\bar{x}_n\bar{y}_n) & \dfrac{\partial f_1}{\partial y}(\bar{x}_n\bar{y}_n) \\ \dfrac{\partial f_2}{\partial x}(\hat{x}_n\hat{y}_n) & \dfrac{\partial f_2}{\partial y}(\hat{x}_n\hat{y}_n) \end{bmatrix} \begin{bmatrix} \alpha - x_n \\ \alpha - y_n \end{bmatrix} \tag{2.11}$$

Define $w_n \equiv \begin{pmatrix} x_n \\ y_n \end{pmatrix}$, $\beta \equiv \begin{pmatrix} \alpha \\ \alpha \end{pmatrix}$ and call the matrix from equation (2.11) F_n. Then we can rewrite equation (2.11) as:

$$\beta - w_{n+1} = F_n(\beta - w_n) \qquad n \geq 1 \tag{2.12}$$

Introduce the Jacobian–matrix for the functions f_1 and f_2:

$$F(w) \equiv \begin{bmatrix} \dfrac{\partial f_1}{\partial x}(w) & \dfrac{\partial f_1}{\partial y}(w) \\ \dfrac{\partial f_2}{\partial x}(w) & \dfrac{\partial f_2}{\partial y}(w) \end{bmatrix} \qquad \text{with } w = \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.13}$$

This implies that when w_n lies in the neighborhood of β, F_n also lies in the neighborhood of $F(\beta)$. From equation (2.12) follows:

$$\|\beta - w_{n+1}\|_\infty \leq \|F_n\|_\infty \|\beta - w_n\|_\infty \tag{2.14}$$

with the vector–norm $\|x\|_\infty \equiv \max\limits_{1\leq i\leq 2} |x_i|$ and the matrix–norm $\|A\|_\infty \equiv \max\limits_{1\leq i\leq 2} \sum\limits_{j=1}^{2} |a_{ij}|$ for $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ and $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$.

Now, the following holds:

$$F(w) = \begin{bmatrix} \frac{\partial f_1}{\partial x}(w) & \frac{\partial f_1}{\partial y}(w) \\ \frac{\partial f_2}{\partial x}(w) & \frac{\partial f_2}{\partial y}(w) \end{bmatrix} = \begin{bmatrix} 1 - \frac{g''(x)}{g''(y)} & \frac{g'(x) \cdot g'''(y)}{(g''(y))^2} \\ 1 & 0 \end{bmatrix} \tag{2.15}$$

and thus (because $\beta \equiv \begin{pmatrix} \alpha \\ \alpha \end{pmatrix}$ and $g'(\alpha) = 0$):

$$F(\beta) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \tag{2.16}$$

The eigenvalue of $F(\beta)$ is: 0, with multiplicity 2 (and thus the magnitude is also equal to 0).

Lemma:

Let (x_n, y_n), $n \geq 0$ be generated by:

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} f_1(x_n, y_n) \\ f_2(x_n, y_n) \end{pmatrix} = \begin{pmatrix} x_n \\ y_n \end{pmatrix} \tag{2.17}$$

Let (ξ, η) satisfy $\begin{pmatrix} \xi \\ \eta \end{pmatrix} = f\begin{pmatrix} \xi \\ \eta \end{pmatrix}$. Let f_1 and f_2 be continuously differentiable around (ξ, η). Define:

$$F(w) = \begin{bmatrix} \frac{\partial f_1}{\partial x}(w) & \frac{\partial f_1}{\partial y}(w) \\ \frac{\partial f_2}{\partial x}(w) & \frac{\partial f_2}{\partial y}(w) \end{bmatrix} \quad \text{with} \quad w = \begin{pmatrix} x \\ y \end{pmatrix}.$$

Assume that the eigenvalues of $F(\xi, \eta)$ are smaller than 1, in magnitude. Then the algorithm of equation (2.17) will converge to (ξ, η), if (x_0, y_0) is chosen sufficiently close to (ξ, η).

Furthermore, for (x_n, y_n) close to (ξ, η):

$$\left[\begin{pmatrix} \xi \\ \eta \end{pmatrix} - \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} \right] \approx \left[F\begin{pmatrix} \xi \\ \eta \end{pmatrix} \cdot \left(\begin{pmatrix} \xi \\ \eta \end{pmatrix} - \begin{pmatrix} x_n \\ y_n \end{pmatrix} \right) \right] \tag{2.18}$$

□

The preceding shows that we are allowed to use this lemma on our algorithm of equation (2.8), under the conditions that g', g'' and g''' are continuous around α and that x_0 is chosen sufficiently close to α. Conclusion:

Theorem

Let $\{x_n, \quad n \geq 0\}$ be generated by:

$$x_{n+1} = x_n - \frac{g'(x_n)}{g''(x_{n-1})}, \quad n \geq 1 \tag{2.19}$$

Let α satisfy $g'(\alpha) = 0$. Assume g', g'' and g''' are continuous around α. Choose x_0 and x_1 sufficiently close to α. Then the algorithm will converge to α.

□

2.4 A First Experiment: The Regula Falsi Method

In this first experiment we make a few assumptions. First of all we assume that we know the interval $[a, b]$ in which the desired root of the function lies. Secondly we assume that $f(a)$ and $f(b)$ have different signs (lie on different sides of the root). These two points a and b are the starting points of our calculations, respectively x_{n-1} and x_n.

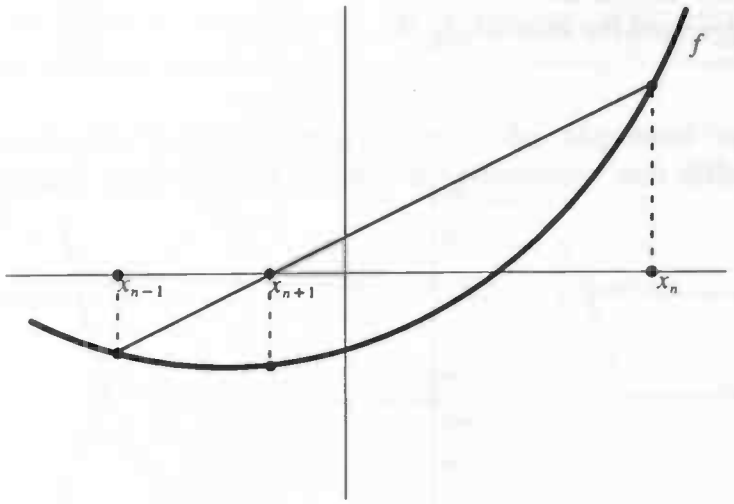


Figure 2.1: *Finding the new approximation of the root (x_{n+1}) of the function f using the Regula Falsi method.*

The second assumption we make is that we may approximate $f'(x_n)$ for any x_n and x_{n-1} by the following equation:

$$f'(x_n) = \frac{f(x_{n-1}) - f(x_n)}{x_{n-1} - x_n} \tag{2.20}$$

$$= \frac{f(a) - f(b)}{a - b} \tag{2.21}$$

Notice that when we substitute $h_n = a - b$ in equation (2.6) we get equation (2.21). In literature this special case of the Newton method is called the Regula Falsi method (see [1] pp. 44–48 and [11] p. 189). In fact the function $f(x)$ is approximated by a straight line through the points $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$. The root of this straight line becomes the new approximation x_{n+1} of the root of the function f. Depending on the sign of $f(x_{n+1})$, the next approximation is made with x_{n-1} and x_{n+1}, or with x_n and x_{n+1}, as long as both function values have opposite signs. This procedure is repeated until the root is approximated closely enough (see figure 2.1). In this case $f(x_{n+1}) < 0$, so the next approximation will be made with x_n and x_{n+1}.

With these assumptions some experiments have been carried out. In the following a distinction will be made between the "sequential" method, where we wait until the right derivative has been calculated, and the "parallel" method, where we don't wait for the right derivative, but take an "old" derivative when necessary.

2.4.1 Results

We consider the following function:

$$f(x) = x^3 - 9x^2 - 66x + 90 \tag{2.22}$$

The function is drawn in figure 2.2. This function has three zero points: $x = -5.6135$, $x = 1.1949$ and $x = 13.4187$. In the following these three roots will be approximated using the Regula Falsi method. We have choosen the following intervals for the different roots:

1. for $x = -5.6135$ we use the interval $[-10, -2]$.
2. for $x = 1.1949$ we use the interval $[-2, 8]$.
3. for $x = 13.4187$ we use the interval $[8, 15]$.

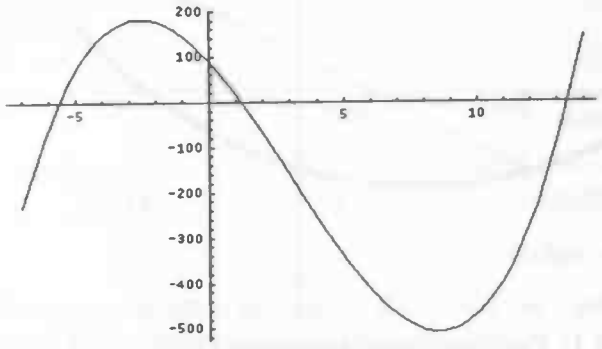


Figure 2.2: *The function* $f(x) = x^3 - 9x^2 - 66x + 90$.

Figures 2.3, 2.4 and 2.5 show the results of the approximation of these three roots.

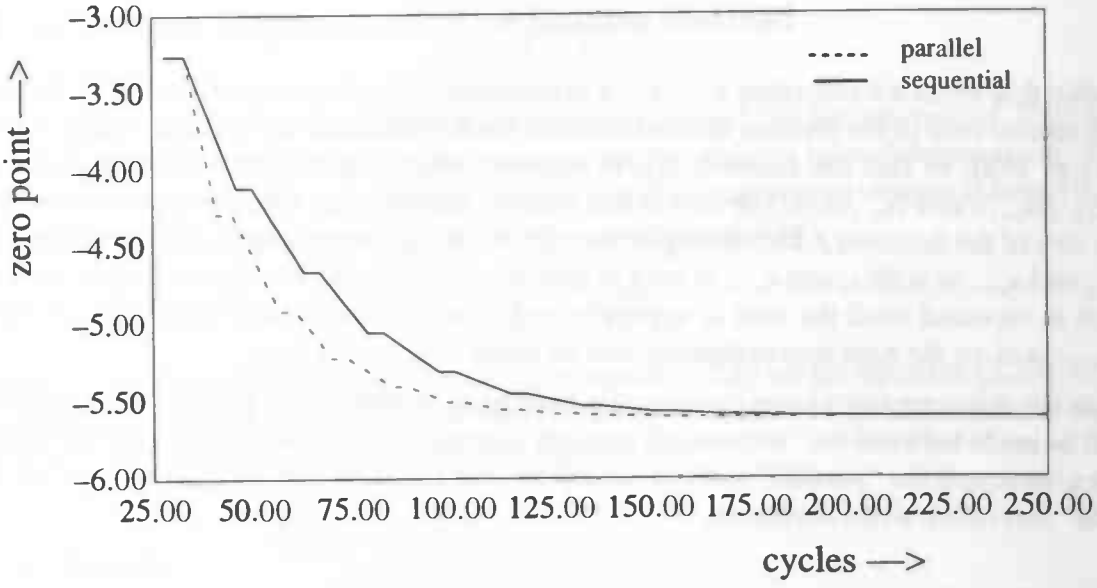


Figure 2.3: *Approximation of root –5.6135 with the "parallel" and the "sequential" method.*

From these figures it can be seen that for this function the method still converges. Furthermore the "parallel" method accelerates the process. For this examples these improvements are:

desired root	**parallel method** number of machine cycles	**sequential method** number of machine cycles	**improvement**
–5.6135	531	649	18%
1.1948	206	213	3%
13.4187	189	262	28%

The profit gained by using the "parallel" method instead of the "sequential" method differs from 3% to 29%. The average profit (over a number of experiments with different equations) is approximately 18%.

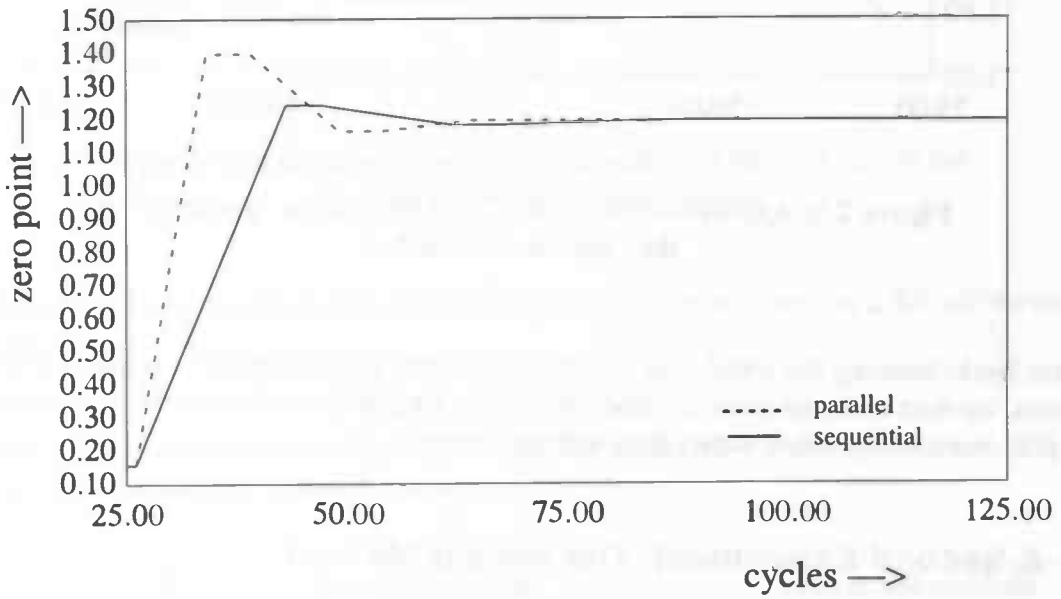


Figure 2.4: *Approximation of root 1.1949 with the "parallel" and the "sequential" method.*

2.4.2 Conclusions

It has been shown that with the "parallel" method the convergence properties are not damaged and the process is accelerated. But for the method we used (Regula Falsi) we have made some important assumptions:

- We know the interval $[a, b]$ in which the root lies.
- We can choose the interval $[a, b]$ in such a way that the signs of the function values $f(a)$ and $f(b)$ have different signs.
- We approximate the derivative by a straight line through the points $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$.

Especially the first two assumptions are rather strict. If we want to apply such an approach on error back–learning, it won't be so easy to choose this interval, if we can do it at all.

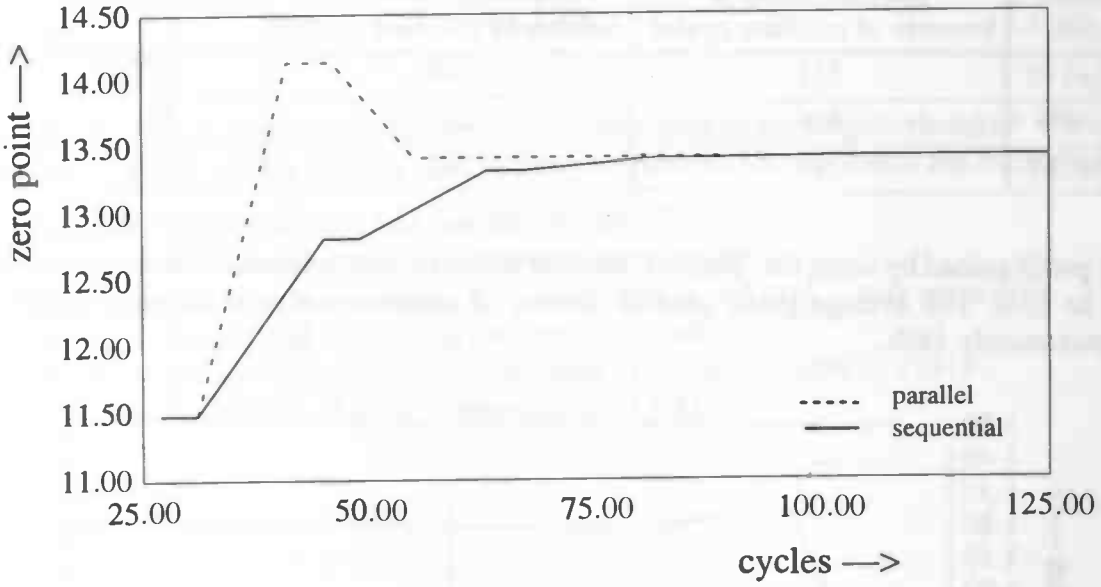


Figure 2.5: *Approximation of root 13.4817 with the "parallel" and the "sequential" method.*

In error back–learning the weight are initialized random. So if we want to minimize the error–function, we start with one random value. In the next section we will show the results of starting with just one starting value, rather than with an interval.

2.5 A Second Experiment: The Secant Method

We now take a look at another version of the Newton method: the Secant method. This method doesn't need the assumption that the function–values $f(a)$ and $f(b)$ have different signs. The Secant method starts with one approximation of the root. The next approximation is found by using equation (2.2). The derivative in this equation is approximated by:

$$f'(x_n) = \frac{f(x_{n-1}) - f(x_n)}{x_{n-1} - x_n} \tag{2.23}$$

Notice that when we substitute $h_n = x_{n-1} - x_n$ in equation (2.6) we get equation (2.23). In literature this is described as a special case of the Newton method (see [11] p. 189). Initially we don't know x_{n-1}, so we use $x_{n-1} = x_n + \epsilon$ (see also figure 2.6).

2.5.1 Results

Results obtained from experiments with the Secant method are less satisfying than those obtained with the Regula Falsi method. Several experiments have been done and in all these experiments

the same phenomenon occurred when we used the "parallel" version. The "parallel" version has mainly difficulties with minima and maxima.

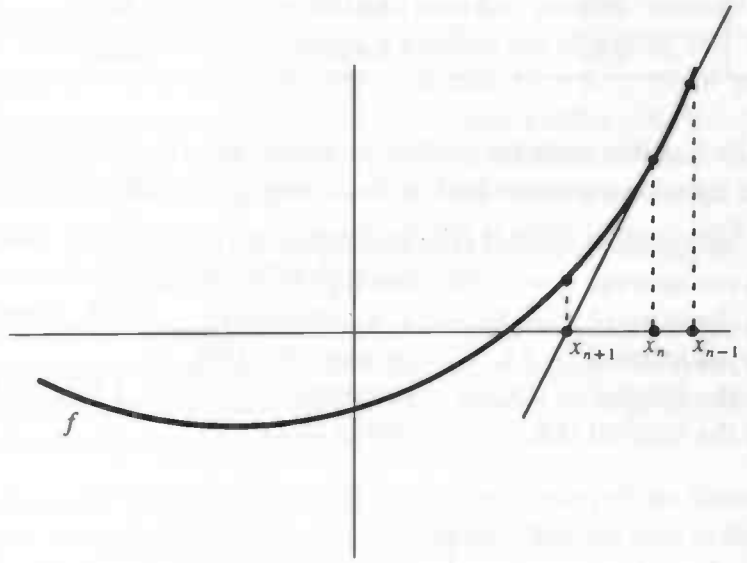


Figure 2.6: *Finding the new approximation of the root x_{n+1} of the function f using the Secant method.*

We'll describe here one example in which the same function as in section 2.4.1 will be used (figure 2.2):

$$f(x) = x^3 - 9x^2 - 66x + 90 \tag{2.24}$$

We have examined the results with different starting values $x_n = -10, -8, \ldots, 14, 16$. The results of these experiments are shown in the next table:

	parallel method		**sequential method**	
starting value	**machine cycles**	**root**	**machine cycles**	**root**
–10	263	–5.6135	255	–5.6135
–8	235	–5.6135	226	–5.6135
–6	166	–5.6135	168	–5.6135
–4	279	–5.6135	224	–5.6135
–2	*diverges*		257	1.1948
0	218	1.1948	197	1.1948
2	174	1.1948	200	1.1948
4	215	1.1948	254	1.1948
6	260	1.1948	247	1.1948
8	*diverges*		649	13.4187
10	*diverges*		333	13.4187
12	211	13.4187	269	13.4187

continued	parallel method		sequential method	
starting value	**machine cycles**	**root**	**machine cycles**	**root**
14	148	13.4187	169	13.4187
16	230	13.4187	229	13.4187

When we take a look at this table we see that problems arise in the neighborhood of -2 and at the interval 8..10. By taking a closer look at these neighborhoods, we notice the following:

When using the "sequential" method (see also figure 2.7):

Points on the interval $<\leftarrow, -3.0]$ converge to -5.6135.
Points on the interval $< -3.0, -2.1 >$ converge to -5.6135, 1.1948 or 13.4187.
Points on the interval $[-2.1, 7.7]$ converge to 1.1948.
Points on the interval $< 7.7, 8.6 >$ converge to -5.6135, 1.1948 or 13.4187.
Points on the interval $[8.6, \rightarrow >$ converge to 13.4187.

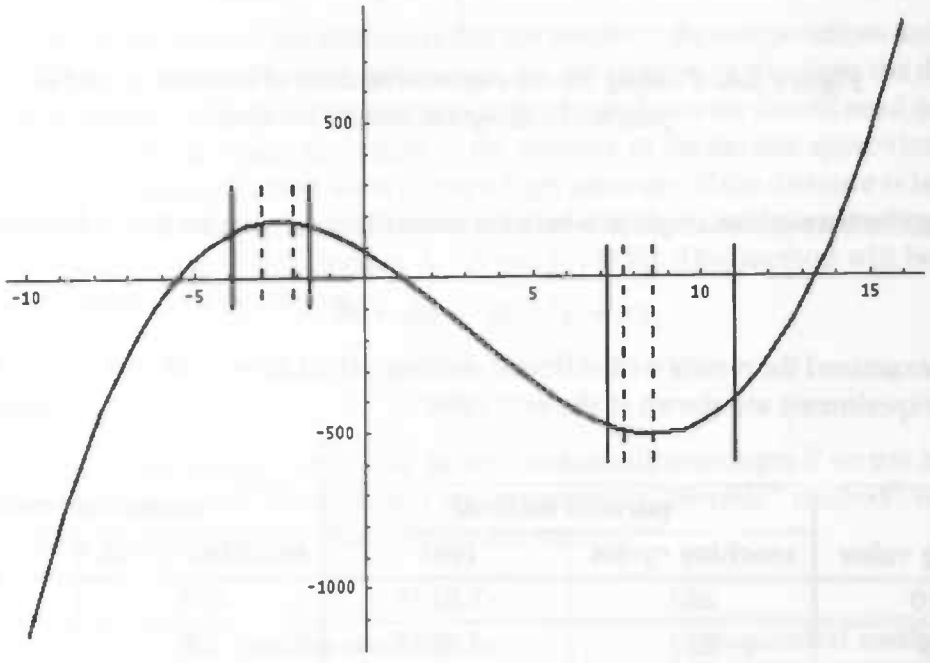


Figure 2.7: *The function f with the intervals in which the problems occur. The dashed lines denote the interval for the sequential method and the straight lines the interval for the parallel method.*

When using the "parallel" method (see also figure 2.7):

Points on the interval $<\leftarrow, -3.9]$ converge to -5.6135.
Points on the interval $< -3.9, -3.4 >$ converge to -5.6135, 1.1948 or 13.4187.
Points on the interval $[-3.4, -1.6]$ diverge.
Points on the interval $[-1.6, 7.2]$ converge to 1.1948.
Points on the interval $< 7.2, 10.1 >$ diverge.
Points on the interval $[10.1, 10.5]$ converge to -5.6135.

Points on the interval $< 10.5, 11.0 >$ diverge.
Points on the interval $[11.0, \rightarrow >$ converge to 13.4187.

The intervals in which the problems occur, this is where the "parallel" method has changing convergence points or diverges and the "sequential" method has changing convergence points, lie around the maximum ($x = -2.56776$) and the minimum ($x = 8.56776$) of the function. In the neighborhood of these points the derivative of the function is rather flat. This means that the next approximation will lie far away from the previous approximation. When the derivative is less flat the method will converge more easily. If we use "old" derivatives, what we do when we use the "parallel" method, we will use these flat derivatives at wrong places, and the effect of it will be enhanced. As a consequence the method can diverge.

More generally, at intervals where the function is monotonous ascending or descending, both methods will converge correctly. However, if the derivative approaches zero (becomes flat), both methods have difficulties, and the "parallel" version can even diverge. By the "parallel" method the area in which the problems occur is larger.

If the function of figure 2.7 will be changed so that the minimum of the function lies just above the x–axis (see figure 2.8a), both methods have problems when we start in the neighborhood of this minimum. The "sequential" method ends in the minimum and the "parallel" method diverges.

Figure 2.8: *Two slightly different functions than figure 2.7.*
(a) *Function with minimum just above the x–axis.*
(b) *Function with a saddle–point and only one root.*

If we change the function in such a way that it has a saddle–point, we experience almost the same problems as with minima and maxima. In the neighborhood of a saddle–point the derivative of the function approaches zero as well. The difference between saddle–points and minima or maxima is that with saddle–points there is no change of sign of the derivative when going from a point at one side of the saddle–point to the other side of the saddle–point. If we choose a simple function with only one saddle point and no minima or maxima it has one root (see figure 2.8b). If we approximate this root using the "sequential" method it will converge correctly (it can't get stuck in a minima or maxima, and it can't converge to a "wrong" zero–point). The "parallel" version does have difficulties near the saddle–point. The reason for this is the flatness of the derivative in this point, which causes the approximations lie far away from each other.

2.5.2 Conclusions

If the secant method is to converge one should choose x_0 close enough to the desired root x_N [1]. The previous sections have shown that when x_0 is chosen near the root, the method will converge correctly, both for the "sequential" and the "parallel" method.

Now that it has been shown that it's possible to use old derivatives by the secant method, we will examine the results of using less accurate computations.

2.6 Using Less Accurate Computations

In this section we will use the same example as described in the previous section: solving $f(x) = 0$ for equation (2.22) (see figure 2.2) with the Secant method. The difference is that we now use less accurate computations, this means that we perform computations with less than the maximum available number of bits. If it's possible to perform computations with less bits, we "save" bits. With these left–over bits we could perform another computation in parallel.

The disadvantage of this form of parallelism is that the results of the computations are sometimes less accurate. When we start a Newton–procedure, we are (relatively) far from the desired root, so a less accurate answer will do. When we are near the desired root we should need more accuracy. For this we adapt the accuracy according to the distance of the current approximation to the desired root. If this distance is small we will use a high accuracy, if the distance is large we will use a lower accuracy and if the distance is somewhere in between we will use an average accuracy (see also source code, appendix A, section A.1.8 and [2] [15]). This method will be referred to as the "flexible" method from now on.

2.6.1 Results

First of all we would like to know wether the Secant method still converges if we use less accurate answers for our computations. To show this we compare the "flexible" method[1] with the "sequential" method. In the next table these results are shown.

	flexible method		sequential method	
starting value	machine cycles	root	machine cycles	root
–10	299	–5.6135	255	–5.6135
–8	228	–5.6135	226	–5.6135
–6	210	–5.6135	168	–5.6135
–4	247	–5.6135	224	–5.6135
–2	285	1.1948	257	1.1948
0	246	1.1948	197	1.1948
2	194	1.1948	200	1.1948

1. We consider the sequential method with changing accuracy. For now we don't consider the parallel version.

continued	flexible method		sequential method	
starting value	**machine cycles**	**root**	**machine cycles**	**root**
4	291	1.1948	254	1.1948
6	282	1.1948	247	1.1948
8	268	13.4187	649	13.4187
10	313	13.4187	333	13.4187
12	298	13.4187	269	13.4187
14	210	13.4187	169	13.4187
16	254	13.4187	229	13.4187

In general the "normal" sequential method is slightly faster in terms of the number of machine cycles needed than the "flexible" one. In some cases the flexible method is faster.

Now that we have shown that the "flexible" method also converges, we want to combine the results of the "flexible" method and the "parallel" method. The results of these two method are shown in the next table.

	parallel flexible method	
starting value	**machine cycles**	**root**
–10	278	–5.6135
–8	244	–5.6135
–6	215	–5.6135
–4	274	–5.6135
–2	*diverges*	
0	223	1.1948
2	147	1.1948
4	209	1.1948
6	225	1.1948
8	*diverges*	
10	diverges	
12	227	13.4187
14	159	13.4187
16	228	13.4187

From this table we see that when we combine the parallel and the sequential method the convergence properties are not damaged. The cases which diverge are caused by the introduction of parallelism (see section 2.5.1).

2.6.2 Conclusions

As shown in the previous section the convergence properties are not damaged by using the "flexible" method. Combining this result with the results of the previous sections, we could introduce parallelism in return for a less accurate intermediate–answers. What we eventually want to accomplish is the following:

> *The degree of parallelism will be determined by the used accuracy. And the accuracy is adapted according to the distance of the current approximation to the desired root.*

This will be shown in the next chapters, where we use the results obtained in this chapter in error back–propagation learning.

Chapter 3

Introducing Parallelism in Error Back–Propagation

In this chapter we introduce parallelism in error back–propagation learning. Especially we would like to introduce parallelism that is not restricted to the layer–wise composition of neural networks.

We will start with a short overview of the error back–propagation algorithm. Furthermore we introduce parallelism in the "original" algorithm. We will show some experiments and we'll examine the results of them.

Analogous to the previous chapter we will denote the normal method used to perform the back–propagation algorithm as the "sequential" method and the parallel version as the "parallel" method.

3.1 Error Back–Propagation

If we want to reduce the learning time of a neural network we may adapt "original" the error back–propagation algorithm [12]. Error back–propagation learning is used to train multi–layer feedforward networks by adapting the weights of the network in a gradient–descent like manner. This process is repeated until the desired network is established. For this we need to know the desired output of the network on forehand (supervised learning). This desired output will be compared with the actual output and the error will then be "propagated" back through the network. Several input–patterns and their outputs are presented to the network. This procedure is repeated until the average sum of the (output)errors is minimized. In the following we will describe this algorithm.

3.1.1 The Algorithm

In this section a brief description of the error back–propagation algorithm is given (as described in [5]). We will start with the notation used.

$x_i(n)$	the i–th input value of the n–th input–pattern.
$y_i^{(l)}(n)$	the output value of neuron i in layer l in response to the n–th pattern.
$v_i^{(l)}(n)$	the internal activation value of neuron i in layer l in response to the n–th pattern.

$w_{ji}^{(l)}(n)$	the synaptic weight from neuron i to neuron j in layer l at the moment the n–th pattern is presented.
$\varphi_i^{(l)}$	the activation function associated with neuron i in layer l.
$\delta_i^{(l)}(n)$	the internal error signal of neuron i in layer l in response to the n–th pattern.
$d_i(n)$	the desired output of neuron i belonging to the n–th pattern.
$e_i(n)$	the error of the output neuron i belonging to the n–th pattern.
α	the momentum term.
η	the learning rate.

Next the algorithm will be described in five steps, at the end of each step, a pseudo code description of the step is given.

1. In the first step the weights $w_{ji}^{(l)}(0)$ of the network are initialized. The biases of the network are represented by the weights $w_{j0}^{(l)}$ which are connected to a fixed input equal to 1.

Initializing weights:

```
for l = first layer to second last layer do
  for i = first neuron in layer l to last neuron in layer l do
    for j = first neuron fed by i to last neuron fed by i do
      w_ji^(l+1)(n) = random
    od
  od
od
```

2. The second step involves the presentation of the input patterns $x_i(n)$, for $n = 1, 2, 3, \ldots$.

Presenting input–patterns:

```
for n = first input pattern to last input pattern do
  x_i(n) = input_i(n)
od
```

3. In the third step the network is evaluated from the input layer to the output layer (the forward pass). This means that for each hidden and output neuron the internal activation v_j and the output value y_j are calculated according to:

$$v_j^{(l)}(n) = \sum_{i=0}^{p} w_{ji}^{(l)}(n)\, y_i^{(l-1)}(n) \tag{3.1}$$

$$y_j^{(l)}(n) = \varphi_j^{(l)}(v_j^{(l)}(n)) \qquad \text{with } l \neq \text{input layer} \tag{3.2}$$

with p the number of neurons in the previous layer $(l - 1)$ and

$$y_j^{(0)}(n) = x_j(n) \qquad \text{with } l = \text{input layer} \tag{3.3}$$

For the activation function φ of the hidden and output neurons we choose the sigmoid function:

$$\varphi_j^{(l)}(v_j^{(l)}(n)) = \frac{1}{1 + \exp(-v_j^{(l)}(n))} \tag{3.4}$$

This sigmoidal nonlinearity is commonly used in multi–layer feedforward neural networks. An activation function should be differentiable. The derivative of this sigmoid function is as follows:

$$\varphi_j'^{(l)}(v_j^{(l)}(n)) = \frac{\exp(-v_j^{(l)}(n))}{[1 + \exp(-v_j^{(l)}(n))]^2} \tag{3.5}$$

As $y_j^{(l)}(n) = \varphi_j^{(l)}(v_j^{(l)}(n)) = \frac{1}{1 + \exp(-v_j^{(l)}(n))}$ and $1 - y_j^{(l)}(n) = \frac{\exp(-v_j^{(l)}(n))}{1 + \exp(-v_j^{(l)}(n))}$ we can simplify the derivative of the activation function to:

$$\varphi_j'^{(l)}(v_j^{(l)}(n)) = y_j^{(l)}(n)[1 - y_j^{(l)}(n)] \tag{3.6}$$

We will use this activation function[2] and it's derivative in the pseudo code descriptions of the error back–propagation algorithm.

For input neurons the identity function will serve as activation function (as can be observed from equation (3.3)). Hence these neurons are nothing more than the interface between the network and the outside world.

Forward pass:

```
for l = first layer to last layer do
  for j = first neuron in layer l to last neuron in layer l do
    if l == input layer then
      y_j^(1)(n) = x_j(n)
    else /* l is not input layer */
      v_j^(l)(n) = 0
      for i = first neuron feeding j to
          last neuron feeding j do
        v_j^(l)(n) = v_j^(l)(n) + w_ji^(l)(n) * y_i^(l-1)(n)
      od
      y_j^(l)(n) = 1 / (1 + exp(-v_j^(l)(n)))
    fi
  od
od
```

Note that "first neuron feeding *j*" represents the bias of neuron *i*.

4. In the fourth step the errors of the neurons are computed starting with the output layer and then propagating these errors back through the network (the backward pass). The errors of the output neurons are calculated by subtracting the calculated output from the desired output:

$$e_j(n) = d_j(n) - y_j^{(l)}(n) \qquad \textit{with } l = \textit{output layer} \tag{3.7}$$

2. In the following, we will use the sigmoid on a range from 0 to 1 as activation function.

The internal error signals δ_j for the output layer are calculated by multiplying $e_j(n)$ with the derivative of the activation function:

$$\delta_j^{(l)}(n) = \varphi_j'^{(l)}(v_j^{(l)}(n))e_j^{(l)}(n) \qquad \textit{with } l = \textit{output layer} \tag{3.8}$$

These errors will be propagated back through the network. The error signals of the neurons in other layers are calculated by:

$$\delta_j^{(l)}(n) = \varphi_j'^{(l)}(v_j^{(l)}(n)) \sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) \qquad \textit{with } l \neq \textit{output layer} \tag{3.9}$$

With these error signals the adjustments to the synaptic weights can be computed according to the following equation (the generalized delta rule):

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha[w_{ji}^{(l)}(n) - w_{ji}^{(l)}(n-1)] + \eta\delta_j^{(l)}(n) y_i^{(l-1)}(n) \tag{3.10}$$

This results in the following algorithm:

Backward pass:

```
for l = first layer to second last layer do
  for i = first neuron in layer l to last neuron in layer l do
    bpe_i^(l)(n) = 0
  od
od
l = output layer
for i = first neuron in layer l to last neuron in layer l do
  bpe_i^(l)(n) = d_i(n) - y_i^(l)(n)
od
for l = output layer to second layer do
  for i = first neuron in layer l to last neuron in layer l do
    δ_i^(l)(n) = y_i^(l)(n) * [1 - y_i^(l)(n)] * bpe_i^(l)(n)
    for j = first neuron feeding i to last neuron feeding i do
      bpe_j^(l-1)(n) = bpe_j^(l-1)(n) + δ_i^(l)(n) * w_ij^(l)(n)
      w_ij^(l)(n + 1) = w_ij^(l)(n) + α * (w_ij^(l)(n) - w_ij^(l)(n - 1)) +
                        η * δ_i^(l)(n) * y_j^(l-1)(n)
    od
  od
od
```

The main idea of the algorithm is that for each neuron (if possible) the error of their feeding neurons are (partially) updated. Furthermore the synapses between this neuron and its feeding neurons are updated too. So

- for each neuron i it's contribution to the error signals $\delta_j^{(l)}$ of all feeding neurons j is calculated,
- for each neuron i the weights w_{ij} on the feeding synapses are updated

5. Steps 2, 3 and 4 are repeated until a stable network is accomplished.

Stability check:

```
repeat
   n=0
   Presenting input–patterns
   Forward pass
   Backward pass
   n++
until network is stable
```

In practice we do not proceed until the network is stable, but until the (root mean square) error of the network drops below a specified level.

3.2 Introducing Parallelism

In this section we introduce parallelism in the error back–propagation algorithm. As described in section 3.1 we distinguish two passes: the forward pass, where the evaluation of the network is done and the backward pass, where the trainings process is performed. In this section we will introduce parallelism in the backward pass.

3.2.1 Motivation for Using Parallelism in the Backward Pass

There are several reason for choosing to introduce parallelism in the backward pass instead of in the forward pass.

First of all, the main profit can be gained in the backward pass as it involves a substantial number of multiplications to calculate each weight update. Admittedly, the forward pass uses multiplications too, but if we compare equation (3.1) to equations (3.8) to (3.10) the difference shows clearly.

Furthermore, if we want to introduce parallelism in the forward pass, the layer–wise composition of the network may lead to inefficient use of the available resources. We will illustrate this with two examples of the exclusive–or problem.

In the first example we will introduce parallelism in the evaluation process of the exclusive–or problem modelled by a 2–2–1 feedforward network. We assume that we can perform two parallel computations. Then the following scheme will be performed:

- the two input neurons will be evaluated,
- the two hidden neurons will be evaluated,
- the output neuron will be evaluated.

Because the parallelism is limited to the layers, this will work fine. It will cost three cycles to evaluate the network. Sequential it would have cost five cycles (one for each neuron). So we will gain two cycles per evaluation of the network.

In the next example we will use three parallel computations, to use all available resources, instead of the two used above. Then the following scheme will be performed to evaluate the network:

- the two input neurons and one hidden neuron will be evaluated,
- the other hidden neuron and the output neuron will be evaluated.

This time we have introduced parallelism over the layers. With this we have introduced some problems. The first, and most serious, problem is that the result at the output is not correct.

Assume the first pattern we present to the network is 0 0, which matches an output 0. The second pattern is 0 1, which matches an output 1. If we evaluate the network with these two input patterns and three parallel computations, the following will happen:

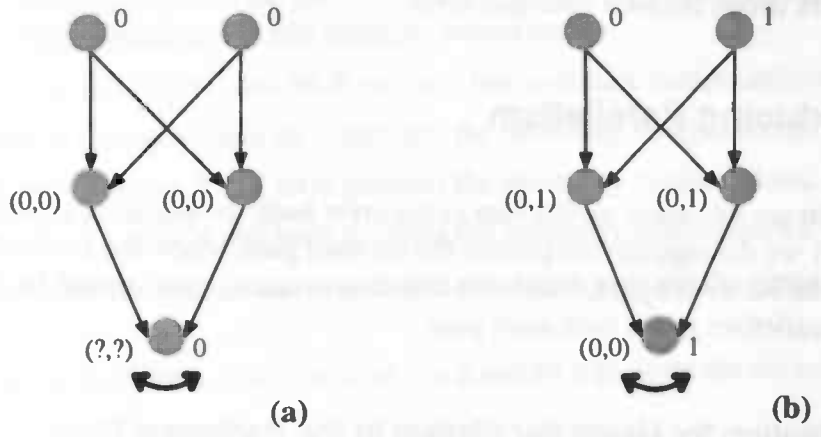


Figure 3.1: *Two steps of the parallel evaluation process for the exclusive–or problem with three parallel computations. The numbers to the right of the neurons denote the input or output patterns. The bracketed numbers to the left of the neurons denote the values that are propagated. The arrow beneath the output neuron denotes that the propagated value is compared with the desired output of the network.*

First we evaluate the network with the first input pattern. Because we perform three computations in parallel, the hidden neurons and the output neuron are considered at the same time. The outputs of the hidden neurons will be computed correctly, according to the input pattern 0 0 and the synaptic weights between the hidden and the input neurons. However, the output of the output neuron, will be computed at the same time. So the output neuron can't use the new outputs of the hidden neurons (because these aren't computed yet), and should use the "old" output values of the hidden neurons, for the computation of its output. Because this is the first evaluation of the network this will be based on the random initialization (see figure 3.1 a).

Then the second input pattern will be presented to the network. This again results in correct output values for the hidden neurons, but because we perform three computations in parallel, the output of the output neuron will again be based an "old output values of the hidden neurons. These "old" values are the values that were computed based on the previous input pattern 0 0. So the output of the output neuron will be based 0 0, instead of on the current pattern 0 1 (see figure 3.1 b).

This output value will be compared to the target value. So what will happen is that an output value based on input pattern 0 0 will be compared to a target value 1 (which belongs to input pattern 0 1). This is obviously not what we want.

Our second example pointed out that we can't tell on forehand if we get the right output values if we perform computations in parallel in the evaluation pass. This depends on the number of parallel computations and especially if these neurons that are updated in parallel are in different layers.

For these reasons, the more expensive computation of the backward pass and the "wrong" output values by introducing parallelism over the layers, we decided to introduce parallelism in the backward pass, instead of in the forward pass. The fact that the learning process resembles the Newton Raphson method (gradient descent) and that we have seen in chapter 2 that it is possible to change the computational order in the Newton Raphson method, are indications that it is possible to introduce parallelism over the layers in error back propagation. In the next section we will introduce a parallel implementations of the error back–propagation algorithm.

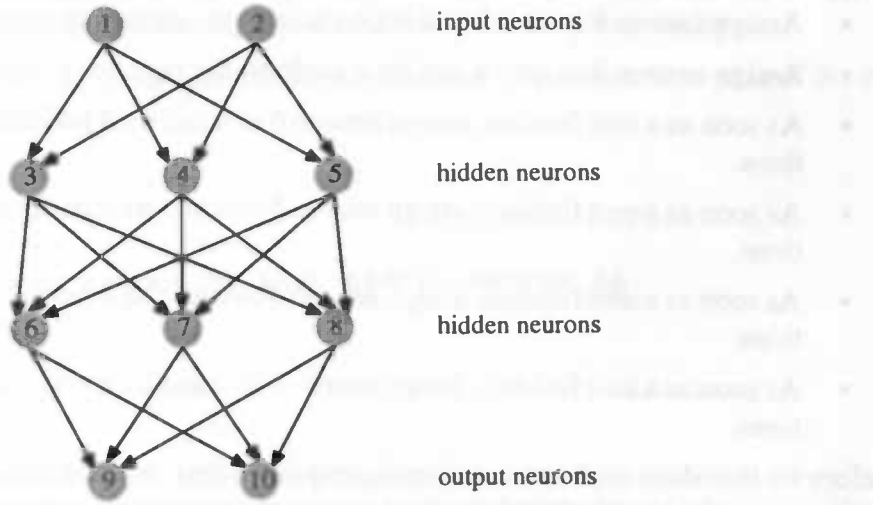


Figure 3.2: *A 2–3–3–2 feedforward neural network. The neurons are numbered from one to ten, starting with the input neurons, followed by the hidden neurons and finishing with the output neurons.*

3.2.2 Adapting the backward pass

We want to introduce parallelism in the backward pass in the following way. We assume we can perform N neural computations in parallel. Furthermore we assume that the neurons are numbered in a way as shown in figure 3.2. The first input neuron gets the lowest number, followed by the next input neuron which gets the next number. This is repeated for all input neurons, hidden neurons (starting with the first hidden layer) and output neurons. We need these numbers to denote the order in which the errors of the neurons should be updated. So we do stick to some kind of order, but we won't be restricted to the layers.

For example, assume $N = 4$ and that we have a network as shown in figure 3.2. In this example we compute the errors of neurons 10, 9, 8 and 7 in parallel in the first cycle and we compute the errors of neurons 6, 5, 4 and 3 in parallel in the next cycle. As the input neurons haven't got any incoming synapses, and thus the errors of these neurons don't have to be computed, we do not consider these neurons here.

3.2.3 Scheduling

What we eventually want to realize is a system that computes N weight updates in parallel. So we have (in some way) N units that can perform these computations. We would like to assign neurons to these units in such a way that the units are used optimally. Thus we would like to have some kind of a scheduling algorithm.

Let's consider the same example of the previous section (figure 3.2). Assume again that $N = 4$. An optimal scheduling algorithm for this example would be:

- Assign neuron 10 to unit 1 and let it perform the necessary computations.
- Assign neuron 9 to unit 2 and let it perform the necessary computations.
- Assign neuron 8 to unit 3 and let it perform the necessary computations.
- Assign neuron 7 to unit 4 and let it perform the necessary computations.
- As soon as a unit finishes, assign neuron 6 to it and let it perform the necessary computations.
- As soon as a unit finishes, assign neuron 5 to it and let it perform the necessary computations.
- As soon as a unit finishes, assign neuron 4 to it and let it perform the necessary computations.
- As soon as a unit finishes, assign neuron 3 to it and let it perform the necessary computations.

Before we introduce such a scheduling algorithm we first need to know whether introducing parallelism over the layers in the backward pass of the error back–propagation algorithm does not damages the convergence properties. For this we introduce a kind of pseudo–parallelism. This means that the computations that should be performed in parallel (N neurons):

- are performed sequential,
- the result of these computations are stored in temporary variables,
- and finally these temporary variables are assigned to the actual variables and these variables are stored.

This should be repeated for the next N neurons.

This kind of simulated parallelism is used in the next sections for the two versions of the error back–propagation algorithm.

3.2.4 Adapting the "original" algorithm

The backward pass of the "original" algorithm is changed to perform N weight updates in parallel. For N neurons, starting at the output layer the error signals of their feeding neurons and the

weights of their incoming synapses are updated in parallel. Then the next N neurons will be considered. This is repeated until we reach the input layer. The variable `i` denotes the neuron that is currently considered.

We assume the neurons are numbered as described in section 3.2.2, so starting with one for the first input neuron and numbering further via the hidden neurons until the output neurons. Furthermore we assume that we have calculated the number of neurons in the network, the number of input neurons and the number of output neurons.

By introducing parallelism we introduce a problem, namely that the back–propagated errors are set to zero at the beginning of the backward pass. When we perform computations in two different layers in parallel, the back–propagated errors are still zero or partially updated, and thus we will propagate this zero (or partial updated) value back to the previous layer. This is probably not what we want[3]. We don't want to wait until the new error is calculated, but rather continue with an old error (not zero or partial updated). So we have to store the old errors before setting them to zero.

This involves another problem, we need to know whether the error is partially updated or totally updated (new). We will do this by using a flag **errorused**, which denotes whether the error of this neuron is already used to calculate the error of a neuron in the previous layer.

This results in the following pseudo–parallel implementation of the backward pass of the error back–propagation algorithm:

Backward pass:

```
/* initialization */
for i = first hidden neuron to last hidden neuron do
   olderror_i = bpe_i
   bpe_i = 0
   errorused_i = false
od
/* set backprop errors for the output neurons */
for i = first output neuron to last output neuron do
   bpe_i(n) = d_i(n) - y_i(n)
   errorused_i = false
od
/* get last neuron */
i = last neuron
/* main loop: repeat until an input neuron is reached */
while i != input neuron do
   /* pseudo parallel loop */
   for nrpar = 1 to N do
      /* if a neuron is not an output neuron check it's error */
      if i!= output neuron then
         /* check if error is new */
```

3. Instead of using an old error we could use a partially updated error. For now we chose to use the old error, or the new error when it is totally updated. Another version of the algorithm could be derived if we didn't initialize the errors to zero, but rather subtracted the contribution of the old error and added the contribution of the new error. The disadvantage of this last version is that it involves a lot of administration.

```
      newerror = true
      for j = first neuron fed by i to last neuron fed by i do
        if !errorused_j then
            newerror = false
            break
        fi
      od
      /* if the error is not totally updated use old error */
      if !newerror then
        bpe_i = olderror_i
      fi
    fi
    /* compute error signals */
    δ_i(n) = y_i(n) * [1 - y_i(n)] * bpe_i(n)
    for j = first neuron feeding i to last neuron feeding i do
      /*compute backprop error and new weight and store them in
        temporary variables */
      tempbpe_j(n) = bpe_j(n) + δ_i(n) * w_ij(n)
      tempw_ij(n + 1) = w_ij(n) + α * [w_ij(n) - w_ij(n - 1)] +
                        η * δ_i(n) * y_j(n)
    od
    /* compute bias */
    tempw_i0(n) = w_i0(n) + α * [w_i0(n) - w_i0(n - 1)] + η * δ_i
    /* get previous neuron */
    i = previous neuron
    /*if we have reached the input layer break from the
      "pseudo parallel loop" */
    if i = input neuron then
      break
    fi
  od /* pseudo parallel loop */
  /*now "nrpar" denotes the number of neurons that are updated
    pseudo-parallel, these "nrpar" neurons should be really
    updated, so copy the temporary variables to the real
    variables, and set errorused to true */
  i = i + nrpar
  for nr = 1 to nrpar do
    for j = first neuron feeding i to last neuron feeding i do
      bpe_j(n) = tempbpe_j(n)
      w_ij(n + 1) = tempw_ij(n + 1)
    od
    w_i0(n) = tempw_i0
    errorused_i = true
    i = previous neuron
  od
od /* main loop */
```

3.3 Experiments

In the next section we consider some experiments we have conducted with the parallel version of error back–propagation. In all these experiments we distinguish "the sequential method", "the parallel method". There are a few question we would like to answer here:

Does the parallel method converge?

Is the parallel method better than the sequential one?

Does the kind of experiment infects the convergence of the parallel method?

We have considered the following problems:

Exclusive–or
Sine function
Three functions problem

In the following sections experiments done with these problems are described and the results of these experiments are analyzed. Each section will start with a description of the problem, followed by some general information. This information holds for all different experiments performed for a specific problem. Then the results of the experiment are given and finally an analysis of these results is presented.

The general information consists of:

- the learning rate (η)
- the momentum term (α)
- the maximum number of training cycles (if this number is reached the training process stops)
- the minimum error (if the absolute error of the network has reached this minimum error the training process stops)
- the number of different data patterns used
- the percentage of the data set that is used for training

Presentation of the results of experiments start with some information about the experiment considered. This information consists of:

- the kind of network,
- a figure in which the network is shown,
- whether input is presented in a random or an ordered way,
- the kind of method that is used for learning (sequential or parallel),
- if a parallel method is used, the number of parallel computations.

Note that the total number of input patterns will be randomly divided in a train–set and a test–set. Presenting the input random means that the train–set will be put in a random order and is then presented to the network. If the whole train–set has been presented, the train–set will be put in an(other) random order again, then this will be presented etc.. Presenting the input ordered means that the train–set (which is chosen random from all the input patterns is presented in an ordered way.

After the information about the experiment, the results of the experiment are given. This includes:

- the number of training cycles needed to obtain the result
- the (final) average absolute error obtained during training
- the (final) average absolute error obtained during testing
- the results of the evaluation of the network (graphical or textual)

Finally some remarks about the results are made.

Note that for all experiments that were done only the most relevant experiments will be presented here. If the same experiment is repeated it is not said that it results in the same trained network. After all, the initial values of the weights depend on the random initialization of the network. This, and the fact that weight updates depend on the (random) order in which the training samples are presented to the network almost ensures that the final network will be slightly different each time an experiment is repeated. For this reason we present here the results of that experiment that is most representative for this type of experiment

3.4 Exclusive–or

The exclusive–or problem (XOR) is a special example of classifying points in an unit hypercube, where only the four corners of this hypercube are considered. There are only two classes, class 0 and class 1, and each point on the hypercube belongs to one of these classes. There are four different input patterns, namely (0,0), (0,1), (1,0) and (1,1). Each of these input patterns belongs either to class 0 or to class 1. To be precise:

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

So input patterns (0,0) and (1,1) belong to class 0 and input patterns (0,1) and (1,0) belong to class 1.

The exclusive–or problem is an example of a non linear separable problem. This can be shown from figure 3.3. If a problem is linear separable the two classes can be divided by a straight line. In the case of the XOR this is not possible.

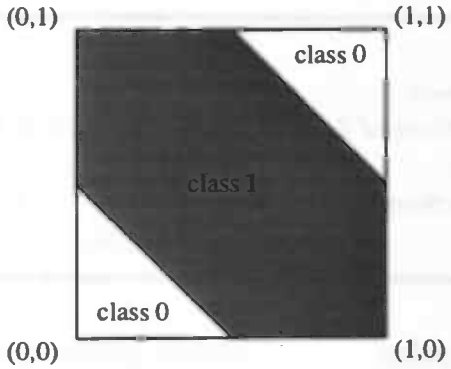


Figure 3.3: *The exclusive–or problem.*

Non linear separable problems can not be solved by a network without a hidden layer. So a hidden layer is definitely needed here.

3.4.1 Results

Because the input data of the exclusive–or hasn't got any kind of order, we can't really talk about input that is presented random or ordered. In this experiment random input means that every time a pattern is chosen randomly from the four possibilities.

The input patterns and the target patterns are scaled. The input patterns to 0.1 and 1.0 and the output patterns to 0.1 and 0.95.

Testing the network of the exclusive–or, consists of the presentation of the four input patterns to the network and the results of an evaluation of the network with these four inputs. Also the performance of the network is given, this is the percentage of the input patterns that are classified correctly. All output larger than 0.7 is considered to be class 1 and all output smaller than 0.3 is considered to be class 0.

Because the four input patterns are completely different for the exclusive–or problem, we do not divide the patterns in training patterns and test patterns.

After each 1000 patterns presented to the network, the error of the network is compared with the minimum error. If the error drops below this minimum the trainings process is ended. The process can only be stopped if a multiple of 1000 patterns has been presented to the network.

General information:

learn rate = 0.5
momentum term = 0.7
maximum number of trainings cycles = 100000
minimum absolute error = 0.00001
number of different data patterns = 4

A) Network information:

learning exclusive–or
uses a random initialized 2 2 1 network
uses random input
uses sequential method

Results:

XOR: after 18000 training cycles:
average error over training–patterns: 0.000006
average error over test–patterns: 0.000005
inputs: 0.100000 0.100000 output: 0.100000
inputs: 0.100000 1.000000 output: 0.949993
inputs: 1.000000 0.100000 output: 0.949993
inputs: 1.000000 1.000000 output: 0.100004
performance 100 %

Remarks: *The exclusive–or can be learned by a 2–2–1 network using the sequential method.*

B) Network information:

learning exclusive–or
uses a random initialized 2 2 1 network
uses random input
uses parallel method
with 1 parallel computation(s)

Results:

XOR: after 18000 training cycles:
average error over training–patterns: 0.000007
average error over test–patterns: 0.000006
inputs: 0.100000 0.100000 output: 0.100001
inputs: 0.100000 1.000000 output: 0.949991
inputs: 1.000000 0.100000 output: 0.949991
inputs: 1.000000 1.000000 output: 0.100005
performance 100 %

Remarks: *As expected the parallel method with one parallel computation also converges correctly. This indicates a (at least partially) correct implementation of the algorithm.*

C) Network information:

learning exclusive–or
uses a random initialized 2 2 1 network
uses random input
uses parallel method
with 2 parallel computation(s)

Results:

XOR: after 100000 (maximum) training cycles:
average error over test–patterns: 0.274800
average error over test–patterns: 0.274800
inputs: 0.100000 0.100000 output: 0.115840
inputs: 0.100000 1.000000 output: 0.715804
inputs: 1.000000 0.100000 output: 0.714294
inputs: 1.000000 1.000000 output: 0.713459
performance 75 %

Remarks: *Introducing two parallel computations results in a not correctly trained network. A performance of 75% doesn't look so bad, but if we take a closer look to the results of this experiment we see that input pattern (1, 1) is always mis–classified. In fact the network has learned to produce the OR instead of the XOR.*

D) Network information:

learning exclusive–or
uses a random initialized 2 2 1 network
uses random input
uses parallel method
with 3 parallel computation(s)

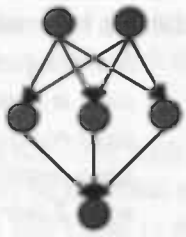
Results:

XOR: after 100000 training cycles:
average error over training–patterns: 0.419670
average error over test–patterns: 0.421528
inputs: 0.100000 0.100000 output: 0.491538
inputs: 0.100000 1.000000 output: 0.509362
inputs: 1.000000 0.100000 output: 0.508265
inputs: 1.000000 1.000000 output: 0.512202
performance 0 %

Remarks: *If we introduce three parallel computations, the network can not even classify one input pattern correctly.*

E) Network information:

learning exclusive–or
uses a random initialized 2 3 1 network
uses random input
uses parallel method
with 2 parallel computation(s)



Results:
XOR: after 19000 training cycles:
average error over training–patterns: 0.000007
average error over test–patterns: 0.000004
inputs: 0.100000 0.100000 output: 0.100000
inputs: 0.100000 1.000000 output: 0.949993
inputs: 1.000000 0.100000 output: 0.949996
inputs: 1.000000 1.000000 output: 0.100005
performance 100 %

Remarks: *If we enlarge the network to a 2 3 1 network, it does converges with two parallel computations.*

3.4.2 Discussion

First of all we notice that the results of the first two experiments (A and B) are nearly the same. This is what we expect because these experiments are: one sequential and one with the parallel method with one parallel computation. The parallel method performed with only one parallel computations should equal the sequential method. The differences are caused by the random initialization.

Furthermore we see that the results of the parallel methods are rather disappointing. The 2–2–1 networks with two and three parallel computations (C and D), have great difficulties learning the exclusive–or. These two examples have in common that, at a certain time, neurons from two different layers are considered at the same time (see figure 3.4).

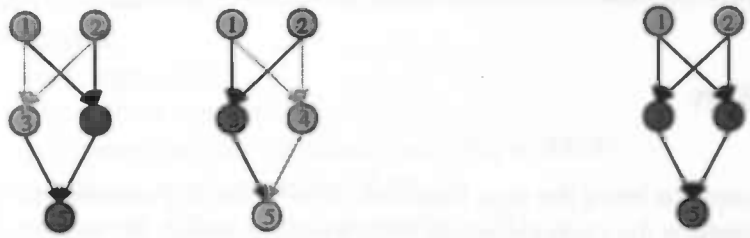
This implies that some neurons in the hidden layers are updated according to "old" values of the error. Obviously the network can't deal with this old values when learning the exclusive–or problem. An "old" error is based on the previous input pattern.

Let's consider the first example that fails (figure 3.4a). In this example two neurons are updated at the same time. Because there is only one output neuron, one of the hidden neurons will be updated at the same time as this neuron (neuron 5 and neuron 4) . So neuron 4 does all it's computation with old values. Assume the next four input patterns were presented to the network: (0, 0), (0, 1), (1, 0) and (1, 1), in this order. Then we observe the following:

- Input pattern (0, 0) corresponds to a target value of 0. The error of neuron 5 based on this target value will be propagated back.
- Next input pattern (0,1) is presented to the network. This input corresponds to a target value of 1. The error of neuron 5 is based on this target value. According to this error

weight updates are made and this error is propagated back. Neuron 3 uses this back propagated error. But neuron 4, which is considered at the same time as neuron 5 uses the back propagated error corresponding to the previous input pattern. This error was based on a target value of 0 (instead of 1) and as the weight updates made by this neuron will be based on this value the training process is corrupted beyond repair.

- The next input patterns cause a similar behavior. This is shown in figure 3.5.



(a)

1: neurons 4 and 5 are considered.
2: neuron 3 is considered.
3: the above is repeated for a new input pattern.

(b)

1: neurons 3, 4 and 5 are considered.
2: this is repeated for a new input pattern.

Figure 3.4: *Two examples of 2–2–1 networks that do not converge when parallelism is introduced. Two parallel computations* **(a)** *and three parallel computations* **(b)** *are used. In both cases neurons from different layers are updated at the same time.*

Summarizing, neuron 4 computes weight updates in a half of the cases with a wrong propagated error.

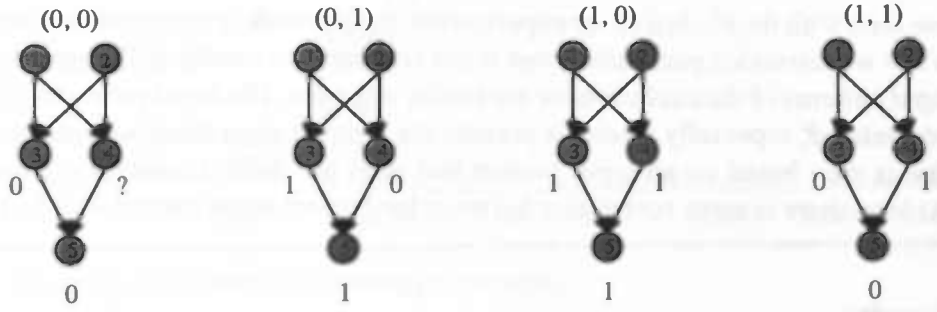


Figure 3.5: *Four input patterns presented to a 2–2–1 network with two parallel computations (neuron 4 and neuron 5). And the effects the input patterns have on the back propagated error. The numbers denote whether this error is based on a target of 0 or 1.*

If we enlarge the network to a 2–3–1 network and use a parallel method with two parallel computations (experiment E) the network does learn the exclusive–or. The reason for this is that the

network learns to "ignore" the hidden neuron that is computed in parallel with the output by setting the weight on its incoming and outgoing synapses (almost) to zero. Effective result: a 2–2–1 network in which neurons in different layers are update sequentially and thus a network that does converge.

As seen from the experiments done with the exclusive–or this network can not cope with parallel computation in different layers. The reason for this is that an old value can be the total opposite of the correct value (0 versus 1). More so this in half of all the cases this old value is wrong.

3.5 Sine Function

This experiment attempts to learn the sine function. A x–value is presented to the network and the network should produce the corresponding output–value, $sin(x)$. As inputs, values on the interval $[0, 2\pi]$ are used, resulting in outputs between $[-1, 1]$. These values will be scaled during training, but are scaled back eventually, in order to generate an output plot using the original intervals. In figure 3.6 the sine function is drawn on the interval used.

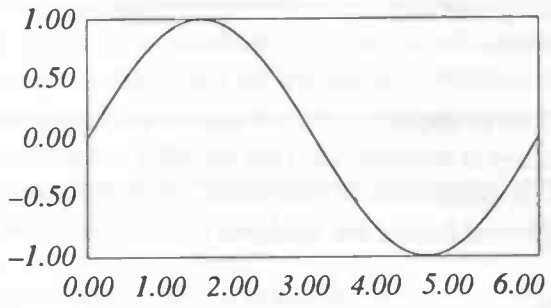


Figure 3.6: *The sine function at interval [0, 2π].*

As we have seen with the exclusive–or experiments, the network is not capable to learn the exclusive–or when we introduce parallelism that is not restricted to one layer. The reason for this was that the input patterns of the exclusive–or are totally opposite. The input patterns of the sine functions are correlated, especially when we present the input in an ordered way to the network. An "old" value is then based on an input pattern that does not differ much from the current input pattern. At least there is some correlation between the current input pattern and the previous one.

3.5.1 Results

To learn the sine function 200 patterns are created on the interval $[0, 2\pi]$. These patterns are divided in a train–set and a test–set. The train–set is presented during learning. Contrary to the exclusive–or problem, the presentation of one input pattern is not denoted as one cycle, but the presentation of the entire train–set is denoted as one cycle.

The test–set is used to test if the network has learned the function correctly. The patterns in the test–set do not occur in the training–set, so to the network these patterns are entirely new. In this case 30% of the patterns is used for training and the other 70% is used for testing.

For each experiment we have done with the sine function a plot will be included with the results of the test–set.

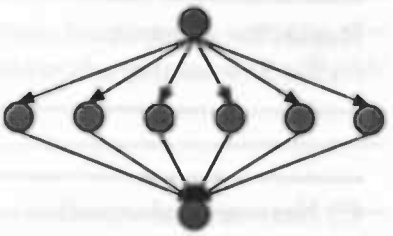
If the train–set is presented 25 times the error of the network will be compared to the minimum error. If the error drops below this minimum the trainings process is ended. The process can only be stopped if a multiple of 25 times the train–set has been presented to the network.

General information:

learn rate = 0.9
momentum term = 0.2
maximum number of trainings cycles = 20000
minimum absolute error = 0.01
number of different data patterns = 200
percentage of data used for training = 30%

A) Network information:

learning sine
uses a random initialized 1 6 1 network
uses random input
uses sequential method

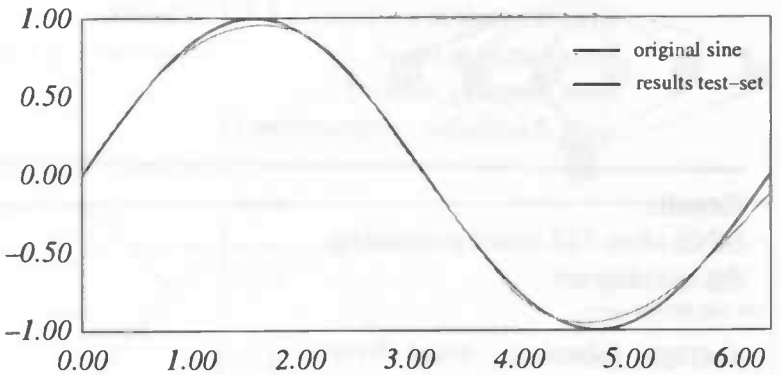


Results:

SINE after 1025 times presenting the training set:

average (absolute) error over training–patterns: 0.009474

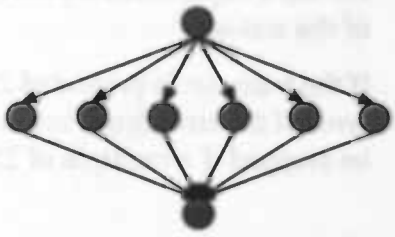
average (absolute) error over test–patterns: 0.011084



Remarks: *The sequential method converges correctly.*

B) Network information:

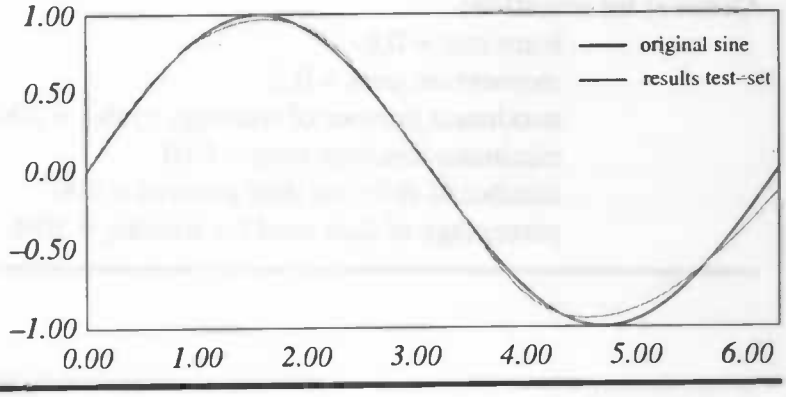
learning sine
uses a random initialized 1 6 1 network
uses random input
uses parallel method
with 1 parallel computation(s)



Results:
SINE after 700 times presenting the training set:

average (absolute) error over training–patterns: 0.008861

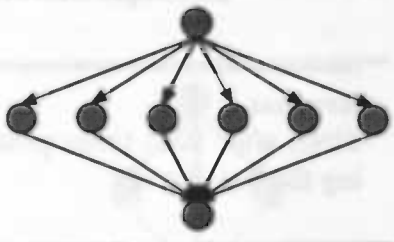
average (absolute) error over test–patterns: 0.012462



Remarks: *As expected, the parallel method with one parallel computation also converges correctly. The number of cycles is depended on the random initialization.*

C) Network information:

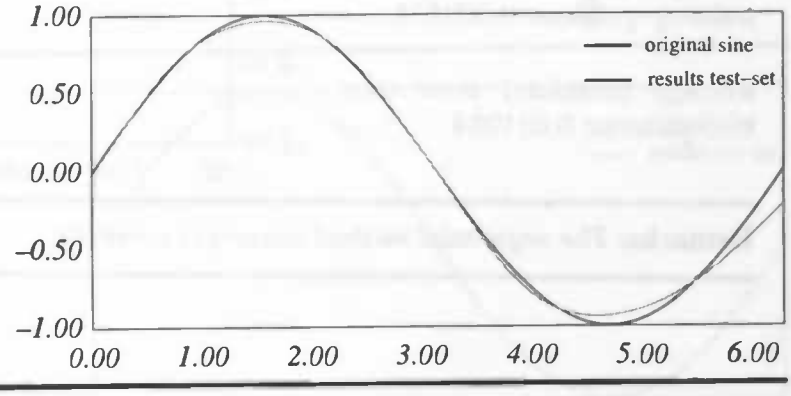
learning sine
uses a random initialized 1 6 1 network
uses random input
uses parallel method
with 2 parallel computation(s)



Results:
SINE after 725 times presenting the training set:

average (absolute) error over training–patterns: 0.009809

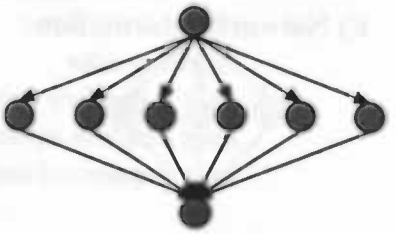
average (absolute) error over test–patterns: 0.014324



Remarks: *Introducing two parallel computations doesn't effect the convergence of the learning of the sine function.*

D) Network information:

learning sine
uses a random initialized 1 6 1 network
uses random input
uses parallel method
with 4 parallel computation(s)

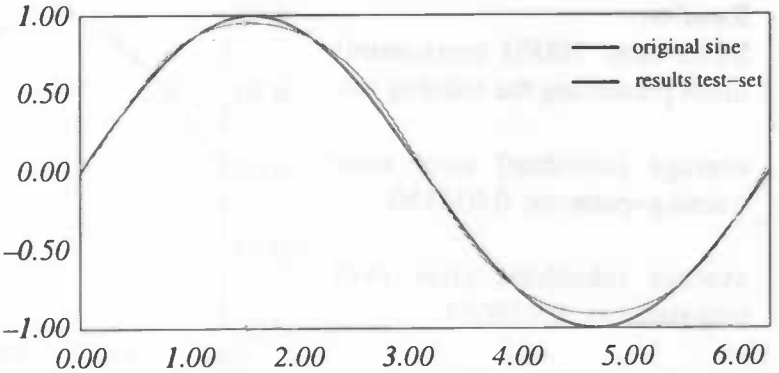


Results:

SINE after 4450 times presenting the training set:

average (absolute) error over training–patterns: 0.009984

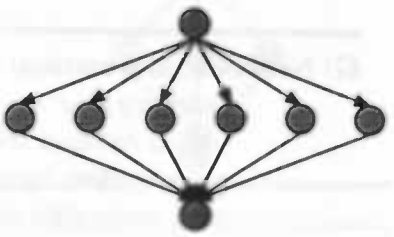
average (absolute) error over test–patterns: 0.013438



Remarks: *Even when we introduce four parallel computations the network still learns the sine function. We notice that the network needs more presentations of the training set, but is still able to learn the sine function correctly.*

E) Network information:

learning sine
uses a random initialized 1 6 1 network
uses random input
uses parallel method
with 5 parallel computation(s)

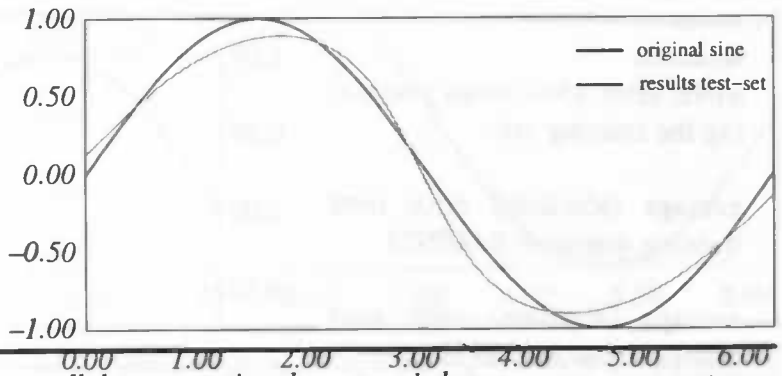


Results:

SINE after 20000 (maximum) times presenting the training set:

average (absolute) error over training–patterns: 0.030174

average (absolute) error over test–patterns: 0.036561



Remarks: *When we introduce five parallel computation the network does not converge, not even after 20000 times presenting the training–set.*

F) Network information:

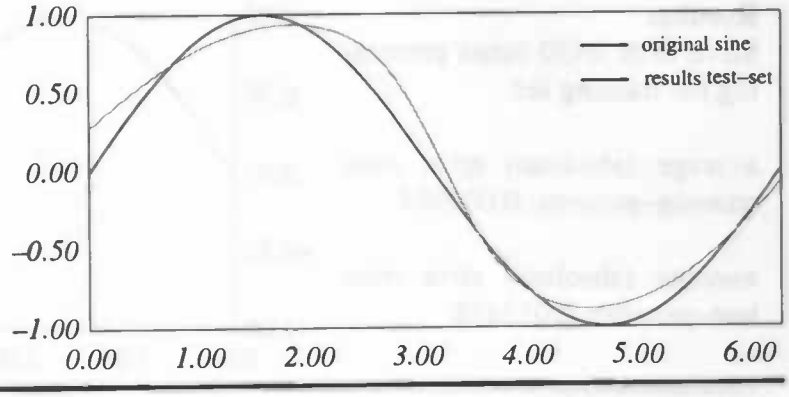
learning sine
uses a random initialized 1 2 1 network
uses random input
uses sequential method

Results:

SINE after 20000 (maximum) times presenting the training set:

average (absolute) error over training–patterns: 0.033150

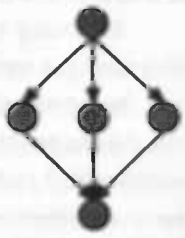
average (absolute) error over test–patterns: 0.039684



Remarks: *Learning the sine function with a 1–2–1 network, causes difficulties. Even after 20000 presentations of the training set, the error is still much higher in comparison to a correctly trained 1–6–1 network.*

G) Network information:

learning sine
uses a random initialized 1 3 1 network
uses random input
uses sequential method

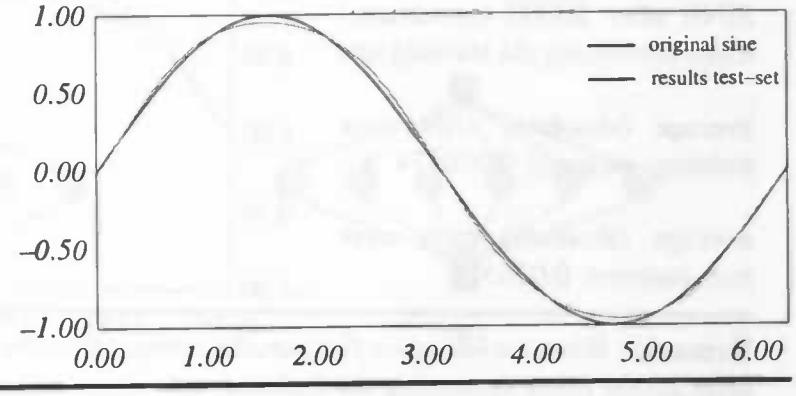


Results:

SINE after 4500 times presenting the training set:

average (absolute) error over training–patterns: 0.009951

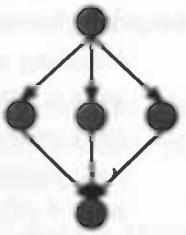
average (absolute) error over test–patterns: 0.009952



Remarks: *The sine function can be learned by using a 1–3–1 network. It cost more cycles than with a 1–6–1 network but it can be done. Notice the resemblance with experiment D.*

H) Network information:

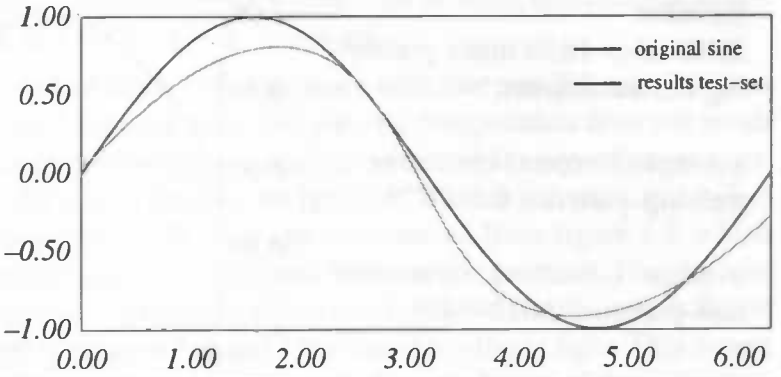
learning sine
uses a random initialized 1 3 1 network
uses random input
uses parallel method
with 2 parallel computation(s)



Results:
SINE after 20000 times presenting the training set:

average (absolute) error over training–patterns: 0.038877

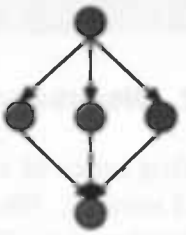
average (absolute) error over test–patterns: 0.045571



Remarks: *If we introduce two parallel computations the network does not converge.*

I) Network information:

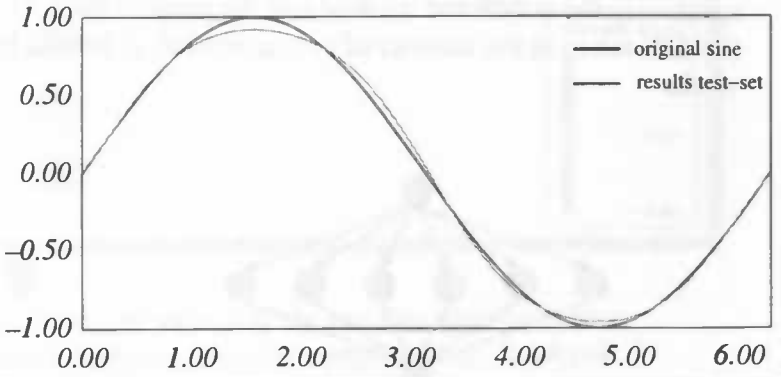
learning sine
uses a random initialized 1 3 1 network
uses ordered input
uses parallel method
with 2 parallel computation(s)



Results:
SINE after 2575 times presenting the training set:

average (absolute) error over training–patterns: 0.009995

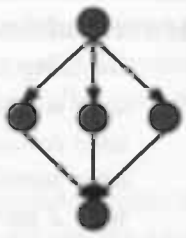
average (absolute) error over test–patterns: 0.012020



Remarks: *If we introduce two parallel computations but present the input ordered, the network does converge.*

J) Network information:

learning sine
uses a random initialized 1 3 1 network
uses ordered input
uses parallel method
with 4 parallel computation(s)

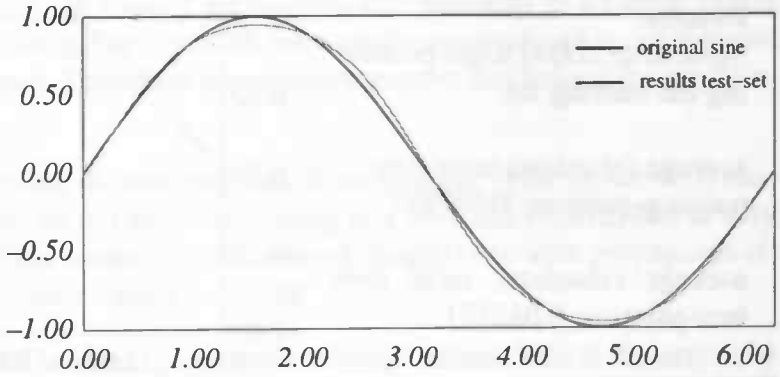


Results:

SINE after 1850 times presenting the training set:

average (absolute) error over training–patterns: 0.009979

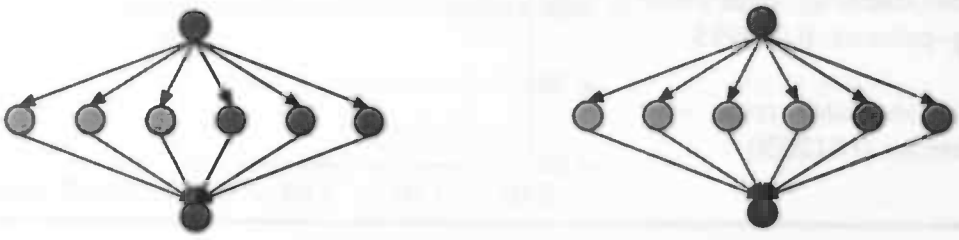
average (absolute) error over test–patterns: 0.012494



Remarks: *If we use ordered input and four parallel computations the network does still converge, even faster than with two parallel computations.*

3.5.2 Discussion

The first series of experiments we have done with the sine function, were experiments with a 1–6–1 network. We choose this size because we knew on forehand that this network could learn the sine function. This was confirmed by the results of the first two experiments (A and B), respectively the sequential method and the parallel method with one parallel computation. Again the difference in the number of cycles needed, is caused by the random initialization of the network.



(a) **(b)**

Figure 3.7: *The effect of introducing respectively* **(a)** *four and* **(b)** *five parallel computations. The network has learned to ignore the lighter colored synapses.*

If we introduce parallelism in the 1–6–1 network, the network still converges (experiments C and D). If we increase the number of parallel computations, the number of cycles needed increases too.

Only when we introduce five parallel computations (experiment E) the network can't learn the sine function correctly. Even after 20000 cycles, which was the maximum number of cycles used, the network still has not learned a proper approximation of the function.

When this occurred we considered some smaller networks (experiments F and G), respectively a 1–2–1 network and a 1–3–1 network. From these experiments can be seen, that a sine function can be learned by a 1–3–1 network, but not by a 1–2–1 network.

This is probably the explanation for the fact that a 1–6–1 network with four parallel computations results in a trained network and a 1–6–1 network with five parallel computation does not result in a trained network. What happens is the following: the neurons in the hidden layer that are computed in parallel with the neuron in the output neuron, are ignored. What is left over, is a 1–3–1 and a 1–2–1 network respectively (see figure 3.7). This can be observed from figure 3.8, which is an observation plot made during training of the weight and biases of the network. The plot was made while a 1–6–1 network with four parallel computations was being trained. From this figure we can see that the weights connected to neurons 5, 6 and 7 are drawn relatively light. This means that these neurons hardly contribute to the result of the network. Also the biases of these neurons have a value near zero. These neurons (5, 6 and 7) are the ones that are computed in parallel with the output neuron.

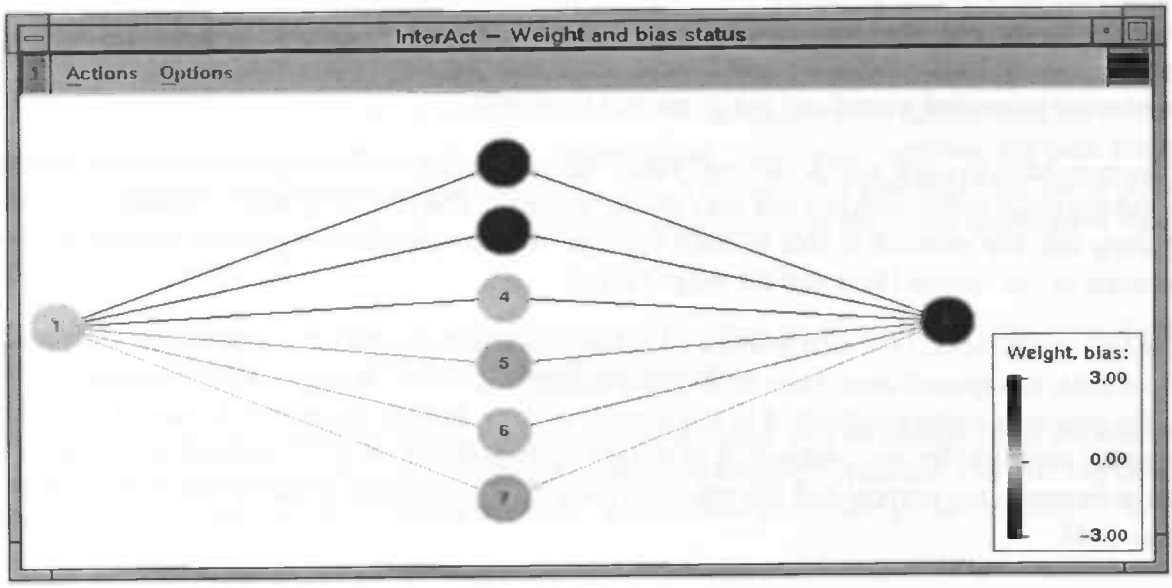


Figure 3.8: *This figure shows the values of the weights and the biases of a typical 1–6–1 network with four parallel computations. Dark colors indicate very high or very low values (much activity), light colors indicate values around zero (less activity).*

Notice that the bias level of neuron 4 is rather low too. The difference with this neuron is that it's incoming and outgoing synapses do have relatively high values and thus do contribute to the output of the network.

From our experiments can be seen that a 1–3–1 network is the minimal network with which the sine function can be learned. For this reason we considered minimal network only from then on. It's unnecessary to look at larger networks, because the redundant neurons will (almost) be ignored and true parallelism does not occur. It is also unnecessary to look at smaller networks, because these networks don't converge at all (most of the time).

From experiment H can be seen that a 1–3–1 network with two parallel computations does not converge. In fact we deal here with the same problem as in the previous example, the exclusive–or. The reason why the network does not learn if we perform computations in different layers in parallel was that the 'old' values we had to cope with were totally uncorrelated in comparison to the values that should be propagated. This result in propagating errors that belong to completely uncorrelated patterns.

This problem also occurs when learning the sine function. In experiment H the inputs were randomly chosen from the train–set. So an old error can belong to a previous pattern, that is very different from the current pattern. This means that the network should cope with propagated errors that differ too much. This obviously causes problems.

Despite this problem the network did learn to ignore these "wrong" values. This is illustrated by figure 3.7 and 3.8. We have seen that the network almost sets the values of the weight and biases of the neurons, that are computed in parallel with the output neuron, to zero.

Clearly the network has problems when uncorrelated data are used when introducing parallelism over the layers. For this reason we have looked what happened if we presented correlated data to the network. In particular we presented the network the sine function again but now the patterns were presented sorted and not presented randomly.

If we now have to cope with a "wrong" value this value does not belong to the current value but to the previous value which is not very much "wrong". The results of these experiments (I and J) show that the network is able to learn the sine function, despite the parallel computations of neurons in the hidden layer and the output layer.

The last experiment (J), which learns a 1–3–1 network with 4 parallel computations and ordered input data, is a special case. Here all the propagated errors are "wrong". All these errors belong to the previous pattern instead of to the current pattern. In fact, the network learned internal to train the network the $sin(x)$ according to weight updates based on $x - \Delta$ instead on x, with Δ the range between the current and the previous pattern. The network actually learns the function $sin(x + \Delta)$.

3.6 Three Functions

In this third experiment we attempt to learn three functions with one neural network. If we present a x–value on the interval [–2, 2] we want to learn the functions $f_1(x) = x^2$, $f_2(x) = x^3$ and $f_3(x) = \sqrt{x + 2}$. So, in this example we have one input and three output neurons. These functions are drawn in figure 3.9.

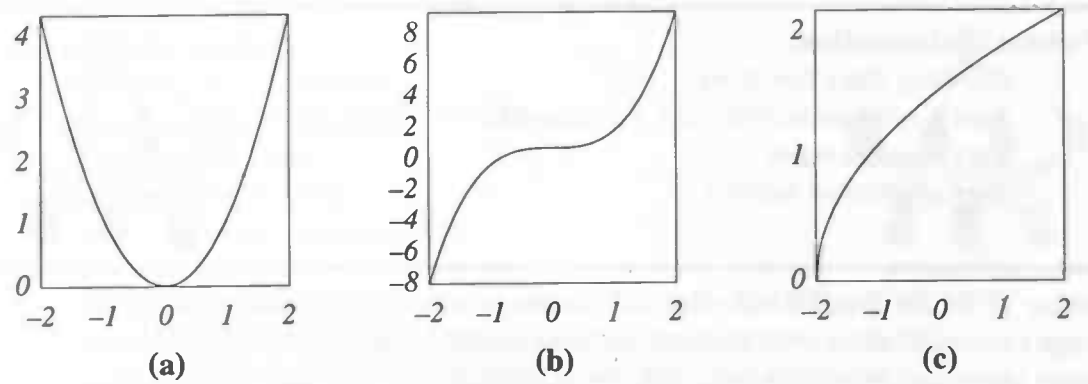


Figure 3.9: *The three functions we would like to learn:*
(a) $f_1(x) = x^2$
(b) $f_2(x) = x^3$
(c) $f_3(x) = \sqrt{x+2}$

3.6.1 Results

For learning these three functions we have created several feedforward neural networks with one input neuron, different numbers of hidden neurons and three output neurons, for each function one. We created 200 patterns on the interval [–2, 2]. Similar to the sine function, these patterns are divided in a train–set and a test–set. Here the presentation of the entire train–set is also denoted with one cycle.

For each experiment we have done with the three function problem, three plots (one for each function) will be included with the results of the test–set.

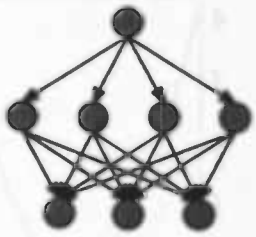
If the train–set is presented 25 times the error of the network will be compared to the minimum error. If the error drops below this minimum the trainings process is ended. The process can only be stopped if a multiple of 25 times the train–set has been presented to the network.

General information:

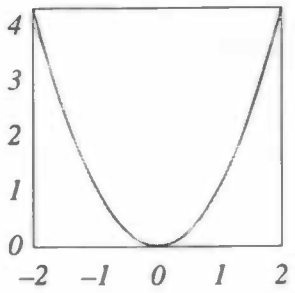
learn rate = 0.9
momentum term = 0.2
maximum number of trainings cycles = 20000
minimum absolute error = 0.005
number of different data patterns = 200
percentage of data used for training = 30%

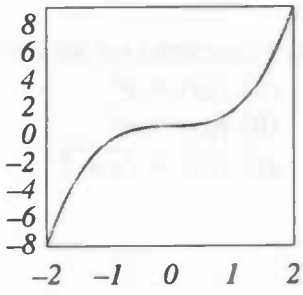
A) Network information:

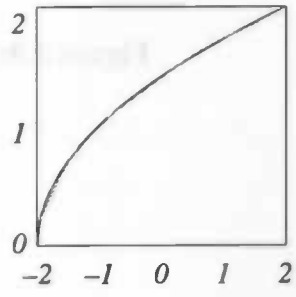
learning three functions
uses a random initialized 1 4 3 network
uses random input
uses sequential method



Results: THREEFUNCTIONS after 1850 times presenting the training set:
average (absolute) error over training–patterns: 0.004708 0.004532 0.004631
average (absolute) error over test–patterns: 0.005926 0.004767 0.004822



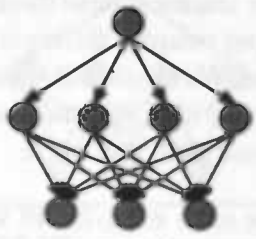




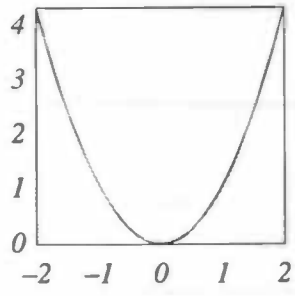
Remarks: *The minimal network that converges correctly, turned out to be a 1–4–3 network.*

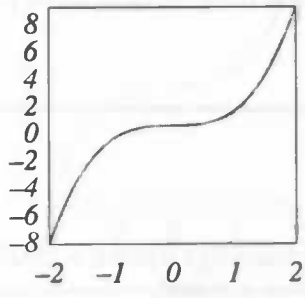
B) Network information:

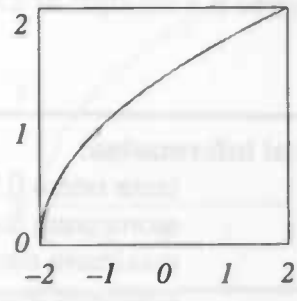
learning three functions
uses a random initialized 1 4 3 network
uses random input
uses parallel method
with 1 parallel computation(s)



Results: THREEFUNCTIONS after 1575 times presenting the training set:
average (absolute) error over training–patterns: 0.004927 0.004287 0.002518
average (absolute) error over test–patterns: 0.004381 0.003932 0.003431



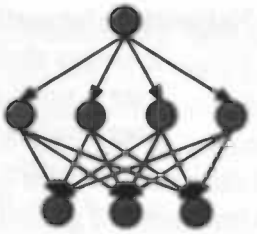




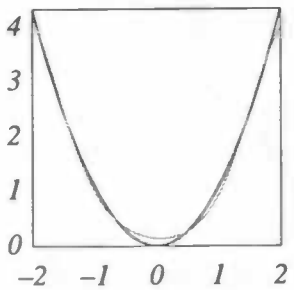
Remarks: *The network also converges when one parallel computation is introduced.*

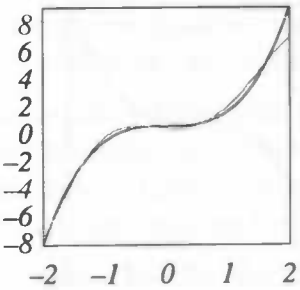
C) Network information:

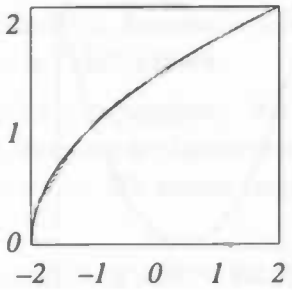
learning three functions
uses a random initialized 1 4 3 network
uses random input
uses parallel method
with 2 parallel computation(s)



Results: THREEFUNCTIONS after 20000 times presenting the training set:
average (absolute) error over training–patterns: 0.014420 0.012340 0.006963
average (absolute) error over test–patterns: 0.016548 0.014082 0.005106



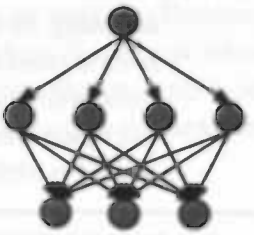




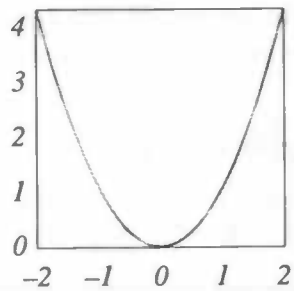
Remarks: *If two parallel computations are introduced, the network doesn't converge correctly. Although it performs better than the non–converging networks in sine experiments.*

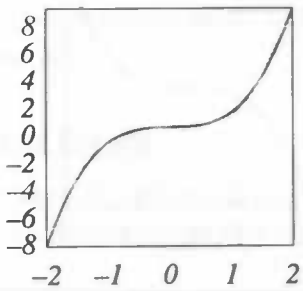
D) Network information:

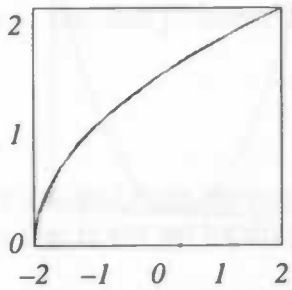
learning three functions
uses a random initialized 1 4 3 network
uses ordered input
uses parallel method
with 2 parallel computation(s)



Results: THREEFUNCTIONS after 1425 times presenting the training set:
average (absolute) error over training–patterns: 0.004921 0.004757 0.004617
average (absolute) error over test–patterns: 0.004366 0.004546 0.003832



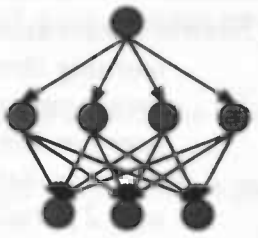




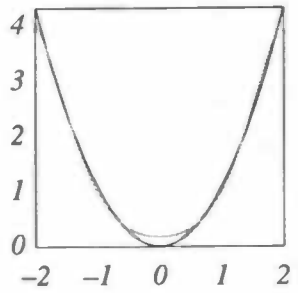
Remarks: *But if the input is presented ordered the network does converge.*

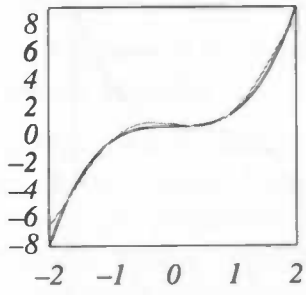
E) Network information:

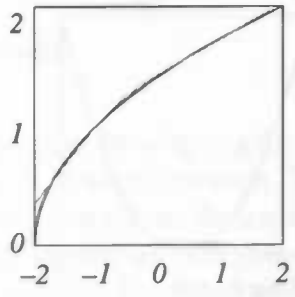
learning three functions
uses a random initialized 1 4 3 network
uses random input
uses parallel method
with 4 parallel computation(s)



Results: THREEFUNCTIONS after 20000 times presenting the training set:
average (absolute) error over training–patterns: 0.015614 0.011402 0.005900
average (absolute) error over test–patterns: 0.016416 0.012256 0.008710



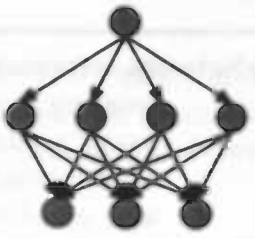




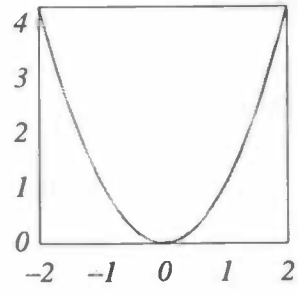
Remarks: *When 4 parallel computation are introduced, the network does not converge.*

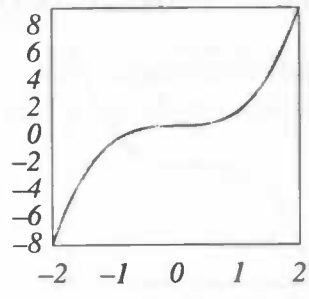
F) Network information:

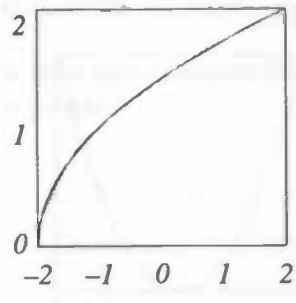
learning three functions
uses a random initialized 1 4 3 network
uses ordered input
uses parallel method
with 4 parallel computation(s)



Results: THREEFUNCTIONS after 1800 times presenting the training set:
average (absolute) error over training–patterns: 0.004975 0.004546 0.003834
average (absolute) error over test–patterns: 0.004536 0.004668 0.003685







Remarks: *When the input is presented in an ordered way, the network does learn the three function problem, even with four parallel computations.*

3.6.2 Discussion

The experiments presented above are a few interesting examples of experiments we have done. We started doing experiments with larger networks than the ones presented here (i.e. with 1–6–3 and 1–5–3 networks). As we have seen with the sine function and the exclusive–or problem parallelism in different layers can be introduced in "too" large networks. We saw that the redundant neurons are (almost completely) ignored. For this reason we started the experiments with three function problem with finding the "minimal" network[4] that converged using the standard error back–propagation algorithm.

This "minimal" network turned out to be a 1–4–3 network. Experiments with this network converge correctly as long as no parallelism is introduced (experiments A and B). If we introduce two parallel computations the network can't learn the three function problem correctly (experiment C). This is kind of the same result as we found with the sine function. As soon as we perform computations with "old" values, the network can't cope with these "old" values.

If we present the input patterns in an ordered way with two parallel computations, the network can learn the three function problem correctly (see experiment D). Because we have three outputs in this case, we didn't do experiments with three parallel computations. We would restrict us to the layers then, and we know this will converge on forehand.

We did some experiments with four parallel computation. These experiments show the same kind of results as the previous. With random input patterns the network does not converge, but with ordered input patterns the network does learn to approximate the functions (experiments E and F).

All the results of the experiments done to (attempt to) learn the three function problem, resemble more or less the results of learning the sine function. There is however an important difference between the three function problem and the sine function: the number of outputs. Because the three function problem has more outputs we can get round the difficulties that occur when we do computations with "old" values. Until now we have discarded any error that was not updated completely and used the previous error associated with the neuron considered. In the next section we will present an algorithm that does use these partially updated errors.

3.7 Another Parallel Version

From the results of the previous section, the wish for using partially updated errors arose. In this section we will discuss the possibilities for achieving this and present a new parallel algorithm. With this algorithm we have done some experiments with the three function problem. The most relevant of them will be presented here.

3.7.1 Using Partially Updated Errors

As mentioned in the footnote on page 31, we could use partially updated error instead of "old" errors. In fact there are several possibilities to deal with an error that is not yet totally updated:

4. Note that the "minimal" network perhaps is not the absolute minimal network. With "minimal" we mean: the minimal number of hidden neurons for which the network converges relatively easy. That is, if the network does not converge after the maximum number of cycles, it is not good enough. Furthermore some optimizations could be made by changing the learn–rate or the momentum term.

- continue with this (possibly) partially updated error.
  disadvantage: if all the output neurons are computed in parallel with a hidden neuron, the error of this neuron will always be zero.
- recompute the error signal of each neuron by inspecting the internal error of the neurons fed by the neuron we wish to compute.
  disadvantage: all contributions to the error have to be computed again, while a part of the error already was determined.
- store which outgoing neurons have contributed to the error of the neuron that is considered and which neurons haven't contributed yet. Add the contribution of these last neurons to the error of the neuron considered.
  disadvantage: costs a lot of administration.
- just take the old error (which is done in the parallel method).
  disadvantage: we don't use all available information.

In fact possibility 3 is the best of the four, but it's hard to implement this by adapting the original error back–propagation algorithm. What is worse, it involves a lot of administration. What we want is an implementation of the error back–propagation algorithm, that allows the use the "old" and new errors together, without administrative overhead. In this section we will give an algorithm to achieve this, we will present some experiments done with this algorithm and we will discuss the results of these experiments. For the experiments we take the three functions problem. We need an example with more output neurons (or more hidden layers) to test this algorithm.

3.7.2 The Algorithm

What we want is an algorithm that does not take an old error if the error is not totally updated, but rather holds the part that is already updated and adds the contribution of the errors that are not yet considered. A way to do this without causing a lot of administration is the following:

- compute the errors of the output neurons,
- consider the hidden neurons,
- update the error of these neurons (by adding the contribution of the neurons that are fed by these neurons, this possibly will be a mixture of "old" and new values),
- update the outgoing synapses.

The main difference with the "original" algorithm is that we now consider a neuron and its outputs, while the "original" algorithm is based on a neuron and its inputs.

Backward pass:

```
/* get last neuron */
j = last neuron
/* main loop: repeat until input layer is reached */
while j >= first neuron do
  /* pseudo parallel loop */
  for nrpar = 1 to N do
    if j = output neuron then
```

```
        /* compute the error of neuron j */
        e_j(n) = d_j(n) - y_j(n)
      else
        /* compute the error of neuron j */
        e_j(n) = 0
        for k = first neuron fed by j to last neuron fed by j do
          e_j(n) = e_j(n) + δ_k(n) * w_kj(n)
          /*compute the new weight and store the new weight in a
            temporary variable */
          tempw_kj = w_kj(n) + α * [w_kj(n) - w_kj(n - 1)] + η * δ_k * y_j(n)
        od
      fi
      /* store the error signal δ in a temporary variable */
      tempδ_j = y_j(n) * [1 - y_j(n)] * e_j(n)
      /*compute new bias and store this in a temporary
        variable */
      tempw_j0 = w_j0(n) + α * [w_j0(n) - w_j0(n - 1)] + η * tempδ_j
      /* get previous neuron */
      j = previous neuron
      /*if we have reached the input layer "break" from the
        "pseudo parallel loop" */
      if j = input neuron then
        break
      fi
    od /* pseudo parallel loop */
    /*now "nrpar" denotes the number of neurons that are updated
      pseudo-parallel, these "nrpar" neurons should be really
      updated, so copy the temporary variables to the real
      variables */
    j = j + nrpar
    for nr = 1 to nrpar do
      δ_j(n) = tempδ_j
      for k = first neuron fed by j to last neuron fed by j do
        w_kj(n + 1) = tempw_kj
      od
      w_j0(n) = tempw_j0
      j = previous neuron
    od
od /* main loop */
```

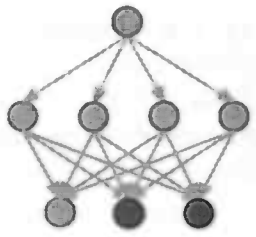
We will call this parallel version the "special" parallel version. From now on, the parallel version based on the original error back–propagation algorithm will be called the "original" parallel version.

As we have discussed above, we wanted a parallel version that uses a combination of old and new errors, instead of totally old errors, but not at the cost of a lot of administration. If we look at the original parallel version we need to store the additional variables `olderror` and `errorused` for all hidden neurons.

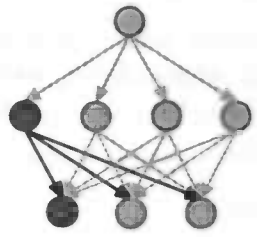
The special parallel version needs to consider the input neurons (to update their outgoing synapses). For this we need to store the error of the input neurons. This involves some multiplications. This can be avoided by a simple test whether the neuron is an input neuron or not. Concluding, the special method does need less administration. Furthermore it doesn't take more time to compute the special parallel method.

3.7.3 Results

We have done some experiments with the three function problem as described in section 3.6 to test the "special" parallel version. We have chosen for this experiment because it has more than one output neuron. If a network has only one output neuron we don't get the combination of old and new errors. The reason for the "special" parallel version was to achieve this. What the algorithm actually does is visualized in figure 3.10.



First the errors of two output neurons are updated in parallel.



Then the error of one hidden and one output neuron is updated in parallel and the outgoing weights of the hidden neuron are updated. The error of the hidden neuron is based on the updated errors of the two first output neurons and the old error of the last output neuron.

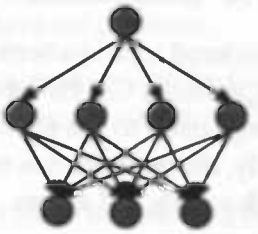
Figure 3.10: *A 1–4–3 network with two parallel computations. Here the error of one of the hidden neurons is based on a mixture of old and new errors.*

General information:

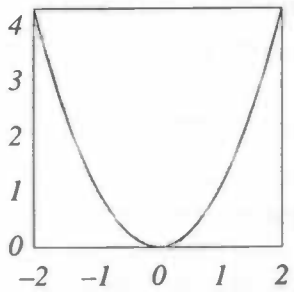
learn rate = 0.9
momentum term = 0.2
maximum number of trainings cycles = 20000
minimum absolute error = 0.005
number of different data patterns = 200
percentage of data used for training = 30%

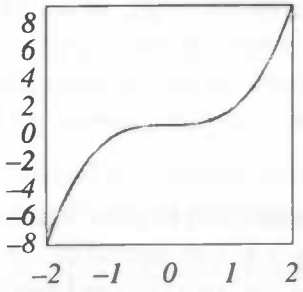
G) Network information:

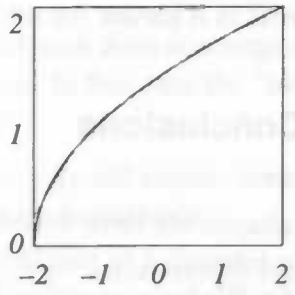
learning three functions
uses a random initialized 1 4 3 network
uses ordered input
uses special parallel method
with 2 parallel computation(s)



Results: THREEFUNCTIONS after 5050 times presenting the training set:
average (absolute) error over training–patterns: 0.004993 0.004143 0.004310
average (absolute) error over test–patterns: 0.005642 0.004645 0.005588



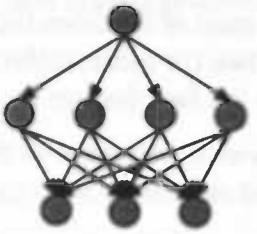




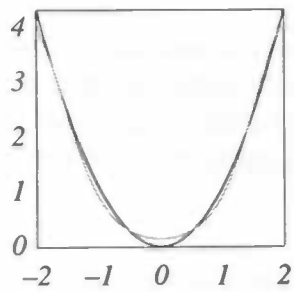
Remarks: *If we use a combination of old and new errors still the network converges when we present the input patterns ordered.*

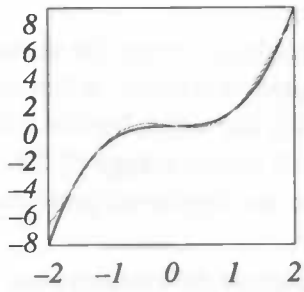
H) Network information:

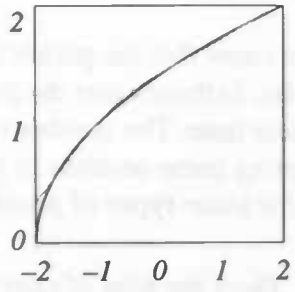
learning three functions
uses a random initialized 1 4 3 network
uses random input
uses special parallel method
with 2 parallel computation(s)



Results: THREEFUNCTIONS after 20000 times presenting the training set:
average (absolute) error over training–patterns: 0.017413 0.010831 0.006543
average (absolute) error over test–patterns: 0.019179 0.011037 0.007787







Remarks: *If we present the input patterns random the network does not converge.*

3.7.4 Discussion

If ordered input patterns are used the network still converges with the "special" parallel method (experiment G). However, on the average the number of cycles needed to converge is larger in comparison to the results of experiment D, where we used the original parallel method. Apparently, the network has more difficulties in dealing with a combination of old and new errors than with totally old errors.

The second experiment done with the special method, training a 1–4–3 network with 2 parallel computations and random input (experiment H), does not converge. Even after 20000 cycles the errors are still relatively high. The results are comparable with the results of experiment C. Obviously, using a combination of old and new values does not help the network to converge. It rather makes it harder for the network to converge, as we have seen with experiment G.

3.8 Conclusions

In this chapter we have introduced parallelism in error back–propagation learning. We were especially interested in parallelism that is not restricted to the layer–wise composition of neural networks. We have considered two parallel versions and discussed the results of the experiments done. Before we started the experiments we formulated a few questions. We will give answers to these question in this section.

Does the parallel method converge?

In most of the cases the parallel method does not converge. This means that if we use parallelism in two (or more) different layers the network does not learn the desired behavior. This is caused by the fact that we have to deal with old errors then.

As we have seen with the special parallel version, the network also can't deal with "wrong" errors that are based on a combination of old and new errors.

However, if we present the input patterns in an ordered way the network does converge. This is only possible if the input data are correlated (i.e. sine function and three functions problem).

Is the parallel method better than the sequential one?

For the cases that the parallel versions do converge the network needs almost the same number of cycles. In these cases the parallel method is better, in the sense that learning the entire network costs less time. The number of cycles is the same, but the time to compute a cycle is reduced by computing some neurons in parallel. A disadvantage of the parallel methods is that they do not work for some types of problems (i.e. exclusive–or problem).

Does the kind of experiment infects the convergence of the parallel method?

The kind of experiment does infect the convergence of the parallel methods. We have seen that the parallel methods do converge when the input is presented ordered and do not converge when

the input is presented randomly. Not for all problems input can be presented in an ordered way, simply because there is no real order. An example of this is the exclusive–or problem.

More general, problems with correlated input, such as function approximation, can be trained with the parallel methods. Problems with uncorrelated input can not be learned with the parallel methods.

Now we have answered these questions, we will draw some conclusions about introducing parallelism in error back–propagation learning:

- A network does not always converge when parallelism in different layers is introduced. This is caused by the "old" errors. The network can't deal with these wrong errors in all situations.
- However, if we present the input in an ordered way the network does converge correctly while performing parallel computations in different layers. In this case the "old" errors are less devastating, and the network can deal with them.
- Using a combination of old and new errors, instead of totally old errors, does also not result in a trained network when input patterns are presented randomly.
- In some situations, the network does learn to exclude "wrong" values and in fact ignores the neurons that have to deal with this "wrong" values. This is done by setting the weights on the synapses of those neurons to very small values, so that those neurons hardly contribute to the output of the neuron. This occurred when too large networks were considered.

Summarizing, the use of parallelism that is not restricted to the layer–wise composition of the network, may be beneficial for some classes of problems such as function approximation.

Chapter 4

Using Less Accuracy in Error Back–Propagation

4.1 Why Less Accuracy?

In the previous chapter two algorithms that introduce parallelism in error back–propagation learning have been presented. The major strong–point of them is that they introduce parallelism that is not restricted to the layers in the network.

Obviously, if we want to introduce parallelism, we need something to realize this with. An ideal situation would be, if we could use multiple processors that perform the necessary calculations in parallel. However, if we are restricted to a hardware realization with a single processor–unit we should cope with the possibilities of this single unit and use it in a way that is most profitable.

Parallelism may be forced on a single processor architecture at the cost of accuracy. By not using the full word width available, several computations may be executed simultaneously. For example, if we have a 64–bits processor at our disposal, we could divide this in two parts of 32 bits and perform one computation in each part. This means that we could perform two computations in parallel. These computation will then be made with an accuracy of 32 bits instead of the maximal 64 bits. So we could perform two computations in parallel with half of the maximal accuracy possible. In the same way we could perform four computations in parallel with an accuracy of 16 bits or eight parallel computations with an accuracy of 8 bits, etc..

Obviously we are not considering a standard processor here, but one that has been provided with extensions to select a (variable) word width.

The key question that has to be answered before we can realize this, is:

> *Is it possible to use less accuracy for the computations in error back–propagation learning?*

If we perform computations with less accuracy, we can reduce the learning time, even if we don't perform computations in parallel. This is because less accuracy implies less iterations to obtain the result of a computation. However, it should not cost much more cycles to converge, because then we don't gain any profit. In this chapter we also answer the following question:

> *Can we adapt the accuracy for computations in error back–propagation learning in a flexible manner?*

In other words, we would like to be able to change the accuracy on the fly. We will start with a low number of bits and add a bit accuracy whenever this is needed.

4.2 Floating Point Representation

A real number can be represented in a fixed point or floating point representation. In this work we use floating point representation. Analysis of using less accuracy in error back–propagation with fixed point representations is treated in [8]. The results of this work were that a minimal number of 13 to 16 bits were needed to obtain an acceptable trained network.

A floating point can be represented as a word that is partitioned in two parts: a mantissa and an exponent. The mantissa is is represented as a fixed point value with the binary point to the right of the leftmost bit. The exponent is used as a scale factor for the mantissa.

If the mantissa consist of M bits and the exponent consist of E bits, then the following notation illustrates the meaning of the representation.

$$m = m_1.m_2m_3 \cdots m_{M-1}m_M \tag{4.1}$$

with m_i the i–th bit of the mantissa and . the binary point.

$$e = e_1e_2 \cdots e_{E-1}e_E \tag{4.2}$$

with e_i the i–th bit of the exponent. The real number denoted in this way can be computed as:

$$\begin{aligned} value &= m \times 2^e \\ &= (m_1 2^0 + m_2 2^{-1} + m_3 2^{-2} + \cdots + m_{M-1} 2^{-(M-2)} + m_M 2^{-(M-1)}) \times \\ &\quad (2^{(e_1 2^{(E-1)} + e_2 2^{(E-2)} + \cdots + e_{E-1} 2^1 + e_E 2^0)}) \end{aligned} \tag{4.3}$$

An example to illustrate this notation: Assume $M = 4$ and $E = 4$, and we represent a floating point as m:e then,

$$\begin{aligned} 0101 : 0000 &= (0.5 + 0.125) \times (2^0) = 0.625 \\ 0101 : 0001 &= (0.5 + 0.125) \times (2^1) = 1.25 \\ 0101 : 0101 &= (0.5 + 0.125) \times (2^5) = 20 \end{aligned} \tag{4.4}$$

The disadvantage of this representation is that it is not unique. For example 20 can either be represented as 0101:0101 or as 1010:0100 or as .... This problem can be solved by assuming that the first bit of the mantissa is always one. In this way we also gain a bit accuracy. To return to the example, 20 should be represented as (1)0100:0100. The 1 between brackets is the bit that is always one. As we know this on forehand we don't have to store this bit.

For the representations of floating points, an IEEE standard exists. This standard, the IEEE–754 standard (see [3]), also uses a first bit that is always set to one for the mantissa. Furthermore a 8–bits exponent and a 23–bits mantissa are used. Finally one bit is used to indicate the sign. From now on we will use this standard, if we talk about floating point numbers.

Note, that although we have a mantissa of 23 bits, we actually have an accuracy of 24 bits because of the leading 1.

4.3 Computations with Less Accuracy

As we have seen in chapter 3 (page 26), the backward pass of the error back propagation algorithm uses several computations. It needs additions, subtractions and multiplications. In compar-

ison to addition and subtraction, multiplication is an expensive operation. For this reason we will perform multiplications with less accuracy (in parallel).

Floating points can be multiplied in the following way:

- add the exponents,
- multiply the mantissas,
- perform the exclusive or on the signs,
- normalize the result.

The multiplication of the mantissas can be implemented as follows. Assume we would like to multiply two mantissas A and B. These can be written as:

$$A = 2^j + A_r \tag{4.5}$$

$$B = 2^k + B_r \tag{4.6}$$

with j the highest one bit in A and k the highest one bit in B. Now $A \cdot B$ can be rewritten as:

$$\begin{aligned} A \cdot B &= (2^j + A_r) \cdot (2^k + B_r) \\ &= 2^j \cdot 2^k + 2^k \cdot A_r + B_r \cdot (2^j + A^r) \\ &= 2^j \cdot 2^k + 2^k \cdot A_r + A \cdot B_r \end{aligned} \tag{4.7}$$

In fact, equation (4.7) is just a reformulation of multiplication in the well–known Logarithmic Number System [9]. In essence, equation (4.7) expresses multiplication in a recursive way. This recursion is with respect to the number B as can be seen from the last term in the equation ($+ A \cdot B_r$). Alternatively, we could express multiplication recursively with respect to both A and B:

$$\begin{aligned} A \cdot B &= (2^j + A_r) \cdot (2^k + B_r) \\ &= 2^j \cdot 2^k + 2^j \cdot B_r + 2^k \cdot A_r + A_r \cdot B_r \end{aligned} \tag{4.8}$$

This equation is the basis for DIGILOG [14]. Here we will concentrate on equation (4.7) however, and compute multiplication in a recursive manner until $B_r = 0$. The number of times this computation is made is the number of iteration needed to obtain the result. If we introduce less accuracy, we will prematurely end the recursion. If the recursion is stopped prematurely the number of iterations is less then if the recursion is only stopped when B reaches zero. If the number of iterations is less we reduce the time for a multiplication and with this the time of the whole process.

The sign of the result is determined very easily: if the signs of both floating point numbers are the same the sign of the result will be positive and if they have opposite signs the sign of the result will be negative.

4.4 Using Less Accuracy

To realize parallel computations we will use less accuracy. To clarify this idea, we will first give an example of how this can be realized.

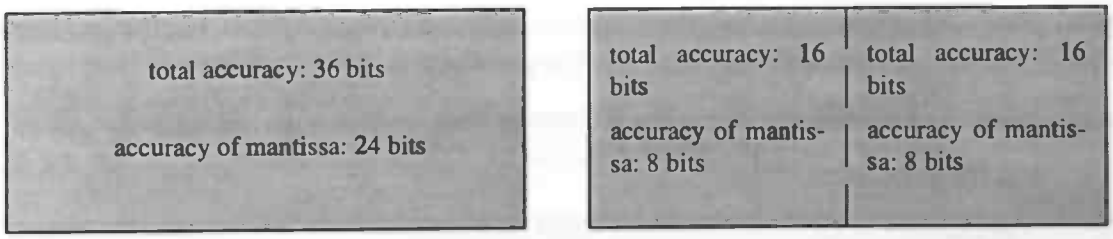


Figure 4.1: *We could perform one computation with an accuracy for the mantissa of 16 bits or two computations in parallel with a accuracy for the mantissa of 16 bits per computation.*

As mentioned in the previous section we need 32 bits to represent floating point numbers, 8 bits for the exponent, 23 bits for the mantissa and 1 bit for the sign. So we can perform one computation with an accuracy for the mantissa of 23 plus the leading one–bit (total: 24 bits). We could also perform two computations with an accuracy of 16 bits. This causes an accuracy of the mantissa of 8 bits ((desired accuracy) – (exponent) – (sign) + (leading one) = 16 – 8 – 1 + 1 = 8). See also figure 4.1.

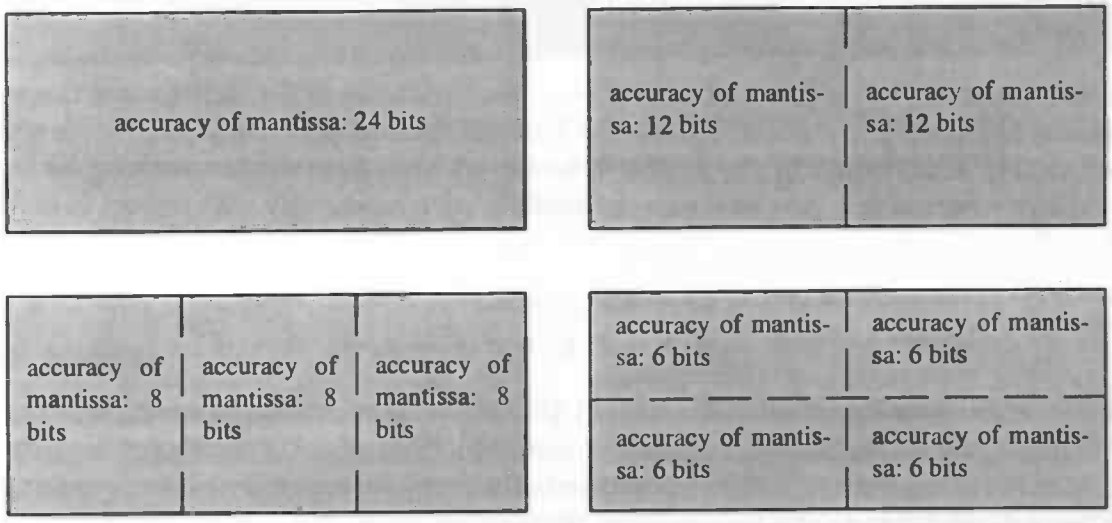


Figure 4.2: *We could perform one computation with an accuracy for the mantissa of 24 bits or two computations in parallel with a accuracy for the mantissa of 12 bits per computation, three with an accuracy of 8 bits and four with an accuracy of 6 bits.*

Another option would be if the exponent and the mantissa were considered completely separately. In this scheme we only have to worry about how to deal with mantissas off reduced accuracy. Then we could divide the mantissa in two parts of 12 bits or three parts of 8 bits or four parts of 6 bits (see figure 4.2).

Note that when we divide the mantissa in two parts, we actually divide 23 bits in two parts. This amounts to a part of 11 bits and a part of 12 bits. However both parts need a sign bit. The first

part already has a sign (according to the IEEE–standard). The other part has a bit extra (12 instead of 11), so this extra bit can be used as sign bit for the second part. In this way both parts satisfy the IEEE standard (apart from the number of bits), as they both have a sign bit and 11 bits for the mantissa. So we end up with two mantissas consisting of 11 bits. As both mantissas have a "hidden" leading one bit (that is not actually stored), we have two mantissas with an actual accuracy of 12 bits accuracy (see figure 4.3).

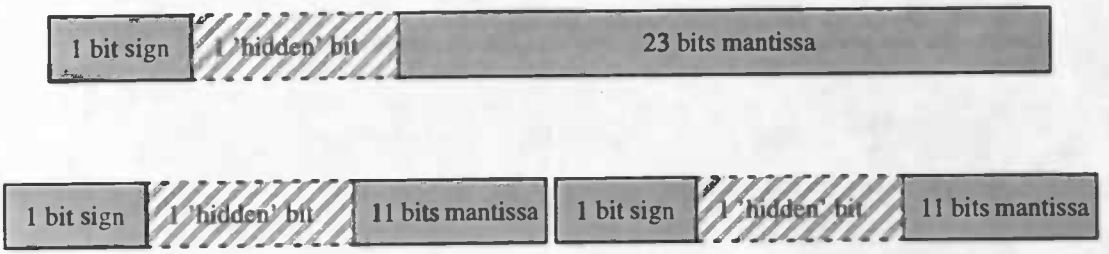


Figure 4.3: *According to the IEEE–standard a floating point number has a 1 bit sign, a 23 bits mantissa (and an 8 bits exponent). Furthermore, a leading one bit is assumed ('hidden' bit), so an accuracy of 24 bits for the mantissa is accomplished. If we divide this in two parts, both parts have a sign bit, a 'hidden' bit and an 11 bits mantissa (and an 8 bits exponent).*

In this work we do not concentrate on how a variable word width will be realized in hardware exactly. The previous suggestions were just two examples to get an idea of how the accuracy of the mantissa can be determined. Here, we just assume that there is some kind of hardware realization that can cope with changing accuracy for the mantissa, and can perform computations in parallel with these accuracies. We will concentrate on the the effects of the use of less accuracy or adaptive accuracy on the error back–propagation algorithm, rather than concentrate on the hardware realization.

The maximum accuracy we can use for a mantissa according to the IEEE standard is 24 bits. Experiments done with the following number of bits will be presented:

- 24 bits, so we can perform one parallel multiplication (so no parallelism)
- 12 bits, so we can perform two parallel multiplications
- 8 bits, so we can perform three parallel multiplications
- 6 bits, so we can perform four parallel multiplications

We have experienced that using an accuracy that is too small will cause a kind of chaotic behavior. This can be seen from figure 4.4 (left plot). In this figure the number of cycles is plot against the average absolute error. We see that the behavior in the first cycles is satisfying, but as soon as the error gets to small the network starts behaving in an oscillatory way. This experiment was done for the sine function using the sequential method with 1 bit accuracy. The same phenomenon occurred when 2 bits accuracy was used.

To get a comparison with a "normal" accuracy, we have made the same plot for an accuracy of 24 bits (see right plot in figure 4.4). Notice that the number of cycles ranges in the left figure from

0 to 20000 and from 0 to 900 for the right figure. When using an accuracy of 24 bits the learning process of the network was ended, at the moment the average absolute error dropped below 0.01.

Obviously, when we use an accuracy that is too small, the network does not converge. For this reason we present experiments here with an accuracy until 6 bits (and no less).

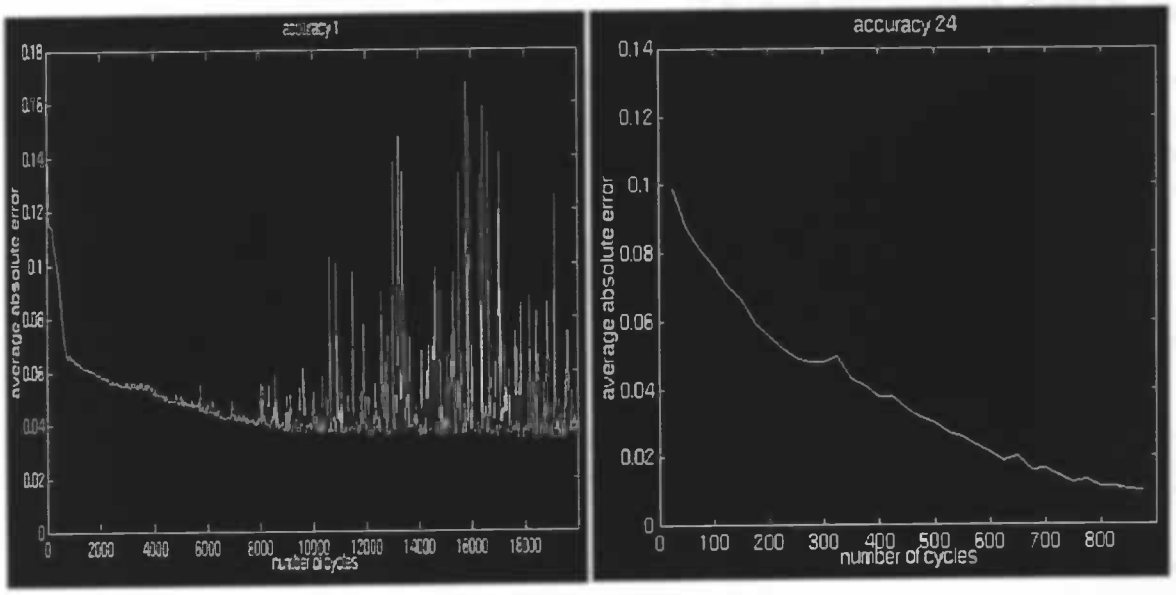


Figure 4.4: *The results obtained by using an accuracy of 1 bit and by using an accuracy of 24 bits in a network that is learning the sine function.*

4.4.1 Results

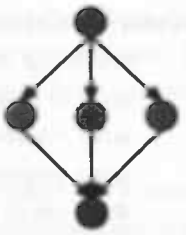
We will present the results obtained by learning the sine function (as described in section 3.5). Again we will use the minimal network for this problem to examine the results of introducing less accuracy in error back–propagation. All experiments are done with the sequential method. The difference in the presentation of the results is that now the number of iterations needed are added. This number of iterations is the total number of iterations needed for all multiplications. The number of multiplication will also be presented. Again, only the most relevant experiments are presented.

General information:

learn rate = 0.9
momentum term = 0.2
maximum number of trainings cycles = 20000
minimum absolute error = 0.01
number of different data patterns = 200
percentage of data used for training = 30%

A) Network information:

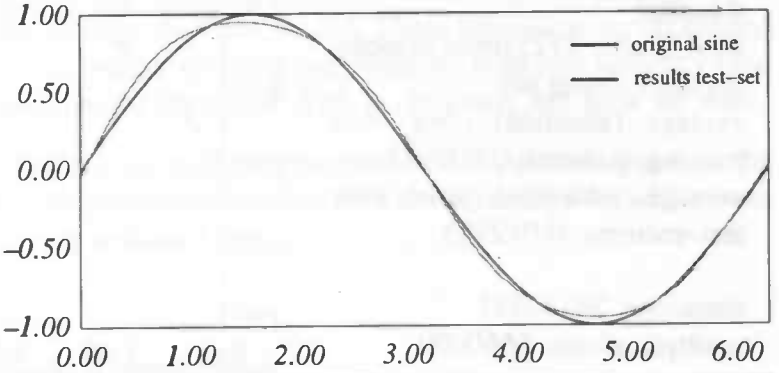
learning sine
uses a random initialized 1 3 1 network
uses random input
uses sequential method
uses an accuracy of 24 bits



Results:

SINE after 2325 times presenting the training set:
average (absolute) error over training–patterns: 0.009934
average (absolute) error over test–patterns: 0.015184

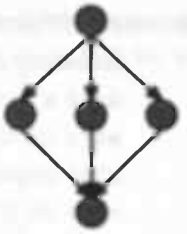
iterations: 59383809
multiplications: 5580000



Remarks: *The network converges correctly when an accuracy of 24 bits is used.*

B) Network information:

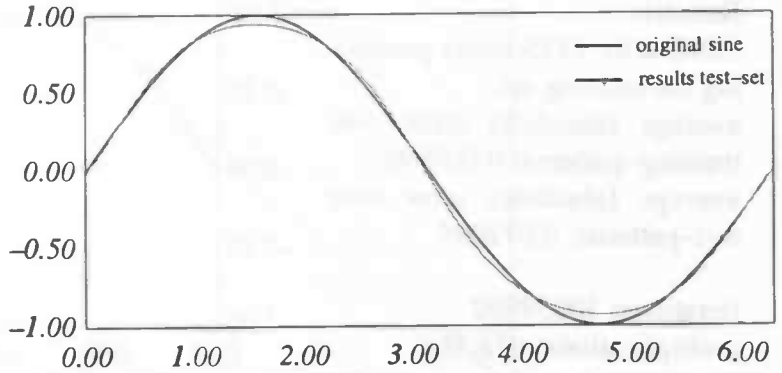
learning sine
uses a random initialized 1 3 1 network
uses random input
uses sequential method
uses an accuracy of 12 bits



Results:

SINUS after 1675 times presenting the training set:
average (absolute) error over training–patterns: 0.009991
average (absolute) error over test–patterns: 0.013022

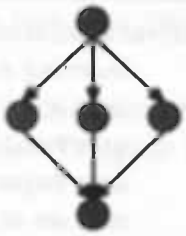
iterations: 28819796
multiplications: 4020000



Remarks: *Also with 12 bits accuracy the sine function can still be learned.*

C) Network information:

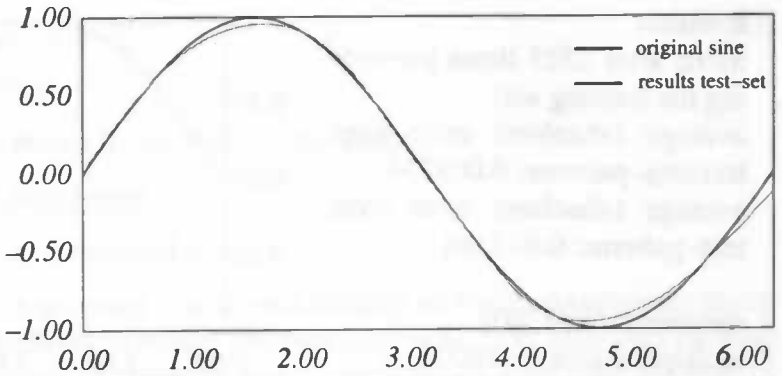
learning sine
uses a random initialized 1 3 1 network
uses random input
uses sequential method
uses an accuracy of 8 bits



Results:

SINUS after 2775 times presenting the training set:
average (absolute) error over training–patterns: 0.009983
average (absolute) error over test–patterns: 0.012985

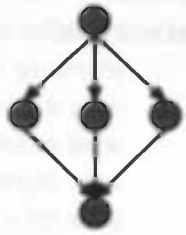
iterations: 36142431
multiplications: 6660000



Remarks: *With an accuracy of 8 bits the network does converge.*

D) Network information:

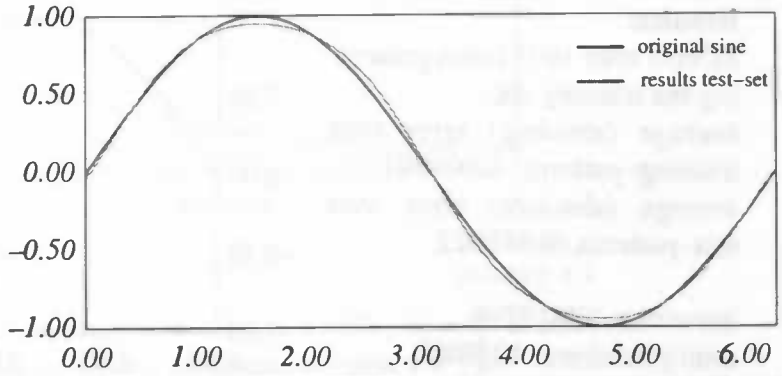
learning sine
uses a random initialized 1 3 1 network
uses random input
uses sequential method
uses an accuracy of 6 bits



Results:

SINE after 2825 times presenting the training set:
average (absolute) error over training–patterns: 0.009996
average (absolute) error over test–patterns: 0.012095

iterations: 30959880
multiplications: 6780000



Remarks: *Even when only 6 bits are used to perform a multiplication the network still converges.*

As we have seen, the sine function converges in all cases presented here. What is most interesting now is the number of iterations needed to achieve this convergence. This is however depended on the number of cycles used. This number of cycles differs if we repeat an experiment, because of the random initialization of the network and the changing input patterns. For this reason it's hard to say which accuracy has the best results.

What we can see is that the number of cycles needed to converge does not raise very much when we use less accuracy. However, if we use less accuracy we can perform computations in parallel. In other words we can introduce less accuracy without damaging the convergence properties and even without a substantial raise in the amount of cycles needed. These amount of cycles needed are very important. If it raises too much, we can perform computations with less accuracy (and thus with parallelism), but at the cost of extra learn time. This is obviously not what we want.

We wanted to use less accuracy so that we can perform computations in parallel. But does the network still converge if we perform computations in parallel with less accuracy? We will answer this question at the end of this chapter in section 4.6.

4.5 Using Flexible Accuracy Adaptation

Using less accuracy can reduce the learning time, if we don't perform computations in parallel. This is because of the reduced number of iterations needed per multiplication. However if less accuracy means that it takes much longer for the network to converge, we don't gain any profit. What we want is to minimize the total number of iterations needed. We could do this by adapting the accuracy in a flexible manner.

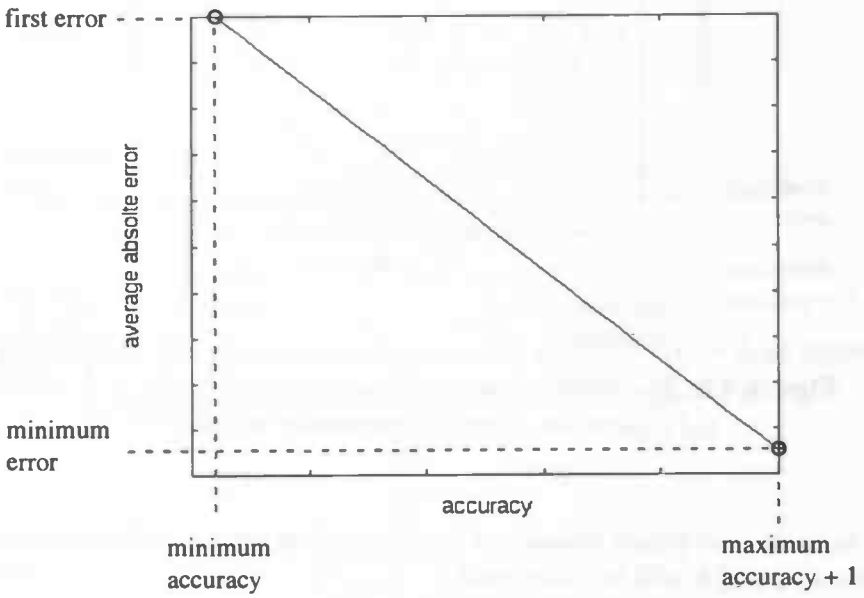


Figure 4.5: *The function adapt(x) = a f(x) + b with f(x) = x at interval [minimum accuracy, maximum accuracy + 1].*

To realize this, we start with less accuracy, because less iterations are needed then. At some point we raise the accuracy. This is repeated until the network converges. Note that the accuracy can never exceed a certain maximal accuracy.

The point at which we have to change the accuracy is hard to determine. We started by changing the accuracy linearly, as done with the Newton Raphson method (see section 2.6, page 20). We have done this according to the function:

$$adapt(x) = a\,f(x) + b \qquad with \tag{4.9}$$

$$f(x) = x \tag{4.10}$$

and with a and b computed as follows:

Assume that the the following two points lie on the function adapt:

(minimal accuracy, first error)

(maximal accuracy + 1, minimal error)

With minimal accuracy we mean the accuracy that is used initially and with maximal accuracy the accuracy which is the maximal possible is meant. The first error is the error after the first time the average absolute error is computed (i.e. with the sine function this is done after the first 25 times the train–set is presented). With the minimal error, the minimal average absolute error is meant (see also figure 4.5).

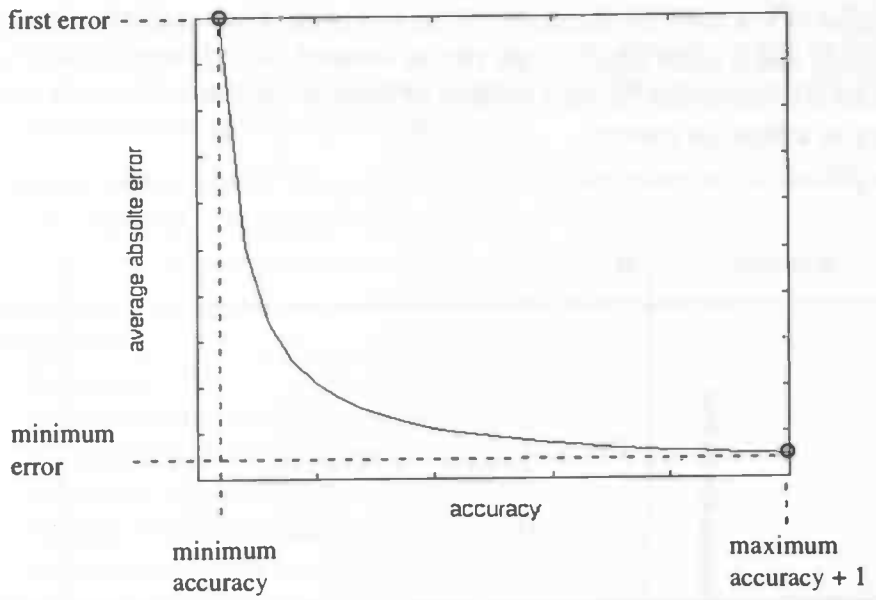


Figure 4.6: *The function* $adapt(x) = a\,f(x) + b$ *with* $f(x) = \frac{1}{x}$ *at interval [minimum accuracy, maximum accuracy + 1].*

According to these two points *(minimum accuracy, first error)* and *(maximum accuracy + 1, minimum error)* a and b will be calculated:

$$a = \frac{(minimum\ error - first\ error)}{f(maximum\ accuracy + 1) - f(minimum\ accuracy)} \tag{4.11}$$

$$b = \textit{first error} - a * f(\textit{minimum accuracy}) \tag{4.12}$$

This can be used to adapt the accuracy flexible in the following way:

- start with a (small) accuracy (the minimum accuracy)
- if the average absolute error of the network drops below *adapt(current accuracy + 1)* raise the accuracy

If the average absolute error drops below *f(current accuracy + 1)* we need to raise the accuracy. This is the reason why the function is considered until *maximum accuracy + 1* and not until *maximum accuracy*. If we would use *maximum accuracy* we would never reach this maximum accuracy because at the point at which the average absolute error drops below *adapt(maximum accuracy)* we have reached the point at which the trainings process is ended (*minimum error*).

We have also done experiments with two other functions:

$$f_2(x) = \frac{1}{x} \tag{4.13}$$

$$f_3(x) = \frac{1}{2^x} \tag{4.14}$$

Again, the variables a and b will be calculated according to equations (4.11) and (4.12) but now for f_2 and f_3 respectively. These functions are visualized in figures 4.6 and 4.7.

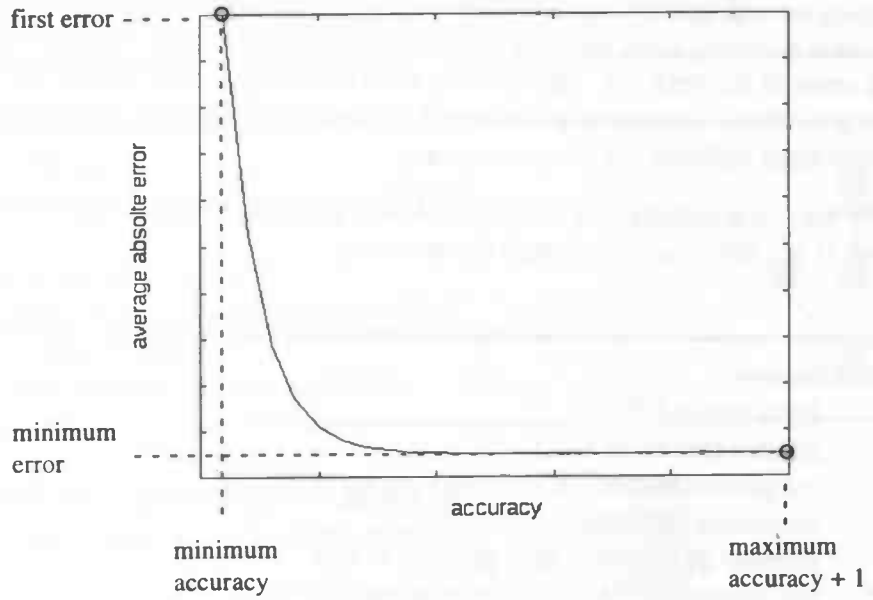


Figure 4.7: *The function adapt(x) = a f(x) + b with* $f(x) = \frac{1}{2^x}$ *at interval [minimum accuracy, maximum accuracy + 1].*

4.5.1 Results

In this section we will present the result of adapting the accuracy in a flexible manner. Again we use the sine function for our experiments. We will adapt the accuracy according to the three func-

tion described above. We will do this in three slightly different ways, namely starting with an accuracy of 1 bit, starting with an accuracy of 4 bits and starting with an accuracy of four bits but changing the accuracy with steps of 4 bits. The next table summarizes this (the step–size is the steps in which the accuracy is changed):

function	minimal accuracy	step–size	experiment
$f(x) = x$	1	1	A
$f(x) = x$	4	1	B
$f(x) = x$	4	4	C
$f(x) = \frac{1}{x}$	1	1	D
$f(x) = \frac{1}{x}$	4	1	E
$f(x) = \frac{1}{x}$	4	4	F
$f(x) = \frac{1}{2^x}$	1	1	G
$f(x) = \frac{1}{2^x}$	4	1	H
$f(x) = \frac{1}{2^x}$	4	4	I

With the results we will present a plot of the function according to which the accuracy was adapted and the actual accuracy used. At the x–axis the number of accuracy is plotted and at the y–axis the absolute error of the network. The actually used accuracy should always be below the function according to which the accuracy is adapted. In these plots the dark lines indicate the function and the lighter lines indicate the actual accuracy.

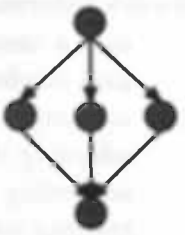
The accuracy may be adapted every 25 cycles. At that point we examine, according to the adaptation function, if it's necessary to change the accuracy.

General information:

learn rate = 0.9
momentum term = 0.2
maximum number of trainings cycles = 20000
minimum absolute error = 0.01
number of different data patterns = 200
percentage of data used for training = 30%

A) Network information:

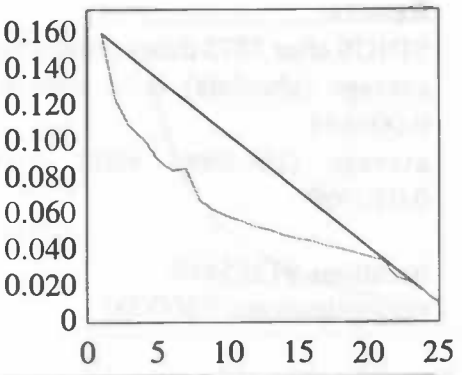
uses a random initialized 1 3 1 network
uses random input
uses sequential method
adapting the accuracy with step–size 1
according to $f(x) = x$
starting with an accuracy of 1 bit



Results:

SINE after 2775 times presenting the training set:
average (absolute) error over training–patterns: 0.009894
average (absolute) error over test–patterns: 0.011135

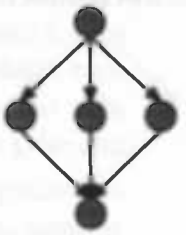
iterations: 65477319
multiplications: 6660000



Remarks: *Only when we reach an accuracy of 20 bits approaches the actual accuracy the adaptation–function. This means that until then the accuracy is changed every 25 cycles.*

B) Network information:

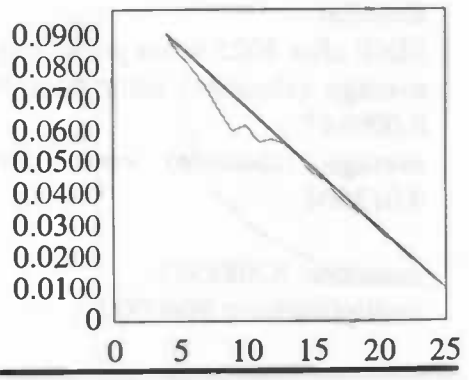
uses a random initialized 1 3 1 network
uses random input
uses sequential method
adapting the accuracy with step–size 1
according to $f(x) = x$
starting with an accuracy of 4 bits



Results:

SINE after 4975 times presenting the training set:
average (absolute) error over training–patterns: 0.009992
average (absolute) error over test–patterns: 0.011364

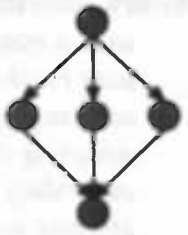
iterations: 119988018
multiplications: 11940000



Remarks: *If we start with 4 bits accuracy the actual accuracy approaches the adaptation function much better than in the previous example.*

C) Network information:

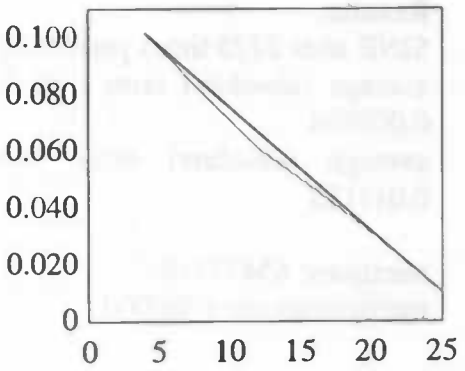
uses a random initialized 1 3 1 network
uses random input
uses sequential method
adapting the accuracy with step–size 4
according to $f(x) = x$
starting with an accuracy of 4 bits



Results:

SINUS after 3875 times presenting the training set:
average (absolute) error over training–patterns: 0.009894
average (absolute) error over test–patterns: 0.010969

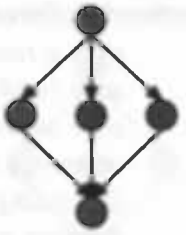
iterations 91265446
multiplications: 9300000



Remarks: *If we change the accuracy in step of size 4 the adaptation function is yet better approximated. Note that the accuracy changes only 5 times now (to 8, 12, 16, 20 and 24).*

D) Network information:

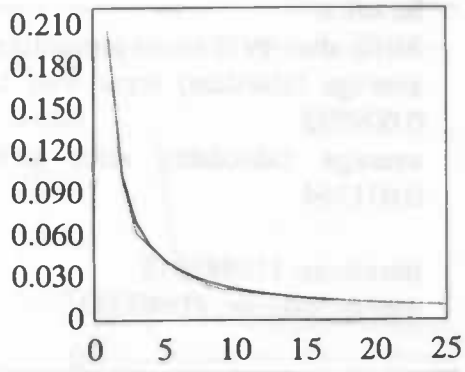
uses a random initialized 1 3 1 network
uses random input
uses sequential method
adapting the accuracy with step–size 1
according to $f(x) = \frac{1}{x}$
starting with an accuracy of 1 bit



Results:

SINE after 4025 times presenting the training set:
average (absolute) error over training–patterns: 0.009847
average (absolute) error over test–patterns: 0.013004

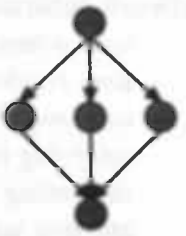
iterations: 82088563
multiplications: 9660000



Remarks: *If we change the accuracy according to* $f(x)=1/x$ *the adaptation function is approximated directly when we start with 1 bit.*

E) Network information:

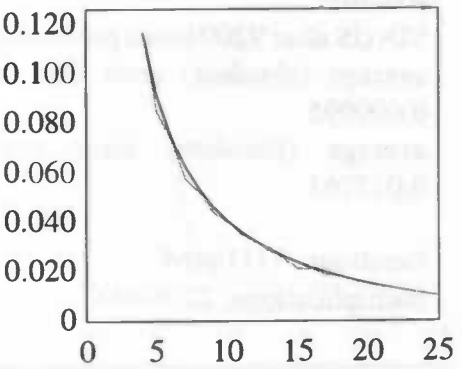
uses a random initialized 1 3 1 network
uses random input
uses sequential method
adapting the accuracy with step–size 1
according to $f(x) = \frac{1}{x}$
starting with an accuracy of 4 bits



Results:

SINE after 2900 times presenting the training set:
average (absolute) error over training–patterns: 0.009913
average (absolute) error over test–patterns: 0.010390

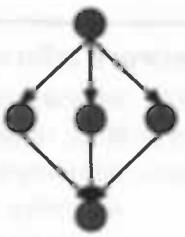
iterations: 64541318
multiplications: 6960000



Remarks: *If we start with 4 bits, we get almost the same result.*

F) Network information:

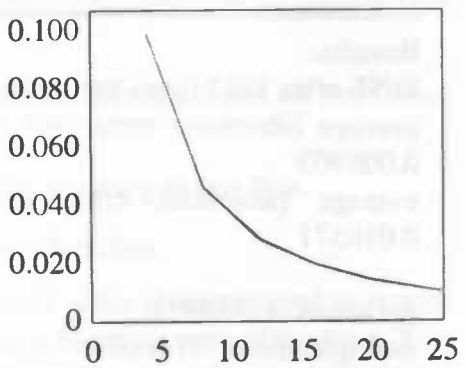
uses a random initialized 1 3 1 network
uses random input
uses sequential method
adapting the accuracy with step–size 4
according to $f(x) = \frac{1}{x}$
starting with an accuracy of 4 bits



Results:

SINE after 3700 times presenting the training set:
average (absolute) error over training–patterns: 0.009999
average (absolute) error over test–patterns: 0.014504

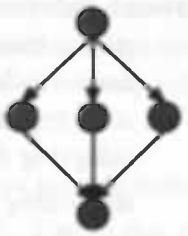
iterations: 75818807
multiplications: 8880000



Remarks: *In this plot we can see clearly that the accuracy is adapted only a few times when we change it with steps of size 4.*

G) Network information:

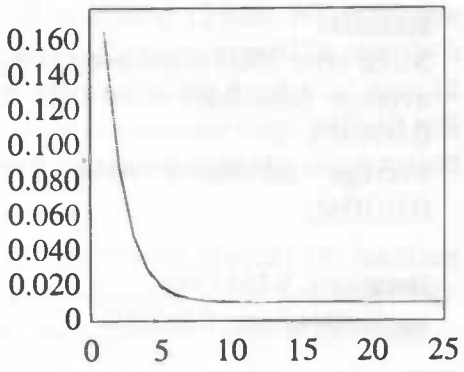
uses a random initialized 1 3 1 network
uses random input
uses sequential method
adapting the accuracy with step–size 1
according to $f(x) = 1/(2^x)$
starting with an accuracy of 1 bit



Results:

SINUS after 9200 times presenting the training set:
average (absolute) error over training–patterns: 0.009995
average (absolute) error over test–patterns: 0.012261

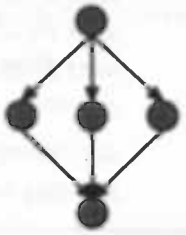
iterations: 97118464
multiplications: 22080000



Remarks: *When we use the function* $f(x) = 1/(2^x)$ *we see that when the accuracy reaches 11 bits it is not changed anymore (straight line).*

H) Network information:

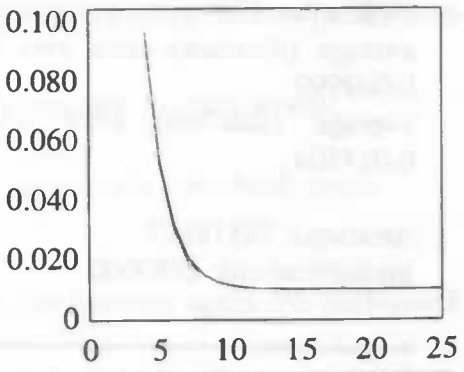
uses a random initialized 1 3 1 network
uses random input
uses sequential method
adapting the accuracy with step–size 1
according to $f(x) = 1/(2^x)$
starting with an accuracy of 4 bits



Results:

SINE after 3225 times presenting the training set:
average (absolute) error over training–patterns: 0.009903
average (absolute) error over test–patterns: 0.010571

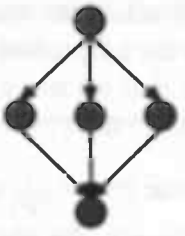
iterations: 41488937
multiplications: 7740000



Remarks: *Here a similar result occurs as with the previous example.*

I) Network information:

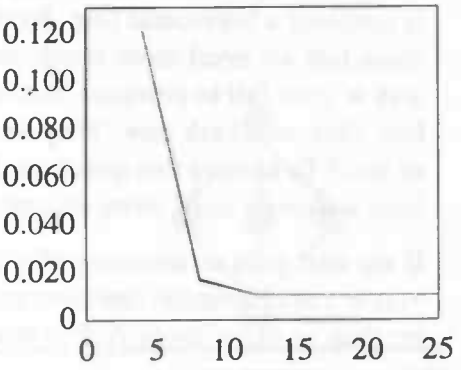
uses a random initialized 1 3 1 network
uses random input
uses sequential method
adapting the accuracy with step–size 4
according to $f(x) = 1/(2^x)$
starting with an accuracy of 4 bits



Results:

SINE after 3300 times presenting the training set:
average (absolute) error over training–patterns:
0.009981
average (absolute) error over test–patterns:
0.010078

iterations: 39454226
multiplications: 7920000



Remarks: *Here the accuracy doesn't change anymore after 12 bits (straight line). This implies that the accuracy changes only two times (to 8 and to 12).*

4.5.2 Discussion

If we use flexible accuracy adaptation, the convergence properties are not damaged. We aimed to do this in such a way, such that the total number of iterations was minimized. However, the results of repeating a particular experiment differ too much to draw a reliable conclusion. For this much more experiments should be done.

We can, however, say something about the adaptation function used in the different experiments. The meaning of the plots of the actual accuracy and the adaptation function can be summarized as follows:

- If the actual accuracy used is below the adaptation function, this means that we need more accuracy.
- If the actual accuracy is on the adaptation function the accuracy is just fine.
- The actual accuracy can never be above the adaptation function.

We started with $f(x) = x$ as adaptation function. If we take a look at the plot presented in experiment A, we see that the actual accuracy is below the adaptation function most of the time. This means that we use too less accuracy (according to the adaptation function). Because the actual accuracy is below the adaptation function all the time until we reach 20 bits, the accuracy is adapted every 25 cycles, the number of cycles after which we check whether we need more accuracy or not.

If we compare this with experiment D, where the adaptation function $f(x) = \frac{1}{x}$ is used, we see that the actual accuracy follows the adaptation function much better. At the points where it drops below the adaptation function we raise the accuracy. The result when we use this adaptation function is that we use a particular accuracy for more than 25 cycles. This is obviously something we want, rather than directly changing the accuracy until 20 bits is reached.

If we use $f(x) = \frac{1}{(2^x)}$ as adaptation function, the actual accuracy also follows the adaptation function closely. We also see that at a certain point (approximately around 12 bits) the accuracy doesn't change anymore (see experiment G). This occurs because from that point on the function is (almost) a horizontal line. So the highest accuracy used will be around 12 bits. When we use more bits we need more iterations. So probably the total number of iterations will be less. In a way it's not fair to compare this with flexible adaptations that will use higher accuracies than 12 bits. One could ask now: Why should we use more bits, if the network can converge with 12 bits or less? To answer this question: If we have a minimum error that is lower than the one we used here we could need more accuracy to converge.

If we start with an accuracy of 4 bits instead of 1 bit, we see that the results of using the function $f(x) = x$ as adaptation function, are better. The actual accuracy approaches the function much better than in experiment A. For other two adaptation functions the results are almost the same.

If we take steps of size 4 instead of 1 bit, the adaptation functions are followed well in all cases. The accuracy doesn't change many times now, only 5 times for adaptation function $f(x) = x$ and $f(x) = \frac{1}{x}$ and only 2 times for adaptation function $f(x) = \frac{1}{(2^x)}$. This, because the last function uses no accuracy higher than 12.

Summarizing, the "perfect adaptation function" (if it exists) is hard to find. Also because of the fact that when an experiment is repeated, the results can differ substantial, much more experiments should be done to get a good impression of the number iterations needed to converge. We have seen that adaptation function $f(x) = \frac{1}{x}$ gives the most acceptable results.

4.6 Parallelism versus Accuracy

As we have seen in section 4.4, it is possible to introduce reduced accuracy in error back–propagation learning. We already new, from the previous chapter, that we could introduce parallelism in error back–propagation learning in certain cases.

The last step is now to combine these two, to see if it works together. In other words:

> *Is it possible to introduce parallelism at the cost of less accuracy in error back–propagation learning?*

To answer this question we did some experiments with the sine function again. We performed the following experiments:

- using an accuracy of 12 bits so we can perform two computations in parallel
- using an accuracy of 8 bits so we can perform three computations in parallel

- using an accuracy of 6 bits so we can perform four computations in parallel

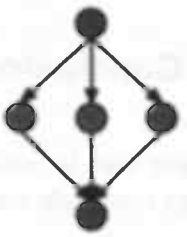
The results of repeating a particular experiments several times differed substantial. For this reason we will present here the result of repeating each experiment three times.

General information:

learn rate = 0.9
momentum term = 0.2
maximum number of trainings cycles = 10000
minimum absolute error = 0.01
number of different data patterns = 200
percentage of data used for training 30%

Network information:

learning sine
uses a random initialized 1 3 1 network
uses ordered input
uses "original" parallel method



---

number of parallel computations	used accuracy	number of cycles needed to converge	(absolute) average error over train–set	(absolute) average error over test–set
2	12	>10000	0.024389	0.054808
2	12	3150	0.009992	0.010377
2	12	2550	0.009962	0.011647

number of parallel computations	used accuracy	number of cycles needed to converge	(absolute) average error over train–set	(absolute) average error over test–set
3	8	>10000	0.011161	0.011279
3	8	>10000	0.010272	0.010169
3	8	>10000	0.010415	0.009744

number of parallel computations	used accuracy	number of cycles needed to converge	(absolute) average error over train–set	(absolute) average error over test–set
4	6	>10000	0.025705	0.024988
4	6	2125	0.009986	0.011517
4	6	>10000	0.030516	0.030910

As can be seen from these tables the results of a particular experiment are very different. With 2 parallel computations and an accuracy of 12 bits, we see that from the three experiments presented here only two converge. With 3 parallel computations and an accuracy 8 bits, not one of the experiments presented converges. However, these experiments do result in an error that almost approximates the minimal error. For the last example, using 4 parallel computations with 6 bits accuracy, only one experiment converges. This particular experiment converges very easily, namely after 2125 cycles.

It's hard to draw conclusions based on these experiments. To draw a reliable conclusion much more experiments should be performed. Because the repeated experiments differ so much, the random initialization is apparently very crucial when less accuracy is used. We also noted this when we adapted the accuracy in a flexible manner.

4.7 Conclusions

We have seen in section 4.4 that using less accuracy in error back–propagation error is possible. We can conclude the following:

- The network did converge when an accuracy of 6, 8, 12 or 24 bits was used
- The network didn't need substantially more cycles to converge when less accuracy was used.

From the experiment done with a flexible accuracy adaptation (section 4.5) we can notice the following:

- To draw a reliable conclusion with respect to the number of iterations needed, much more research and experiments should be done. Especially because the initialization of the network strongly influences the results of an experiment.
- From the different adaptation functions used, the function $f(x) = \frac{1}{x}$ gives the best results. Furthermore starting with an accuracy of 4 bits instead of 1 seems to give rise to better results.
- In general we can conclude that we need to do much more research to find the "optimal adaptation function" (if this exist at all).
- We only presented results of experiments done with the sine function. It could be very well possible that there does not exist a unique "optimal" adaptation function.

If we combine the introduction of parallelism with the use of less accuracy, we've got very different results.

From all experiments done we noted one imported issue, namely that the initialization of the network apparently has great influence on the convergence of the network when we use less accuracy.

We have seen that it's possible (under certain circumstances) to use less accuracy in error back–propagation learning. We have also seen that when we combine the introduction of parallelism with the usage of less accuracy the network did not always converge. What we eventually would like to achieve is a way to adapt the accuracy flexible together with the usage of parallelism.

This could be realized in the following way: We start for example with an accuracy of 6 bits and 4 parallel computation, at some point we will change the accuracy to 8 bits, this implies that we can't use more than 3 parallel computations. Then at a certain point we change the accuracy to 12 bits, which implies 2 parallel computations and we finish with an accuracy of 24 bits with no parallel computations. The point as which the accuracy should be changed has to been researched. As we saw in section 4.5 this point is hard to determine. But the function $f(x) = \frac{1}{x}$ looks a acceptable adaptation function.

In our example we learned the sine function until an error of 0.01 was reached. For some applications this error is probably too high. If we use smaller errors, the accuracy which with we perform calculation becomes more important. This is also a reason to adapt the accuracy flexible, starting for example with 6 bits and raising it until 24 bits. Before this could be realized much more research should be done.

Chapter 5

Conclusions and Recommendations

This work concentrated on the possibilities of introducing parallelism in error back–propagation learning that is not restricted to the layer–wise composition of neural networks. We also looked at a possibility to achieve this parallelism on a relative simple architecture, namely by introducing less accuracy in exchange for parallelism.

The key question of this work was:

> *Is it possible to introduce parallelism that is not restricted to the layer–wise composition of neural networks in error back–propagation learning by using less accuracy?*

To answer this question we started by looking at traditional gradient descent methods (chapter 2) . We saw that the Newton methods still converged when we used old derivatives. We also saw that it was possible to change the accuracy without damaging the convergence properties.

After this we turned to error back–propagation learning (chapter 3). Because of the resemblance with gradient descent we thought that it could be possible to introduce parallelism that is not restricted to the layers. Using such a parallelism implies that we use old values (old errors) and this was also the case with the Newton methods. It turned out that this was only possible for a certain class of problems, namely those by whom the input patterns are presented in an ordered way, such as function approximation problems.

We also looked at the possibilities of using a partially old error, that is use a combination of old and new errors to update the weights. Although this gives the network more correct information than the usage of totally old errors does, it does not converge any faster. Even worse, it seems the network has more difficulties with using a combination of old an new errors than with using totally old errors.

Another interesting point we observed when using parallelism over the layers was, that when we used a "too large" network, it learned to ignore the hidden neurons that were computed in parallel with the output neurons by setting the weights on their incoming and outgoing synapses to values around zero. In other words parallelism can be introduced if we there is enough redundancy.

Next we observed the possibilities of using less accuracy in error back–propagation learning (chapter 4). We saw that it was very well possible to use less accuracy without damaging the convergence properties. However, the random initialization becomes more crucial when we use less accuracy.

We also tried to find a way to adapt the accuracy of multiplications in a flexible manner. It turned out that the point at which we have to change the accuracy is very hard to determine and we need

more research to find an "optimal" solution (if one exists at all). For now the adaptation function $f(x) = \frac{1}{x}$ gives the best results.

Finally, we looked at the results of combining the use of less accuracy with the usage of parallelism. Again, it turned out that the random initialization is very important for the convergence of the network.

What we eventually like to realize is a way of adapting the accuracy flexible, starting with a small accuracy and much parallelism. At the right point we have to change to a higher accuracy and less parallelism. Eventually, we need all the accuracy possible to achieve an optimal convergence. Especially, when we use very small errors the accuracy becomes crucial. In this work we made a step in the right direction. It can be concluded that before we can realize such a combination of flexible accuracy adaptation and parallelism, a lot a of extra research need to be done, especially on the possibilities of flexible accuracy adaptation.

Acknowledgement

I would like to thank everyone who has contributed to this work in one way or another. Specially I would like to thank my supervisor Prof. Dr. Ir. L. Spaanenburg. Furthermore I would like to thank Rienk Venema for his help on the mathematical foundation of the gradient descent method. And last but certainly not least I would like to thank Marco Diepenhorst for all his support.

Literature

[1] "An Introduction to Numerical Analysis", **Kendall E. Atkinson**, *John Wiley & Sons*, 1978.

[2] "Arithmetic for Relative Accuracy", **R. van Drunen, L.Spaanenburg, P. Lucassen, J.A.G. Nijhuis and J.T. Udding**, *IEEE Symp. on Computer Arithmetic*, pp. 239–250, Bath, England, July 1995.

[3] "Computer Architecture and Organization", **Ian East**, University of Buckingham, *Pitman Publishing*, ISBN 0–273–03038–8, 1990.

[4] "An Empirical Study of Learning Speed in Back–Propagation Networks", **Scott E. Fahlman**, *CMU–CS–88–162*, 1988.

[5] "Neural Networks A Comprehensive Foundation", **Simon Haykin**, *Macmillan College Publishing Company*, 1994.

[6] "Encyclopaedia of Mathematics", **M. Hazewinkel**, *Kluwer Academic Publishers*, vol. 4, p. 292, 1994.

[7] "Learning with Limited Numerical Precision Using the Cascade–Correlation Algorithm", **Markus Hoehfeld and Scott E. Fahlman**, *CMU–CS–91–130*, May, 1991.

[8] "Finite Precision Error Analysis of Neural Network Hardware Implementations", **Jordan L. Holt and Jenq–Neng Hwang**, *IEEE Transactions on Computers*, vol. 42, no. 3, pp. 281–290, March 1993.

[9] "Computer Multiplication and Division Using Binary Logarithms", **J.N. Mitchell jr.**, *IRE Transactions on Electronic Computers*, vol. 11, pp. 512–517, August 1962.

[10] "Hardware–Friendly Learning Algorithms for Neural Networks: an Overview", **P.D. Moerland and E. Fiesler**, *MicroNeuro '96*, Lausanne, Switzerland, February, 1996.

[11] "Iterative Solutions of Nonlinear Equations in Several Variables", **J.M. Ortega and W.C. Rheinboldt**, Computer Science and Applied Mathematics, *Academic Press*, 1970.

[12] "Learning Representations by Back–Propagating Errors", **David E. Rumelhart, Geoffrey E. Hinton and Ronald J. Williams**, *Nature*, vol. 323, pp. 533–536, October 1986.

[13] "An Accuracy–Driven Complex Arithmetic Unit", **L. Spaanenburg, R. van Drunen, A. Pops, A.C.W. Postma and R. Sytsma**, *Digest EUROMICRO '94*, pp. 491–498, Liverpool, September 1994.

[14] "A Multiplier–Less Digital Neural Network", **L.Spaanenburg, B. Höfflinger, S. Neußer, J.A.G. Nijhuis and A.J. Siggelkow**, *2nd International Conference on Microelectronics of Neural Networks*, Munich, October 1991.

[15] "An Arithmetic Technique for Accuracy–Driven VLSI Systems", **L. Spaanenburg, A. SiggelKow and M. Luft**, *Digest ECCTD*, pp. 161–166, Davos, September, 1993.

[16] "Introduction to Artificial Neural Systems", **Jacek M. Zurada**, *West Publishing Company*, 1992.

Appendix A

The Source Code of the Secant Method

A.1 Description of the Source Code

In this section the source code of `secant.c` is described as presented in section A.2.

The original source code computes roots of an equation using the Regula Falsi method (section 2.4). This program has been adapted to the Secant method (section 2.5).

A.1.1 Declarations

The program can be ran with several options, the most important are:

–m	the parallel version,
–l	the sequential version,
–f	flexible accuracy adaptation,
–x	output written to an Interleaf file.

For the implementation of the Secant method the variables **`first`** and **`second`** (l. 39–40, 71–72) have been introduced (instead of the variables **`left`** and **`right`**). The variable **`first`** denotes the first approximation of the root and **`second`** the second approximation. We need two points to calculate the derivative. The variable **`first`** will be read from input. The variable **`second`** is calculated by adding **`epsilon`** to **`first`**. Because we choose **`epsilon`** small (l. 27), the first approximation of the derivative is a rather good one.

A.1.2 `Main`

First of all the mode (parallel, sequential etc.) is determined. Then this mode is printed on the screen. Next the function **`get_equation`** is called and the first approximation is read. Finally the equation is solved by calling the function **`equation_solver`**.

A.1.3 `Get_equation`

This function reads the equation that has to be solved from a file. Further the file **`MATH`** is used for generating an **`mathematica`** input file.

A.1.4 Equation_solver

This functions starts with the initialization of some variables (l. 228–233). Then the starting values are computed. The variable `first` was read from input, the other starting point is computed by adding `epsilon` to `first`. The function `fill_node` is called twice to computed the function values belonging to `first` and `second`. After this the time this has taken is computed and the result is generated (`get_result`) (l. 248–254). Next the derivative is computed (l. 259–264).

Now the initialization is done and the procedure can be repeated (l. 269). There are three types that we distinguish:

`type=1` get the new function value
`type=2` get the new x–value
`type=3` get the derivative.

Get the derivative:
The derivative is determined and if we perform the procedure sequential, a node is filled with the information for the computation of the new x–value (l. 276–285).

Get the new x–value:
The new x–value is determined and a node is filled with information to compute the new function value (l. 290–295).

Get the new function value:
The new function value is determined (l. 301) and if we use the flexible option, the new accuracy is determined (l. 303–306). Next the function shift_on is called, to determine with which points we continue our procedure (l. 313). Further if we perform the procedure in parallel, a node is filled with information to compute the new x–value. Then we fill a(nother) node with information to compute the derivative (l. 314–316).

A.1.5 Fill_node

This function finds a free node (l. 367–369), and fills it with information to perform the computation (l. 371–378). Next it distinguishes the three types described in the previous section, to compute the next x–value, function value or derivative. And calls the appropriate hardware call. Furthermore the time these calls take is stored.

A.1.6 Get_result

This function returns the value of the currently active node.

A.1.7 Shift_on

This function shifts past, now and future values, in the appropriate way. In this case (the secant method) this means that `past` gets the value from `now` and `now` gets the value from `future`.

A.1.8 SetPrecision

This function sets the precision according to the current function value (x):

```
x < -3                    precision = 9
```

```
-3 <= x < -0.01          precision = 24 + 15 * x / 3
-0.01 <= x <= 0.01       precision = 24
0.01 < x <= 3            precision = 24 - 15 * x / 3
x > 3                    precision = 9
```

A.2 Source Code

In this section the source code of `secant.c` is presented. Underlined line numbers are added or adapted.

A.2.1 Declarations

```
1   /*_____________________________________________________________________*/
2   /* FILE:     secant.c                                    VERSION: 3.1  */
3   /* IN:       ~/afstudeerwerk/digi/secant                 DATE:20/05/96 */
4   /*                                                                     */
5   /* PURPOSE: To search the relation between the number of machine cycles, */
6   /* required to find the solution to an equation, and the maximum       */
7   /* #iterations/computation.                                            */
8   /*                                                                     */
9   /* USAGE:    Newton [-<s|m|l|f|x>] filename                           */
10  /*            s provides parallel operation for all max. iterations.   */
11  /*            m provides parallel operation max. 24 iterations.        */
12  /*            l provides sequential operation for max. 24 iterations.  */
13  /*            f provides for flexible accuracy adaption.               */
14  /*            x provides an Interleaf input file.                      */
15  /*                                                                     */
16  /* VERSION: 20/05/96  G.F.Meijering (secant method)                    */
17  /*          18/08/92  L.Spaanenburg & M.Luft (flexible accuracy)       */
18  /*          17/07/92  L.Spaanenburg (provides run-time options)        */
19  /*          17/06/92  L.Spaanenburg (original C-code)                  */
20  /*_____________________________________________________________________*/
21
22  #include    <stdio.h>
23  #include    "/home2/rug4/ben/digilog/cmodels/include/digilog/hardware.rtn"
24
25  /* _________ DEFINITIONS ____________________________________________*/
26
27  #define epsilon 0.001
28
29  /*__________ INTERNAL STRUCTURES ____________________________________*/
30
31  typedef struct collect {
32    float     x;                                        /* x-value */
33    float     f;                          /* function value at the x-value */
34  } POINT;                              /* substructure to store an x,f(x)-pair */
35
36  typedef struct pe {
37    int       type;                          /* type of computation (1, 2 or 3) */
38    int       time;                 /* time at which the computation is finished */
39    POINT     first;                              /* x, f-pair to start with */
40    POINT     second;                               /* the second x, f-pair */
41    float     dfinv;          /* derivative at the time the node was activated */
42    float     res;                                /* result of the computation */
43  } PE;                         /* contains all data for a single node computation */
44
45  /*__________ INTERNAL ROUTINES ____________________________________*/
46
47  void      get_equation();
48  int       equation_solver(short int flag);
49  int       fill_node(int type, int time, POINT *first, POINT *second,
```

```
50                          float dif);
51  float       get_result();
52  int         shift_on();
53  void        SetPrecision(float value);
54
55  /*_____________ INTERNAL VARIABLES ____________________________________________*/
56
57  POINT       now, past, future;        /* from past/pairs a future pair is found */
58  PE          units[10];                          /* the Processing Elements */
59  int         unitcount = 10;                /* the number of Processing Elements */
60  short int max_it;          /* maximum number of iterations per basic operation */
61  short int default_it = 24;                    /* default setting of max_it */
62  short int it;               /* total amount of iterations in a basic operation */
63  short int mode = 0;                        /* mode of operations (see header) */
64  short int output = 0;                   /* interleaf file request(see header) */
65  short int flexible = 0;            /* flag for accuracy adaption; 1 if active */
66  int         tot_it;                           /* total amount of iterations */
67  short int mult_prec = 24;                /* hardware multiplication precision */
68  short int div_prec = 2;                       /* hardware division precision */
69  short int frl = 32;                             /* finite register length */
70  char        filename[20];                    /* name of the parameter file */
71  float       first;                                  /* first approximation */
72  float       second;                                /* second approximation */
73  int         coeffcount;                  /* amount of equation coefficients */
74  int         coeff[20];                        /* the equation coefficients */
75  FILE        *MATH;                        /* output port for MATH data file */
76
```

A.2.2 Main

```
77  main(int argc, char **argv)
78        /*_____________________________________________________________________*/
79        /* ACTION: the main procedure calls the equation solver for different   */
80        /*         values for max_it: the maximum number of DIGILOG iterations. */
81        /*_____________________________________________________________________*/
82  {
83    int       result;        /* # machine cycles required for equation solving */
84    int       i, j;                                  /* local loop variables */
85
86
87    /* get from the call arguments, the execution mode and parameter filename  */
88    for (mode = 0, filename[0] = '\0', i=1; i < argc; i++) {
89      if (argv[i][0] == '-') {
90        for (j = 1; argv[i][j] != '\0'; j++) {
91          switch (argv[i][j]) {
92          case 's':
93            if (mode == 0)
94              mode = 1;
95            break;
96          case 'm':
97            if (mode == 0)
98              mode = 2;
99            break;
100         case 'l':
101           if (mode == 0)
102             mode = 3;
103           break;
104         case 'f':
105           flexible = 1;
106           break;
107         case 'x':
108           output = 1;
109           break;
110         default : printf("Invalid argument\n"); exit();
111         }
112       }
113     }
114     else {
115       if (strcmp(filename, "\0") == 0) strcpy(filename, argv[i]);
```

```
116         else {
117           printf("Duplicate filename\n"); exit();
118         }
119       }
120     }
121
122     /* check on valid call arguments and provide a default for the mode .....*/
123     if (mode == 0) mode = 1;
124     if (strcmp(filename, "\0") == 0) {
125       printf("No parameter file\n");
126       printf("\nProper call is TestDigilog [-<s|m|l|f|x>] <filename>\n");
127       printf("  with 's' for a short overview varying iterations/operation\n");
128       printf("  with 'm' for a single parallel run with full documentation\n");
129       printf("  with 'l' for a single sequential run with full documentation\n");
130       printf("  with 'f' for flexible accuracy adaption\n");
131       printf("  with 'x' for an Interleaf input file\n");
132       printf("  default is 's'\n");
133       exit();
134     }
135
136     if (mode == 1) {
137       printf("short overview\n");
138     }
139     if (mode == 2) {
140       printf("parallel\n");
141     }
142     if (mode == 3) {
143       printf("sequential\n");
144     }
145     if (flexible) {
146       printf("flexible\n");
147     }
148     if (output) {
149       MATH = fopen("math\0", "a");
150       printf("output\n");
151     }
152
153     /* get the parameters from the named file ..............................*/
154     get_equation();
155
156     printf("\n First approximation:");
157     scanf("%f", &first);
158
159     /* execute the parallel newton-raphson for the given equation .....*/
160     if (mode > 1)
161       i = default_it - 1;
162     else
163       i=0;
164     for (max_it = default_it; max_it > i; max_it--) {
165       result = equation_solver(max_it == default_it);
166       j = fill_node(1, result, &past, &future, 0.0);
167       if (mode > 1) printf("\nFinal result:\n");
168       printf("(max_it=%d) f=%10.7g at x=%10.7g found in %d machine cycles\n",
169              max_it, get_result(j), future.x, result);
170     }
171     /*****************************************************************************/
172     /* EXIT main                                                                 */
173     /*****************************************************************************/
174   }
175
```

A.2.3 Get_equation

```
176 void        get_equation(void)
177       /*_____________________________________________________________________*/
178       /* ACTION:  collect from file the equation and the search interval.     */
179       /*_____________________________________________________________________*/
180 {
181   FILE      *FROM;                                  /* input file port */
```

```
182   int       i;                                          /* local loop variable */
183
184   /* read from file the equation coefficients, followed by the interval .....*/
185   if ( (FROM = fopen(filename, "r")) == NULL) {
186     printf("no parameter file %s found\n");
187     exit();
188   }
189   fscanf(FROM, "%d", &coeffcount);
190   for (i=0; i<coeffcount; i++)
191     fscanf(FROM, "%d", &coeff[i]);                        /* coeffficients */
192   fclose (FROM);
193
194   /* echo the read parameters on the standard output .....................*/
195   printf("\nFile %s contains a %d-order polynome\n", filename, coeffcount-1);
196   printf("   the coefficients are:");
197   for (i=0; i<coeffcount; i++)
198     printf(" %d", coeff[i]);
199
200   fprintf(MATH, "f =");
201   for (i = 0; i < coeffcount; i++) {
202     fprintf(MATH, " + %d x ^ %d ", coeff[i], coeffcount - i - 1);
203   }
204   fprintf(MATH, "\nShow[Plot[f, {x, a, b}], Graphics[{");
205   max_it = 24;
206   /*************************************************************************/
207   /* EXIT get_equation                                                     */
208   /*************************************************************************/
209 }
210
```

A.2.4 Equation_solver

```
211 int         equation_solver(short int flag)
212       /*_____________________________________________________________*/
213       /* ACTION:  to solve an equation.                              */
214       /*                                                             */
215       /* INPUT:   short int flag  > 1 for initial print request; otherwise 0  */
216       /*                                                             */
217       /* OUTPUT:  int       ()    > total amount of machine cycles   */
218       /*_____________________________________________________________*/
219 {
220   int       time;             /* current time                       */
221   int       ltime;            /* time of last update                */
222   int       silence;          /* 1 if there are still active units  */
223   int       index;            /* index to active unit               */
224   int       clock;            /* scheduling clock                   */
225   float     dfinv;            /* inverse derivative                 */
226   FILE      *INTERLEAF;       /* output port for INTERLEAF data file */
227
228   /* initialize the variables ..........................................*/
229   for (index=0; index < unitcount; index++)
230     units[index].time = -1;
231   past.x = now.x = future.x = 0;
232   past.f = now.f = future.f = 0;
233   tot_it = time = clock = 0;
234
235   if (output == 1) {
236     INTERLEAF = fopen("interleaf\0", "a");
237   }
238
239   /* compute the starting values .......................................*/
240   past.x = first + epsilon;
241   fill_node(1, time, &future, &past, 0.0);
242   now.x = first;
243   fill_node(1, time, &future, &now, 0.0);
244   if (flag == 1)
245     printf("\nStarting search at point (%10.7g,%10.7g)\n",
246           first, units[1].res);
247
```

```
248   ltime = 0;
249   if (ltime < units[0].time)
250     ltime = units[0].time;
251   past.f = get_result(0);
252   if (ltime < units[1].time)
253     ltime = units[1].time;
254   now.f = get_result(1);
255   if (flexible) {
256     SetPrecision(now.f);
257     printf("precision %d\n", mult_prec);
258   }
259   index = fill_node(3, ltime, &past, &now, 0.0);
260   if (ltime < units[0].time)
261     ltime = units[0].time;
262   dfinv = get_result(0);
263   if (flag == 1) {
264     printf("   Initial derivative is %10.7g\n\n", dfinv);
265   }
266   /* perform the iterative computation over the nodes ..........*/
267   time = clock = ltime;
268   fill_node(2, time, &past, &now, dfinv);
269   for (ltime = 0, time = clock + 1; time < (clock + 10000); time++) {
270     silence = 1;
271     for (index=0; index < unitcount; index++) {
272       if (units[index].time != -1) silence = 0;
273       if (units[index].time == time) {
274
275         /* get the new derivative ..........................*/
276         if (units[index].type == 3) {
277           dfinv = get_result(index);
278           if (mode > 1) {
279             printf("(#cycles=%d) ", time);
280             printf("xvalue=%10.7g function=%10.7g derivative=%10.7g\n",
281                    future.x, future.f, dfinv);
282             fprintf(INTERLEAF,"%d %10.7g \n", time, future.x);
283           }
284           if (mode == 3)
285             fill_node(2, time, &past, &now, dfinv);
286
287           /* get the new x-value ..........................*/
288         }
289         else
290           if (units[index].type == 2) {
291             printf("(%5.5g,%5.5g)\n", now.x, now.f);
292             fprintf(MATH, "\nPoint[{%5.5g,%5.5g)],", now.x, now.f);
293             units[index].second.x = get_result(index);
294             fill_node(1, time, &(units[index].first), &(units[index].second),
295                       dfinv);
296
297             /* get the new function value ..................*/
298           }
299           else
300             if (units[index].type == 1) {
301               future.f = get_result(index);
302               future.x = units[index].second.x;
303               if (flexible) {
304                 SetPrecision(future.f);
305                 printf("future precision %d\n", mult_prec);
306               }
307               if (mode > 1) {
308                 printf("(#cycles=%d) ", time);
309                 printf("xvalue=%10.7g function=%10.7g derivative=%10.7g\n",
310                        future.x, future.f, dfinv);
311                 fprintf(INTERLEAF,"%d %10.7g \n", time, future.x);
312               }
313               if (shift_on() != 0) {
314                 if (mode != 3)
315                   fill_node(2, time, &past, &now, dfinv);
316                 fill_node(3, time, &past, &now, dfinv);
317               }
318
319               /* do not know what to get ..................*/
```

```
320             }
321             else {
322               printf("illegal operation on a node\n");
323               exit();
324             }
325         }
326       }
327       if (silence == 1)
328         break;
329       ltime++;
330     }
331     if (future.f > 0.02 || future.f < -0.02)
332       ltime = -1 * (clock + ltime);
333     else
334       ltime += clock;
335     if (output == 1) {
336       fclose(INTERLEAF);
337       fprintf(MATH, "\b}]]\n");
338       fprintf(MATH, "cycles = %d", time);
339       fclose(MATH);
340     }
341     /*****************************************************************************/
342     /* EXIT equation_solver                                                      */
343     /*****************************************************************************/
344     return (ltime);
345   }
346
```

A.2.5 Fill_node

```
347   int       fill_node(int type, int time, POINT *first, POINT *second, float dif)
348         /*_____________________________________________________________________*/
349         /* ACTION:  to fill a computing node with information, perform the     */
350         /* desired computation and schedule the result for the future.         */
351         /*                                                                     */
352         /* INPUT:   int       type        > type of calculation to be performed */
353         /*                                  by the node                        */
354         /*          int       time        > current time                       */
355         /*          POINT     *first      > first approximation                */
356         /*          POINT     *second     > second approximation               */
357         /*          float     dif         > last computed derivative           */
358         /*                                                                     */
359         /* OUTPUT:  int       ()          > index of the activated node        */
360         /*_____________________________________________________________________*/
361   {
362     int       i;              /* local loop variable                           */
363     int       index;          /* free unit                                     */
364
365     /* finds a free unit and filles it with past, present and future         */
366     /* information ..........................................,.....  */
367     for (index = -1, i = 0; i < unitcount && index == -1; i++) {
368       if (units[i].time == -1)
369         index = i;
370     }
371     if (index != -1) {
372       units[index].type = type;
373       units[index].time = time;
374       units[index].first.x = (*first).x;
375       units[index].second.x = (*second).x;
376       units[index].first.f = (*first).f;
377       units[index].second.f = (*second).f;
378       units[index].dfinv = dif;
379
380
381       /* compute the next function value .....................................*/
382       if (type == 1) {
383         units[index].res = (float) coeff[0];
384         for (i=1; i<coeffcount; i++) {
385           units[index].res = coeff[i] +
```

```
386             (float) HW__multiply(units[index].second.x,units[index].res,
387                                  &it, max_it, mult_prec, frl);
388           units[index].time += it+1;
389         }
390         /* compute the next x-value ..........................................*/
391       }
392       else
393         if (type == 2) {
394           units[index].res = units[index].second.x -
395             (float) HW__multiply( units[index].second.f, units[index].dfinv,
396                                  &it, max_it, mult_prec, frl);
397           units[index].time += it + 1;
398           /* compute the derivative ..........................................*/
399         }
400         else {
401           units[index].res =
402             HW__divide((double) (units[index].second.x - units[index].first.x),
403                        (units[index].second.f - units[index].first.f),
404                        &it, max_it, div_prec, frl);
405           units[index].time += it + 2;
406         }
407
408     }
409     /*****************************************************************************/
410     /* EXIT fill_node                                                            */
411     /*****************************************************************************/
412     return (index);
413   }
414
```

A.2.6 Get_result

```
415   float       get_result(int index)
416         /*_____________________________________________________________________*/
417         /* ACTION:  to reset an active computing node.                         */
418         /*                                                                     */
419         /* INPUT:   int    index    > index to node involved in the computation */
420         /*                                                                     */
421         /* OUTPUT:  int    ()       > result of last performed computation     */
422         /*_____________________________________________________________________*/
423   {
424     units[index].time = -1;
425     /*****************************************************************************/
426     /* EXIT get_result                                                           */
427     /*****************************************************************************/
428     return (units[index].res);
429   }
430
```

A.2.7 Shift_on

```
431   int         shift_on(void)
432         /*_____________________________________________________________________*/
433         /* ACTION:  to store a future solution as now.          */
434         /*                                                                     */
435         /* OUTPUT:  int   ()  > 1 or -1 if information is updated; otherwise 0  */
436         /*_____________________________________________________________________*/
437   {
438     int       active = 0;     /* if 0, no value update required; otherwise 1/-1 */
439
440     /* find out whether the future value is really different ..................*/
441
442     if (now.x != future.x) {
443       active = 1;
444       past.x = now.x;
```

```
445       past.f = now.f;
446       now.x = future.x;
447       now.f = future.f;
448     }
449     /*****************************************************************************/
450     /* EXIT shift_on                                                             */
451     /*****************************************************************************/
452     return (active);
453   }
454
```

A.2.8 SetPrecision

```
455   void          SetPrecision(float value)
456         /*_____________________________________________________________________*/
457         /* ACTION:  to adapt the precision according to the current function   */
458         /*          value.                                                     */
459         /*                                                                     */
460         /* OUTPUT:  void       ()              > new multiplication precision. */
461         /*_____________________________________________________________________*/
462   {
463     if (value >= 0.0) {
464       if (value > 3)
465         mult_prec = 9;
466       else
467         if (value > 0.01)
468           mult_prec = 24 - 15 * value/3;
469         else
470           mult_prec = 24;
471     }
472     else {
473       if (value < -3)
474         mult_prec = 9;
475       else
476         if (value < -0.01)
477           mult_prec = 24 + 15 * value/3;
478         else
479           mult_prec = 24;
480     }
481     /*****************************************************************************/
482     /* EXIT SetPrecision                                                         */
483     /*****************************************************************************/
484   }
```

Appendix B

Parallel Implementations of the Error Back–Propagation Learning in *InterAct*

B.1 Error Back–Propagation in *InterAct*

For the development and training of neural networks we use *InterAct*. With–in *InterAct* a wide range of standard calls are available. For example calls for creating a network, adding or deleting a neuron, choosing a learn–rule, initializing a network, evaluating a network, learning a network, and many more. Furthermore it's possible to start some observations to visualize how a network is learning.

If we have a network, this network can be trained, according to the original (sequential) back–propagation algorithm, using the following *InterAct*–calls:

Forward pass:

- `set_Sinput(inputs, nr_inputs, &status)`
  This function sets the input–patterns, from the array `inputs`, on the input neurons.
- `calc_Start_evalu(0L, hidden_Slist, &status)`
  This function evaluates the network for the specified neurons. In this case it updates the values of the hidden neurons (`hidden_Slist`) according to equations (3.1) and (3.2).
- `calc_Start_evalu(0L, output_Slist, &status)`
  The same as the previous function, but now the values of the output neurons are updated.

Backward pass:

- `set_Starget(targets, nr_targets, &status)`
  This function calculates the errors of the output neurons, using the targets specified in the array `targets`. Further all errors of the other neurons are set to zero.
- `calc_Start_learn(0L, output_Slist, &status)`
  This function performs a learn–cycle for the specified neurons. In this case all weights on the incoming synapses of the output neurons (`output_Slist`) are adapted, according to equation (3.10). Further the errors of these neurons are propagated back through

the network to the neurons in the previous layer. This means that the errors of these last neurons are raised accordingly.

- `calc_Start_learn(0L, hidden_Slist, &status)`
  The same as the previous function, but now the weights of the incoming synapses of the hidden neurons are adapted.

These calls take for granted that the neurons are updated layer by layer, which is usually the case by error back–propagation. But if we want to change the computational order in which the neurons are updated, we find some difficulties with this implementation.

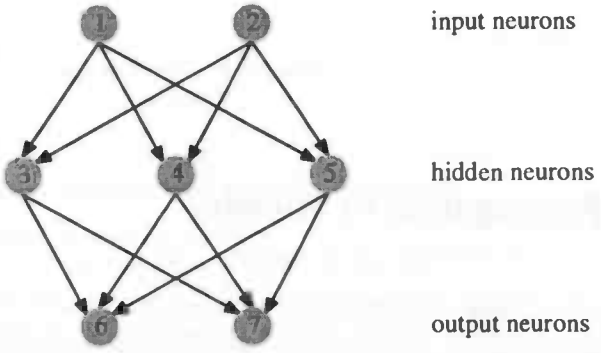


Figure B.1: *A 2–3–2 feedforward neural network.*

These difficulties will be explained by the following example. Assume we have a 2–3–2 feedforward neural network as shown in figure B.1. The main problem lies in the fact that when the call `set_Starget` is performed, the errors of neurons 3, 4 and 5 are set to zero. Normally, when using the "sequential" procedure, this is not a problem but even a necessity. Namely, if we perform `calc_Start_learn` for neuron 6, the function performs two major task:

- The weights on the incoming synapses of neuron 6 are updated using the error of neuron 6 (see equation (3.10)).
- The errors of neurons 3, 4 and 5 are raised according to the back–propagated error of neuron 6 (see equation (3.9)).

If this is repeated for neuron 7, the errors of neurons 3, 4 and 5, will get the right values. (Using the "sequential" procedure this is repeated, because we perform it for the whole output list `calc_Start_learn(0L, output_Slist, &status)`.)

If we use the "parallel" procedure, that is perform some updates in parallel, the above does give a problem. When we for example want to perform `calc_Start_learn` for neurons 7, 6 and 5 in parallel, the error of neuron 5 is become zero. While want we to find an "old" error. So we would update the weights by using an error with value zero, instead of updating it with an "old" value.

B.2 Parallel *InterAct*–Implementation of the "Original" Version

As mentioned in section B.1 adapting the learning–procedure to a parallel version gives some problem. We can solve these problems by adapting the procedure in the following way. Hereby we assume that the forward pass has been performed.

B.2.1 Description of the Source Code

The original version is implemented as presented below. The source code can be found in section B.2.2. The line numbers refer to the numbers in this section. The source code is direct implementation of the algorithm as presented in section 3.2.4.

1. Copy the errors of all neurons and store them in the array `old_error` (lines 19–20).
2. Initialize the array `error_used` to zero. This array will be used to check whether an error is already propagated back or not (line 23).
3. Compute the errors of the output neurons, using the targets specified in the array `targets` and set all errors of the other neurons are to zero (line 27). This is done by the *InterAct*–call `set_Starget`.
4. If the error is not a new error, that is if the error is zero or partially changed, than we will use the old error. For this we use the old error of the neuron, which is stored in `old_error` (lines 42–62).
5. Compute the error signal of the neuron and store the result in a temporary variable (lines 64–69).
6. Compute for all feeding neurons the contribution to the error of these neurons and compute the update for the synapses. Store these results in temporary variables (lines 71–98).
7. Compute the new bias and store this in a temporary variable (lines 100–105).
8. Repeat steps **4**, **5**, **6** and **7** for all neurons that should be performed in parallel (line 35–108).
9. Copy the temporary variables to the real variables (line 111–128).
10. Set flag `error_used`. This means that the error of this neuron has been propagated back (lines 129–130).
11. Repeat steps **9** and **10** for the same neurons that are performed in parallel in step **8** (line 110–132).
12. Steps **8**, **9**, **10** and **11** should be repeated for all groups of neurons that should be performed in parallel (line 33–133).

B.2.2 Source Code of the Original Version

```
1    void OriginalParallel (void)
2    {
3      neuron_Sid_t       i, j, output_info[NR_NEURONS];
4      get_Sinput_info_t  input_info[NR_NEURONS];
```

```
5       out_St              bpe, y, delta, tempbpe[NR_NEURONS + 1];
6       long                nrpar, nr, nr_i, nr_outputs, nr_inputs;
7       weight_St           w, deltaw,
8                           tempw[NR_NEURONS + 1][NR_NEURONS + 1];
9       bias_St             bias, deltabias;
10      int                 newerror;
11
12
13      /* backward pass */
14
15      /* initialization */
16      for (i = 1; i <= NR_NEURONS; i++) {
17
18        /* olderror i = bpe i */
19        get_Sneuron_error(i, &bpe, &status);
20        olderror[i] = bpe;
21
22        /* errorused = false */
23        errorused[i] = 0;
24      }
25
26      /* bpe i = y - d */
27      set_Starget(outputs, NR_OUTPUTS, &status);
28
29      /* learn */
30      /* get last neuron */
31      i = NR_NEURONS;
32      /* main loop: repeat until input layer is reached */
33      while (!IsInput(i)) {
34        /* pseudo parallel loop */
35        for (nrpar = 0; (nrpar<N) && (!IsInput(i)); nrpar++) {
36
37
38          /* if a neuron is not an output neuron check it's
39             error */
40          if (!IsOutput(i)) {
41
42            /* newerror = true */
43            newerror = 1;
44            get_Sneuron_outputs(i, output_info, &nr_outputs,
45                                &status);
46
47            for (nr = 0; nr < nr_outputs; nr++) {
48              j = output_info[nr];
49              if (!errorused[j]) {
50
51                /* newerror = false */
52                newerror = 0;
53              }
54            }
55            /* if the error is not totally updated use old
56               error */
57            if (!newerror) {
58              /* bpe i = olderror i */
59              set_Serror_random(i, 0, olderror[i],
60                                olderror[i], olderror[i], 1.0,
61                                rand_Sdummy, &status);
62            }
63          }
64          get_Sneuron_error(i, &bpe, &status);
65          get_Sneuron_output(i, &y, &status);
66
67
68          /* delta i = y i * (1 - y i) * bpe i */
69          delta = y * (1 - y) * bpe;
70
71          /* get feeding neurons */
72          get_Sneuron_inputs(i, input_info, &nr_inputs,
73                             &status);
74          for (nr = 0; nr < nr_inputs; nr++) {
75            j = input_info[nr].from_id;
76            get_Sneuron_error(j, &bpe, &status);
77            w = input_info[nr].weight;
78
```

```
79          /* tempbpe j = bpe j + delta i * w ij */
80          if (nrpar == 0) {
81            /* for every neuron that is considered as first
82               parallel: tempbpe j = bpe j + delta i * w ij*/
83            tempbpe[j] = bpe + delta * w;
84          }
85          else {
86            /* for all other neurons:
87               tempbpe j = tempbpe j + delta i * w ij */
88            tempbpe[j] = tempbpe[j] + delta * w;
89          }
90          deltaw = w - oldw[i][j];
91          oldw[i][j] = w;
92          get_Sneuron_output(j, &y, &status);
93
94          /* tempw ij = w ij + alpha * deltaw +
95             eta * delta i * y j */
96          tempw[i][j] = w + LEARN_MOMENTUM * deltaw +
97            LEARN_RATE * delta * y;
98        }
99
100       /* biases */
101       get_Sneuron_bias(i, &bias, &status);
102       deltabias = bias - oldw[i][0];
103       oldw[i][0] = bias;
104       tempw[i][0] = bias + LEARN_MOMENTUM * deltabias +
105         LEARN_RATE * delta;
106
107       i--;
108     }
109     i = i + nrpar;
110     for (nr = 1; nr <= nrpar; nr++) {
111       /* get feeding neurons */
112       get_Sneuron_inputs(i, input_info, &nr_inputs,
113                          &status);
114       for (nr_i = 0; nr_i < nr_inputs; nr_i++) {
115         j = input_info[nr_i].from_id;
116         /* bpe j = tempbpe j */
117         set_Serror_random(j, 0, tempbpe[j], tempbpe[j],
118                           tempbpe[j], 1.0, rand_Sdummy,
119                           &status);
120
121         /* w ij = tempw ij */
122         set_Sweight_random(j, i, 0, 0, 1, tempw[i][j],
123                            tempw[i][j], tempw[i][j], 1.0,
124                            rand_Suniform, &status);
125       }
126       set_Sbias_random(i, 0, tempw[i][0], tempw[i][0],
127                        tempw[i][0], 1.0, rand_Sdummy,
128                        &status);
129       /* errorused i = true */
130       errorused[i] = 1;
131       i--;
132     }
133   }
134 }
```

B.3 Parallel *InterAct*-Implementation of the "Special" Version

In this section the implementation of the special version is presented. Instead of updating incoming synapses, this version updates outgoing synapses. Again, we assume that the forward pass has been performed.

B.3.1 Description of the Source Code

The special version is implemented as presented below. The source code can be found in section B.3.2. The line numbers refer to the numbers in this section. The source code is direct implementation of the algorithm as presented in section 3.7.2.

1. Compute the error of the current neuron (lines 29 and 33–48).
2. For hidden neurons compute the weight updates for the outgoing synapses and store the results in temporary variables (lines 51–59).
3. Compute the error signal of the current neuron and store this in a temporary variable (lines 67).
4. Compute the new bias and store this in a temporary variable (lines 69–74).
5. Repeat steps **1, 2, 3** and **4** for all neurons that should be performed in parallel (line 21–79).
6. Copy the temporary variables to the real variables (line 111–128).
7. Repeat step **6** for the same neurons that are performed in parallel in step **5** (line 87–114).
12. Finally steps **5** , **6** and **7** should be repeated for all groups of neurons that should be performed in parallel (line 19–115).

B.3.2 Source Code of the Special Version

```
1   void SpecialParallel (void)
2   {
3     neuron_Sid_t            j, k, output_info[NR_NEURONS];
4     out_St                  e, y, delta,
5                             tempdelta[NR_NEURONS + 1];
6     long                    nrpar, nr, nr_o, nr_outputs;
7     get_Sconnection_info_t synapse;
8     weight_St               w, deltaw,
9                             tempw[NR_NEURONS + 1][NR_NEURONS + 1];
10    bias_St                 bias, deltabias;
11    neuron_Sid_t            first_output;
12
13    get_Sid_neuron (get_Soption_first, 0L, output_Slist,
14                    NULL, &first_output, &status);
15    /* backward pass */
16    /* get last neuron */
17    j = NR_NEURONS;
18    /* main loop: repeat until input layer is reached */
19    while (j >= 1) {
20      /* pseudo parallel loop */
21      for (nrpar = 0; (nrpar < N) && (j >= 1); nrpar++) {
22
23        /* get y j */
24        get_Sneuron_output(j, &y, &status);
25
26        if (IsOutput(j)) {
27          /* compute the error of neuron j */
28          /* Get proper index in output array */
29          e = outputs[j - first_output] - y;
30        }
31        else {
32          /* compute the error of neuron j */
33          e = 0.0;
34          get_Sneuron_outputs(j, output_info, &nr_outputs,
35                              &status);
36          for (nr_o = 0; nr_o < nr_outputs; nr_o++) {
37            k = output_info[nr_o];
38
39            /* get delta k */
40            get_Sneuron_error(k, &delta, &status);
41
42            /* get w kj */
43            synapse.to_id = k;
44            synapse.from_id = j;
45            synapse.to_site = site_Sdefault_site;
46            get_Sinfo_connection(&synapse, &status);
```

```
47            w = synapse.weight;
48            e = e + delta * w;
49
50            /* compute deltaw kj */
51            deltaw = w - oldw[k][j];
52
53            /* store 'old' w kj */
54            oldw[k][j] = w;
55
56            /* compute the new weight and store the new
57               weight in a temporary variable */
58            tempw[k][j] = w + LEARN_MOMENTUM * deltaw +
59              LEARN_RATE * delta * y;
60
61          }
62        }
63
64        if (!IsInput(j)) {
65          /* store the error signal delta in a temporary
66             variable */
67          tempdelta[j] = y * (1 - y) * e;
68
69          /*bias*/
70          get_Sneuron_bias(j, &bias, &status);
71          deltabias = bias - oldw[j][0];
72          oldw[j][0] = bias;
73          tempw[j][0] = bias + LEARN_MOMENTUM * deltabias +
74            LEARN_RATE * tempdelta[j];
75        }
76
77        /* get previous neuron */
78        j--;
79      } /* pseudo parallel loop */
80
81
82      /* no "nrpar" denotes the number of neurons that are
83         updates pseudo-parallel, these "nrpar" neurons
84         should be really updated, so copy the temporary
85         variables to the real variables */
86      j = j + nrpar;
87      for (nr = 1; nr <= nrpar; nr++) {
88        if (!IsInput(j)) {
89          /* copy delta */
90          set_Serror_random(j, 0L, tempdelta[j],
91                            tempdelta[j], tempdelta[j], 1.0,
92                            rand_Sdummy, &status);
93
94          /* copy bias */
95          set_Sbias_random(j, 0L, tempw[j][0], tempw[j][0],
96                           tempw[j][0], 1.0, rand_Sdummy,
97                           &status);
98        }
99
100       if (!IsOutput(j)) {
101         /* copy weights */
102         get_Sneuron_outputs(j, output_info, &nr_outputs,
103                             &status);
104         for (nr_o = 0; nr_o < nr_outputs; nr_o++) {
105           k = output_info[nr_o];
106           set_Sweight_random(j, k, 0, 0,
107                              site_Sdefault_site,
108                              tempw[k][j], tempw[k][j],
109                              tempw[k][j], 1.0, rand_Sdummy,
110                              &status);
111         }
112       }
113       j--;
114     }
115   }/* main loop */
116 /* end of backward pass */
117 }
```

Appendix C

Implementation of Using Less Accuracy in Error Back–Propagation

C.1 Description of the Source Code

Here will describe the implementation of the flexible accuracy adaptation as presented in C.2. To use less accuracy, the function `HW__Multiply` (lines 1–60) stops its iteration if the desired accuracy is reached (lines 53–54).

In the two parallel methods described before, every multiplication `a*b`, will be replaced with `Multiply(a, b)`. This function performs the `HW__Multiply` and thus performs a multiplication with a certain accuracy. Furthermore it computes the number of iterations and multiplications (lines 63–73).

The function `f(x)` performs the appropriate function for x (lines 76–86).

The function `get_adaptation` computes the variables a and b by solving `y = a f(x)+b` for (`min_acc`, `error_init`) and (`max_acc + 1`, `error_des`) (lines 89–109).

The function `adapt` computes `y = a f(x)+b` (lines 111–115).

In lines 120–143 we check whether the accuracy needs to be adapted or not. After the first 25 cycles, we compute `a` and `b` according to `get_adaptation` (lines 124–129). After all other multiples of 25 we check according to `adapt` whether we need to raise the accuracy. If this is needed we raise the accuracy with `flex_step` (lines 131–138).

C.2 Source Code

```
1   double HW__multiply(float a, float b, short *it,
2                       short mit, short prec, short frl)
3   /*_____________________________________________________________*/
4   /* ACTION:  multiplies A with B by the usual series/          */
5   /*          parallel principle                                */
6   /*  a * b= 2^(j+k) + 2^k * oa          for all '1' in b      */
7   /*  wherein a= 2^j + oa and b= 2^k + ob                       */
8   /*  taking 24-bits inputs and leaving a 49-bit result.        */
9   /*                                                            */
```

```
10  /* INPUT:  float  a    >  multiplicand value                    */
11  /*         float  b    >  multiplier value                      */
12  /*         short  mit  >  maximum number of iterations          */
13  /*         short  prec >  max. # bits in result w.r.t.          */
14  /*                        leading '1'                           */
15  /*         short  frl  >  finite register length                */
16  /*                        (inoperative if 32)                   */
17  /*                                                              */
18  /* OUTPUT: short  *it  >  number of iterations                  */
19  /*         double ()   >  the result of the                     */
20  /*                        multiplication                        */
21  /*                                                              */
22  /* REMARK:  An IEEE-standard floating-point number has         */
23  /*          a 23-bits mantissa, but the leading '1' is          */
24  /*          not explicitly stored. This brings the range        */
25  /*          to 24-bits.                                         */
26  /*--------------------------------------------------------------*/
27  {
28    /* default parameter settings ..........................*/
29    result = *it = 0; factor = sign = 1;
30
31    /* make a and b positive and save sign of the            */
32       multiplication result ..............................*/
33    if (a < 0) { a = -a; sign = -sign; }
34    if (b < 0) { b = -b; sign = -sign; }
35
36    /* separate exponent and mantissa for both the numbers   */
37    while (a < constant1 && a != 0) { a *= 2; factor *= 2; }
38    while (b < constant1 && b != 0) { b *= 2; factor *= 2; }
39    if ( (A = (int) a) == 0 || (B = (int) b) == 0) return 0;
40
41    /* multiply according to the USUAL recursive formula .....*/
42    for (; A !=0 && B !=0 && *it < mit;)
43      {   first(A, &fA); A -= (1 << fA);
44          if ( (shA = fA - frl) < 0) shA = 0;
45          first(B, &fB); B -= (1 << fB);
46          result += interm =
47            (double) (1 << fA) * (double) (1 << fB) +
48              (double) (1 << fB) * ((A >> shA) << shA);
49          A += (1 << fA); (*it)++;
50
51          /* break on exceeding the required computational
52             precision ..................................*/
53          fR = inspect(result); fI = inspect(interm);
54          if ( (fR - fI) > prec) { result -= interm; break; }
55      }
56  /*************************************************************/
57  /* EXIT HW__multiply                                         */
58  /*************************************************************/
59    return ((double) ((sign * result))/factor);
60  }
61
62
63  out_St Multiply(float a, float b)
64  {
65    double        result;
66    short         it;
67
68    result = HW__multiply(a, b, &it, 24, ACCURACY, 32);
69    nr_it += it;
70    nr_mul ++;
71
72    return((out_St) result);
73  }
74
75
76  float f(int x)
77  {
78    if (FLEX_VERSION == 3) {
79      return(1 / (float) (2 << (x-1)));
80    }
81    else if (FLEX_VERSION == 2) {
82      return(1 / (float) x);
83    }
```

```
84      else
85        return((float) x);
86    }
87
88
89    void get_adaptation(int min_acc, float error_init,
90                        int max_acc, float error_des,
91                        float *a, float *b)
92          /* solve y = a f(x) + b for (min_acc, error_init) and
93             (max_acc + 1, error_des) */
94
95    {
96      float min_acc_f;
97      float max_acc_f;
98
99
100     max_acc ++;
101
102     min_acc_f = f(min_acc);
103     max_acc_f = f(max_acc);
104
105
106     (*a) = (error_des - error_init) /
107       (max_acc_f - min_acc_f);
108     (*b) = error_init - (*a) * min_acc_f;
109   }
110
111   float adapt(float a, float b, int x)
112   {
113     /* compute y = a f(x) + b */
114     return(a * f(x) + b);
115   }
116
117
118   ...
119   ...
120   if (cycles % 25 == 0) {
121     ...
122     ...
123     if (flexible) {
124       if (first_time) {
125         get_adaptation(ACCURACY,
126                        train_error[0] / nr_train_patterns,
127                        24, MIN_ERROR, &a, &b);
128         first_time = 0;
129       }
130       else {
131         if (adapt(a, b, ACCURACY + flex_step) >
132             train_error[0] / nr_train_patterns) {
133           if (ACCURACY < 24) {
134             ACCURACY += flex_step;
135             printf("changing the accuracy to %d\n",
136                    ACCURACY);
137           }
138         }
139       }
140     }
141     ...
142     ...
143   }
144   ...
145   ...
```