



On the Road again  
Moving a Neural Net in Hardware

Henrickus M. G. Ter Haseborg

July 1999

---

Abstract



**On the Road again:**

***Moving a Neural Net in Hardware***

by : Ter Haseborg, Henrickus M. G.

19-07-1999.



At The department of computing Science

Rijksuniversiteit Groningen,

Groningen, The Netherlands,

July, 1999.

Supervisors:

Prof. Dr. Ir. L. Spaanenburg

Dr. Ir. J. A. G. Nijhuis

A thesis submitted in fulfillment of the requirements for  
the degree Master of Science  
at the Rijksuniversiteit Groningen

---

---

# Abstract

---

Artificial neural networks are a new and promising generation of information processing systems. In the last couple of years they have shown to outperform classical algorithms in such areas as pattern recognition, image processing and data clustering. For several (notably embedded) applications the requirements for physical size, power consumption and especially raw speed dictate the use of specific hardware realisations. However, the transformation of a neural network into a hardware platform introduces a number of new problems, as studied in this thesis.

A structured design approach is based on three phases: architecture, implementation and realisation. In this thesis we focus especially on the first two of these three phases. The architecture creates a description (if not specification) of the neural network behaviour, while the implementation transforms this description into a buildable model, that is optimal for the envisaged realisation technology. In this thesis the implementation will be given in the VHDL hardware description language. This VHDL description is a widely accepted basis for system simulation and synthesis, which facilitates a comfortable parameterisation and test of a buildable model.

A number of transformations are required for the optimal technology mapping of a neural system, of which the current literature does not give a clear evaluation. Therefore this thesis presents a number of fundamental experiments to find the actual degree of freedom in the design space. The performance of a neural network is dependent on the envisaged application (such as function approximation or classification). It is generally anticipated that the representation of the discriminatory function (sigmoid) will have a major impact. Our research has shown that the impact of the sigmoid representation is unmistakably present; however, this impact is not dependent on the application area. Rather will the "area of effectiveness" be dimensioned on basis of the envisaged usage.

This thesis also pays attention to the impact of number representation techniques. We have focussed especially on the finite wordlength effects that will be encountered in ASIC technology. It is found that rounding is the best technique for mapping arbitrary integers on finite-length computer words. Further, we have no empirical evidence that restricting the representation of the input signals has a major effect on the system performance, not even by rounding. These and related results have been used in the design and implementation of a neural network ASIC.

---

# Abstract

The first part of the paper is devoted to a review of the existing literature on the topic. It is found that the majority of the studies have focused on the role of the state in the provision of social services. However, there is a growing body of research that suggests that the private sector can also play a significant role in this regard. This paper aims to explore the potential of the private sector in the provision of social services, with a particular focus on the role of non-profit organizations.

The second part of the paper presents a theoretical framework for the analysis. It is argued that the provision of social services is a complex task that requires a combination of public and private resources. The paper develops a model that takes into account the different types of social services and the different actors involved in their provision. The model is then used to analyze the role of the private sector in the provision of social services, with a particular focus on the role of non-profit organizations.

The third part of the paper presents the empirical analysis. It is based on a survey of non-profit organizations that provide social services. The survey covers a wide range of issues, including the organization's mission, its sources of funding, and its methods of service provision. The results of the survey are then used to test the theoretical framework developed in the second part of the paper. It is found that the private sector, and in particular non-profit organizations, can play a significant role in the provision of social services. However, this role is often limited by a number of factors, including the availability of funding and the capacity of the organizations to provide services.

The final part of the paper presents the conclusions and policy implications of the study. It is concluded that the private sector, and in particular non-profit organizations, can play a significant role in the provision of social services. However, this role is often limited by a number of factors, including the availability of funding and the capacity of the organizations to provide services. The paper suggests a number of policy measures that could be taken to support the role of the private sector in the provision of social services, including the provision of funding and the development of a regulatory framework.

---

# Samenvatting

---

Kunstmatige neurale netwerken zijn een nieuwe en veelbelovende generatie van informatie-verwerkende systemen. Ze hebben in de afgelopen jaren bewezen goed te presteren op gebieden in patroonherkenning, beeldverwerking en data clustering, waar conventionele algoritmes aan hun grenzen gekomen zijn. Voor een aantal embedded toepassingen zijn specifieke hardware oplossingen gewenst. De afbeelding van een neurale netwerk op platforms met begrensde representatie leidt tot een aantal nieuwe problemen, waaraan in dit afstudeerwerk aandacht gegeven is.

Het ontwerptraject bestaat uit een drietal fasen, te weten architectuur, implementatie en realisatie. In dit rapport wordt alleen aandacht gegeven aan de eerste twee van de genoemde fasen. De architectuur geeft een beschrijving van het gedrag van het neurale systeem, terwijl de implementatie de werkelijke systeemopzet beschrijft zoals die voor de beoogde realisatie techniek gewenst is. In dit rapport wordt voor de implementatie de hardware beschrijvingstaal VHDL gebruikt. Deze VHDL beschrijving is simuleerbaar en synthetiseerbaar zodat de feitelijke functionaliteit getest en gedimensioneerd kan worden.

Om het systeem optimaal te dimensioneren is een aantal aanpassingen noodzakelijk, waarvan in de literatuur de precieze uitwerking nog niet bekend is. Daarom zijn diverse experimenten doorgevoerd om een beter inzicht te verschaffen in de ontwerp vrijheid. De prestaties van het hardware systeem zijn afhankelijk van het toepassingsgebied van het neurale netwerk (b.v. functie approximatie en classificatie). Daarbij wordt een grote rol toegedacht aan de representatie van de beslisfunctie (sigmoïd). Uit het onderzoek blijkt echter dat de invloed van de sigmoïd representatie weliswaar groot is, maar onafhankelijk van het toepassingsgebied. Deze komt eerder tot uiting in de keuze van het werkgebied, waarbinnen de sigmoïd van toepassing is.

Verder is aandacht geschonken aan diverse technieken ter beperking van de getalsrepresentatie. Bij de hardware afbeelding zullen getallen opgeslagen moeten worden in woorden met een beperkte breedte. Het blijkt dat een implementatie op basis van "rounding" de beste resultaten geeft. Verder lijken afrondingen van de ingangssignalen slechts zeer beperkt van invloed te zijn op de prestatie van het neurale netwerk. Op basis van deze resultaten wordt tenslotte een implementatie van een neurale netwerk volledig uitgewerkt en middels simulatie geverifieerd.

# Zamenyavning

...

...

...

...

---

## Abbreviations and Symbols

---

### Abbreviations

ANN	Artificial Neural Network
MLP	Multi Layer Perceptron
IO	Input – Output
ROM	Read Only Memory
RAM	Random Access Memory
DoD	Department of Defence
Err <sub>abs</sub>	Absolute error
Err <sub>ems</sub>	the mean squared error
Err <sub>rms</sub>	the root mean squared error
VHDL	VHSIC (Very High speed Intergrated Circuits) Hardware Decription Language

### Important Symbols

$x_i(n)$	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 that the $n$ -th pattern is presented.
$\varphi_i^{(l)}$	the activation function associated with neuron $i$ in layer $l$ .
$\delta_i^{(l)}(n)$	the local gradient 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.
$\eta$	the learning rate parameter.

# Table of Contents

iii

Chapter 1: Introduction	1
1.1 The Role of the Engineer	1
1.2 The Engineering Profession	2
1.3 The Engineering Code of Ethics	3
1.4 The Engineering Process	4
1.5 The Engineering Society	5
Chapter 2: Fundamentals of Engineering	6
2.1 The Engineering Process	6
2.2 The Engineering Design Process	7
2.3 The Engineering Design Process	8
2.4 The Engineering Design Process	9
2.5 The Engineering Design Process	10
2.6 The Engineering Design Process	11
2.7 The Engineering Design Process	12
2.8 The Engineering Design Process	13
2.9 The Engineering Design Process	14
2.10 The Engineering Design Process	15
2.11 The Engineering Design Process	16
2.12 The Engineering Design Process	17
2.13 The Engineering Design Process	18
2.14 The Engineering Design Process	19
2.15 The Engineering Design Process	20
2.16 The Engineering Design Process	21
2.17 The Engineering Design Process	22
2.18 The Engineering Design Process	23
2.19 The Engineering Design Process	24
2.20 The Engineering Design Process	25
2.21 The Engineering Design Process	26
2.22 The Engineering Design Process	27
2.23 The Engineering Design Process	28
2.24 The Engineering Design Process	29
2.25 The Engineering Design Process	30
2.26 The Engineering Design Process	31
2.27 The Engineering Design Process	32
2.28 The Engineering Design Process	33
2.29 The Engineering Design Process	34
2.30 The Engineering Design Process	35
2.31 The Engineering Design Process	36
2.32 The Engineering Design Process	37
2.33 The Engineering Design Process	38
2.34 The Engineering Design Process	39
2.35 The Engineering Design Process	40
2.36 The Engineering Design Process	41
2.37 The Engineering Design Process	42
2.38 The Engineering Design Process	43
2.39 The Engineering Design Process	44
2.40 The Engineering Design Process	45
2.41 The Engineering Design Process	46
2.42 The Engineering Design Process	47
2.43 The Engineering Design Process	48
2.44 The Engineering Design Process	49
2.45 The Engineering Design Process	50
2.46 The Engineering Design Process	51
2.47 The Engineering Design Process	52
2.48 The Engineering Design Process	53
2.49 The Engineering Design Process	54
2.50 The Engineering Design Process	55
2.51 The Engineering Design Process	56
2.52 The Engineering Design Process	57
2.53 The Engineering Design Process	58
2.54 The Engineering Design Process	59
2.55 The Engineering Design Process	60
2.56 The Engineering Design Process	61
2.57 The Engineering Design Process	62
2.58 The Engineering Design Process	63
2.59 The Engineering Design Process	64
2.60 The Engineering Design Process	65
2.61 The Engineering Design Process	66
2.62 The Engineering Design Process	67
2.63 The Engineering Design Process	68
2.64 The Engineering Design Process	69
2.65 The Engineering Design Process	70
2.66 The Engineering Design Process	71
2.67 The Engineering Design Process	72
2.68 The Engineering Design Process	73
2.69 The Engineering Design Process	74
2.70 The Engineering Design Process	75
2.71 The Engineering Design Process	76
2.72 The Engineering Design Process	77
2.73 The Engineering Design Process	78
2.74 The Engineering Design Process	79
2.75 The Engineering Design Process	80
2.76 The Engineering Design Process	81
2.77 The Engineering Design Process	82
2.78 The Engineering Design Process	83
2.79 The Engineering Design Process	84
2.80 The Engineering Design Process	85
2.81 The Engineering Design Process	86
2.82 The Engineering Design Process	87
2.83 The Engineering Design Process	88
2.84 The Engineering Design Process	89
2.85 The Engineering Design Process	90
2.86 The Engineering Design Process	91
2.87 The Engineering Design Process	92
2.88 The Engineering Design Process	93
2.89 The Engineering Design Process	94
2.90 The Engineering Design Process	95
2.91 The Engineering Design Process	96
2.92 The Engineering Design Process	97
2.93 The Engineering Design Process	98
2.94 The Engineering Design Process	99
2.95 The Engineering Design Process	100

# Table Of Contents

---

<b>Abstract .....</b>	<b>i</b>
<b>Samenvatting .....</b>	<b>iii</b>
Abbreviations and Symbols .....	v
<b>Introduction .....</b>	<b>1</b>
1.1 Goals .....	1
1.2 Historical notes .....	2
1.3 Content and Organization .....	5
<b>Chapter 2 : The functional behavior of an artificial neural network .....</b>	<b>7</b>
2.1 Fundamental Concepts of Artificial Neural Networks .....	7
2.1.1 A Neuron Model .....	8
2.1.2 The connections .....	9
2.1.3 Learning rule .....	10
<b>Chapter 3 : The basic concept of hardware design .....</b>	<b>15</b>
3.1 Design concept .....	15
3.2 Architectural Level .....	16
3.2.1 The Functional Behavior .....	17
3.2.2 The Conceptual Structure .....	19
3.3 Implementation .....	22
3.3.1 A Neural System .....	23
3.3.2 Interface Module .....	23
3.3.2.1 Input Behavior .....	24
3.3.2.2 Output Behavior .....	25
3.3.3 Memory Module .....	25
3.3.4 Processing Unit .....	26
3.4 Description Verification .....	27
3.4.1 IO interface .....	27
3.4.2 Memory .....	29
3.4.3 Processing Unit .....	30
3.5 Realization .....	31
3.6 Summary .....	31

<b>Chapter 4 : Tools used before Simulation of the Neural System. ....</b>	<b>33</b>
4.1 Simulations .....	33
4.1.1 Extraction Network Characteristics .....	34
4.1.2 A Simulation Environment .....	35
4.2 Diagnostic Checking .....	36
4.2.1 Goodness of fit .....	36
4.2.2 Performance .....	39
 <b>Chapter 5 : The discriminatory function with an area of effectiveness .....</b>	<b>41</b>
5.1 Activation function implementation .....	42
5.2 An Effective Operating Area .....	43
5.2.1 Experiments .....	44
5.2.1.1 Sine Function .....	45
5.2.1.1.1 Results .....	45
5.2.1.1.2 Analysis of the Results .....	47
5.2.1.2 Exclusive-OR .....	49
5.2.1.2.1 Results .....	49
5.2.1.2.2 Analysis of the Results .....	50
5.2.1.3 A Valve .....	52
5.2.1.3.1 Analysis of the Results .....	52
5.2.1.4 The Iris Classifier .....	53
5.2.1.4.1 Analysis of the Results .....	53
5.2.1.5 Summary .....	54
5.2.2 The Discussion .....	54
5.3 Reduction of Accuracy .....	56
5.3.1 Experimental results .....	56
5.3.1.1 Results Sinewave function .....	56
5.3.1.2 Results Iris classifier .....	57
5.3.2 Discussion .....	60
5.4 Conclusions and recommendations .....	62
 <b>Chapter 6 : An evaluation of the effects of the error generation and propagation. ....</b>	<b>63</b>
6.1 Sources of Quantization Errors .....	63
6.1.1 Rounding techniques .....	64
6.1.2 Jamming techniques .....	65
6.1.3 Truncation techniques .....	66
6.2 The Influence of Rounding Techniques .....	66
6.2.1 The Experiments .....	66
6.2.1.1 Experimental results of the sine function .....	67
6.2.1.2 Experimental results of the Iris classifier .....	70
6.2.2 Discussion .....	73
6.3 Addition of Noise .....	74
6.3.1 The Experiments .....	74
6.3.1.1 Results by Addition of Noise at point IV .....	75
6.3.1.2 Results by Addition of Noise at point III .....	77
6.3.2 Discussion .....	78
6.4 Conclusions and Recommendations .....	80

<b>Chapter 7 : Conclusions and recommendations</b> .....	<b>81</b>
<b>Acknowledgement</b> .....	<b>83</b>
<b>References</b> .....	<b>85</b>
<b>Appendix A : The arithmetic principles of Logarithms</b> .....	<b>87</b>
A.1 The idea of the logarithm .....	87
<b>Appendix B : The Hardware Neural System Description</b> .....	<b>89</b>
B.1 Neural System description .....	89
B.1.1 Interface module .....	90
B.1.1.1 The Interface Controller .....	92
B.1.2 Memory module .....	94
B.1.3 Processing unit .....	96
B.1.3.1 Repeated Adder .....	100
B.1.3.2 Digilog Multiplier .....	103
B.1.3.3 Sequence Controller .....	108
B.1.3.4 Processing Units Controller .....	110
B.1.3.5 Bias Array .....	112
B.1.3.6 Synaptic Weight Array .....	114
B.1.3.7 Discriminatory Function .....	116
B.1.4 Main Controller .....	116
<b>Appendix C : The Generation Tools and Simulation Environments</b> .....	<b>119</b>
C.1 Characteristic Extraction Tool .....	119
C.1.1 An Example .....	129
C.2 An Example of a Simulation Environment .....	131
C.3 Generation Tool Discriminatory Function .....	135
C.3.1 A generated Discriminatory function .....	139
<b>Appendix D : The experimental result of the Valve and Iris problem</b> .....	<b>141</b>
D.1 The experimental Results of the Valve problem .....	141
D.2 Iris classifier results .....	143

---

# Introduction

---

For several millennia, humans have tried for several centuries to understand natural phenomena. Starting from a single “natural science” and later in the off-spring disciplines known as biology, physics and chemistry, one succeeded to construct mathematical models of some phenomena. This research makes it possible to understand what’s happening around us. One interesting thing is, that only a century ago a model has been proposed to understand the brain, i.e. the biological neuron. This model makes it possible to formulate and simulate a small network of neurons. These simulated neurons are known as artificial neurons. Men has discovered that neurons are getting more powerful when they are connected to each other. The network structure of neurons (or topology), is able to encapsulate knowledge by use of a learning rule. When such a network is trained, the learned knowledge can be recalled on short notice. These networks of artificial neurons are called artificial neural networks (ANNs).

The artificial neural networks as we know nowadays are systems that are deliberately constructed to make use of some organizational principles resembling those of the human brain. They represent a promising new generation of information processing systems. ANNs have proven over time that they are good at tasks such as pattern matching, vector quantization, and data clustering, while traditional computers are inefficient at these tasks. However, the traditional computers as known nowadays are faster in algorithmic computational tasks and precise arithmetic operations. The reason for this phenomenon lies within the architecture of these systems. Thus, there must be a way to combine the enormous computational power of a computer and the advantages of ANNs in hardware.

## 1.1 Goals

Different researchers have influenced the development of *artificial neural networks*. Still, a lot of research is needed to obtain the ideal representation of ANNs. This report discusses the experiments performed and the solutions found in developing such a representation. The primary goal has been to investigate the possibility to represent a feedforward neural network in a given hardware architecture.

This hardware device holding an ANN can be used in real-world applications. These real-world applications demand several properties of a hardware implementation, such as: high speed, high accuracy, small area, real-time response, embeddable within other systems, on-chip learnable, etc. Lots of research must be done to create such a hardware device.

In the past, a first impulse has been given to construct a hardware architecture named GREMLIN [1]. This architecture is a DSP-like (Digital Signal Processor) architecture that aims at the integration of data pre-processing and network emulation. Back propagation learning can be an option for adaptivity. This architecture can be used to emulate feedforward multilayer Perceptron networks. My goal is to develop a VHDL (Very high speed integrated circuit Hardware Description Language) description for this architecture. Another part of my research is to take a look at the calculation accuracy and bit representations of the synaptic weights and channels within the GREMLIN architecture.

First, some historical notes will be given. The reason of this is that *Sir Winston Churchill* once said: "To know the history, is to know the future", this saying confirms that it is necessary to describe some issues from the past. After that, some fundamental concepts of ANNs are explained, to understand the conversion from theoretical to practical use. The last section gives an overview of what to expect in the next chapters of this report.

## 1.2 Historical notes

Many researchers have been inspired by the fact that humans, and animals have the ability to adapt their knowledge. The source of this phenomenon lies within a complex system, called the brain. The brain is an immensely complex network of neurons, synapses, axons, dendrites, and so forth. To explain the working, and the secrets behind it biologists, psychologists, and other researchers have tried to model the brain in a mathematical way. This section gives a historical survey of the developments around neural networks.

One of the first pioneers is the American psychologist *William James* who published a theory on neural networks in 1890. In this period of time, researchers thought that the brain has to be an unstructured randomly connected web of fibers that propagated electrical currents in all directions. *William James* theories about the functionality of neurons and learning itself, are described in his book "Principles of Psychology". He assumes, that learning consisted of changing the current paths and of forming new paths by using the following rule:

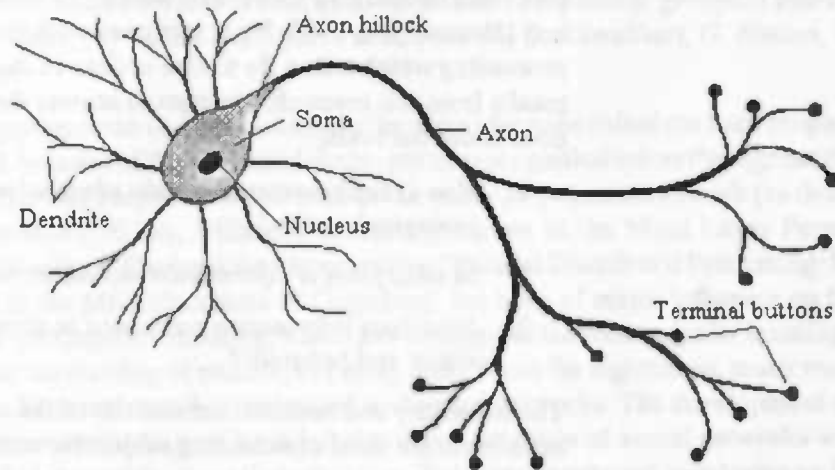
*"When two elementary brain-processes have been active together or in immediate succession, one of them, on reoccurring, tends to propagate its excitement into the other."* (James 1890, p. 566)

From a biological view, the nerve action (consisting of a burst of action potentials traveling only in one direction down a single celled neuron) was established between 1890 and 1910. This quickly led to the standard neuron model in which the dendrites of a neuron sum all the facilitatory inputs from the synapses of other connecting neurons, see figure 1-1. This sum of the neuron inputs is compared with a threshold located at the beginning of the axon, then an action potential is produced. The larger some senso-

ry stimulus intensity the larger is the frequency of the action potentials at the end of the axon.

**Figure 1-1;**

The basic structure of a biological neuron.



Later, in 1943, when *Warren McCulloch*, a psychiatrist and neuroanatomist by training, and *Walter Pitts*, a young mathematician, realize that the natural consequence of the standard neuron model's threshold in combination with binary action potentials produces another type of logic (known as threshold logic). This work is usually considered as the beginning of neurocomputing. After a couple of years, in 1949, *Donald Hebb* proposes a specific learning rule for the synapses of the neuron. He assumes that the connectivity of the brain is continually changing as an organism learns other functional tasks. His book "*The Organization of Behavior*" has been immensely read by psychologists, but had little or no impact on the engineering community.

*"Let us assume that the persistence of a reverberatory activity (or trace) tends to induce lasting cellular changes that add to its stability. The assumption can be precisely stated as follows: When an axon of cell A is near enough to excite cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place on one or both cells so that A's efficiency as one of the cells firing B is increased" (Hebb 1949, p62)*

This book has inspired even more researchers (*Utteley, Caianiello, Ashby*, and others) to take a look at computational model for learning and adaptive systems. Till 1954, the improvements of the theory on neural networks has rapidly increased. In this year *Marvin Minsky* writes a "neural networks" Doctorate Thesis at Princeton University, entitled "*Theory of Neural-Analog Reinforcement Systems and Its Application to the Brain-Model Problem*." Also in 1954, the first such *Hebbian* inspired network is simulated by *Farley and Clark* on an early digital computer at M.I.T. Their network consists of nodes representing neurons randomly connected with each other by unidirectional lines having multiplication factors called weights. To make this simulation work they have to modify Hebb's learning rule. With this rule the network is able to

successfully discriminate between two widely differing patterns as long as they are presented alternately.

Many researchers get stuck on the idea that the neural connections in the brain are mostly random. This means that the random neural networks are not having much success yet. The next step forward is taken by *Frank Rosenblatt*, a neuro-biologist at Cornell University, in 1958. He is intrigued with the operation eye of a fly. Much of the processing which tells a fly to flee is done in its eye. The original Perceptron, which results from this research, attempts to answer the last two of three fundamental questions about the brain:

1. *How is information about the physical world sensed, or detected, by the biological system?*
2. *In what form is information stored, or remembered?*
3. *How does information contained in storage, or in memory, influence recognition and behavior?*

The simplicity and random connectivity of the original Perceptron and the later Perceptrons make them a fascinating subject for mathematical analysis using probability. The Perceptron in a single-layer architecture is found to be useful in classifying a continuous-valued set of inputs into two or more classes. This concept has not only been established theoretically, but is also built in hardware and is still in use today. The hardware implementation of the network was realized by using electric motors and potentiometer. It has a 400 pixel image sensor and 512 programmable weights and was successfully used for character recognition.

In the early sixties, *Bernard Widrow* and *Marcian Hoff* of Stanford develop models they called ADALINE (ADaptive LINEar Elements) and MADALINE (Multiple ADaptive LINEar Elements). The purpose of these models is the recognition of binary patterns in order to predict the next bit from a stream of bits. The Madaline model has been the first neural network to be applied to a real-world problem: an adaptive filter which eliminates echoes on phone lines still in commercial use.

A turning point in the development of theory as well as in the practical use of neural networks came in the year 1969. The work done by *Marvin Minsky* and *Seymour Papert* reports on the computational limits of Perceptrons with one layer of modifiable connections. They use elegant mathematics to demonstrate that there are fundamental limits on what one-layer Perceptrons can compute. But these limitations do not occur in networks of Perceptrons that consist of multiple layers. They also speculate that the study of multilayered perceptrons will be "sterile" in the absence of an algorithm to usefully adjust the connection of such architectures. Since the Perceptron has been the most sophisticated neural network idea at that time, the book written by *Minsky* and *Papert* almost killed neural network research in the United States.

In the following years only a few scientists work (with a minimum of financial support) on neural networks, but they achieve remarkable results. A new stage in neural network research begins in 1972 with the publication of two papers. One is by *James Anderson* who was inspired by the *William James - Donald Hebb* model. The second is written by *Teuvo Kohonen* from Helsinki in Finland. He was inspired by the idea that memories may be holographic in nature. The result of *Kohonen's* research is a network identical that proposed by *Anderson*, and *v.d. Malsburg* in Germany: an associative memory based on neural nets with competitive neurons.

In 1982, *Hopfield* revives interest in neural networks in the United States, and introduces a new kind of network topology. This network topology differs from the earlier versions by using bi-directional lines between summation nodes instead of unidirectional lines and emphasized individual cells instead of cell assemblies. Before these developments, *Grossberg* establishes the basis of a new class of neural networks known as adaptive resonance theory (ART). In 1986 three independent groups of researchers come in focus, (1) *Y. Le Cun*, (2) *D. Parker*, and (3) *D. Rumelhart, G. Hinton, and R. Williams*.

These groups come up with essentially the same idea to be called the back propagation network because of the way it distributes pattern recognition errors throughout the network. The basic repeatable unit used in the back-propagation network (as described by *Rumelhart, Hilton, Williams*) is recently known as the Multi Layer Perceptron (MLP) topology. The book they have written "Parallel Distributed Processing: Explorations in the Microstructures of Cognition" has been of major influence on the use of back-propagation learning, which has emerged as the most popular learning algorithm for the training of multilayer Perceptrons. From the eighties on, many researchers have been interested in the behavior of neural networks. The development are not only towards a theoretical basis but also representations of neural networks within a hardware environments get their attention. Today many network topologies and learning rules are available, each having it's own application area. The main interest of today's research in neural networks lies within (a) improvement of experimental techniques, (b) searching for application areas, and (c) developing hardware for practical use. With this in mind the research on the area of neural networks is not done yet.

NOTE: this section uses the literature sources [2], [3], [4], and [5].

### 1.3 Content and Organization

This last section of this chapter shows the organization and the contents of the following chapters. Until now, the long and interesting history and the main objectives are shown, but the real functional behavior of an artificial neural network has not been shown yet. For the understanding of the artificial neural networks, and especially the network known as feedforward multi layered Perceptron networks, will be outlined in chapter 2. After showing the functionality of such neural networks, a basic concept of a neural hardware system will be shown in the following chapter. The functionality of the neural hardware system must behave in the same way as described earlier, therefore the functionality will be verified to assure that the theoretical behavior of an artificial neural network and the practical implementation are the same. But the hardware description in the language VHDL must adapted the characteristics of a trained neural network, which is performed by the software package *InterAct*. For the extraction of the network parameters software tools are developed, so that the characteristics of a trained neural network can be offered to the hardware system. These tools for extraction of the characteristics and the generation of parameters as, a discriminatory function, translation of the input vectors are mentioned in chapter 4. Another phenomenon which is outlined in that chapter, is the way of comparing the systems performances. From now on the hardware neural system can be exposed to experimental use. The first experiments will perform changes to the discriminatory function, so that the systems behavior can be analyzed in order to select a proper setting for the discriminatory function. These experiments are outlined in chapter 5. For an investigation of rounding effects exposed to the network characteristics will be mentioned

the the following chapter 6. This chapter also describes an experiment which is used by real-time systems, the addition of random (or white noise) numbers to results can speed-up the performance of the neural hardware system. Finally, our conclusions and directions for further research on hardware implementations of artificial neural networks are presented in chapter 7.

---

# Chapter 2

## *The functional behavior of an artificial neural network*

---

The artificial neurons we use to build our neural networks are truly primitive in comparison to those found in the brain. It is also true, that those networks as we are presently able to design are just as primitive, compared to the local circuits and the interregional circuits in the brain. Nevertheless those networks we can design have the ability to learn and therefore generalize; generalization refers to the neural network producing reasonable output for inputs not encountered during training (learning). This information-processing capability makes it possible for neural networks to solve complex (large-scale) problems that are currently intractable. In practice, however, neural networks cannot provide the solution in isolation. Rather, they need to be integrated into a consistent system engineering approach.

The primary interest in this chapter is confined to artificial neural networks from an engineering perspective, in other words, the functional behavior of such networks, to which we refer simple as neural networks.

### **2.1 Fundamental Concepts of Artificial Neural Networks**

The source of inspiration for ANNs is biology. By adopting parts of the functional and structural properties of the brain, it is hoped that ANNs will inherit some of their extraordinary computational properties.

The power behind ANNs is the amount of highly interconnected processing elements (nodes or units) that usually operate in parallel and are configured in regular architectures. The collective behavior of an ANN demonstrates the ability to learn, recall and generalize from training patterns or data.

Models of ANNs are specified by three basic entries: models of the neurons themselves, models of synaptic interconnections and structures, and the training or learning rules for updating the connecting weights. These basics will be introduced in the following sub-sections.

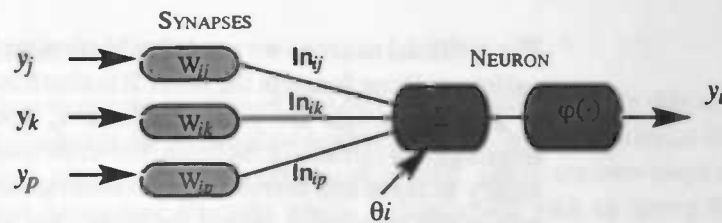
### 2.1.1 A Neuron Model

The processing elements in an ANN are called artificial neurons, or simply neurons. Figure 2-1 shows the model for a neuron. We may identify two basic elements of the neuron model, as described here :

- **Synapses :** The connections (or junctions) between neurons are made by synapses. These connections with their weight values are responsible for the information storage. The weighting coefficient of the synapse from the  $j$ -th neuron to the  $i$ -th neuron is given by  $w_{ij}$ . The functionality of a synapses is to multiply the weight of the connection and its input signal.
- **Neurons :** These are viewed as the processing elements of the network. As a matter of fact it contains two functions, an adder and a discriminatory function. The adder will sum all the input signals of the neuron, and subsequently adds the neurons bias  $\theta_i$ . Finally this result will pass through the discriminatory function.

**Figure 2-1;**

The mathematical model of the  $i$ -th artificial neuron.



The discriminatory function, denoted by  $\varphi(\cdot)$ , defines the output of a neuron in terms of the activity level at its input. This non-linear output function will limit the output amplitude of a neuron. Typically, the amplitude range of the output signal is written as the closed interval  $[0,1]$  or  $[-1,1]$ . There are several types of discriminatory functions, see figure 2-2 for some examples. Also more complicated functions can be used.

In mathematical terms, we describe a neuron  $i$  by writing the following pair of equations:

$$y_i = \varphi(v_i + \theta_i) \quad (2-1)$$

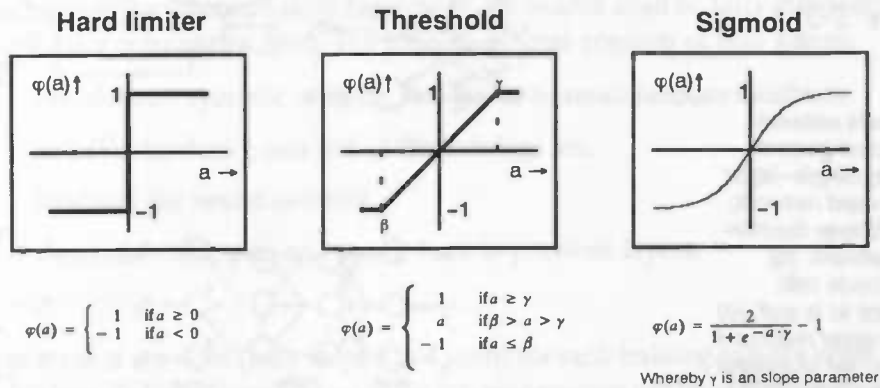
and (whereby  $In_i$  a set is of all the neurons that deliver inputs to the  $i$ -th neuron)

$$v_i = \sum_{j \in In_i} w_{ij} \cdot y_j \quad (2-2)$$

where  $w_{ij}$  are the synaptic weights,  $\varphi(\cdot)$  is the discriminatory function, and  $\theta_i$  is the bias. As mentioned, ANNs consist of many neurons. The way to connect these elements, several architectures have been developed, and these will be discussed in the next section.

**Figure 2-2;**

Sample discriminatory(transfer) functions, a Hard limiter, a threshold, and a sigmoid function.



### 2.1.2 The connections

At the moment there exist many different ANN models and their variations, and their number is still rising. In this section we shall describe some network topologies, but first how to connect those processing elements.

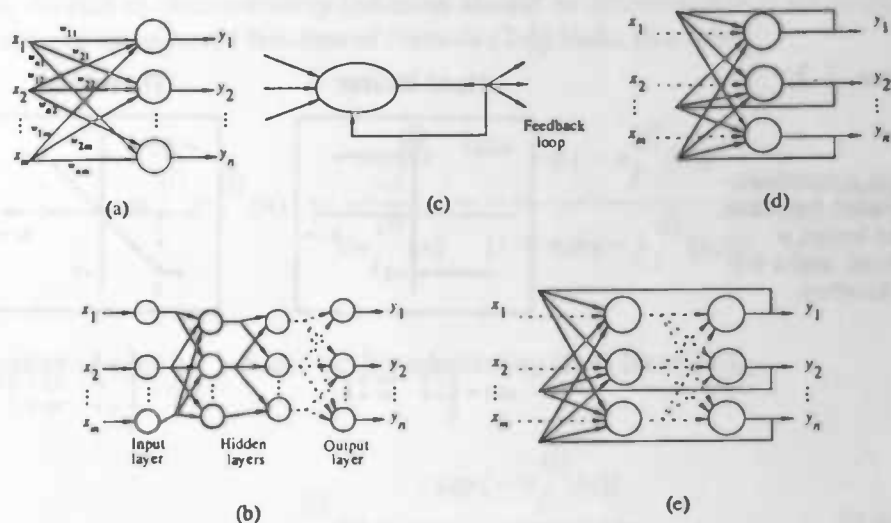
The neurons and synapses pass information to each other following a fixed communication scheme called the neural network topology. As said, an ANN consists of a set of highly interconnected processing elements such that each neuron output is connected, through (weighted) synapses, to other neurons or to it self; both delayed and lag-free connections are allowed, see figure 2-3.

The first architectures have been the single layer feed-forward networks, where each input connects to all processing units but no feedback connections are allowed. The next step is taken by *Rumelhart* and others, to combine single feed-forward networks into one large feedforward network, like the multi-layer Perceptron (MLP) topology. This network topology has some nice properties, notably the range of the input values and the discriminatory function of the neuron. The input values can be either binary or multi-valued (continuous or discrete). The discriminatory function can vary from the simplest heaviside one to very complex mathematical operations with time dependencies.

These properties make the MLP network to one of the most popular and successful neural network architectures. It is suited to a wide range of applications such as pattern discrimination and classification, pattern recognition, interpolation, prediction and forecasting, and process modelling. It is not only the geometry that makes the MLP popular but also its learning algorithm. The next section will tell us how the MLP architecture adapts its knowledge.

**Figure 2-3;**

Five basis network connection geometries, (a) Single-layer feedforward network, (b) Multilayer feedforward network. (c) Single node with feedback to is self. (d) Single-layer recurrent network. (e) Multilayer recurrent network.

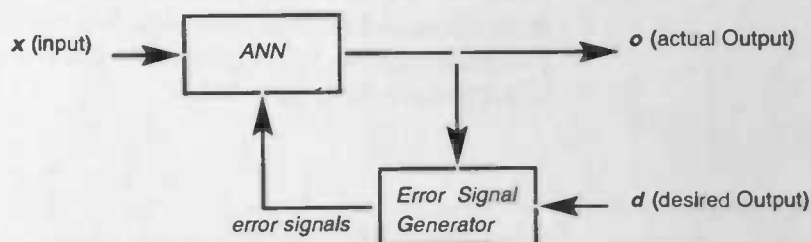


### 2.1.3 Learning rule

The final part in our description of an ANN is the learning rule. In this section the main question is "what is the mechanism behind adapting knowledge by an ANN?". Before the answer is given something else must be told. There are two ways of learning by ANNs: first *parameter learning* which is concerned with updating the connection weights in an ANN, and second *structure learning* which focuses on the change in the network structure [6], [7], and [8]. Over the years many of those learning rules have been developed and they can be classified into three categories: (1) supervised learning, (2) reinforcement learning, and (3) unsupervised learning. The category that will be discussed here is parameter learning rules in the category supervised learning.

In supervised learning, the corresponding desired response  $d$  of the system is given at each instant of time when input is applied to an ANN. The network is thus told precisely what it should be emitted as output. More clearly, in the supervised learning mode, an ANN is supplied with a sequence,  $(x^{(1)}, d^{(1)})$ ,  $(x^{(2)}, d^{(2)})$ , ...,  $(x^{(n)}, d^{(n)})$ , of desired input-output vector pairs.

When each input  $x^{(n)}$  is presented to the ANN, the corresponding desired output  $d^{(n)}$  is also supplied. As shown in figure 2-4, the difference between the actual output  $o^{(n)}$  and the desired output  $d^{(n)}$  is measured in the error signal generator which then produces error signals for the ANN to correct its weights in such a way that the actual output will move closer to the desired output.



**Figure 2-4;**

The supervised learning schema.

The way to correct the synaptic weights is done by a learning rule. The most popular one is the back-propagation learning algorithm. Backpropagation is a systematic method for training artificial multi-layer feed-forward neural networks (Figure 2-3b). In this method the discriminatory function of the neuron must be fully differentiable, a hard limiter is no option here. The training process consists of four stages:

1. Initialize all synaptic weights, and biases to small random numbers.
2. Select at random a pair out of the trainings set.
3. Evaluate the neural network.
4. Determine the error, and pass it back to previous layers.

This process is repeated (only steps 2 to 4) until for each training pair the error is acceptably low. Before of some stages in the learning process can be explained the mathematical notation we use must be clear; therefore they are mentioned in a special list in the beginning of this thesis.

The initialization of the biases  $\theta_i$  and the synaptic weights  $w_{ij}$  will be performed by walking through the network structure. The second stage is to pick at random a input pattern  $x_i(n)$ , to evaluate the network.

In this part the biases of the network are represented by the weights  $w_{j0}^{(l)}$  which are connected to a fixed input equal to 1. 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  $y_j$  is calculated according to:

$$v_i^{(l)} = \sum_{i=0}^p w_{ij}^{(l)}(n) \cdot y_i^{(l-1)}(n) \quad (2-3)$$

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

$$y_i^{(l)}(n) = \varphi_j^{(l)}(v_j^{(l)}(n)) \quad \text{with } l \neq \text{input layer} \quad (2-4)$$

for a neuron in the input layer.

$$y_j^{(0)}(n) = x_j(n) \quad \text{with } l = \text{input layer} \quad (2-5)$$

For the discriminatory function  $\varphi(\cdot)$  of the hidden and output neurons we choose the sigmoid function. A mathematical function is as follows:

$$\varphi_j^{(l)}(v_j^{(l)}(n)) = \frac{1}{1 + \exp(-v_j^{(l)}(n))} \quad (2-6)$$

This sigmoidal nonlinearity is commonly used in multi-layer feedforward neural networks. As said an discriminatory functions should be differentiable at all times, the derivative of the sigmoid function of formula (2-6) looks like this:

$$\varphi_j^{(l)}(v_j^{(l)}(n)) = \frac{d\varphi_j^{(l)}}{dv_j^{(l)}(n)} = \frac{\exp(-v_j^{(l)}(n))}{[1 + \exp(-v_j^{(l)}(n))]^2} \quad (2-7)$$

The output of neuron  $j$  is found as: By substituting (2-6) into (2-4),

$$1 - y_j^{(l)}(n) = \frac{\exp(-v_j^{(l)}(n))}{1 + \exp(-v_j^{(l)}(n))} \quad (2-8)$$

Formula (2-8) can be simplify the derivative of the discriminatory function to:

$$\varphi_j^{(l)}(v_j^{(l)}(n)) = y_j^{(l)}(n)[1 - y_j^{(l)}(n)] \quad (2-9)$$

This discriminatory function and its derivative are used in the summary of the error backpropagation algorithm in sidebar 2-2 .

### Sidebar 2-1;

The algorithm of the forward pass in a neural network, [9].

#### Summary Recall phase

```

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
       $y_j^{(l)}(n) = x_j(n)$ 
    else /* l is not input layer */
       $v_j^{(l)}(n) = 0$ 
      for i = first neuron feeding j to last neuron feeding j do
         $v_j^{(l)}(n) = v_j^{(l)}(n) + w_{ji}^{(l)} * y_i^{(l-1)}(n)$ 
      od
       $y_j^{(l)}(n) = \frac{1}{1 + \exp(-v_j^{(l)}(n))}$ 
    fi
  od
od

```

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

For input neurons the identity function will serve as discriminatory function (as can be seen in equation (2-5)). Hence these neurons are nothing more than the interface between the network and the outside world.

In the fourth step the error of the neurons are computed starting the 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:

$$e_j(n) = d_j(n) - y_j^{(l)}(n) \quad \text{with } l = \text{output layer} \quad (2-10)$$

The interval error signals  $\delta_j$  for the output layer are calculated by multiplying  $e_j(n)$  with the derivative of the discriminatory function:

$$\delta_j^{(l)}(n) = \varphi_j^{(l)}(v_j^{(l)}(n)) \cdot e_j^{(l)}(n) \quad \text{with } l = \text{output layer} \quad (2-11)$$

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

$$\delta_j^{(l)}(n) = \varphi_j^{(l)}(v_j^{(l)}(n)) \cdot \sum_k \delta_j^{(l+1)}(n) \cdot w_{kj}^{(l+1)}(n) \quad \text{with } l \neq \text{output layer} \quad (2-12)$$

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

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha[w_{ji}^{(l)}(n) - w_{ji}^{(l)}(n-1)] + \eta \delta_j^{(l)}(n) \cdot y_i^{(l-1)}(n) \quad (2-13)$$

The parameter  $\eta$  is the rate of learning. The smaller we make the learning rate parameter, the smaller will the changes to the synaptic weights in the network be from one iteration to the next and the smoother will be the trajectory in weight space. If, on the other hand, we make the learning-rate  $\eta$  too large so as to speed up the rate of learning, the resulting large changes in the synaptic weights assume such a form that the network may become unstable. Therefore *Rumelhart et al.* have introduced a momentum term  $\alpha$ , in the back-propagation algorithm represents a minor modification to the weight update, and yet it can have highly beneficial effects on learning behavior of the algorithm. The momentum term may also have the benefit of preventing the learning process from terminating in a shallow local minimum on the error surface.

### Sidebar 2-2;

A summary of the error back propagation algorithm in pseudo code, [9].

#### Summary 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
         $\delta_i^{(l)}(n) = y_i^{(l)}(n) \cdot [1 - y_i^{(l)}(n)] \cdot bpe_i^{(l)}(n)$ 
        for j = first neuron feeding l to last neuron feeding l do
             $bpe_j^{(l-1)}(n) = bpe_j^{(l-1)}(n) + \delta_i^{(l)}(n) \cdot w_{ij}^{(l)}(n)$ 
             $w_{ij}^{(l)}(n+1) = w_{ij}^{(l)}(n) + \alpha \cdot [w_{ij}^{(l)}(n) - w_{ij}^{(l)}(n-1)] +$ 
                 $\eta \cdot \delta_i^{(l)}(n) \cdot y_j^{(l-1)}(n)$ 
        od
    od
od

```

The main idea of the error back propagation 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 updates too. So

- for each neuron  $i$  its 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

---

# Chapter 3

## *The basic concept of hardware design*

---

The design of a digital circuit or software application is strongly subject to developments over years. In the early days often unstructured design methods have been used to fit the problem. These design methods lead frequently to an enlargement of the complexity and a badly arranged result, that can lead to mistakes. Another phenomenon of minor importance is the rapid and continuous offer of new techniques and technologies which make the complexity more and more worse to handle. From this point of view, designers have developed several methodologies to conquer complexity. These methodologies or design trajectories constrain the designer to follow a strict path from the begin tile the end of the project, without coming to a death end in slumbering details. One of the first articles that use such a design style, is on the development of the IBM-360 architecture. After this, many have followed the principle of leveled or stepped design, for large and small projects.

But in the last decades of the century, software-engineers face the same problem, the applications are growing and growing and the complexity strikes again. This problem must be solved by the introduction of software architecture [10], and [11].

This chapter reports on design steps, that will lead to a workable representation of artificial feedforward neural networks. Also the constraints and decisions will be mentioned.

### 3.1 Design concept

It is a challenge to achieve a solution of a project without having to much trouble, therefore a design strategy had to be chosen. To inform which kind of method will be used, this section will tell its origin.

At first, in a classic 1964 paper *Amdahl, Blaauw, and Brooks* propose dividing the design description of a system into three levels: architecture, implementation, and realization [12]. Such a development, to separate the whole designing space into three stages was a step forward to handle the complexity increase. Each stage gets besides a name also a description. Architecture refers to the attributes of the system, that is,

the conceptual structure and functional behavior. Implementation is defined as the actual hardware organization, including data paths, logic units, and control units. While the realization level is the actual physical structure, including logic technologies, board layouts, interconnections, and power supplies.

**Figure 3-1;**

A flow chart of the design concept. Where the relations is shown between the three stages of design; Architecture, Implementation, and Realization

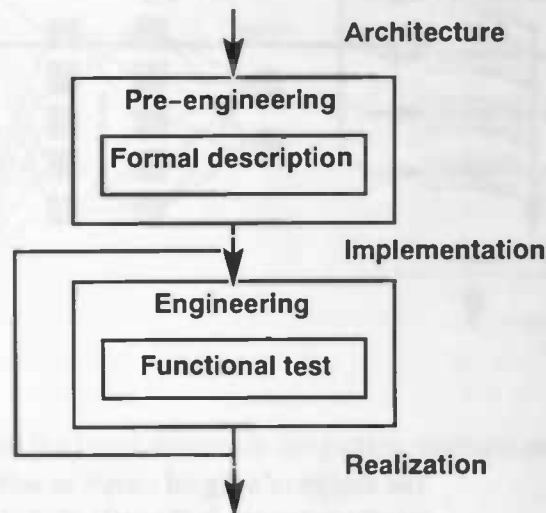


Figure 3-1 shows us the idea of the design concept. On architecture consisting of a functional behavior and a conceptual structure will be transformed in the pre-engineering phase to a formal model description. This model can be tested, so that the specification can be verified. From this stage on, the model can go from implementation to realization through an engineering phase. That holds the actual design of the chip, in a certain technology and platform. This design will be tested in its own environment to show that it meets the required specifications.

In the following sections of this chapter the design stages as architecture, implementation and realization are described. The simulation of the result of the implementation phase shall be tested and some experiments will follow in later chapters. These experiments shall dimension the neural system, so that it satisfies the required performance.

### 3.2 Architectural Level

The architectural level of a design consists of two parts, namely the conceptual structure and the functional behavior. How to manage the structure of a feedforward neural network into a hardware environment is the main part that will be outlined in the conceptual structure. It also describes how the system is divided into several subsystems, these systems or building blocks have their own functionality and conceptual structure. On the other hand the functional behavior describes the functionality of several building blocks, and the behavior of the system including the algorithms of specific blocks. These two parts must be clearly described before a next step of development is started, to avoid errors in the specification, to cause failures within the designed system.

Another reason to divide the architectural level of design into two parts is that it gives us an advantage in describing it in a hardware description language, as VHDL. These

languages for describing hardware make a difference between the functional behavior and its conceptual structure. The representation of the conceptual structure is easy to convert into a description in a hardware language. The same story applies to the behavior of the system. Now the advantages are known, let's describe the functional behavior and the conceptual structure of the neural system.

### 3.2.1 The Functional Behavior

This section will explain the functionality and its architecture, but before that some things must be mentioned. As seen in chapter three, the functionality of a neural network depends on a variety of attributes, such as the type of activation function used by a neuron, the way of connecting neurons, and their types (lag-free or not), etc. The spectrum of usable neural network is so wide, that it is impossible to create a hardware design generic enough to support all those types. Therefore our architecture has a number of constraints.

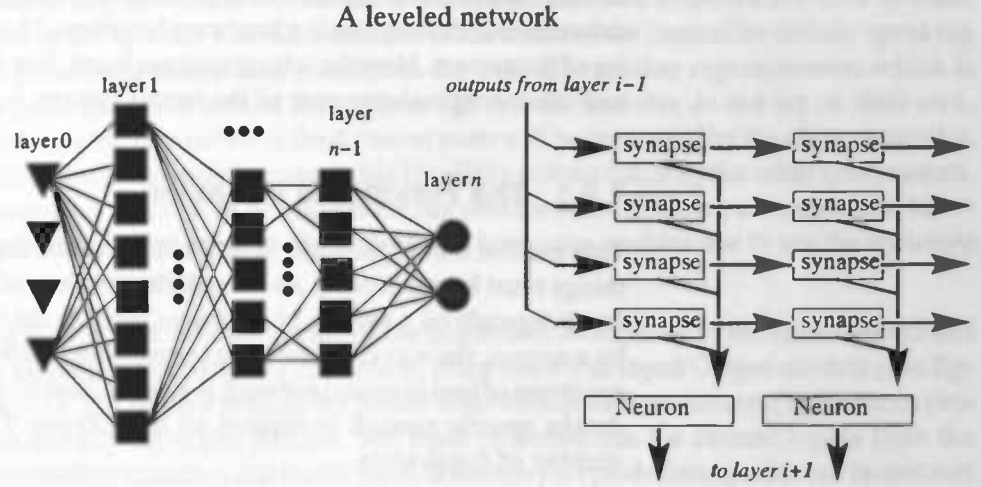
The supported network topology in the design is a feedforward structure of neurons. These feedforward neural network is an assembly of neurons, that are connected through synapses in such a way that only a single direction of the dataflow is supported, i.e. all signals stream from the inputs to the outputs. A second restriction is based on the structuring of the network into so-called levels: all signals pass all levels in consecutive order, or, in other words, all synapses that emanate from level  $i$  connect to level  $i+1$ . A consequence is that by moving from the input level through the intermediate (hidden) levels to the output levels all calculations are guaranteed to be based on fresh synapse values. Another restriction is, that the synaptic weights and the biases of the neurons are no longer variable once a representation of a neural network on hardware is accomplished. This means, the training of the network is finished and it meets conform the requirements, in a simulation environment, such as *InterAct*.

The computational meaning of respectively synapse and neuron is illustrated in figure 3-2 of a layered network. The network shown has a succession of layers, secondly a single layer is detailed to be a computational matrix. Each synapse takes a value from the output of a designated neuron in the previous layer and multiplies this value with the dedicated synapse weight. The result is passed onto the input of a designated neuron in the present layer. There it will be summed with the multiplication results from all other synapses that feed this specific neuron. Now, each neuron will add the summed result of the feeding synapses to the internal bias and pass the result to a non-linear decision function to compute the neurons output value. The most salient non-linear decision function within a feedforward neural network is the sigmoid function. To enlarge the applicability of the design other functions can be chosen. The algorithm for an artificial feedforward neural network in the forward pass can be found in sidebar 2-1.

The following analysis gives an overview on the boundaries of the signals within a multi-layer feedforward neural network. The understanding of the signals flowing through the network will be an indication of the accuracy, but gives also a better view on the network itself. For a proper working of a neural network the input and target patterns have to be scaled. The reason for this lies within the used transfer function as the proper work area of a sigmoid function lies around zero, its maximum derivation.

**Figure 3-2;**

On the left, a MLP network is shown with four inputs, and two outputs. The right side shows a detailed layer.



Lets assume that all the input-vectors in the pattern database are scaled, such that

$$x_i(n) \in [0, 1]; \forall i \in [0, N] \quad (3-1)$$

with N the number of neurons in the input layer. From equation (2-5), we learn the transfer function of an input neuron. Combining it with the known boundaries of the input vectors, follows,

$$y_i^{(0)}(n) \in [0, 1]; \forall i \in [0, N] \quad (3-2)$$

For the determination of the range of the internal activation, equation (2-3) will be used. Assuming that the synaptic weights of a network are between  $-W_{max}$  and  $W_{max}$ , the following can be concluded,

$$v_i^{(l)}(n) \in [-M \cdot W_{max}, M \cdot W_{max}]; \forall i \in [0, M]; l > 0 \quad (3-3)$$

with M the number of inputs of the  $i$ -th neuron. The internal activation level can take extreme values. To bear up against this phenomenon the determination of the boundaries of the activation function could give an answer. This can be done by taking the limit of the activation function (used function the well know sigmoid).

$$\lim_{v_j^{(l)}(n) \rightarrow \infty} \frac{1}{1 + \exp(-v_j^{(l)}(n))} = 1 \quad (3-4)$$

$$\lim_{v_j^{(l)} \rightarrow -\infty} \frac{1}{1 + \exp(-v_j^{(l)}(n))} = 0 \quad (3-5)$$

We see that the activation function with a range of  $(-\infty, \infty)$  will map on the range  $[0, 1]$ . This means that the system output as well as the inputs are bounded in the same range.

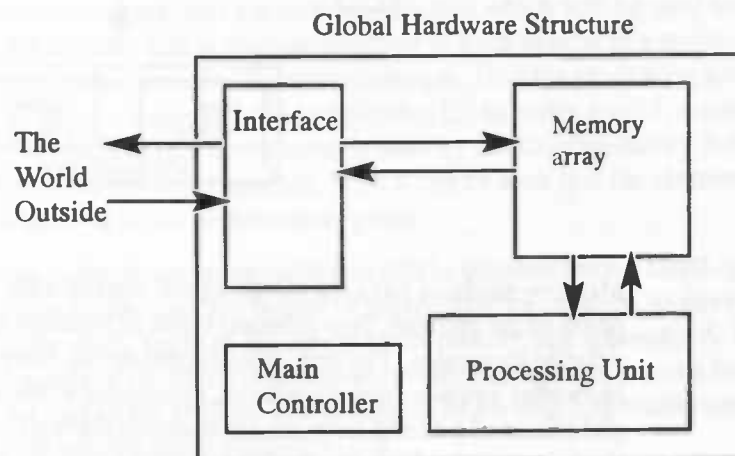
A system option which could be handy to enlarge the applicability of a hardware representation will be outlined. The system must be developed in such a manner that an indefinite number of identical systems can be connected in a sequential or parallel order. This gives the design such an advantage, that even larger neural networks can be constructed by using a number of processing units. Another possibility that can lead to a speed improvement is a parallel composition and decomposition of the network structure. The decomposition of the network structure in different components makes it possible to create gated expert systems, where each expert is individually processed by a separate system. But also classifiers where each output determined by another chip.

### 3.2.2 The Conceptual Structure

The application area of neural networks in a hardware setting is often embedded as a controlling system. The architecture of such embedded systems resemble each other. These controllers can be classified into two groups; (1) machines that use an instruction set for programming (instruction set machines), (2) and application specific machine. Application specific machines consist of state-machines where the calculations and actions are defined before producing the system; often only one task can be performed. Instruction set machines are larger in size, and are not as easy to develop as application specific machines. The calculation performance of predefined machines with respect to the instruction set machine is larger, if the design is of good quality.

**Figure 3-3;**

A global system structure of the neural hardware processor.



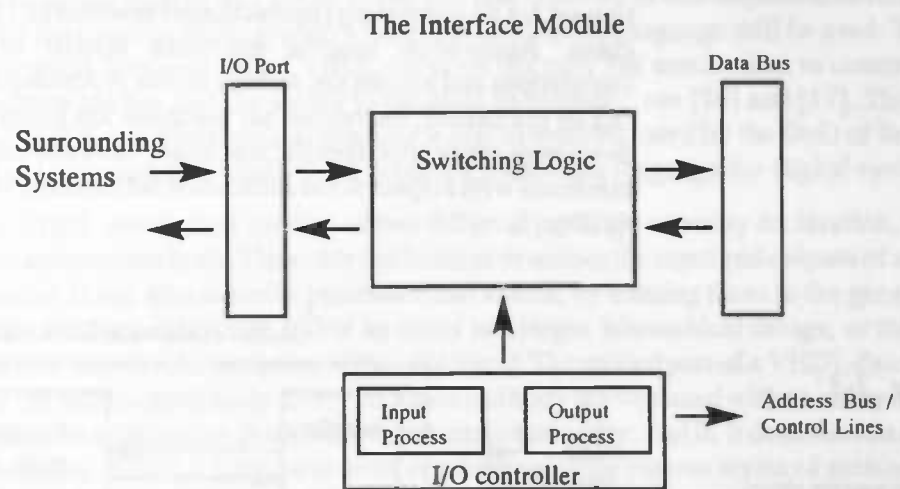
The conceptual structure of many controllers looks similar from high abstraction level. These controllers contains mostly three major parts, see figure 3-3. The first one

is the interface which supports the system with data, or produces information to its surrounding environment. Second is a memory that will store/release data of other parts of the system. The third element is the heart of the controller strictly speaking the processing unit, which transforms the data into another representation which is useful to the world around the system. The system modules do not act on their own, but the interaction between the different parts will be organized by the main controller. This controller is mainly responsible for all the actions taken by the other components. The system that will be developed carries also the same structure as mentioned. Moving from this high level of abstraction to a lower one enables use to see the structure of each component.

The interface module takes care of the communication between the environment and the internal organization of the system, often known as Input/Output module (see figure 3-4). The control module is divided into two separate processes, (1) the input process and (2) the output process. The input process reads the offered inputs from the surrounding actuators and stores them in memory on predefined positions. In contrast during to the output process a memory location will be read and its content be set on the output pins, such that the environment of the system can act properly. The logic between the I/O port and the data bus of the internal system acts as a galvanical separation of the input and output data. The access from or to the environment can only happen by use of this interface, so a strict communication protocol is at hand. This protocol (a sort of hand-shaking) is necessary to communicate with the internal controller that handles the input or output behavior of the interface module.

**Figure 3-4;**

A detailed scheme of the interface module is presented. The module is responsible for the I/O behavior.



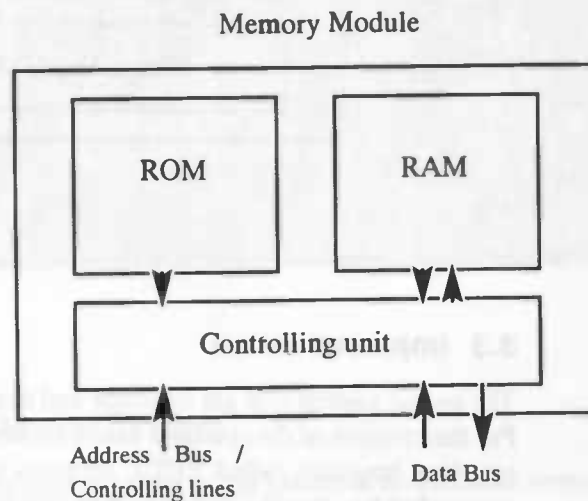
Memory modules have the advantage of storing data, that they can release in another phase of the process. Two different types of memory are known: the non-volatile and volatile memories. The non-volatile memories keep their state (no memory loses), even after a power-down situation, in contrast to the volatile memories. The weights and biases of a neural network are constant during the recall process. Therefore they can be memorized in a non-volatile memory without updating them, the memory used is a ROM or Read Only Memory. On the other hand the data that changes over time, such as inputs or the outcomes of calculations, are variable. A perfect place to store this type of data is within a RAM or Random Access Memory. Figure 3-5 shows the

internal structure of the memory module. The two types of memory and the controller must take care to store data, and to release the asked data on the data bus. The input signals of the module are a data bus, an address bus, and control lines.

The access toward the memories must be controlled in such a way that no writing or reading can happen at the same time. That means that each action which contains an access to a memory must be an atomic one. The memory controller is informed by the main system, by use of controlling lines, what action must be taken. If it is a read action the address on the address bus will tell the controller which memory to activate. By a data write signal the only memory to access is RAM, and the location of the memory cell that will store the data is encapsulated in the address on the bus. The data that must be stored or is released will always be present on the data bus.

**Figure 3-5;**

The figure shows the internal structure of the memory module. It contains a ROM, a RAM, and a controller unit to perform its task.



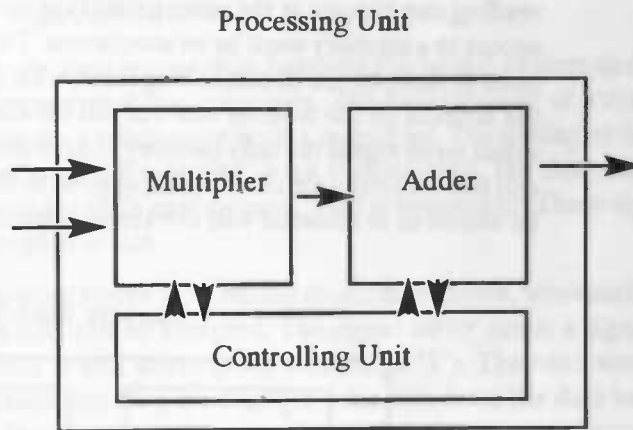
The processing unit of the system is where the calculation of the neural network will take place. The formulas for the behavior of a neural network, can be concluded from the following. The behavior formula of a synapse corresponds with a multiplication of the synaptic weight and its input. The neuron itself consists of a summation of the neurons inputs, and a activation function which will be used to determine the output of a neuron. The activation function of each neuron in a multi-layer Perceptron network is the same, and therefore constant. Thus the processing unit of the neural system has three components (1) a multiplier, (2) an adder, and (3) a controller, see figure 3-6. The result of this component is used by the discriminatory function which is represented in the ROM module. Now it can be seen that the structure of a neuron is fully described in the architectural phase.

The multiplying component that will be constructed is a DigiLog multiplier. This type of multiplier uses the theory of logarithmic calculation, as developed by *John Napier* in the early years of the seventeenth century, see appendix A. His theory is the base concept of the digital logarithmic multiplier that will be used here within the processing unit of the system. The principles of the DigiLog multiplier can be found in [13], [14] and for detailed design [15].

The description above gives enough information to start the next stage of the design trajectory, the implementation phase.

**Figure 3-6;**

The processing unit of the neural system consists of three modules, an adder, a multiplier and a controller.



### 3.3 Implementation

The neural system will get its shape and functionality in this implementation phase. For the creation of the system a hardware description language will be used. This description language called VHDL supports not only the mechanism to construct the system but has also the ability to simulate its behavior, see [16] and [17]. The origin of VHDL is a description language for digital systems, used by the DoD of the USA, but has grown to a world-wide accepted simulation language for digital systems.

A VHDL description consists of two different parts, (1) an entity declaration, and (2) an architecture body. The entity declaration describes the input and outputs of a design entity. It can also describe parameterized values, by naming them in the generic list. The I/O list could be the I/O of an entity in a larger, hierarchical design, or the entity is a device-level description of the chip it itself. The second part of a VHDL description is the architectural body. Every architectural body is associated with an entity declaration. An architecture describes the contents of an entity; that is, it describes an entity's function. VHDL allows us to write our designs using various styles of architectures. The styles are behavioral, dataflow, and structural, or any combination. These styles allow us to describe a design at different levels of abstraction, from algorithms to gate-level primitives.

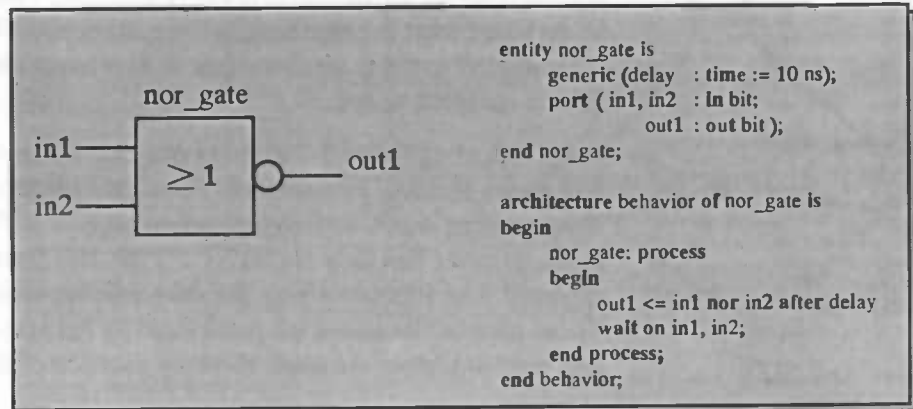
An example of a VHDL description from a NOR-gate can be found in sidebar 3-1. The entity of the NOR-gate has three ports namely, two inputs (named IN1 and IN2), and one output (named OUT1). In the example the static information tells us that the component has a gate delay of 10ns. The architecture tells us that the description that follows is the behavior of the NOR-gate. The "process" description contains sequential statements just like an imperatively program language, as C, Pascal, and Cobol. This description shows that the output-signal OUT1 is the logical NOR operation between IN1 and IN2. After a change of the output-value OUT1 the system will wait 10ns

before putting it on the output-channel. The VHDL-code for a NOR-gate can be tested by using a VHDL-simulator like Warp, V-system, or VeriLog. These simulators show timing diagrams, and after mapping on the chip the propagation delay can be viewed and examined. The testing phase of the VHDL-code is very important, because if each chip doesn't perform its objectives, lots of money is wasted.

#### Example of VHDL-code

##### Sidebar 3-1;

A VHDL example of a NOR-gate. The two parts in the description of the NOR-gate can be easily recognized



### 3.3.1 A Neural System

The implementation of the neural system is divided into several part, to describe each function of its architecture. Combining all these modules together gives the neural system its functionality. Our goal is to design a neural system that can be used in experiments. These experiments require that the binary word length is scalable in size. This means that all the control lines, data-busses, and address-busses are variable in size. In a language as VHDL this can be solved using the *generic* type, that supports a static variable during compilation of the system. As known from the conceptual structure the system exists of modules, each with its own functionality and responsibilities. In the following sections each module gets their attention. For the connoisseurs among us, the appendix B holds the total VHDL-code of the neural system.

### 3.3.2 Interface Module

The interface module is cut in two pieces for its I/O behavior. The first one is concerned with how to get the data from an input port to memory. The second reads data from memory and puts it on the output port. Those two processes do not use one and the same output port, because this is handy in simulating the total system. For a possible realization of the system the use of one I/O port is advisable so that a minimum of output pins are necessary. But in some cases a separation of input and output ports is granted if the embedded neural system has two separate data-busses available.

The interface module has on the environment side a variety of signal pins. The I/O port receives data from the setting of the system. A pin called start can activate the input process, so that the data available on the data I/O port will be accepted by the system

and will be memorized. For the communication between the system and its environment, signal pins are added to feed the environment information that is needed to know what is happening inside the neural system.

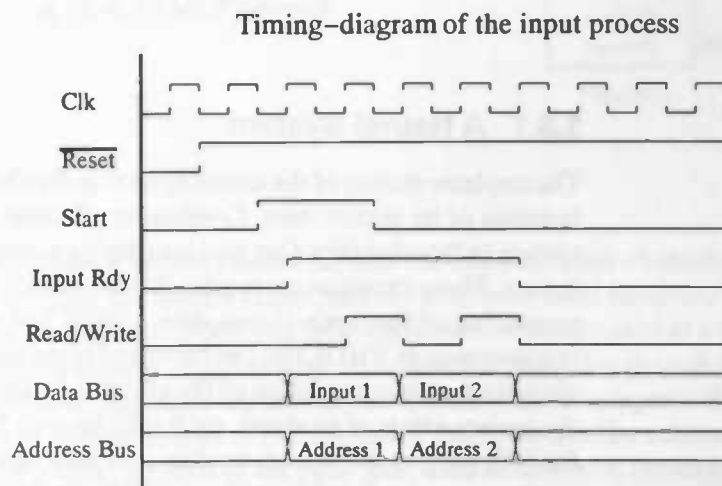
### 3.3.2.1 Input Behavior

For a well-defined cooperation between the neural system and its setting, timing is the most important thing to manage. The development of a system, that takes input signals from the environment is very important. The setting of the neural system must be aware of what is happening in the system itself. For this reason signals are added to the system to tell its environment what is happening. These signals will tell whether data is accepted or not.

In the following figure 3-7 a timing diagram is shown, wherein the neural system gets two inputs that will be accepted. The signal *INPUT\_RDY* is a signal that shows that the input process is still active (only when high '1'). The *READ/WRITE* signal tells us that at the moment that they are high ('1') the data from the data bus will be copied into memory. The location where the data will be written is shown by the address bus. Those memory locations are predefined by the system so the environment of the system does not know the exact memory location of the data.

**Figure 3-7;**

The timing of the input process, if two inputs are needed by the system. These inputs will be stored in memory for further processing. The signals who are colored are not seen by the systems environment.

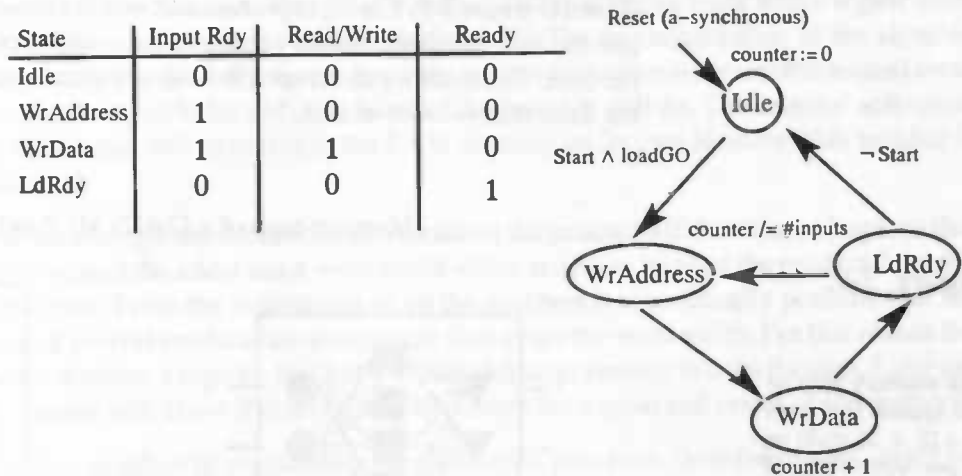


For the creation of the timing specification as given above, a state machine is developed. This state machine has three things to do: (1) wait for the input data, (2) determine the exact memory location for the input data, and (3) pass the input data to the data bus so the input data can reach the memory. The machine will get the command to load input data from the overall controller by means of a signal *LOADGO*. But also the setting of the system talks to the I/O controller. The controller can not start without a *START* signal turning high from the environment. The first state of the machine is waiting for the permission to start from the internal main machine and from the environment. In the second phase, the controller will set the memory location on the address bus. The next step is to pass the input data through from the I/O port to the data bus.

The final state of the machine is to wait for the next time the system and its setting will pass/ask for data. This entire story is shown in a finite state machine in figure 3-8.

**Figure 3-8;**

A diagram of the state-machine of the data input process. It contains four states in a sequential order (Idle - WrAddress - WrData - LdRdy).



### 3.3.2.2 Output Behavior

The machine of the output-process is almost the same as the input-process. That means also that the timing of this process is nearly the same. The difference is that the output process reads data from the memory and puts it on the I/O port. This process will start, when the all calculations are performed, only then the start sign will be given by the main controller. The memory location that will be accessed is the highest filled one. For example, if the system contains a (2, 4, 2) neural network, that means two input, four hidden neurons and two output neurons. The memory locations of the output neurons are six and seven, this can be calculated on forehand, by summing the number of input neurons to the number of hidden neurons. This will give us the beginning of the locations of the output neurons of the network. A closer look on the memory structure, and the way data is organized, can be found in the next section.

### 3.3.3 Memory Module

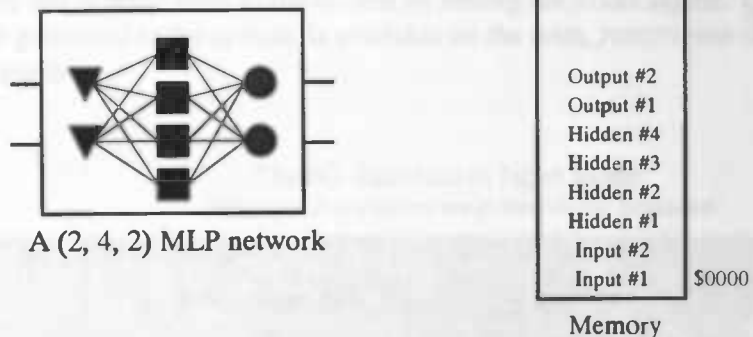
This module is responsible for data. If data is stored, the same data can be recalled without loss of information. Memory is no more than a pile of information (for example a pile of bills). On a time dependent moment information must be recalled, in our example some bills are paid and removed from the pile of bills. It is possible that the wrong bill has been removed from the pile, and there is loss of information. This can happen in the electronic memories that we will use. The stored information must be carefully handled, so that there is no loss of information. There are two ways to access a memory: (1) data can be stored, and (2) the stored data can be recalled. The first memory that will be discussed here is the RAM (Random Access Memory), after that, the storage of constants in a ROM (Read Only Memory) will be outlined.

The memory (RAM) is used to store variable information, that is produced during the calculation of the network. The design of the system holds a trained multi layered Perceptron neural network, thus much information does not change over time. The only variable data that will be stored are the determined results of each neuron. The data is stored in such an order, that the calculation process can find the output or input data in a second. The memory map of the memory module of a (2, 4, 2) MLP network, is shown in figure 3-9. The input process delivers the input data from the I/O port directly into the first available memory cells, so the calculation process knows where to find the data. The memory developed is fully scalable in number of cells, or the width of the data, that must be stored.

Memory map of a (2,4,2) MLP network

Figure 3-9;

The memory map of the system is shown of a (2, 4, 2) multi layer perceptron neural network. The memory locations starts with address 0 to maximum number of locations.



Another type of memory stores the data that has a constant character, namely the synaptic weights, the neuron biases, and the used activation function. This data is available in some sort of ROM. All these constants are memorized in a weight in a bias or in a discriminortoy function table. A reason for doing this in such a way, are the nice properties of such look-up tables. Those tables can be compressed and/or optimized by use of an synthesis tool. The amount of space required by a memory is much larger than for a look-up table of the same size. This results from the structure of memories that use lots of flip-flops, while a look-up table can be expressed in Boolean logic. Another advantage of look-up tables is that the speed for accessing data is much higher than for a memory call. Two look-up tables have been equipped with a part of logic so that with one signal the next value can be accessed. So it holds a calculation sequence within the look-up table.

### 3.3.4 Processing Unit

The unit schematics shown in figure 3-6 consists of three parts; (1) a controller, (2) a multiplier, and (3) an adder. These parts together are able to fill the functionality of a neural network. The analysis of the boundaries of the values that cross the parts that perform the calculations, shows that only the multiplier and the neuron calculation results can be negative. The DigiLog multiplier which performs the product of the synaptic weights and the output of a previous neuron, can result in a negative number.

Only the weights can be negative, so the representation of that number will be in signed magnitude. These types of number only can be multiplied. The result of the multiplier is an signed magnitude number. The summation that will follow, can only perform its actions on two's complement numbers, therefore a translation must be realized. The output sum of the adder gives us the internal activation of a neuron. This internal activation level is used to find the external activation level by use of the sigmoid decision function. This discriminatory function has an input with a higher resolution than the output (or external activation). The implementation of the sigmoid function is a little tricky. As we know the sigmoid function is symmetric around zero. So a look-up table for only one half of the function will do. The external activation of the neuron will be stored in the RAM memory on its own location (this number is positive).

For the multiplication of two binary numbers the product will be twice as large, so that the inputs of the adder has a word width which is also as large as the product from the multiplier. From the summation of all the numbers it is practically possible that the sum of several products are even larger than twice the word width. For this reason the adder contains a register that has a wordwidth large enough to hold the sum. Later experiments will show if this is really necessary for a good end result of the network.

Another problem to overcome is the calculation sequence, therefore a state-machine is developed which determines this sequence. This machine is a 1-to-1 mapping of the sidebar 2-1 given in chapter 2. It determines which weight from the look-up table must be multiplied with which synaptic input for the RAM memory. The result of this machine will tell the environment on which location the input is stored in the RAM memory. For a proper working of the processing unit a second controller is developed. This machine takes care of all the interaction between the memory, multiplier, adder, and the calculation sequence machine. Now the total system is implemented, the source VHDL-code is as shown, in appendix B.

### 3.4 Description Verification

A time-consuming but crucial part of the design process is design verification; the ability to simulate a VHDL model can greatly increase the efficiency. This allows the functional verification of a design before synthesis and place & route. In our case, the functionality is an artificial feed-forward neural network, as in chapter 2. The description produced in VHDL must have the same functionality so that we can speak of an neural hardware model.

This section verifies the functionality of the main parts of the neural system. These three main parts are (1) the IO-interface, (2) the RAM module, and (3) the processing unit. The verification of each part is performed by a VHDL simulator, called *V-System*. This simulator can handle a description without running synthesis, place & route tools. This means: there is no technology mapping necessary before the description can be simulated. The verification of the three parts will be performed in the order: IO-interface, RAM module, processing unit.

#### 3.4.1 IO interface

The functionality of the IO-interface, which comprises the communication between the internals of and the environment around the neural system, will be simulated to

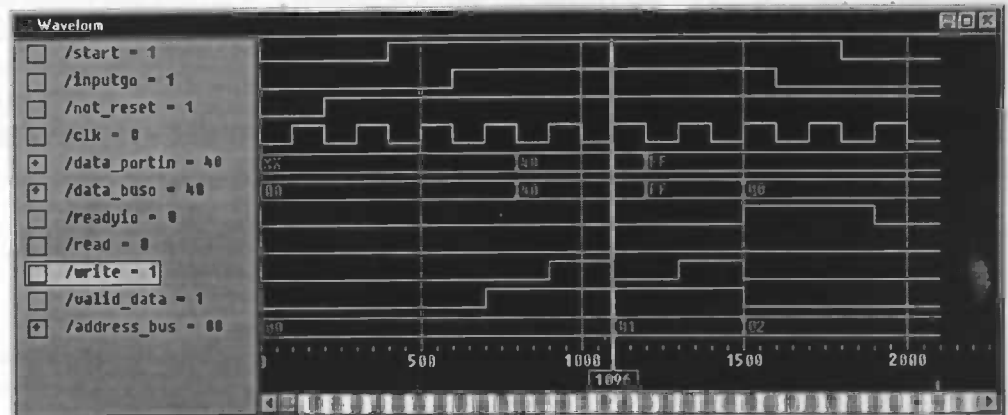
see of it corresponds to the specification earlier in this chapter. The simulation of the IO-interface is split in two separate parts, because it describes two kinds of function. These two functions are an input and an output function.

The input behavior of the IO-interface is verified by simulation. Two inputs are offered to the interface. The signal from the environment as well as the signals from the main controller are set by hand. This is done to eliminate a wrong behavior caused by other parts of the neural hardware system. Figure 3-10 shows behavior of the interface by using a time table, which illustrates the events and the reactions of the interface module. The signals set by the environment of the system are: DATA\_PORTIN, NOT\_RESET, CLK, and START. The clock signal (CLK) is the signal on which the system synchronizes with its environment. The state machine of the IO-interface module is reset by the signal NOT\_RESET, so that the machine can start from its idle state. The reset of the system is independent of the clock signal (CLK), thus an asynchronous reset. The environment of the system tells that there is data available which can be processed by the system; this is made clear to the system by setting the START signal. The data, which will be presented to the system, is available on the DATA\_PORTIN port after setting the START signal.

**The IO-Interface in Input Mode**  
Two input data patterns are presented and processed

**Figure 3-10;**

The input behavior of the IO interface module. Two separate input signals are present and processed.



The signal from the main controller is also simulated. That signal tells the interface that the controller is ready to accept data from its environment, is called INPUTGO. The IO-interface tells the main controller that the input is processed when the READYIO signal becomes high. Before that happens the interface must write the input data available on its input port to memory. The inputs are presented to the memory in the same order as the environment is presenting them. The memory locations of the input data are calculated by the interface module and presented on the ADDRESS\_BUS of the system. With the signals READ and WRITE the memory knows exactly what to do, in this case the data must be stored. The data that must be stored in memory is presented by the DATA\_BUS0. Some signals are presented also to the environment so that a handshake can be performed, those signals are VALID\_DATA, READ and WRITE.

The second behavior of the interface is the output behavior. Now the interface must ask the memory module to present data on certain memory locations. The following

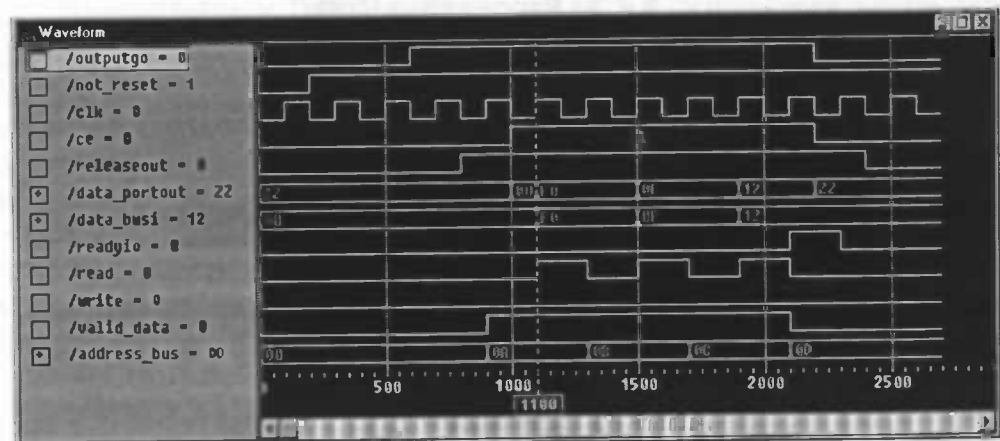
figure shows the time table of the output behavior. Lets consider that the outputs are available on the memory locations 0Ah, 0Bh, 0Ch (h=hex notation), and that the contents of those locations is F0h, 0Fh and 12h. After a start of the interface by the main controller (OUTPUTGO) the handshake will start. The data corresponding to the memory location, which is presented on the ADDRESS\_BUS, will be presented on the DATA\_BUSI. For the environment the data is presented on the DATA\_PORTOUT. The signals CE (chip enable) and RELEASEOUT are signals that are set by the environment to influence the handshake with the system. If the output process is ended the main controller gets a signal for the interface to say its ready (READYIO).

The verification of this module seems to correspond with the demands that are made to design the interface module. The following step is to look at the behavior of the memory module.

The behavior of the IO interface, by output

Figure 3-11;

The output behavior of the interface module, three memory locations are accessed. The contents of them is presented to the environment.



### 3.4.2 Memory

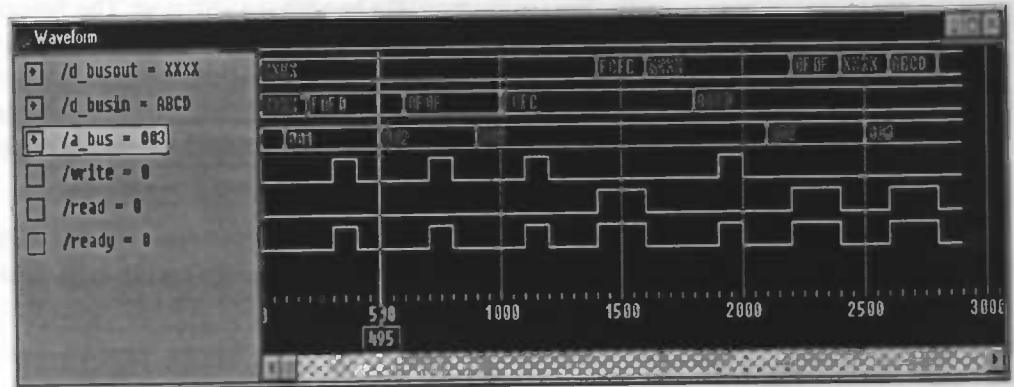
The memory module of the system that has the purpose to store data after or before a calculation, and the release of it. The verification of this module can easily be done by storing some information and retrieving it. For a correct functionality the stored and retrieved data must be the same.

The following operations are done to ensure a proper working. First three write operations are performed, and the data F0F0h, 0F0Fh and FCFCh are written to the respective the addresses 001h, 002h, and 003h. After the write actions, the content of memory location 003h is recalled, the value presented (FCFCh) on the D\_BUSOUT corresponds to the value which was stored. Now a write action follows on the same address and will overwrite the data stored on position 003h; the value stored is ABCDh. The following step is to read memory location 002h, its content is 0F0Fh and is correct. The last action on the memory module is a read operation on the memory location 003h, which must hold the value ABCDh. From the output databus the retrieved data is ABCDh. The memory module seems to work properly and the verification is complete.

## The verification of the memory module

**Figure 3-12;**

A timing diagram of the verification of the memory module. It shows the response of the module by applying several test vectors.



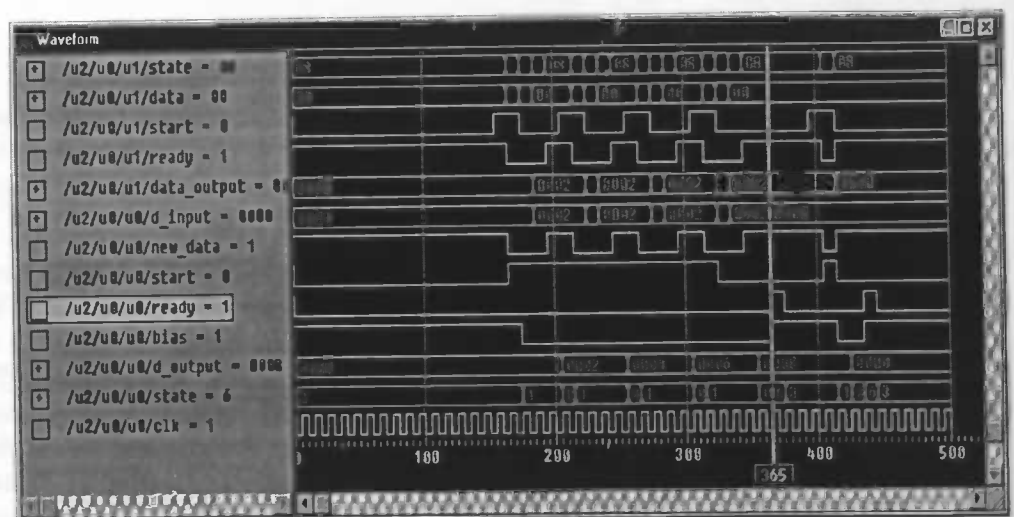
### 3.4.3 Processing Unit

The processing unit consists of a DigiLog multiplier and a recursive adder. Together these two components produce the required sum of products. The verification of these two components will be outlined in this section. The synaptic weights, biases or the discriminatory function which are constants or terms in the process have no influence on the verification; so these component are left out of consideration. The verification of both components will be performed as if it is one component. One of the verifications performed is shown in figure 3-13. The verification holds the following operations, first a sum of four products is calculated where each product is 1 times 2. The

result of the sum must be:  $\sum_{i=0}^{i=3} 1 * 2 = 8$ ; This result can be seen when the READY signal of the adder become high.

**Figure 3-13;**

A timing diagram of the adder and multiplier together. The signal of the multiplier starts with  $u_2/u_0/u_1$ , while the adder signal starts with  $u_2/u_0/u_0$ . A sum of four products and a sum of one product term can be seen.



The second calculation is that the adder sums the result of the product  $0 * 2$  with 0; the answer is presented on the D\_OUTPUT, when the READY signal of the adder becomes high for the second time. The signals START and NEW\_DATA give the repeated adder the

information which is needed to perform multiple additions in sequence. Each time new data is available for the adder the NEW\_DATA signal becomes high, while the signal START is high multiple additions are performed.

During the verification performed on the DigiLog multiplier and the recursive adder it can be concluded that both components are performing in the way as described in the specification. All components are working properly. Another step which can be made are the simulations of the entire neural hardware system. The content of the following chapter describes the translation of the characteristics of a trained neural network, such that the components with their functionality described in this chapter, combined with the correct characteristics of a trained neural network, would perform its task like the neural software in *InterAct*.

### 3.5 Realization

Manufacturers have seen the advantages of using a hardware description language to catch system behavior. Designers, who use this method for developing systems, have got the idea to use the VHDL language to transform the description of a system into a hardware realization. This transformation, called synthesis of VHDL-code, is chip, technology and manufacturer dependent. A shortcoming of synthesis tools is that not each description in the language gives a workable mapping on the used technology. Designers who use VHDL create a simulatable version of a system very fast, but the main problem is to compile it with a synthesis tool.

The system as described above is a simulatable system that performs well in simulations. But it is far from an optimized neural system solution. An optimized system can be produced only when a platform is chosen. The platform that will be chosen poses its own restrictions on the VHDL-code. These restrictions can mean, that an optimized system can not be synthesized at all. Therefore the system that is developed is only there for simulation purposes.

### 3.6 Summary

This chapter introduced the design trajectory used to translate the behavior of an artificial neural network into an hardware description. The design trajectory shows the three stages of design: architecture, implementation, and realization. The architectural phase of design describes the functional and a conceptual structure of the hardware system. This description of the functionality and the conceptual structure is used to create the VHDL description. The descriptions of these components are verified, which shows us, that the VHDL description conform to what was mentioned in the architectural phase. What follows is that the functionality of an artificial neural network and the description in a hardware description language, as VHDL, are acting in the same manner. So that we can speak of a hardware description of a neural network. Further the realization phase tells us that this system which performs to the same functionality as a neural network, is only developed for simulation purposes. A further development of the system after a platform is chosen, is time-consuming.

The first part of the paper discusses the importance of the study of the history of the world, and the second part discusses the importance of the study of the history of the world.

The first part of the paper discusses the importance of the study of the history of the world, and the second part discusses the importance of the study of the history of the world.

The first part of the paper discusses the importance of the study of the history of the world, and the second part discusses the importance of the study of the history of the world.

The first part of the paper discusses the importance of the study of the history of the world, and the second part discusses the importance of the study of the history of the world.



### 3.1 Summary

The first part of the paper discusses the importance of the study of the history of the world, and the second part discusses the importance of the study of the history of the world.

---

# Chapter 4

## *Tools used before Simulation of the Neural System.*

---

Until now the description of the neural hardware can only perform the functionality of a neural network. The neural network characteristics, which holds the synaptic weights and the biases, must be obtained from a trained neural network. Only by using those characteristics of an artificial neural network, gets the neural hardware description the property of an application-specific neural network. These network characteristics are translated from a trained neural network. The trained network is obtained from the software package *InterAct*, where it is tested and trained. This section describes in which way the network characteristics are translated to our hardware description. This in order to get an useful hardware description which holds the property of the translated application.

For the simulation of the whole system, the used pattern data base to train and test the network in *InterAct* must also be translated in away such that it is useful for our system. To handle each input pattern of the network for the simulation is a times taking business. Therefore a simulation environment is created in order to handle all patterns automatically. This simulation environment can be seen as the testing environment of the hardware system. In this chapter it will be told in what way this is achieved.

After simulation, the results which are obtained from the testing environment are analyzed to see the performance of the system. That problem: how do we measure the goodness of the fit performed by the hardware system with regard to the one trained by *InterAct*, or to the real train and test data set. This chapter will end with a discussion of these problems.

### 4.1 Simulations

Before simulations can be performed by the hardware description of a neural network, the characteristics and data to test with should be extracted from the *InterAct* environment. For this purpose a software tool is developed to achieve look-up tables with the characteristics of the artificial neural network, and tables with the test patterns. The tables with the test patterns are used by the simulation environment which supplies

the network of input. But also it must retrieve the output data from the hardware neural system, and store them in a file such that the behavior of the system can be analyzed. This section outlines the path to go, from a trained network in *InterAct* to a hardware model of a neural network such that it fits the *InterAct* model.

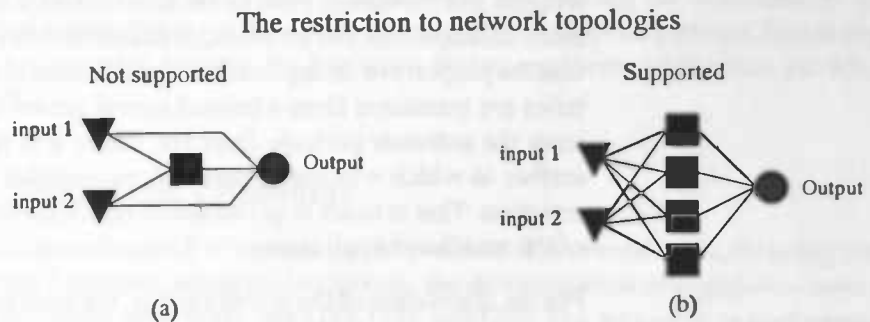
#### 4.1.1 Extraction Network Characteristics

Before a start is made, with extracting the characteristics of a neural network, it is important to know which properties satisfy a good translation: (1) the network topology must be a feed-forward neural network, (2) the network must be trained with input in the range  $[0, 1]$ , and (3) there must be an *InterAct* data file for a good translation.

The first constraint is on the network topology: only a topology of a feedforward structure of neurons is supported. This means that the neurons must be connected in such a way that only a single direction data-flow is permitted. A second restriction on the topology is that all connections are lag free, and that all signals pass all levels in consecutive order. Figure 4-1 shows two types of feed-forward multi-layer Perceptron networks. The topology of figure 4-1a is not supported by the hardware neural system, because the signals from the input to the output do not pass the hidden level. While the right figure shows the topology which is supported.

**Figure 4-1;**

The left network topology is not supported by the hardware neural system, because of the way of connecting the neurons together. The right figure is supported by the system.



The second constraint mentioned above is that the inputs must be in the range of  $[0, 1]$ , because the hardware system can only react on positive input numbers. This means also that the network which will be trained by the software package *InterAct*, must be trained with inputs in the range  $[0, 1]$ . The neural network obtained after the train and test phase in *InterAct* must be saved in the proper format of the software package. The saved network (net file) and the test data (pbf files) must both be named in the same way, such that our software application can extract the network characteristics and respectively the test and train data of the network.

The software which extracts the characteristics of the network and the pattern database, is outlined in appendix C. This tool is an *InterAct* application, which needs only the name of the network which will be translated, because the network and pattern database file are named with that name. The developed software application generates tables in the VHDL package file. Two VHDL files are constructed, the first one describes the characteristics of the network, while the second is a translation of the pattern database file.

The characteristics of the artificial neural network holds the information of the topology, and secondly the parameters of the network. The software tool makes the translation by searching the absolute largest parameter (bias or synaptic weight). If that parameter is found, the software application walks through the network and translates all the weights and biases to a finite set of numbers. The largest absolute parameter limits the range of all the parameters in the network, let say that the parameter is  $W_{\max}$  than the range is  $[-W_{\max}, W_{\max}]$ . All the network characteristics are uniformly quantized in  $2^N-1$  steps, where  $N$  is the number of bits that will be used by the hardware neural system. An exception is made to  $N$  if the value which will be converted to a discrete number is a bias, than  $N$  will be twice as large. All these values are then stored in the tables for the weights and one for the biases. The neural hardware system can use these tables in order to produce a proper output. Another characteristic must be entered to the neural system. The system must know in what order the network will be calculated. Therefore a part must be entered to that file by hand, as this is not generated. This part is the topology of the artificial neural network. The following four lines must be appended to that file;

```
constant Nr_layers : integer := 3;
type topology is array (0 to Nr_layers-1) of integer;
constant network: topology := ( 4, 5, 1 );
constant Nr_Neurons : integer := 10;
```

These four lines tell that the hardware neural system holds an artificial neural network with three layers (an input layer, one hidden layer and an output layer). The constant network shows the topology of the network, it says that the input layer consists of 4 input neurons, the hidden layer takes five neurons, and that the network has one output neuron. The last line of VHDL code tell that there are 10 neurons are involved in the network topology.

The second file generated contains the test and train patterns. Because the range is known (namely between 0 and 1), the uniform quantizer translates the input as well as the target patterns. For each input a separated table is constructed, that is also true of the targets. An example of these files is shown in appendix C, together with the C-code of the *InterAct* application.

#### 4.1.2 A Simulation Environment

The neural hardware system can be simulated by using the software package *V-System*. By simulation the system must act on input patterns to achieve a response of the system. Each input pattern consists of  $N$ -dimensional input vectors, where  $N$  is the number of inputs to the neural network. Each input vector should be offered to the hardware system. Because the system can consume only one pattern at the time, the pattern should be offered sequentially. This is also true for the output of the system. In case of a multiple output network the outputs vector are offered to the systems environment one by one. For the simulations that will be performed, the pattern database does not consist of only one input pattern. For the way of offering input vectors and retrieving output vectors, the issue of good timing for sending and receiving data is important. For this reason a simulation environment is written to achieve that each

---

pattern is correctly send to or received from the system. Another reason is that the large amount of data generated from the system can also be written to a file, without a description change of the system itself.

The simulation environment is also written in the VHDL language, and uses the table produced by the converting software describe above. The contents of the table are the patterns which are offered to the system, where each pattern exists of the input vectors and the corresponding output vectors of the network. The only task of the simulation environment is to offer the input data to the system in a proper way and to retrieve the output and store the data on a file. A disadvantage of the simulation environment is the dependence of the network topology. The simulation environment description changes when more or less inputs or outputs are described by the neural network topology. When the simulation environment is started by a signal, the whole pattern database will be offered to the digital neural system, and each output pattern of the system will be stored on file. After all the input patterns have passed, the simulation environment gives the end signal ready. This file with the response of the functional behavior of the neural hardware system can so be analyzed and verified. For the description of the simulation environment, see appendix C.

All the simulations are performed by the VHDL simulator *V-System*. This VHDL-*interpreter* can simulated VHDL-code after compiling it, without synthesing the system nor placing & routing the components on a chosen device. This advantage creates a designing environment which is suitable for researching the functionality of a described system behavior. This software package developed by Model Technology Inc. is therefore very suitable for hardware design engineers with some knowledge of VHDL.

## 4.2 Diagnostic Checking

The diagnostic check of models can be performed in several ways, by using auto correlation functions, using periodograms, the determination of signal-to-noise ratio's and in several other ways, see [18], [19], and [20]. The way of fit or performance between two or more systems, by comparing them to one another, residuals or errors can be obtained. The behavior of the error between the fitted system and the neural behavior of a network in *InterAct*, and the errors obtained by looking to the real train and test data gives us enough information for a good check. How to examine the plots which we obtain for the system errors will be based on some examples. The examples are shown in two categories, namely a good fit and a better performance.

### 4.2.1 Goodness of fit

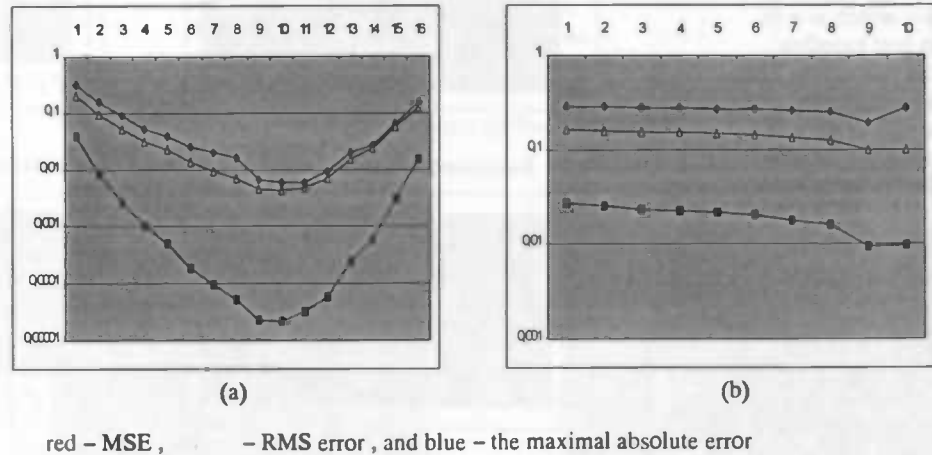
Typically the goodness of fit of a neural model to a set of data is judged by comparing the observed values with the corresponding values obtained from the fitted model. If the fitted model is appropriate, then the residuals should behave in a manner which is consistent with the model. This definition of the goodness of fit give an exact description. In our case the *InterAct* model of an artificial neural network is fitted into a hardware environment. The *InterAct* model of the network uses a infinite set of numbers to represent the variables of the network, while the network in a digital environment can only use a finite set of numbers.

Lets consider in figure 4-2, the errors between the two systems, where the errors are the mean square error, the mean square root error and the maximal absolute error. In figure 4-2a, the best fit is reached by a value of 10, and in figure 4-2b by 9 because the error values are the minimum. Thus the error functions between the InterAct system and the hardware neural system is the best when the error functions are minimal.

#### The Goodness of Fit

**Figure 4-2;**

Two example of goodness of fit. The shown error are the MSE, RMS, and the absolute error, on a logarithmic scale.



The maximal absolute error shows the largest error made between both systems. The digital system uses a binary representation for the numbers, thus (by using  $N$ ) the numbers of bit and the range of those numbers are known. The maximum absolute error tells us how many quantization steps we are wrong in the worst case. The MSE shows us the dispersal of the errors, or variance ( $\sigma^2$ ) of the error, and a reliability interval can be shown of the networks answers. The RMS error is showing the root of the MSE and is equal to  $\sigma$ . For the goodness of fit these plots are showing enough information to say on which point the best fit is reached.

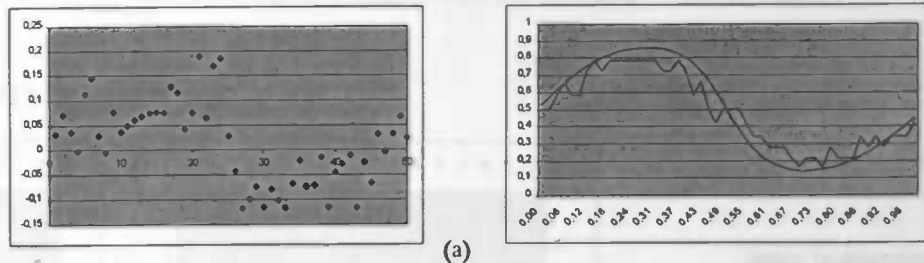
Another way to analyze the results is to look at the errors, namely the errors between the fitted system and the system which is fitted. Figure 4-3 is showing three different results, displayed by using scatter-plots and the actual response of the system. The goodness of fit is a measure for the structure of these errors. Unstructured errors (white noise) mean that the errors are identical independent distributed (i.i.d.). This holds that the goodness of fit can be related to the structure of the errors. By structured errors a worse fit is achieved than by not or less structured errors.

## Goodness of Fit by analyzing generated errors

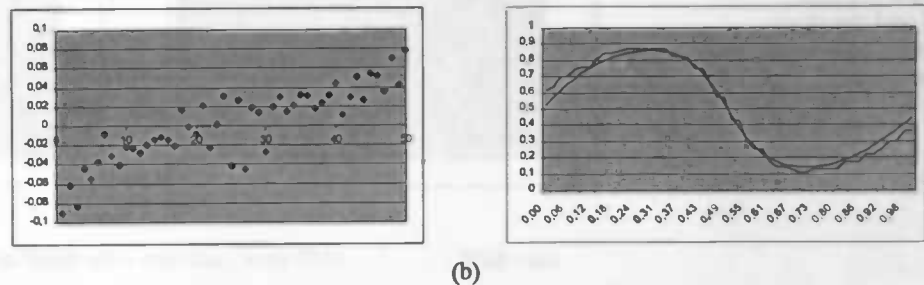
**Figure 4-3;**

Three examples of errors between the actual system and the system which is a fit of the first together with its response. The scatter-plots are showing the error of each pattern in relation to the amplitude.

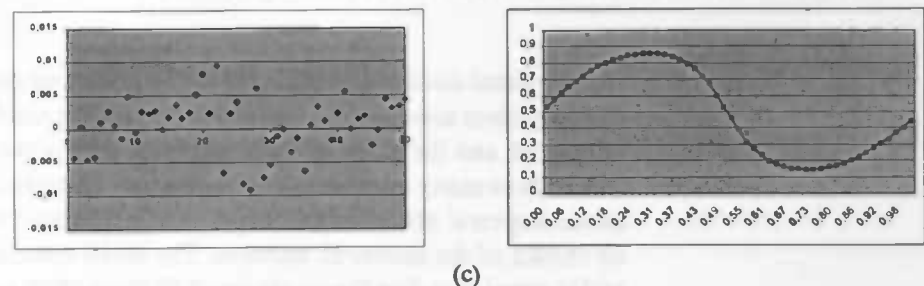
Response of a coding the discriminatory function by 6 bits



Response by 6 bits, where the synaptic weights are rounded and the inputs are jammed



One of the responses of a 7 bits synaptic weight, by addition of 1 random bit.



NOTE : in the right figures the blue line represents the fitted system, while the red line shown the response of the systems which is fitted.

The upper figures are showing the response of using a discriminatory function coded with 6 bits. The scatter-plot of the errors between both systems looks like a sinusoid, especially the errors belonging to the patterns in the range [10, 17]. Another observation can be made by looking at the dispersal of the errors: a larger dispersal indicates a worse fit while a small one is acceptable. The middle figure shows an error distribution with an upward trend (a line), which indicates a worse fit because it has no random behavior. For the last figure (bottom) the error is randomly distributed (unstructured) and has a small dispersal of the errors so a good fit can be concluded. The gradation of the fit is shown for the upper figures till the bottom. The response of the fitted system with regard to the system which is fitted is shown on the right side of the scatter plots in order to see what response belongs to the errors in the scatter plots.

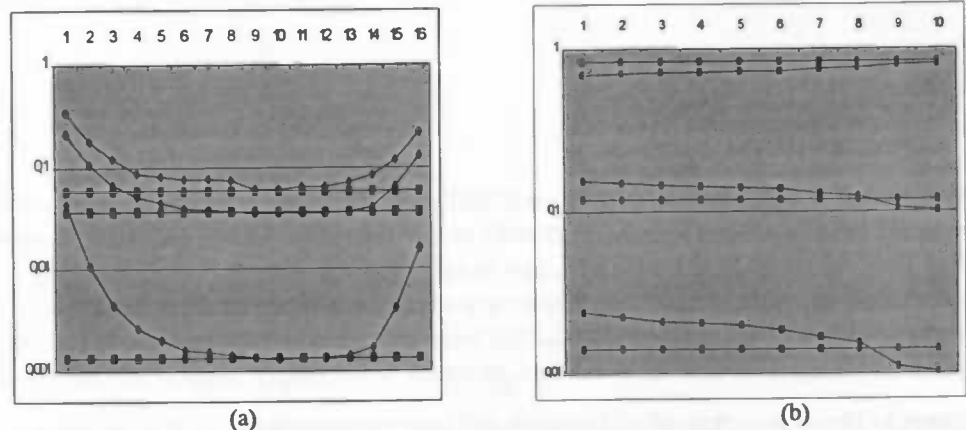
### 4.2.2 Performance

Neural networks are estimators of a real-world application, thus there is an error between the real world and both systems (the *InterAct* and our digital neural system). From figure 4-2 can the performance to the real-world not be shown. The goodness of fit, even when it is perfect, does not mean the best performance. To see which system performs the best, the errors must be calculated between the real world data and both systems. Lets show two examples, where parameters in the *InterAct* network are all constant while the parameters in the digital neural system uses several values for parameters.

Performance of both systems

Figure 4-4;

The performance of the two systems in regard to the real world. The upper error is the max. abs. error, middle RMS, lower error the MSE.



From the figure 4-4a, we see that the MSE and the RMS error are dropping under the constant line of the *InterAct* system, and mean a better performance. While the maximum absolute error stays above the *InterAct* network thus there is no improvement of the performance in terms of the maximum error. In figure 4-4b, all the errors are showing improvement to the performance, while the maximum absolute error is rising but the RMS and the MSE are dropping.

Thus judging the best fit does not gives information about the performance, or otherwise. For the best results a compromise between both constraints is much more likely to choose for the parameters of the digital neural system.

## 2.2. The $q$ -analogue of the binomial theorem

The  $q$ -analogue of the binomial theorem is a generalization of the binomial theorem to the case where  $q$  is a positive integer. It is given by the following formula:

$$(x+y)_q^n = \sum_{k=0}^n \binom{n}{k}_q x^k y^{n-k}.$$

where  $\binom{n}{k}_q$  is the  $q$ -binomial coefficient, defined by

$$\binom{n}{k}_q = \frac{(n)_q!}{(k)_q! (n-k)_q!},$$

and  $(n)_q!$  is the  $q$ -factorial, defined by

$$(n)_q! = (n)_q (n-1)_q \cdots 1,$$

where  $(n)_q = 1 + q + q^2 + \cdots + q^{n-1}$ .

It is easy to see that the  $q$ -binomial coefficient is a polynomial in  $q$  with integer coefficients. For example,  $\binom{n}{k}_q$  is a polynomial in  $q$  of degree  $k(n-k)$ . This is because the  $q$ -factorial is a polynomial in  $q$  of degree  $n(n-1)/2$ .

Figure 1



It is also easy to see that the  $q$ -binomial coefficient is a polynomial in  $q$  with integer coefficients. For example,  $\binom{n}{k}_q$  is a polynomial in  $q$  of degree  $k(n-k)$ . This is because the  $q$ -factorial is a polynomial in  $q$  of degree  $n(n-1)/2$ .

It is also easy to see that the  $q$ -binomial coefficient is a polynomial in  $q$  with integer coefficients. For example,  $\binom{n}{k}_q$  is a polynomial in  $q$  of degree  $k(n-k)$ . This is because the  $q$ -factorial is a polynomial in  $q$  of degree  $n(n-1)/2$ .

$$\binom{n}{k}_q = \frac{(n)_q!}{(k)_q! (n-k)_q!}$$

The  $q$ -binomial coefficient is a polynomial in  $q$  with integer coefficients. For example,  $\binom{n}{k}_q$  is a polynomial in  $q$  of degree  $k(n-k)$ . This is because the  $q$ -factorial is a polynomial in  $q$  of degree  $n(n-1)/2$ .

---

# Chapter 5

## *The discriminatory function with an area of effectiveness*

---

Non-linear threshold or discriminatory functions give an artificial neural network the property that there behavior is non-linear. Over time several functions have been developed and tested to get neural behavior as well, see figure 2-2. Some of them have no potential to realize practical solutions to a variety of problem, and have been developed out of academical research. The most useful and used ones are: piecewise linear functions, the tangent hyperbolical function, and the well-known sigmoid function.

The sigmoid is the most commonly used function within the software world of practical neural networks. But in hardware, other types of discriminatory function appear. A reason for this phenomenon is the practical implementation of such kind of functions. In the early years of neural hardware, the hard-limiter was a commonly seen function. These functions have been adopted, because they offer a good compromise between simplicity, with associated small area, and complexity [21]. Nowadays in our technical and technological inspired world, where digital memories and silicon wafers are cheap, the way of implementing is most used discriminatory function is based on the use of look-up tables. An connected phenomenon is that the technology of Boolean logic also has improved so that a hardware description language can be logical by optimized. Thus the logic requirements on silicon space are reduced. That is the advantage to move all the static data, that is necessary for a neural hardware system to play its role, in a table that can be optimized.

This chapter shows us how to implement an activation function like the sigmoid. But also the way of handling of such kind of implementation. And a little foretaste of accuracy and the influence will be given.

## 5.1 Activation function Implementation

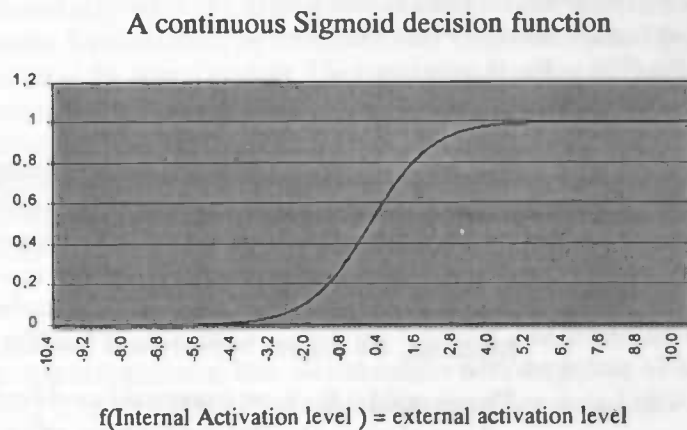
The sigmoid activation function is by far the most common one in the construction of artificial neural networks. It is defined as a strictly increasing function that exhibits smoothness and asymptotic properties. An example of the sigmoid is the logistic function, defined by

$$\varphi(x) = \frac{1}{1 + e^{-ax}} \quad (5-1)$$

where  $a$  is the slope parameter of the sigmoid function, see figure 5-1 with  $a=1$ . Whereas a threshold function assumes the value of 0 or 1, a sigmoid function assumes a continuous range of values from 0 to 1.

**Figure 5-1;**

This figure represents a continuous sigmoid signal. That can be obtained by using :  
 $f(x) = 1/(1+\exp(-ax))$



In chapter 3 an analysis of the value range within a network is given. The internal activation level of a neuron is not limited in value, and lies within the range  $[-\infty, \infty]$ . However the external activation is limited, what can be seen in the figure but also in the following equations.

$$\lim_{v_j^{(l)} \rightarrow \infty} \frac{1}{1 + \exp(-v_j^{(l)}(n))} = 1 \quad (5-2)$$

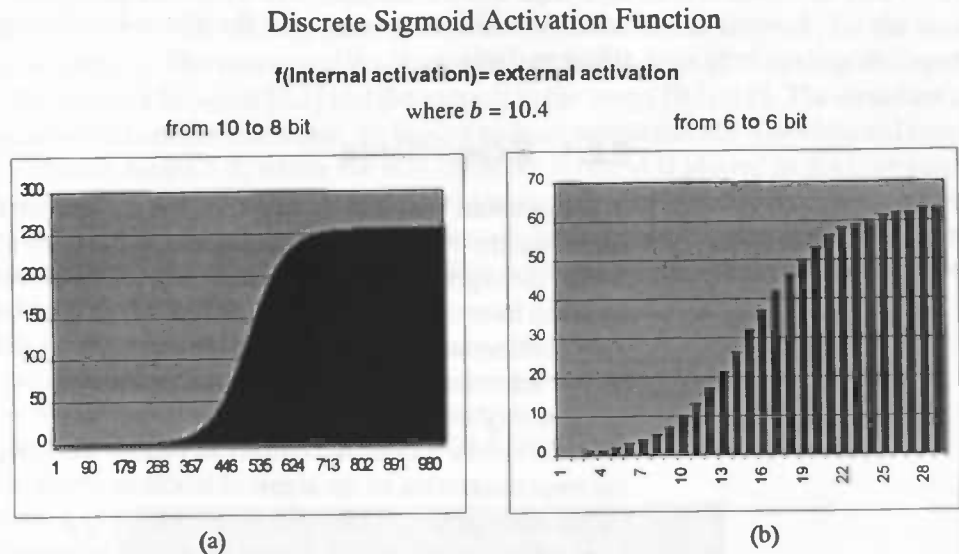
$$\lim_{v_j^{(l)} \rightarrow -\infty} \frac{1}{1 + \exp(-v_j^{(l)}(n))} = 0$$

The infinity of the internal activation range complicates the discretization of the function. To overcome this problem the internal activation range had to be limited. Lets assume that  $a$  and  $b$  are respectively the lower and the upper range of the internal activation. If we choose  $b = -a$  and  $b = 10.4$ , then figure 5-2a gives a graph of a discrete-sized sigmoid function, where the range  $[a, b]$  is divided into  $2^{10} - 1$  quantization steps (i.e. 10 bits precision). Each quantization step is at size  $b+a$  divided by the num-

ber of steps. The external activation is in this example divided into 255 quantization steps (8 bits precision) as large as  $1/255$ . From this graph the discretisation can not be seen; therefore figure b shows the same function but for less precision to effect that the steps of the digital function can be seen.

**Figure 5-2;**

Both figures are discrete sigmoid functions. The left figure (a), is a digital signal but its resolution is too high to see the steps of the function. Figure (b) shows a region for a lower resolution, the discretization steps are visible.



The VHDL description of the activation function is generated as a look-up table by a piece of software that is mentioned in appendix C. This look-up table for the sigmoid is not complete one. For the implementation the presence of symmetry is used: the table is filled with only the values right from the zero (in the range  $[0, b]$ ). If a internal activation is negative, the external activation is looked up for the positive corresponding value of the internal activation. The output of the sigmoid function will be one minus the look-up value.

For this implementation of the activation function the choice of  $a$  and  $b$  are now no longer two independent values. The reason is that if  $a$  and  $b$  are not each others opposite, the sigmoid function is not a symmetrical function anymore and can not be generated by the software tool. Thus  $a$  and  $b$  are the same variables, the choice of this variable is a little bit tricky. The value that is chosen for this parameter sets an operating area of the sigmoid discriminatory function. The following section will discuss the proper setting of that operating area and will explain why there is one: strange things can happen to the effectiveness of the network if the values to  $a$  and  $b$  is not set properly set.

## 5.2 An Effective Operating Area

The discriminatory function of the neural system is generated from set of parameters by a software program. This set describes the whole sigmoid activation function. The set of parameters consists of: the slope of the function, the lower and upper limit of the internal activation level, and the number of bits that will be used to code the values of the internal and external activation levels. The setting of these parameters have lots of influence on the systems overall behavior. For example if the range of the internal activation is not set properly, the result of the network is not corresponding with what

we think that will happen. Therefore this section will investigate the influence on the behavior of the neural system, by using several implementations of the internal activation. In other words, different values for  $a$  and  $b$ . The values in that range are called to be the effective operating region of the sigmoid activation function. The effective operating area will be selected from the performance of the neural hardware system with regard to a neural network with an infinite calculation precision. These experiments with different ranges for the internal activation will be followed by more detailed analysis.

### 5.2.1 Experiments

Before we start with the experiments, a short summary of the in's and out's is presented. As we know a neuron contains a (non-)linear transfer or activation function, in our design the sigmoid function is taken to fulfill that task. The sigmoid function transforms an internal activation of a neuron into an external activation (or neuron output). This continuous function must be translated into a discrete valued function to fit the digital behavior of the neural system. The translation is rather difficult because the internal activation lies within an infinite range. To overcome that a effective range must be taken before the discretization can follow. This range must be symmetric owing to the implementation of the sigmoid function. By choosing a set of parameters the behavior of the activation function is generated by a piece of software. This set of parameters describes the whole sigmoid function in order to create a VHDL-file.

In figure 5-3, a sigmoid is drawn to show the parameters which will be used in the generation of the discrete sigmoid activation function. The implementation of the activation function is symmetric, thus we can suffice with the points  $-b$  and  $b$ . The slope of the sigmoid is taken 1.0 for all the cases, except when stated otherwise.

**Figure 5-3;**

A sigmoid function with its equations to make it a discrete valued function.

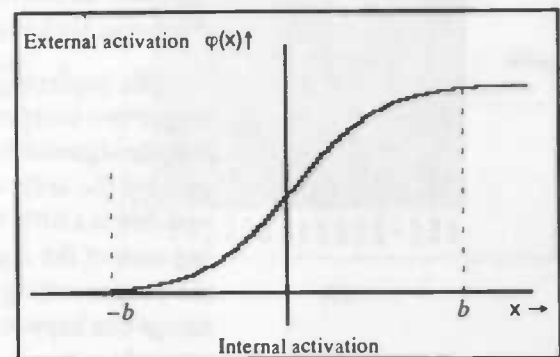
**Sigmoid function**

$$\varphi(x) = \frac{2^N - 1}{1 + e^{-s \cdot \mathcal{A}}}$$

Where  $\mathcal{A}$ ,

$$\mathcal{A} = \frac{2 \cdot b}{2^M - 1} \cdot x - b$$

$M$  is #bits coding of the internal activation  
 $N$  is #bits coding of the external activation  
 $s$  is the slope of the sigmoid function



Our goal is to set an effective operating area bounded in the range  $[-b, b]$ . Therefore experiments are done by using different values of  $b$ , such that the performance of the digital system can be compared with an artificial neural network simulated in *InterAct*. The simulation with *InterAct* can be seen as a hardware implementation that uses a much higher amount of bits, so that we can say that those simulations are in fact using a infinite range of numbers.

The following subsections describe the experiments and their results. Each experiment will show the performance of a simulation with infinite precision and the limited

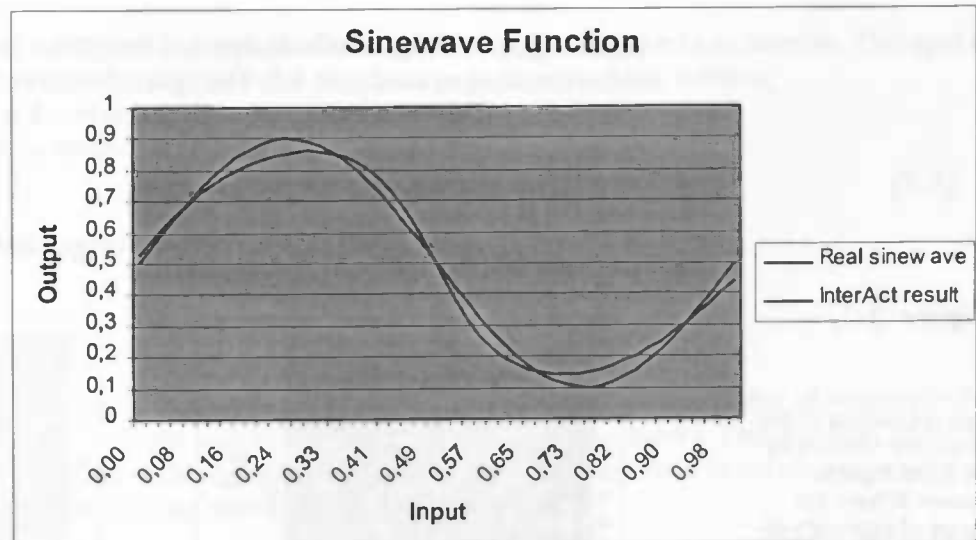
one. After the results of the experiments are given, a short analysis of the experiment will show what is done.

### 5.2.1.1 Sine Function

This experiment shows what happens if the internal activation is discretized for different values of  $b$ . The starting point is a trained artificial neural network, for the sine-wave function. The training of the sine-wave function is done after scaling the inputs of the network between  $[0,1]$  and the outputs in the range  $[0.1, 0.9]$ . The structure of the network contains one input, six hidden and one output neuron. The obtained result is shown in figure 5-4, where the real sine-wave function is plotted in the blue color, and the network response in red. From the trained network the synaptic weights and the neuron biases are obtained, in order to translated them to digital values which can be used by the neural hardware system. This translation is also performed on the input and target of the train data.

**Figure 5-4;**

The sine-wave function, where the input is scaled in the interval  $[0,1]$  and output in  $[0.1, 0.9]$ . The same story for the response of the trained sine-wave network.



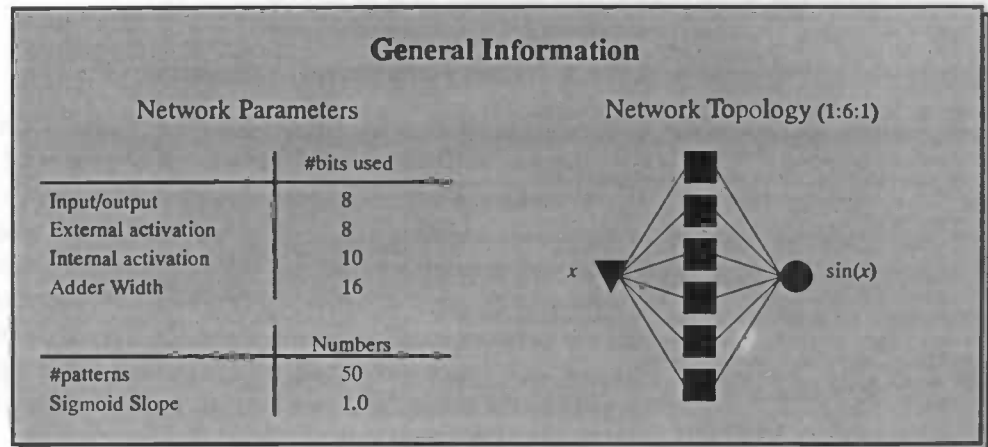
From the figure 5-3 can be seen that the trained network with infinite data precision matches not exactly the real sine-wave function. The *InterAct* network has an error with respect to the real sine function. These errors are approximately:  $\max.\text{error}_{\text{abs}} = 0.0591$ ,  $\text{error}_{\text{mse}} = 0.0013$ , and  $\text{error}_{\text{rms}} = 0.0363$ . These errors are a measure of the approximation quality of the hardware system, and a goodness of fit can be concluded from these errors.

#### 5.2.1.1.1 Results

The neural hardware system is simulated by use of the VHDL-simulator *V-System*. These experiments, sixteen in number, differ in the selection of the sigmoid activation range (or the value of  $b$ ). The other parameters of the network are constants. The following Sidebar 5-1 shows these constant factors.

### Sidebar 5-1;

General information on the sinewave experiments. It contains the network parameters and the network topology.

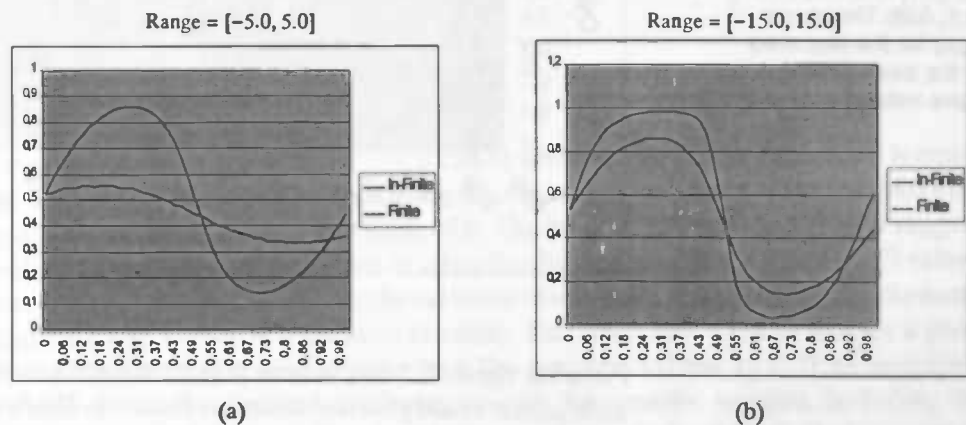


By changing the range of effectiveness of the sigmoid (other values for  $b$ ), the output behavior changes, see figure 5-5. The figure shows the response of the InterAct (infinite precision) vs. the response of the digital network with an 8 bit precision. The difference is only the range of the internal activation of the neuron.

The output behavior by two different ranges of the sigmoid's effectiveness

Figure 5-5;

Two responses of the sinewave function by an 8 bits input is shown. Where the range of sigmoid's effectiveness are different, shown is the behavior by the ranges  $[-5, 5]$  (left) and  $[-15, 15]$  (right), with regard to the InterAct response.

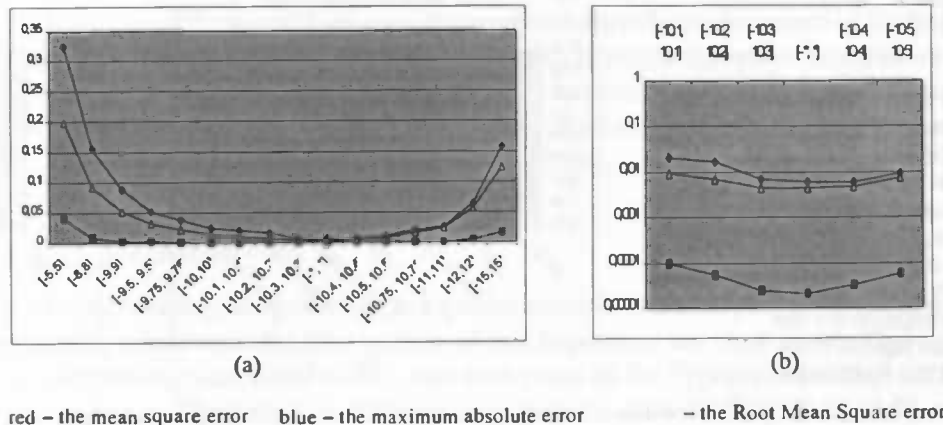


Another result is the error, that can be calculated for the available data. The interesting errors types, are the maximum absolute error, the mean square error, and the root mean square error. These kinds of errors can be viewed with respect to the real data, the InterAct response and the data obtained by hardware simulations. Figure 5-6 shows the errors between the output response of a system with infinite calculation precision (InterAct) and the digital neural system.

**Figure 5-6;**

The left draft (the bath-tub-curve), shows the three errors over all the experiments done, for a variety of values for  $b$ . On the right a small part is shown on a logarithmic scale.

The errors between a the networks with finite and infinite precision.  
For several ranges of the sigmoid's effectiveness



In this figure one typical number is missing, this is given as an asterisk. This spot in the table carries the value of  $b$  that can be formulated as follows,

$$* = \text{Max} \left| w_{ij}^{(l)} \right| \quad (5-3)$$

with,

$$\forall l \in [1, L]; \forall i \in [1, N]; \forall j \in [0, M]$$

and where  $L$  the number of layers of the network,  $N$  as the number of neurons in the preceding layer, and  $M$  the number of neurons in the layer  $l$ . This equation gives the maximum of the absolute weights and biases in the network. In the sinewave network the value for the asterisk is 10.32.

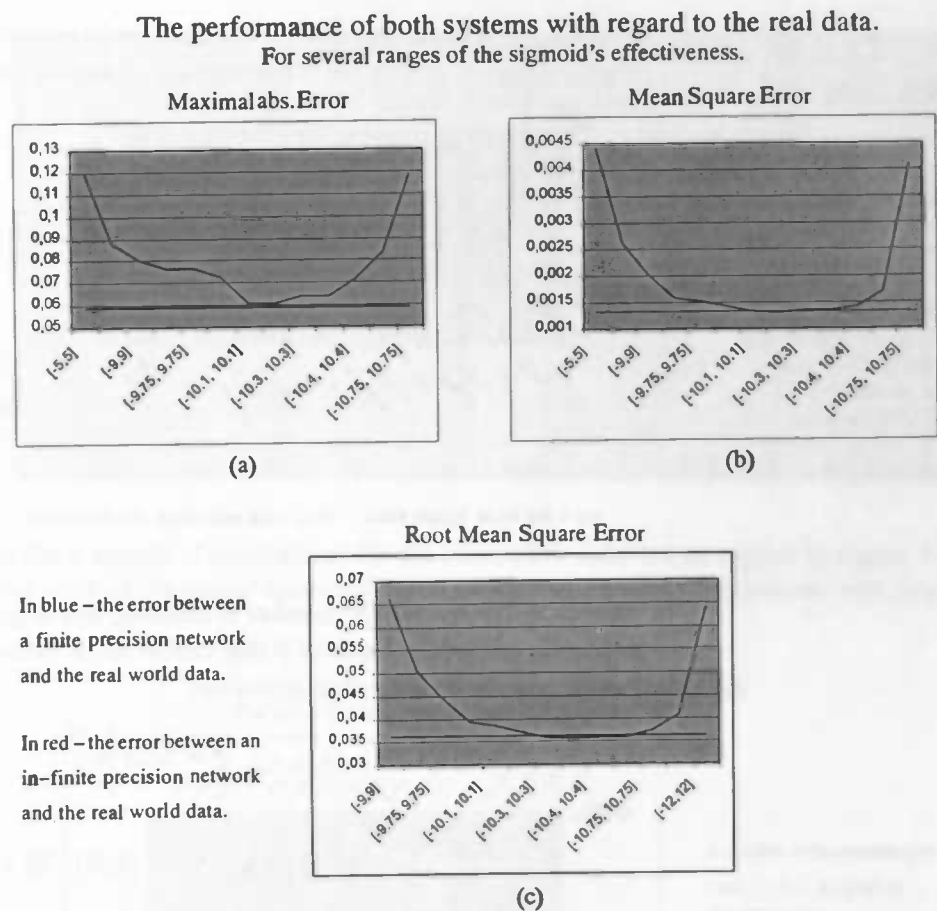
The results between the infinite (*InterAct*) and finite calculation precision (digital neural system) is nice to know, but what are the consequences with regard to the real data? In figure 5-7, these results are shown. In the upper left (a) the maximum absolute error is shown, in blue for the finite calculation system to the real world data, and in red for the infinite (*InterAct*) response. The same story can be told for the upper right figure (b), but the error that is presented here is the mean square error, and the graph at the bottom shows the root mean square error. The following section outlines an analysis of the shown sinewave problem results.

#### 5.2.1.1.2 Analysis of the Results

This section will tell us what can be observed for the graph shown in the previous section. The explanation why the neural hardware systems behavior changes by using other effective ranges of the sigmoid function, will be deferred until all experiments are done.

**Figure 5-7;**

The performance of both system in regard to the real world sine-data by using several values of  $b$  is shown here. Figure a shows the max.abs.error of both systems, (b) and (c) respectively the mean square error and the RMS error.



From the graph shown in figure 5-5, it can be observed that if the value for  $b$  is small the response of the hardware system does not match the response of the *InterAct* system. This is also true for large values of  $b$ . These responses to a change of  $b$  suggest that we force the sigmoid function in saturation by large values for  $b$ . For small values the sigmoid function behavior is almost linear instead of a non-linear (these phenomena are also known in transistors circuits). This observation tells us that for a good fit of both systems the usable value for  $b$  lies between 5.0 and 15.0. If we take for  $b$  a value around the absolute maximum over all the synaptic weights, including the biases, the best fit is given. This conclusion can be made from the bath-tub curve in figure 5-6.

By the maximum absolute value over all the weights and biases as a setting for  $b$ , the error between the simulation with an infinite and finite precision matches almost with 0.5% root mean square error. The mean square error is then on its lowest point, and means that the variance of the errors decreases if the values for  $b$  are around point \*. The largest error (the absolute) describes also some curve like that of the RMS-error, but stays the largest one (0.006). This value for the absolute error is large as the input and output have a resolution of 8 bit. That means that the maximum error made is as large as two quantization level.

To know which of both networks matches best on the sinewave function, we must take a look at the performance of both networks with regard to the real sinewave data (fig-

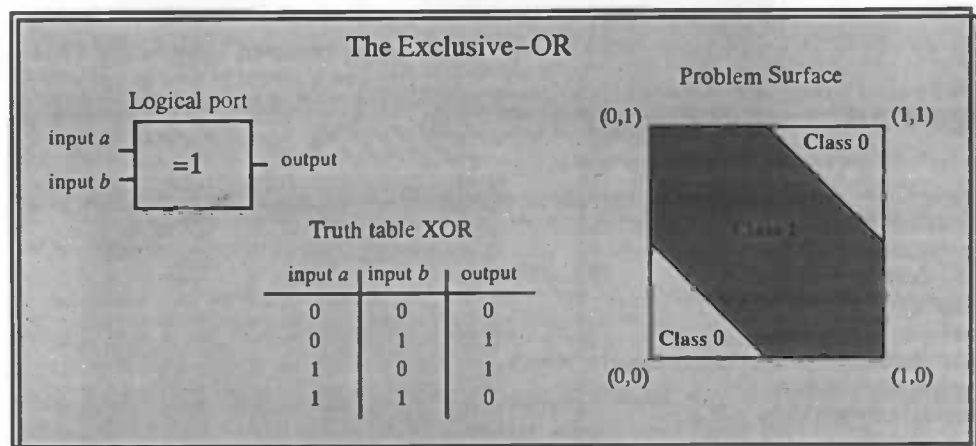
ure 5-7). The straight line in the performance of the network reflects the use of infinite numbers. The explanation why these error are constant is that that system has not changed at all. The curved line shows the performance of the hardware system. When both systems have the same performance with regard to real sinewave data, they are joint. If the line of the neural hardware system goes below the performance of the InterAct network, then the system with finite precision matches the real-world sinewave function better than the infinite one. The neural hardware system performs the task of the sinewave function better than the *InterAct* version when  $b$  is equal to the maximum absolute values over all the weights and biases.

### 5.2.1.2 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.

#### Sidebar 5-2;

This Sidebar shows the behavior of the XOR, and the problem surface map. Also a truth table is included.



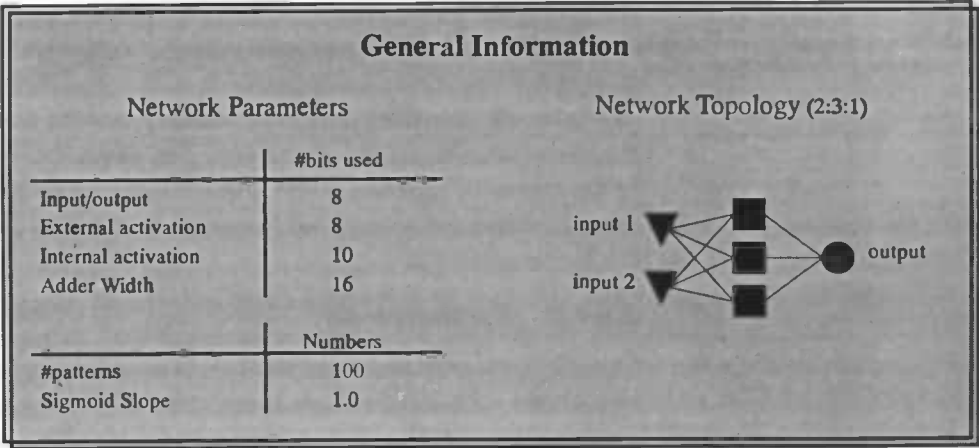
The exclusive-OR problem is not a linearly separable. This shows from Sidebar 5-2. If a problem is linearly separable the two classes can be divided by a straight line. In the case of an XOR this is not possible.

#### 5.2.1.2.1 Results

This classifying experiment uses the same constant parameters as the sinewave function experiment so that they can be compared. This experiment uses all the information as given by the Sidebar 5-3. The network topology is different as the XOR network has 2 input neuron, 3 hidden, and 1 output neuron. Another thing that is different is the amount of data patterns that are offered to the network.

**Sidebar 5-3;**

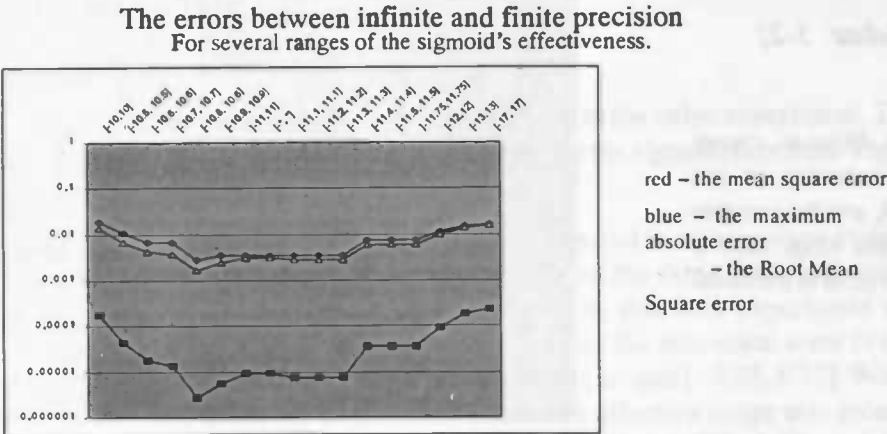
General information on the XOR experiments. It contains the network parameters and the network topology.



After a couple of simulations the the results are obtained as appear in figure 5-8 for the error of the neural hardware system with regard to the simulations with *InterAct*.

**Figure 5-8;**

The errors between the systems with finite and infinite precision by using different values for *b*. The graph show the MSE, RMS, and the maximum absolute error on a logarithmic scale.



In the graphs (figure 5-9) of the errors of a finite and infinite precision with regard to the real-world data, the red lines are the error signals of the XOR problem produced by *InterAct* (infinite data and calculation precision). The blue error signals are retrieved from calculations of the real-world data with regard to finite data.

5.2.1.2.2 Analysis of the Results

The experimental results of the XOR experiment which are shown in the previous section will be analyzed next from a comparison between the sinewave results and the XOR results. First we will start with the observations that can be made from the results outlined in the previous section.

**The performance of both systems in regard to the real XOR data**  
For several ranges of the sigmoid's effectiveness.

**Figure 5-9;**

The performance of both system in regard to real XOR data, by several values for  $b$ , is shown here. Figure (a) shows the max.abs.error of both systems, (b) and (c) respectively the mean square error and the RMS error.

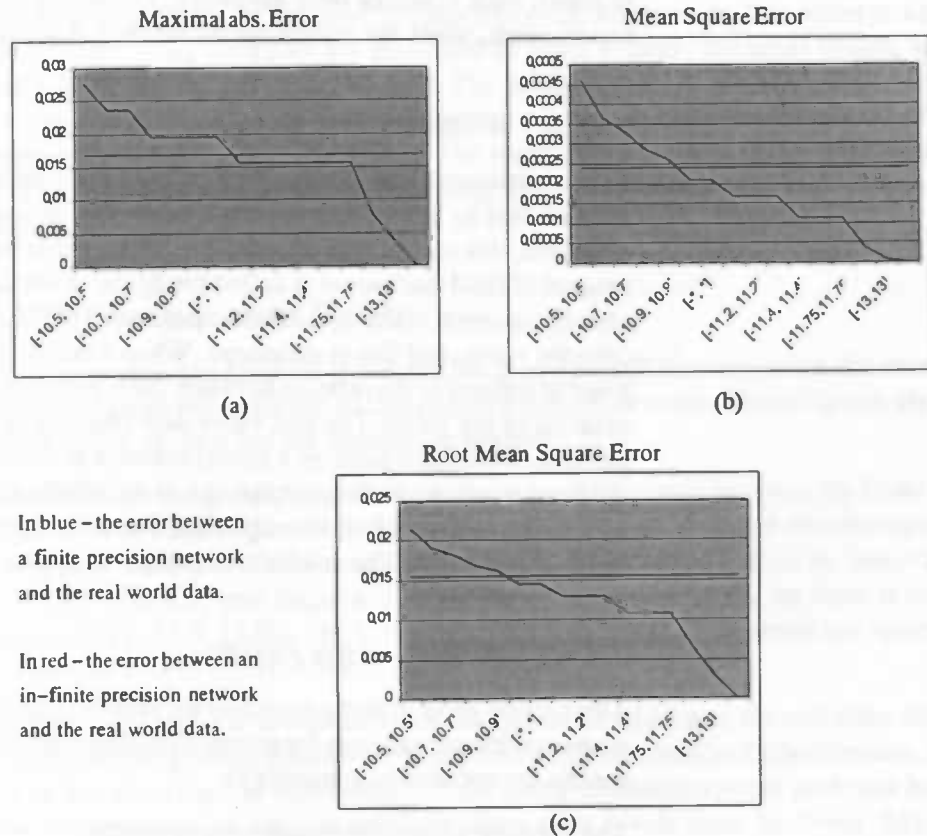


Figure 5-8 shows the errors of the digital system when compared to the *InterAct* network. These error-curves show that the range of internal activation of a neuron has influence on the performance of the network which uses finite precision. The error gives a kind of quality mark of the mapping between the systems with finite and infinite calculation precision. When the errors in both systems are on the lowest level, the best mapping of a neural network with infinite precision into a network with finite precision is achieved. That means: if  $b = 10.8$ , the best mapping of both system is achieved. But if we look at the graphs that show the mapping of the performance of both systems compared with real data, another conclusion can be made. These performances appear in figure 5-9. When the blue line is lower than the constant red line of the *InterAct* simulation, the performance of the digital network is better compared to the real data. The performance with regard to real data is improving, when larger values are chosen for  $b$ . The real XOR data is matched by  $b = 15.0$ . This explains why the error curves in figure 5-8, are rising again. The mapping becomes bad for large values for  $b$ , while the hardware systems performance compare to the real data is better.

From the results of the XOR experiment compared to those of the sinewave experiment, it can be observed that the mappings have almost the same shape. But with regard to the real data, the performance is rising in the XOR but dropping in the sinewave experiment for larger values for  $b$ .

---

To ensure that our discussion is correct, two other experiments have to be done. One of these experiments is a classifier and the other one approximates a function. Maybe these two classes of applications have different properties, when a translation is made to match their behavior onto hardware. The following sections will introduce those experiments, while the results are mentioned in appendix D.

### 5.2.1.3 A Valve

This experiment considers the flow of fluid that passes a valve. The amount of fluid is controlled by the valve's position. When the amount of fluid that passes the valve increases, this means that the position of the valve has changed. The change of the amount of fluid that passes is called the  $\Delta flow$  or the difference in amount before and after the moment of change. Another parameter of this valve is the *pressure* of the fluid onto the valve, and this is measured. When a neural controller must be designed the point of interest is the relation between these parameter ( $\Delta flow$  and *pressure*) and the position of the valve. The real valve data consists of 1000 input patterns and target patterns. With these patterns a neural network is trained and recalled by two systems the first system performs calculations on an infinite range of numbers and the second uses only a sub-set of numbers (finite). The following section shows the analysis for those experiments; The results are outlined in appendix D.

#### 5.2.1.3.1 Analysis of the Results

This section gives a short analysis of the results obtained in the valve experiment. The nine experiments are done for several effective ranges for the sigmoid function. These results are outlined in appendix D.

The valve experiment indicates a lot of similarities compared to the sinewave experiment. The experimental results of the valve shows almost the same bath-tube curve as in the sinewave experiment. An explanation could be that both experiment are approximating a function. But the error graph shows that the minimum error is obtained if the effectiveness range of the sigmoid function is around  $[-8.75, 8.75]$ . When this is compared to the sinewave experiment, where the effective range was around the maximal absolute values of the synaptic weight and biases, a slide difference can be seen. This range gives in comparison to the infinite system the best mapping of the valve application. Lets see what the systems performance is when it is compared to the real valve data.

In figure D-2, where both systems are shown with regard to the real valve data, it can be seen that only the maximal absolute error is on the lowest level for a particular value of  $b$  (by 9.0), even below the line of the system that uses the infinite set of numbers. But the other errors (MSE and RMS error) do not cross the line of the infinite system: they are rising again if  $b$  passes the value of \*. It can be observed that for all possible values of  $b$  (by a reasonable error in comparison to the real valve data or to the infinite system), the effective range for the sigmoid is much larger compared with the sinewave experiment. Another observation can be done with regard to the errors: the errors in the valve experiment are also larger than the one produced by the sinewave network. The discussion why these behavior changes happen if only the range of effectiveness of the sigmoid function is changed, will be outlined after the analysis of the next experiment, the Iris classifier.

#### 5.2.1.4 The Iris Classifier

This is perhaps the best known database in the pattern recognition literature. Fisher's paper [22] is a classic in the field and is referenced frequently to this day, see [23]. The data set contains 3 classes of 50 instances each, where each class refers to a type of Iris plant. Four instances are presented as input vectors; the sepal length, sepal width, petal length, and the petal width. The problem consists of one class that is linearly separable from the other 2; the latter are not linearly separable from each other. Predicted attribute: class of Iris plant. The experimental results of the Iris classifier are shown in the appendix D. The following section contains the analysis of those results.

##### 5.2.1.4.1 Analysis of the Results

The results obtained from the simulated Iris classifier will be analyzed in this section. Here the figures are analyzed so that conclusions can be made. These figures are not found in this chapter but are depicted in appendix D.

A measure for the goodness of fit can be seen from the errors between the finite and the infinite system. Figure D-3 shows the error for several areas of effectiveness of the sigmoid function. Those errors are continuously decreasing when the interval of initially  $[-8.0, 8.0]$  gets larger and larger until a certain moment, the error is rising again (after  $[-15.0, 15.0]$ ). The best fit can be found where these errors are minimal, at a certain area of effectiveness.

On the other hand, the performance of the system in relation to the real data, shows us if the mapping onto the hardware improves by different area's of effectiveness. Figure D-4 is showing this relationship. If the neural hardware system performs better than the system with infinite precision, then error levels must be lower than the constant line (red) of the system with infinite precision. The results of the Iris class Setosa stays above the red line and the system is not performing better than the infinite system. But as the area of effectiveness is chosen in the range of  $[-*, *]$  the errors, with regard to the real data, are dropping rapidly. But if the range becomes larger, the errors are rising again. Until at a certain point the errors drop again and are staying there, but the system performance is not improved. For the other two Iris classes an otherstory is told, these error are dropping smoothly while the area of effectiveness becomes larger. Until the range of the sigmoid is limited between  $-10.0$  and  $10.0$ , at this turning point the errors are smaller than the ones obtain for the system with in-finite number representation. The change in performance is also clear in table D-1, the number of patterns which are correctly classified increases after a area of effectiveness of  $[-10.0, 10.0]$ . This means that the area of effectiveness enlarges the distance between the winner and the runner-up. This phenomenon is also seen in the XOR experiment, the larger the effective range of the sigmoid function the better the system with the finite set of number will perform.

#### 5.2.1.5 Summary

From all the analyzed figures, the first thing to mention is that the class of application has influence on the area of effectiveness. In function approximation, such as the sine-wave and the valve experiments, the errors compared to the real data follow a bathtub curve. At the minimal value of that curve (errors minimal) the system performs

best. Those two experiments show no particular improvement of the performance with regard to the system that uses infinite representation. On the other hand, the two classification applications are showing a performance increase, when the effective range of the sigmoid is large enough. At that moment that the neural hardware system improves the performance compared to the real data system, but the mapping of the infinite system becomes worse.

Another phenomenon is that the area of effectiveness for a good performance is different for all the experiments, done in this chapter. For the performed experiments the minimal area of effectiveness is  $[-8.0, 8.0]$  and the largest  $[-20.0, 20.0]$ . But the largest interval of this area is not researched.

### 5.2.2 The Discussion

The phenomenon is that the neural hardware system's behavior is changing by modifying the area of effectiveness. For the change in behavior of the system a statement is outlined in this section. The problem is the internal activation of a neuron: it is not limited to a range of values. Therefore it is difficult to translate to a world where only limited values can be presented, thus the internal activation of the sigmoid function must be limited by specifying a certain range, let's say  $[-b, b]$  for a value  $b$ . Now the translation to a limited set of values is much easier. The problem is how to choose a proper value for  $b$ .

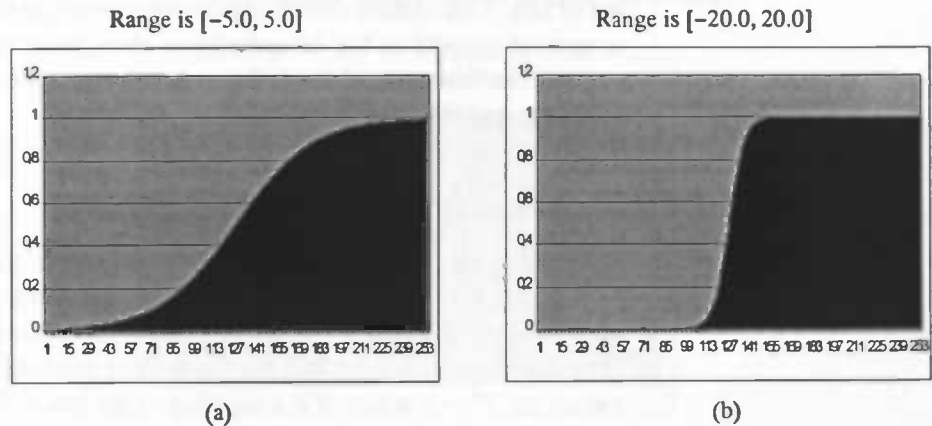
What is happening to the discriminatory function for a certain value for  $b$ ? The non-linear sigmoid, which is used by neural networks as an advantage above other controllers, is now the bottle neck. Assume a small value of  $b$ , then only the range of  $[-b, b]$  is translated. If  $b$  is for example 5.0, then the range of translation is  $[-5.0, 5.0]$  of the sigmoid function. In figure 5-1, a sigmoid is shown in the range of  $[-10.4, 10.4]$ , if only the part in the range of  $[-5.0, 5.0]$  is translated the function result looks like a linear function. Vice versa, for large values for  $b$ , the sigmoid is then translated over its full range the sigmoid's result is practically 1 or 0. This can be realized if the value for  $b$  is chosen in such a way that the error which is made can be neglected. But then another problem arises.

Let's assume that the translation that must be made uses only 255 values. That means that by translation the whole range of  $[-b, b]$  must be divided into 255 values. If other values for  $b$  are chosen the number of useable values stays the same, and means that the distance between them is changing. Let's assume that  $b$  is small, then the distance between two values is  $A$ . If another value for  $b$  is chosen, which is larger than the one mentioned before, the distance between two values becomes larger than  $A$ . Which means that how larger the interval, how the larger the steps of the translation are. The consequence of this phenomenon is that the steps may become so large that the effect of the non-linear behavior is totally eliminated. The result is a linear or in worst case a hard limiter. This is shown in figure 5-10. The left figure shows a translation where the number of values is limited to 255, and the value for  $b$  is small. The right part shows for the same number of limited values a translation, with a large value of  $b$ . The right part of the figure 5-10 shows almost a hard limiter, while left appears to be linear. Another observation is that the slope of the sigmoid is changing for other values of  $b$ .

### The effect of modifying the effective area of the sigmoid function

**Figure 5-10;**

The left figure is a translation of a sigmoid function with a area of effectiveness of  $[-5.0, 5.0]$ , and the right one with  $[-20.0, 20.0]$ .



Another statement that is made is why each application has its own area of effectiveness. From the results obtained in the experiments it is clear that the change in behavior of the systems by modifying the area of effectiveness can be divided in two classes. These two classes are the known application area's of the networks. The first class contains networks for function approximation, and the second one holds classification networks. The first class response on a slight difference in the area of effectiveness, is followed by an decrease in performance. The second class of problems (classifiers), the story is a little bit different. If the area of effectiveness is large, or the sigmoid function is replaced by almost a hard limiter, the performance increases.

Lets consider a problem of the first class, as know the characteristics of a neural network correspond to the synaptic weights and the neuron biases. The values of these characteristic parameter of the network are also translated to a binary code. The full range of synaptic weights must also be divided into a limited amount of values. Lets assume that those characteristic parameter lies in the range  $[-W_{\max}, W_{\max}]$ . And the input vector to the neural hardware system are in the interval  $[0, 1]$ . The parameters are translated to integer values, the number of values are for example  $2^N - 1$  where  $N$  is a member of the interval  $[4, 12]$ . These numbers divide the whole range of the parameters of the network. The input vectors are also coded, but the number of values is smaller,  $2^{N-1} - 1$  for the same value for  $N$ . The distance between the coding of the parameters and of the input vectors is not the same. This sources errors during processing, which can be illuminated by use of the discriminatory function, by setting a good area of effectiveness. Because each neural network has different weights and biases, this explains the different areas of effectiveness for each problem of that class.

A problem of the second class performs only better by an area of effectiveness which is large. That means that the discriminatory function changes from a sigmoid to a linear or even a hard-limiter. An explanation can be that classifying systems must decide to which class the input vector belong. To this purpose the sigmoid function may be too smooth and another discriminatory function works better.

### 5.3 Reduction of Accuracy

The area of effectiveness researched in the previous section, is related to a certain number of bits. This section shows experiments wherein the area of effectiveness is coded in several amount of bits to investigate changes in behavior of the neural hardware system. For this research the following two experiments are performed: the sinewave problem, and the Iris classifier.

#### 5.3.1 Experimental results

For the experiments a certain area of effectiveness must be chosen. For the sinewave experiment and for the Iris classifier is the maximum chosen of the absolute weight and biases. This is for the sinewave problem the area of  $[-10.31, 10.31]$  and for the Iris classifier  $[-8.63, 8.63]$ . This range is translated to integer values: the number of values are  $2^N - 1$  where  $N$  is a member of the interval  $[4, 13]$ . Those  $2^N - 1$  numbers are dividing the whole area of effectiveness into equal parts. For each part the function result of the sigmoid is calculated and stored in the look-up table of the hardware neural system. The two experiments are repeated for several values of  $N$ . The way of passing the sum of products to the discriminatory function is implemented in such a way that from the 16 bits result only the  $N$  highest order bits will be passed. The lowest order bits are disposed by the rounding technique truncation.

##### 5.3.1.1 Results Sinewave function

The parameters which are used to obtain the following results are mentioned in Sidebar 5-4. For the values of  $N$  the interval  $[6, 13]$  is chosen. The following graphs will show, (a) the response of the finite system with regard to the in finite system, and (b) the finite and infinite system compared to the real data. For both graphs hold that it is done with respect to several values for  $N$ .

#### Sidebar 5-4;

General information on the sinewave experiments. It contains the network parameters and the network topology.

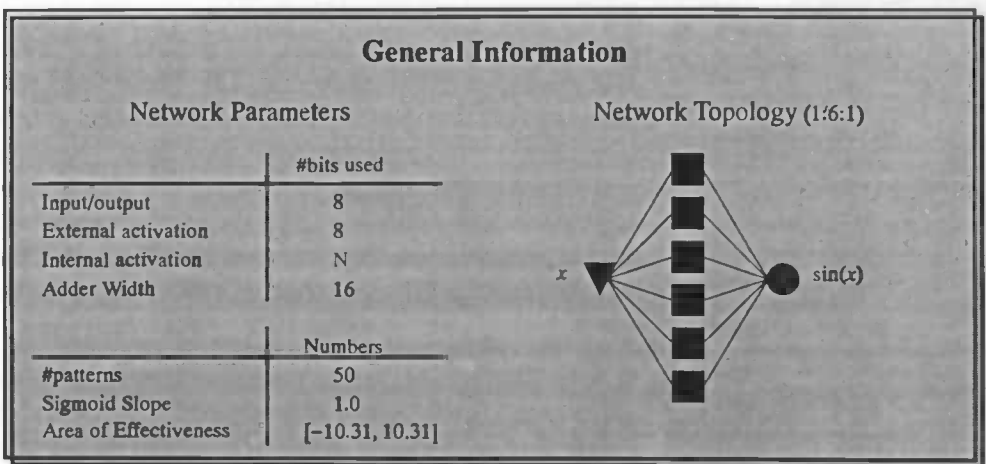


Figure 5-11 shows the relationship between the finite and infinite system, where the curves describes the change of the errors between both system for various values of  $N$ . From this graph a measure for the goodness of fit can be obtained. The best fit is given by the smallest values of the errors.

### The errors between infinite and finite precision For several numbers of bits to code the sigmoids effectiveness.

**Figure 5-11;**

This figure show the relationship between the finite and infinite system, with respect to the number of bits use that codes the area of effectiveness.

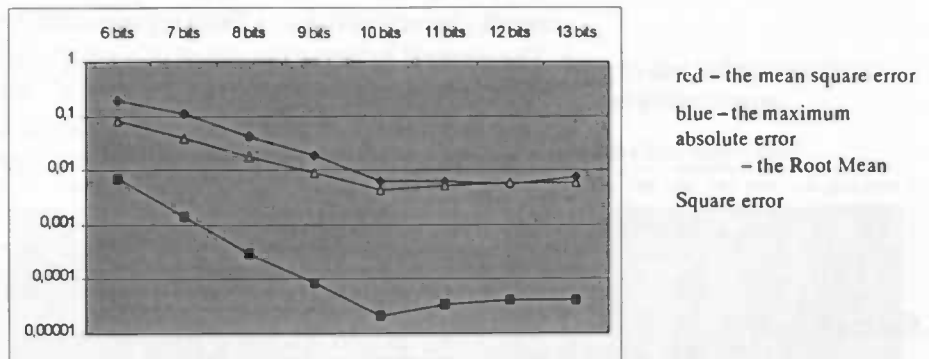
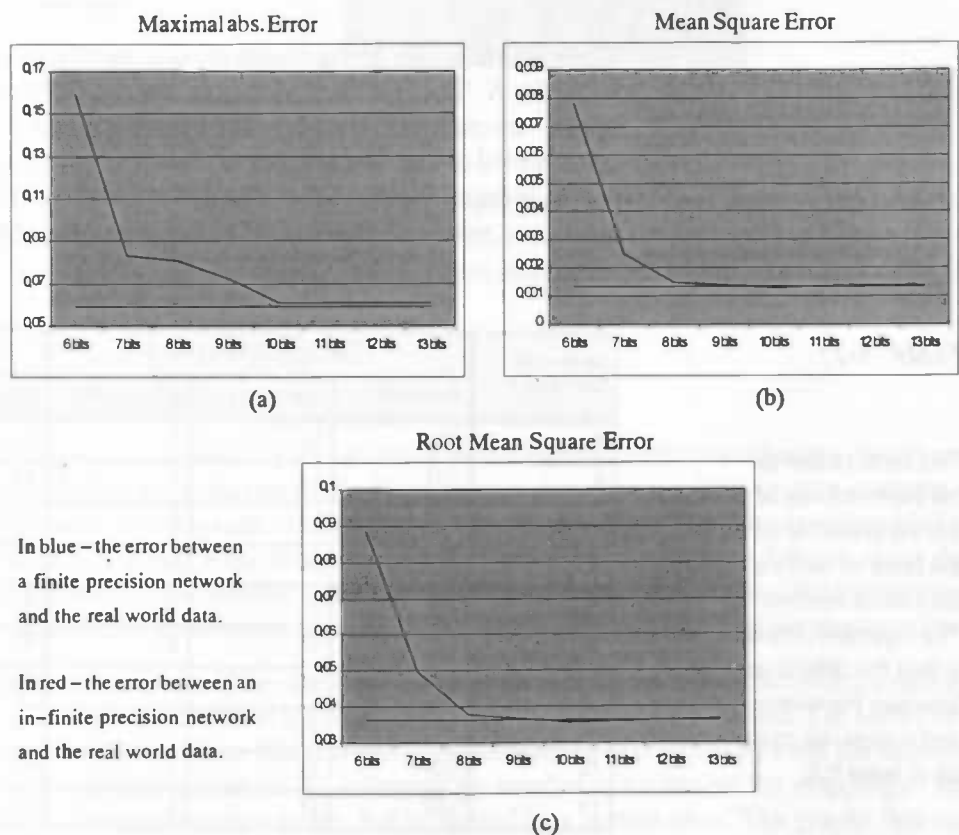


Figure 5-12 shows both system (finite and infinite) compared with the real sinewave data, for several values for  $N$ . This plot gives us information about the improvement that is occurred between both systems. If the line of the finite system crosses the line of the infinite system, an improvement of performance is a fact. That means only that the errors of the finite system are smaller than the errors of the infinite system.

### The performance of both systems with regard to the real Sinewave data For several number of bits used for coding the sigmoid's effectiveness.

**Figure 5-12;**

The graphs shown represent the error made between both systems compared to the real data set. This all under the circumstances that various value for  $N$  are taken.



### 5.3.1.2 Results Iris classifier

This experiment discusses the results obtained from simulations performed by *Inter-Act* using the infinite set of numbers, while V-system uses the finite set of numbers on the neural hardware system, as describe in chapter 4. The conversion of the infinite network characteristics to a finite representation is performed as described in chapter 5. The general information about this experiment can be found in the Sidebar 5-5. These experiments are performed for several values of  $N$  to get information about the change in behavior as well as the performances of the finite system compared to the real data and in comparison with the infinite system.

#### Sidebar 5-5;

General information on the Iris experiments. It contains the network parameters and the network topology.

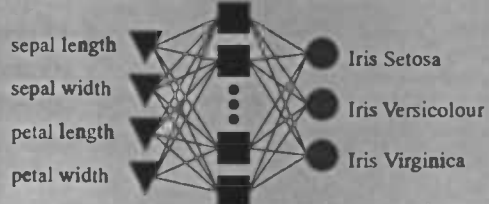
General Information		
Network Parameters		Network Topology (4:8:3)
	#bits used	
Input/output	8	
External activation	8	
Internal activation	$N$	
Adder Width	16	
	Numbers	
#patterns	118	
Sigmoid Slope	1.0	
Area of Effectiveness	$[-8.63, 8.63]$	

Table 5-1 shows the performance of both classifying system with respect to the chosen value for  $N$ . The rejection criteria is that the difference between the winner and the runner-up must be at least 0.5. The infinite system is misclassifying one input pattern out of a total of 118. The number of classified patterns and the rejected ones is constant for the infinite system, it classifies 107 correctly and it rejects 11. The number of classified and rejected patterns for the finite system, changes with the values of  $N$ .

Table 5-1;

This table presents the performance of the Iris classifier for the finite as well as the infinite system. The rejection criteria is that the difference between the winner and runner-up must be at least 0.5.

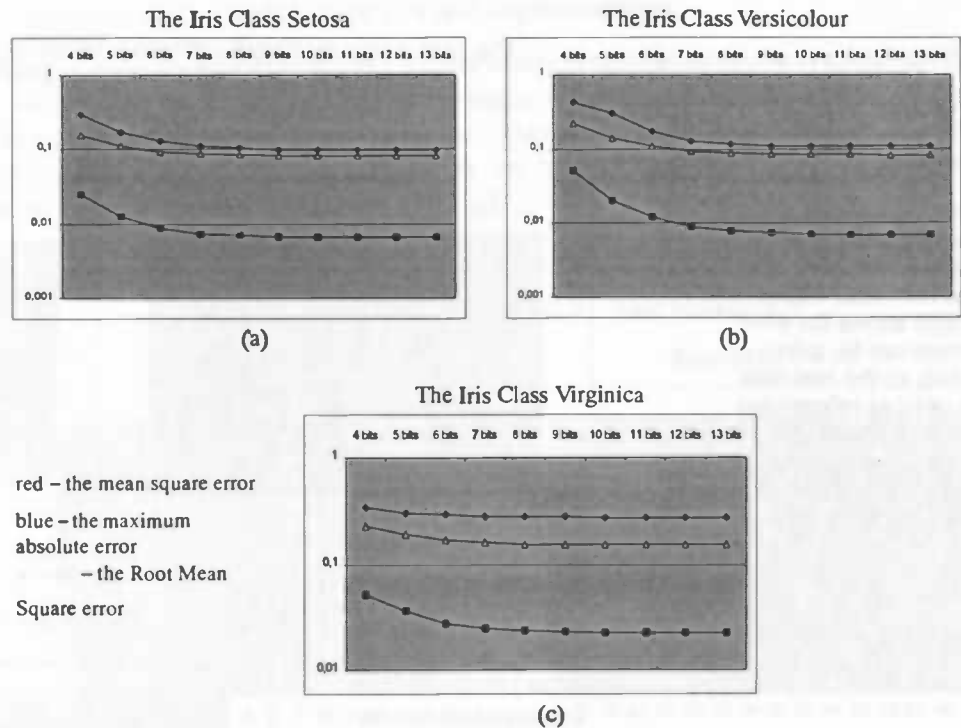
number of bits ( $N$ )	The in-finite system			The finite system		
	classified	rejected	misclassified	classified	rejected	misclassified
4	107	11	1	11	99	8
5	107	11	1	45	70	3
6	107	11	1	73	43	2
7	107	11	1	82	34	2
8	107	11	1	87	30	1
9	107	11	1	89	28	1
10	107	11	1	90	27	1
11	107	11	1	90	27	1
12	107	11	1	91	26	1
13	107	11	1	91	26	1

Figure 5-13 shows the performance of the digital network in relation with the one trained and tested by InterAct, for each Iris class. This graph shows also the goodness of fit between both systems, for various values for  $N$ . For this experiment the used integer values for  $N$  are in the interval  $[4, 13]$ .

**Figure 5-13;**

For each class of Iris the performance is shown in relation with the use number of bits to code the area of effectiveness. As reference is taken the infinite system.

The performance of the finite system in comparison to the infinite system  
For several number of bits used for coding the sigmoid's effectiveness.



The last result which is obtained is the response of both systems compared with the Iris database. Figure 5-14, shows for each class of Irises the errors of the finite system in blue and the infinite system in red. These errors are calculated by several value of  $N$ .

### 5.3.2 Discussion

The experiments whereby the accuracy of the internal activation changes (by  $N$ ) are showing that the behavior of the finite system also changes. This section contains a discussion of the results obtained from both experiments. The first experiment that will be mentioned is the sinewave problem, after that a discussion follows about the results of the Iris classifier. The last discussion of this section is about the similarities between both experiments. But first we start with the results of the sinewave problem.

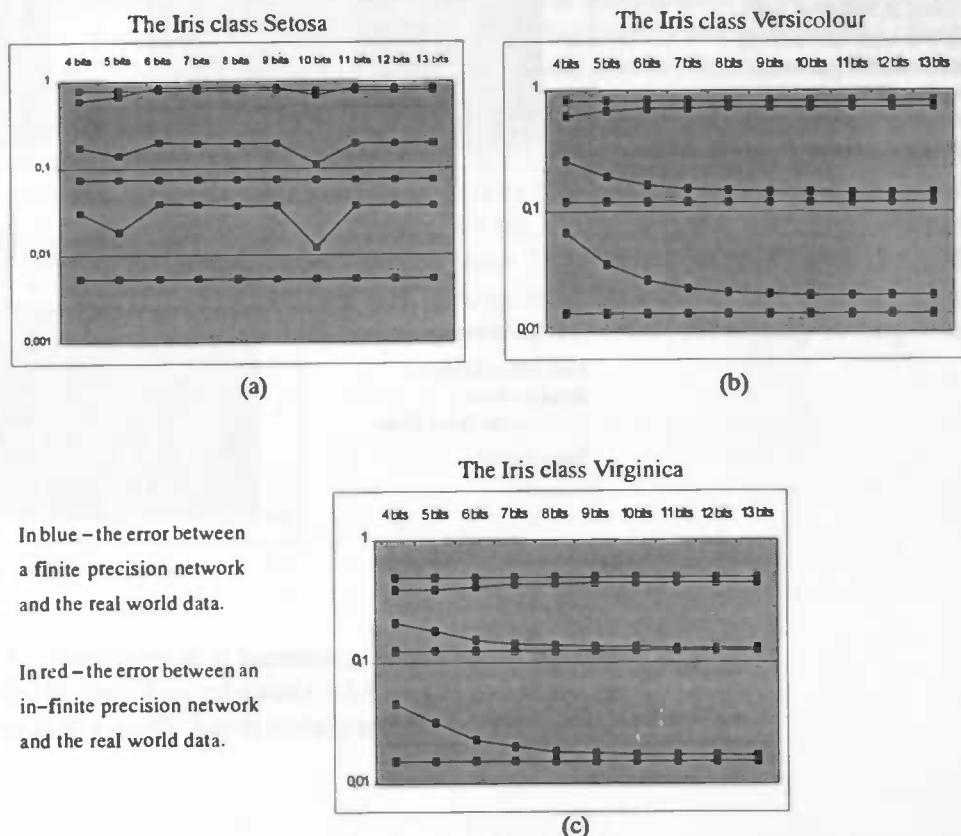
The approximation of the sinewave function shows that by coding the internal activation through small values for  $N$  gives relatively the larger error values. The errors between the finite and in-finite system, are descending exponentially until the number number of bits reaches 10. By enlarging the number of bit further the mapping of the infinite system becomes worse, but is limited by a certain error. The graphs that are

showing the finite and infinite results in comparison with the real sinewave function, show a similar behavior. For small values for  $N$ , the error compared with the real data is large, but for approximately 8 or 9 bits the mean square error and the root mean square error are stable close to the performance of the infinite system. The absolute maximal error shows that kind of behavior only after using 9 bits. For  $N = 10$  bits on only a very little improvement can be observed of the RMS error, (smaller than 0.1 %).

The performance of both systems compared to the real Iris data base  
For several number of bits used for coding the sigmoid's effectiveness.

Figure 5-14;

This view shows the performance of both system in regard to the real data. Each graph shows the error which can be calculated, as the real data is used as referencing point. It shows of each class the max. absolute error, MSE, And the RMS error.



NOTE: The upper lines are the maximal absolute errors, the middle is the RMS error, and the lowest lines represents the MSE.

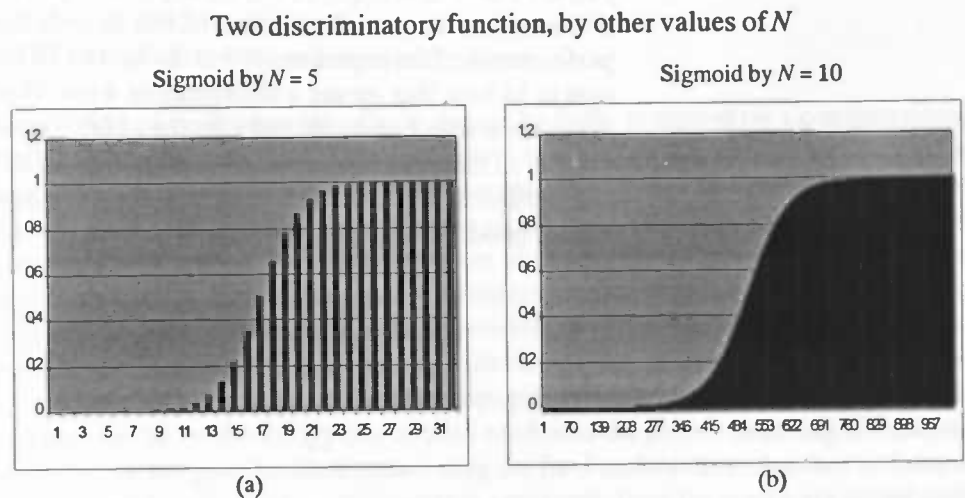
The Iris classifier shows approximately the same results. The errors between the finite and infinite system is also descending exponentially, but the error settles on a certain value at about  $N=8$ . This is true for all three Iris classes. The performance of both systems related to the real Iris database shows that the performance of the infinite system is the asymptotic minimum of the errors for the finite system. Except for the Iris Setosa class, the errors to the real data are at two points the lowest, namely by an accuracy of 5 and of 10 bits ( $N$ ). For all further number of  $N$  the performance is showing no improvement.

These two experiments are showing a lot of similarity by changing the accuracy for coding the area of effectiveness of the discrimination function. It can be seen that for a small amount of bits the performance is becoming worse. But also that after a certain number of bits the performance settles at an error level, it has than no use to enlarge the number of bits for coding the area of effectiveness. Overall the performance between the finite system and real data does not improve, for any number of  $N$ . For small numbers of  $N$  the loss of information is so high that the performance is bad, but at the same time a large number is carrying no useful information to the system and holds an enlarging of the needed space in a real implementation.

What is changing about the discriminatory function that has such power to change the systems behavior. Figure 5-15 shows two sigmoid functions. The left part of the figure shows a sigmoid function for  $N=5$ , and the right one for  $N=10$ . From this point of view the sigmoid function shows no shape loss, but only a loss of accuracy in comparison to the sigmoid functions of  $N=10$ , and  $N=5$ .

**Figure 5-15;**

Two sigmoid functions are shown each with another value for  $N$ , but for the same range that must be coded. Left is the value for  $N=5$ , and right  $N=10$ .



## 5.4 Conclusions and recommendations

The conclusions which can be made after the interesting experiments of this chapter will be mentioned in this section. After the conclusions some recommendations will follow to stimulate the research in the area of hardware neural systems. First the conclusion which can be made of the area of effectiveness of the discriminatory function will be given, followed by those for the research of using more or less accuracy for coding the internal activation. And last but not least the recommendations are given at the end of this chapter.

The area of effectiveness of the discriminatory function is different for two type of applications, if the performance on the real data set is the main objective. For function approximation the area of effectiveness is bounded to obtain the best performance. Another conclusion can be made if the application is a classifier. The performance of the classifier shows by a large area of effectiveness a higher performance than the one which is obtained by the system that uses the infinite set of numbers. The area of effectiveness has direct influence on the slope of the discriminatory function, and by the classifying system even on the distance between winner and runner-up (large area of

effectiveness  $\rightarrow$  distance also larger). If the criterium is the best mapping of the infinite system, then for both types of applications there can be a solution.

The accuracy needed to code the area of effectiveness has an optimal value for the number of bits. By a small number of bits the performance with regard to the real data as well as to the mapping of the infinite system becomes worse. But for a large number of bits the effect of performing better is canceled out. The performance by changing the number of bits does not lead to a better performance than for the infinite system or for real data. The response of the system, by any number of bits to code the internal activation of the discriminatory function, has the same effect on classification or function approximating systems.

An overall conclusion is that the performance to real data will be improved for classification systems by a wide area of effectiveness, such that the slope of the discriminatory function changes. It can not be improved by using more or less bits to code a chosen area of effectiveness, because the slope of the discriminatory function stays the same.

Further research is required on, the dependency or independency of the area of effectiveness in relation to the number of bits to code that area. Another point is that the performance of the experiments was the best by 10 bits of coding. But the sum of products is 16 bits, that means a difference of 4 bit. That means that the lowest order bit are calculated but are truncated after the adder. The question is: can the adder be made smaller in size, and can we stop the DigiLog multiplier earlier when the residuals of the multiplication have the highest order on the  $p$ 's position, for which  $p$  is the performance good.

---

# Chapter 6

*An evaluation of the effects of the error generation and propagation.*

---

The problem of evaluating the effects caused by finite precision on a computational flow plays an important role in designing and configuring the architecture related to an algorithm before its implementation on a physical device. In general this aspect is tackled empirically by configuring first the architecture and the precision to represent values and then testing the loss in accuracy at the device output with respect to the ideal algorithm. A different approach, based on sensitivity analysis as done in [24], [25] and [26], may be considered, which models finite precision errors and their propagation along the computational chain up to device output. In this chapter an evaluation of the effects of the error generation and propagation along feedforward network sheds light on how the methodology can be used to choose the proper rounding techniques for the inputs, weights, hence dimensioning the final architecture. Another technique which is used in real-time environments is evaluated. Here the inputs are added with an i.i.d. (identical independent distributed) values, hence to use a smaller representation for the input vectors, or even the synaptic weights of a neural system, in order to achieve a good fit and performance.

## 6.1 Sources of Quantization Errors

In a digital signal processing system, the most common signals that are processed are signals with a discrete time scale and the corresponding values are discrete. But in the real world of engineering, these signals are seldom present in this form. Therefore a conversion must take place before the signal can be processed. After processing, the result must be converted back into its real world environment, such that it can react.

Quantization is the conversion of a discrete-time continuous-valued signal into a discrete-time, discrete-valued signal. The value of each signal is represented by a value selected from a finite set of possible values. The difference between the unquantized input and the quantized output is called the quantization error (or noise) and it is desirable to minimize the perceived magnitude of this error. In order to achieve this objective, several quantization techniques can be used, e.g. uniform quantization, logarithmic quantization, non-uniform quantization and vector quantization. All

these quantization schemes can be made to adapt to the input waveform's statistics so that the quantizer will always be locally optimum and provide the highest possible quality with the lowest possible bit rate. Unfortunately, in practice, as the efficiency of the quantizer increases so does its complexity and, therefore, its cost.

In this work, the method used is a linear or uniform quantization. This means that the distances between all the reconstruction levels are the same. They make no assumptions about the nature of the signal being quantized. For this reason they normally do not give the best perceived performance. If a number does not map exactly to one of the quantization levels, it must be truncated, jammed or rounded to the closest level. These three common methods used for finite precision computation are analyzed by [26] and [27].

The error generated by truncating, jamming, or rounding techniques maybe considered to be a discrete random variable distributed over a range determined by the specific technique being employed. The following assumptions are made:

- Errors are random variables with uniform distributions except as noted.
- Errors are independent of each other and of all input and outputs.

For a statistical view of the error, it is desirable to know the mean and variance of the error generated by each of these three techniques.

### 6.1.1 Rounding techniques

Here, we consider a number quantized to a reasonable large number of bits ( $N_{max}$ ), whereby  $N_{max}$  is much larger than the representation of the number in  $N$  bits. Then the rounding operator chops off the  $q$  lowest order bits of a number which will have its new lowest bit in the  $2^q$ -th place. If the  $q$  bit value chopped off is greater than or equal to  $2^{q-1}$ , the resulting value is incremented by "1"; otherwise, it remains unchanged, ( $N = N_{max} - q$ ). The error  $e$  generated by the rounding operator which is uniformly distributed in the range of:

$$\left[ -\frac{\Delta_{max} \cdot 2^q}{2}, \frac{\Delta_{max} \cdot 2^q}{2} \right] \quad (6-1)$$

whereby,  $\Delta_{max}$  is the quantization level of the least significant bit, by the representation of  $N_{max}$  bits. The mean error can be calculated by

$$E_e = \frac{\frac{\Delta_{max} \cdot 2^q}{2} + \frac{-\Delta_{max} \cdot 2^q}{2}}{2} = 0 \quad (6-2)$$

and the variance is

$$\sigma_e^2 = \frac{\left( \frac{\Delta_{max} \cdot 2^q}{2} - \frac{-\Delta_{max} \cdot 2^q}{2} \right)^2}{12} = \frac{\Delta_{max}^2 \cdot 2^{2q}}{12} \quad (6-3)$$

### 6.1.2 Jamming techniques

The conditional jamming operator<sup>1)</sup> chops off the  $q$  lowest order bits of a number and forces the new lowest order bit, in the  $2^q$ -th place, to be a "1" if any of the  $q$  removed bits were a "1"; otherwise, the new lowest order bit retains its value. This operation is equivalent to replacing the  $2^q$ -th bit with the logical OR of the  $2^q$ -th bit and the  $q$  bits which are chopped off. The error  $e$  generated by the jamming operator is not exactly uniformly distributed as the probability of the error being zero is twice the probability of the error holding any of its other possible values.

$$p_e = \begin{cases} \frac{1}{3}, & e = \Delta_{max} \cdot 2^q \\ \frac{2}{3}, & e = 0 \end{cases} \quad (6-4)$$

For the statistical view of the error  $e$ , the mean is given by,

$$E_x = \sum_i p_i x_i \quad (6-5)$$

The mean of the error distribution is then,

$$E_e = \frac{2 \cdot 0}{3} + \frac{\Delta_{max} \cdot 2^q}{3} = \frac{1}{3} \cdot \Delta_{max} \cdot 2^q \quad (6-6)$$

For determination of the variance, equation (6-7) will be used as follows,

$$\sigma^2 = E[(x - E_x)^2] = \sum_i p_i (x_i - E_x)^2 \quad (6-7)$$

Substituting the mean and its probability, values in expression (6-7), gives

$$\sigma_e^2 = (0 - \frac{1}{3} \cdot \Delta_{max} \cdot 2^q)^2 \cdot \frac{2}{3} + (\Delta_{max} \cdot 2^q - \frac{1}{3} \cdot \Delta_{max} \cdot 2^q)^2 \cdot \frac{1}{3} \quad (6-8)$$

$$\sigma_e^2 = \frac{2}{3} \cdot \frac{1}{9} \cdot \Delta_{max}^2 \cdot 2^{2q} + \frac{1}{3} \cdot \frac{4}{9} \cdot \Delta_{max}^2 \cdot 2^{2q} = \frac{6}{27} \cdot \Delta_{max}^2 \cdot 2^{2q} \quad (6-9)$$

<sup>1)</sup> Patent Pending, Adaptive Solutions, Inc., 1990.

### 6.1.3 Truncation techniques

The truncation operator simply chops the  $q$  lowest bits off of a number and leaves the new lowest order bit, in the  $2^q$ -th place, unchanged. The error  $e$  generated by truncation is a uniformly distributed in the range of

$$[0, \Delta_{max} \cdot 2^q] \quad (6-10)$$

with each of the  $2^q$  possible error values being of equal probability. Therefore, the mean and variance of the error  $e$  are as follows,

$$E_e = \frac{\Delta_{max} \cdot 2^q}{2} \quad (6-11)$$

$$\sigma_e^2 = \frac{(\Delta_{max} \cdot 2^q)^2}{12} = \frac{\Delta_{max}^2 \cdot 2^{2q}}{12} \quad (6-12)$$

## 6.2 The Influence of Rounding Techniques

In order to see the consequences of the three different rounding techniques, two different type of networks will pass this experiment. For these experiments the network characteristics and test data must be quantized by a large number of bit, to be precise 12 bits for the network characteristics and a 14 bits representations for the input patterns. Now the above mentioned rounding techniques can be use to round, or to truncate, or to jam the values to reduce the number of bits. The input vectors will be reduced to a 8 bit representation. The reduction of the network characteristics holds for both the synaptic weights and the neuron biases. The only reduction is made to the synaptic weights in order to get no word width difference in producing the product. The neuron biases are in all the cases truncated to 2 times the word width of the synaptic weights. The influence of the biases is canceled by the input of the discriminatory function. Because the word resolution of the adders result is 16 bits, while the input of the discriminatory function will be 10 bits which means that four bit will be lost. This means that the influence of any way of rounding performed on the biases are canceled out after the discriminatory function. Hence the results of the entire system only reacts on the way of rounding the inputs and synaptic weights.

### 6.2.1 The Experiments

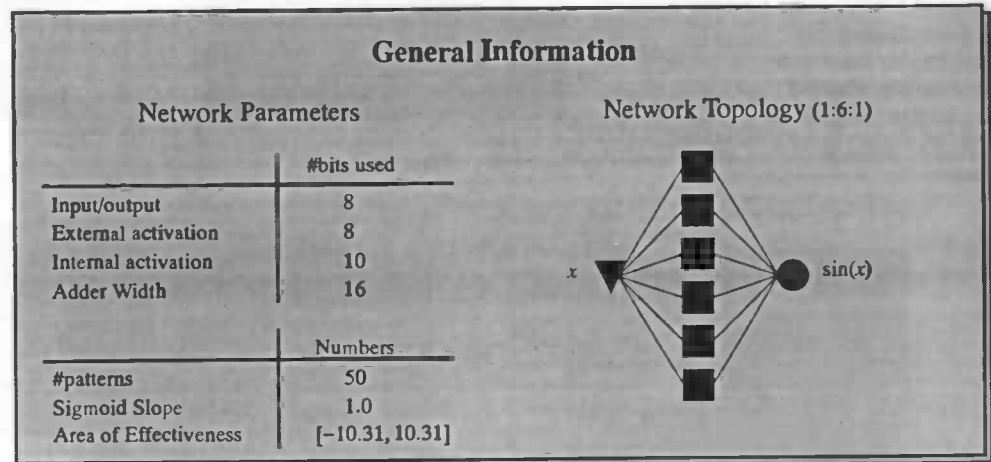
These experiments are performing all possible combinations of the three rounding operators on the input as well as the synaptic weights of the network. The neural applications used are the sine function and the Iris classifier. The following section will show the experimental results obtained from both experiments. Each result is accompanied with an observation and a short analysis.

### 6.2.1.1 Experimental results of the sine function

The experiment performs all nine combinations (of the three rounding techniques in combination with the input and the synaptic weights) on the sine function. Sidebar 6-1 is showing the general information about the setting of the internals of the hardware system. Besides the network parameters the network topology is shown.

#### Sidebar 6-1;

The general information about the systems parameters, and its network topology.



The rounding techniques are creating small differences in the input and synaptic weights, such that the output of the network shows a slide difference in behavior. The changes in behavior are so small, that a graphical representation as used before is impractical. Therefore another way of presenting the results is chosen. The results are presented as scatter plots of the errors between the finite and infinite system. These plots are accompanied by the mean square error and the root means square error. The goodness of fit can be measured by the dispersal of the errors. But another observation can be made. If the errors are showing a random and independent structure, the errors are noise. A good fit is characterized by a random and independent error (noise). If some sort of structure can be found (as a sinusoid, parabolic, or linear relationship exists between the error values), the fit not optimal.

Figure 6-1 shows three scatter plots where the jamming method is used for the input vectors, while the synaptic weights for each experiment uses another rounding method. The dispersal of the error signal in the left upper plot and the plot at the bottom are larger than in the upper right plot, where the synaptic weights are rounded and the input is jammed. For the lower plot, which generally holds negative errors, no structure in the data can be found. That means: the errors are randomly distributed, but is negative biased and confirms that it is no perfect match to the fitted system. The upper left figure shows some sort of sinusiod structure in the error signal; the goodness of fit is therefore smaller than by the other plots. The best fit is given in the right upper figure, while the distribution of the errors is small and is randomly structured. That means that the best fit is given by the following combination of rounding techniques: the input is jammed and the synaptic weights are rounded to the closed quantization level.

**Figure 6-1;**

The three figures shows the influence on the mapping. By applying for the input a truncating method and for the weights all three rounding techniques.

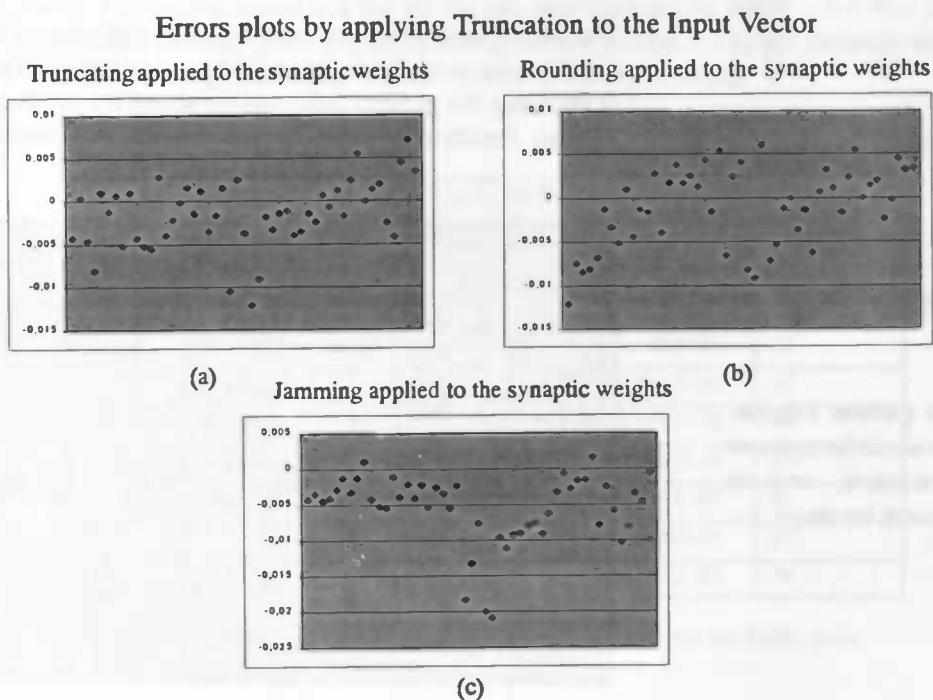
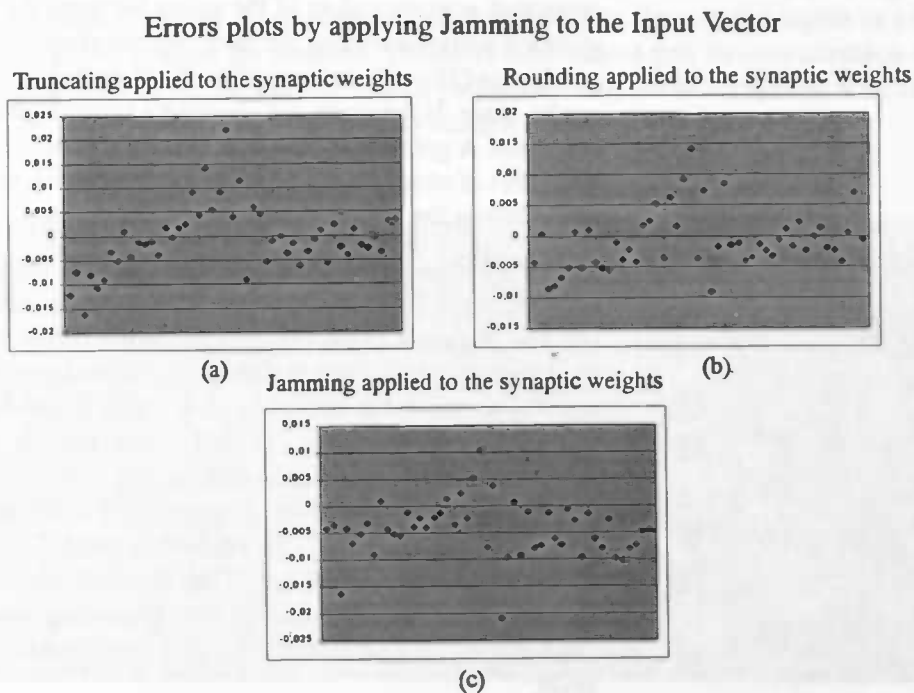


Figure 6-2 shows the following series of experiments. Here the rounding techniques is used: for all the experiments truncation is used for the input vectors, and a different rounding method is chosen for the synaptic weights in each experiment.

**Figure 6-2;**

The three figures shows the influence on the mapping. By applying for the input a jamming method and for the weights all three rounding techniques.



In the scatter plot at the bottom it can be seen that mostly all the errors are negative. What means that the results of the digital system are larger than the system which must be fitted, a worse fit. For a comparison between the upper left (a) and right (b) figures

a decision about the goodness of fit can be made on the principle that the distribution of the right figure is smaller than the left one. Thus, by using rounding for the weights combined with a truncating technique for the input vectors gives the best fit.

By a comparison between the best fit in this series and the best of the results mentioned above, the following can be concluded. The maximum error is smaller in this series of experiments and so is the distribution of the error. That means that the best fit is obtained by the combination rounding and truncating, where truncating is used for the input vectors and rounding for the synaptic weights.

### Errors plots by applying Rounding to the Input Vector

**Figure 6-3;**

The three figures shows the influence on the mapping. By applying for the input a rounding method and for the weights all three rounding techniques.

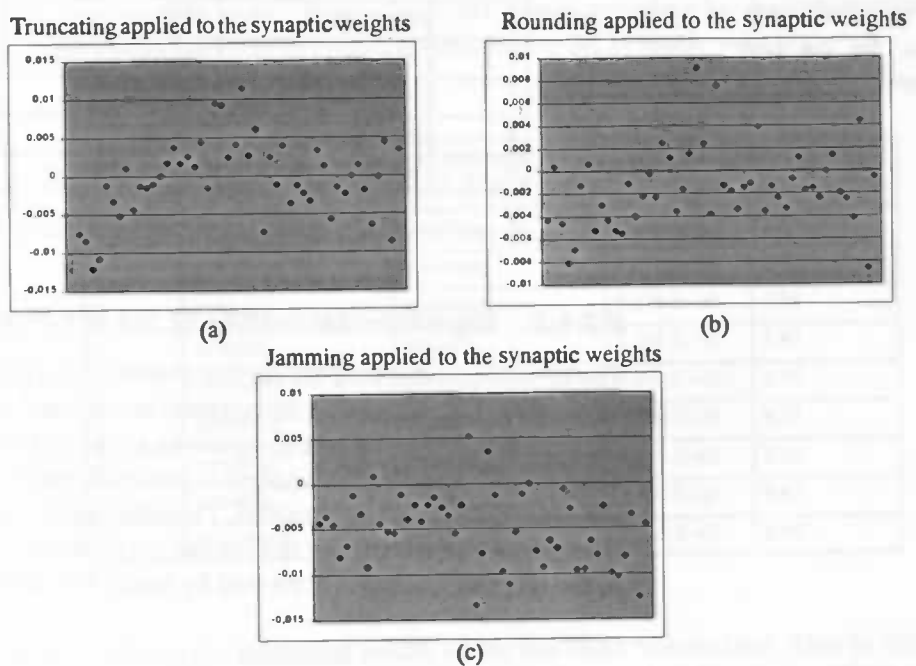


Figure 6-3 shows three scatter plots, where each experiment uses the rounding technique for the systems input, while the rounding technique for the synaptic weights are different. An observation which can be directly made is that the variance of the error signals is smaller in the right (b) figure. The error signals in figure 6-3c, are mostly negative, which means that the hardware system produces larger responses than the system which is fitted. This behavior of the error signals is not recommendable for a good fit. The upper left (a) figure shows some sort of structure in the error signal, namely a sinusoid. The error signal in figure 6-3b and c are showing no structure at all, an indication of a good fit. The conclusion is that the right figure (b) gives the best fit, by using rounding for the synaptic weights and the input vectors.

By comparing all the results obtained by these experiments the following gradation of rounding techniques for the sine function can be made, see table 6-1. From the table it can be concluded that for the sinewave function the fit is worse by using the jamming technique for the synaptic weights. The best fit can be obtained by combinations of rounding and the truncating techniques.

About the influence of rounding techniques on the performance to the real-world, can be short. The performance lies for all the use combination by  $\text{RMS} - 3.4 \% \pm .2 \%$ . Which means that the influence of rounding techniques have a larger dependency on the fit than on the performance of the system to the real-world.

**Table 6-1;**

The change in the goodness of fit, for various rounding techniques, for the sine function.

Best fit	technique for the synaptic weights	technique for the input vectors	MSE	RMS error (in %)
1	rounding	rounding	1.387 E-05	.37
2	rounding	truncating	1.695 E-05	.41
3	truncating	truncating	2.140 E-05	.43
4	rounding	jamming	2.286 E-05	.47
5	truncating	rounding	2.707 E-05	.49
6	jamming	rounding	3.985 E-05	.62
7	truncating	jamming	4.511 E-05	.65
8	jamming	jamming	4.573 E-05	.67
9	jamming	truncating	5.456 E-05	.74

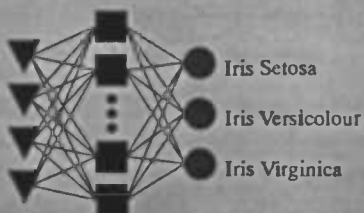
NOTE : The exact gradation is based on the calculated MSE and the RMS-error.

#### 6.2.1.2 Experimental results of the Iris classifier

The experimental results for the Iris classifier, for each combination of the rounding techniques, are included in this section. In sidebar 6-2, the general information about the performed experiments is mentioned. The Iris classifier contains four input and three output neurons. Each output reacts differently on the way the inputs as well as the synaptic weights are rounded. Therefore each output gets its own attention in the examination of the fit. For the overall performance of the fitted system with regard to the real-world data it is viewed by using a rejection criterion.

**Sidebar 6-2;**

The general information about the experiments performed by using various rounding techniques. The rounding techniques are performed on the synaptic weights and the input.

General Information			
Network Parameters		Network Topology (4:8:3)	
	#bits used		
Input/output	8		sepal length
External activation	8		sepal width
Internal activation	10		petal length
Adder Width	16		petal width
	Numbers		
#patterns	118		
Sigmoid Slope	1.0		
Area of Effectiveness	[-8.63, 8.63]		

The order of showing the results obtained for the Iris simulations, are: (1) the fit of the Iris Setosa class, (2) the fit of the Iris Versicolour, (3) the fit of the Iris Virginica

class, and (4) the performance of the discrete and continue values systems with regard to real-world data.

The change in fit when various rounding techniques are applied to the synaptic weights and the input vectors, is presented in table 6-2 for the Iris class Setosa. The errors are obtained between the fitted system and the system which is fitted. The variance of the error signals are calculated and used to create a gradation for the fit. The RMS error shows for all combinations of the applied rounding techniques the approximately same fit ( $\sim 1\%$  between the best fit and the worse fit). The best fit is obtained by using the rounding technique for the synaptic weights and a jamming technique for the input vectors. An observation which can be made is that within the four best fits only the different techniques are applied to the synaptic weights, namely rounding and truncating. While the techniques applied for the inputs get the best results by jamming and truncating. The worse fits are obtained by using rounding for the input vector and by using any of the three rounding techniques to the synaptic weights.

**Table 6-2;**

The change in the goodness of fit, for various rounding techniques, for the Setosa Iris class.

Best fit	technique for the synaptic weights	technique for the input vectors	MSE	RMS error (in %)
1	rounding	jamming	6.107 E-03	7.82
2	rounding	truncating	6.114 E-03	7.82
3	jamming	truncating	6.179 E-03	7.86
4	jamming	jamming	6.194 E-03	7.87
5	truncating	jamming	6.694 E-03	8.18
6	truncating	truncating	6.709 E-03	8.19
7	rounding	rounding	7.415 E-03	8.66
8	jamming	rounding	7.507 E-03	8.67
9	truncating	rounding	8.010 E-03	8.95

Table 6-3 shows the statistical results of the Iris class Versicolour. Also in this case the goodness of fit between both systems is calculated. An observation which can be directly made is that the RMS and MSE values by applying rounding to the input is so large that almost 6 % is lost between both systems. This is not recommendable.

**Table 6-3;**

The change in the goodness of fit, for various rounding techniques, for the Versicolour class.

Best fit	technique for the synaptic weights	technique for the input vectors	MSE	RMS error (in %)
1	rounding	jamming	6.703 E-03	8.18
2	rounding	truncating	6.715 E-03	8.19
3	jamming	truncating	6.891 E-03	8.30
4	jamming	jamming	6.892 E-03	8.30
5	truncating	jamming	7.092 E-03	8.42
6	truncating	truncating	7.127 E-02	8.44
7	rounding	rounding	21.279 E-03	14.59
8	truncating	rounding	21.635 E-03	14.71
9	jamming	rounding	21.694 E-03	14.72

By comparing this table (table 6-3) with table 6-2 it can be seen that the order in goodness of fit is practically the same. The best fit by Iris Versicolour is therefore also by applying rounding for the synaptic weights and by applying jamming for the inputs.

The results of the last class, the Iris Virginica, are summed up in table 6-4. The RMS error and the MSE are in this case the largest. Which means that the fit will be not as good as the Iris classes Setosa and Versicolour. The largest difference in the calculated errors is approximately 5%. In this case the worse fit is also obtained by applying rounding for the input vectors. While for jamming and truncating techniques, the best fit is obtained by applying them to the synaptic weights and to the input vectors. The order in which the best fit is obtain, is different by comparing these results to the results above table 6-2 and 6-3.

**Table 6-4;**

The change in the goodness of fit, for various rounding techniques, for the Versicolour class.

Best fit	technique for the synaptic weights	technique for the input vectors	MSE	RMS error (in %)
1	jamming	truncating	2.152 E-02	14.67
2	jamming	jamming	2.162 E-02	14.70
3	truncating	truncating	2.263 E-02	15.04
4	truncating	jamming	2.275 E-02	15.08
5	rounding	truncating	2.289 E-02	15.13
6	rounding	jamming	2.299 E-02	15.16
7	jamming	rounding	3.685 E-02	19.20
8	truncating	rounding	3.791 E-02	19.47
9	rounding	rounding	3.811 E-02	19.52

Before the performance results are showed of both systems, a summary of the results mentioned above about the goodness of fit, will be presented. By comparing the results of the Iris Setosa class with the Iris Versicolour class it can be concluded that applying combinations of rounding and jamming for the synaptic weights and for the inputs results in one of the best fits. But the gradation of the Iris Virginica class compared with the other Iris classes shows a best fit for other applied rounding techniques. The gradation of the best fitted class is: (1) Iris Setosa, (2) Iris Versicolour, and the worse fit for the (3) Iris class Virginica. For all three classes holds that the worst fit is obtained by applying for the input vector the rounding operator, together with any rounding technique for the synaptic weights. Then it can be concluded that a worst fit is independent of the applied rounding technique for the synaptic weights.

The following table 6-5 shows the classifying performance of both systems, whereby a rejection criteria is used to reject unreliable classifications. The used technique is that the distance between the winner and its runner-up must be at least 0.5. Table 6-5 show a orange bar, which holds the best performance made, which is achieved with applying rounding for the synaptic weights and truncating for the input vectors. The difference between the best (94) classification and the worst (87) is 7 patterns, it is a percentual difference of 5,9 %. The best fit classifies 94 good ( and 1 wrongly classified pattern) which means a classified percentage of 78%. It is not strange that the worst fit is performed by using truncation for the synaptic weights while the input is rounded, because the results above shows that this couple is in most cases the worst.

The overall performance of the classifier shows that for applying the rounding operator to the synaptic weights the best performance, is achieved independent of the applied technique for the input vectors.

**Table 6-5;**

The performances by using a rejection criterion by various combination of applied rounding techniques. The difference between winner and runner-up must be at least .5 for a classified pattern.

rounding technique weights	rounding technique for input	InterAct reference system			Digital fitted system		
		classified	rejected	wrong classified	classified	rejected	wrong classified
truncating	rounding	107	11	1	87	31	1
truncating	truncating	107	11	1	90	28	1
truncating	jamming	107	11	1	90	28	1
rounding	rounding	107	11	1	91	27	1
rounding	truncating	107	11	1	94	24	1
rounding	jamming	107	11	1	93	25	1
jamming	rounding	107	11	1	89	29	1
jamming	truncating	107	11	1	90	28	1
jamming	jamming	107	11	1	90	28	1

NOTE : the Orange bar presents the best performance by applying rounding techniques

## 6.2.2 Discussion

The experiments performed to observe the influence of rounding effects show some relations. The influence of each rounding technique (rounding, truncating and jamming applied on the synaptic weights or input vectors, or even a binary value) is maximal one quantization level. For the input vectors with a range of  $[0, 1]$ , the influence is at maximum  $1/2^8$  by using an 8 bits representation. The range of the synaptic weights is larger than the ones used by the input vectors, which means that by using the same 8 bits of representation the quantization levels are larger. This results in a more vulnerable solution for the coding of the synaptic weights. This vulnerability to the different rounding techniques is observed in the experiments above. Thus the influence on the neural system by applying rounding techniques is mainly depending on the rounding technique applied to the synaptic weights. The influence on the technique used for the input vectors does not carry a large influence on the performance. The best performing rounding technique for these experiments is by applying rounding to the synaptic weights. The choice of a rounding technique for the input vectors is not so important, while the influence is very small. The influence on the input gets important when the number of bits used for a representation is decreasing. But the most important and largest influence on the fit, and on the performance, will always be the weights.

From another point of view, recalling the sine experiment, the influences on the goodness of fit are so small that they can be eliminated. But for the Iris classifier the number of correctly classified patterns are increasing or decreasing by a good or wrong choice of rounding techniques. A practical recommendation is, that it is useful to check the influences of rounding techniques, especially for the synaptic weights. For a realization of a network in hardware, it holds that in some cases the mapping and the performance will increase by a proper choice of rounding techniques.

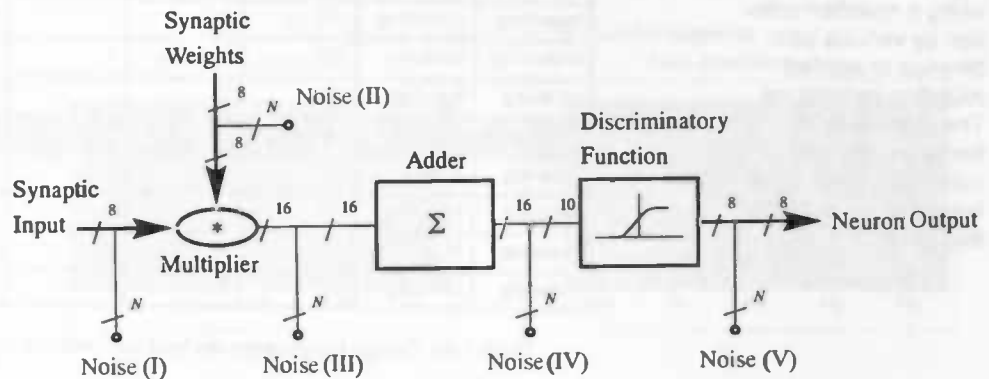
### 6.3 Addition of Noise

For further optimization of the neural hardware system a closer look at the components is recommended. This section investigates the influence of adding noise/disturbances to the signals of the system. Figure 6-4 shows at which points in the system an addition of the noise disturbances can be placed.

The Points where Noise can be Added

Figure 6-4;

A neuron is shown with one synaptic input, and the point where noise can be added. Each line shows the connection between a component and its predecessor. Where each line shows its digital word width by a number.



In figure 6-4 five points are shown, where noise can be added to the signal. But not each adding point is recommended for an addition of noise. Lets consider noise point I, the noise will be added to a synaptic input signal which holds for a input neuron that another input vector is obtained. This means when an input is offered to the neural system; instead of the real-world signal another one is being calculated. This addition point (I) of a noise signal is not recommended, because the environment is responsible for the correctness of the input signals. Another addition point which is equal to addition point I is point V, because for a hidden layer the output of a neuron leads to the input of the following layer. At addition point II the addition of noise to the synaptic weights (or the bias of a neuron, which is not drawn) will bring us back to the moment before a network is trained when the synaptic weights are randomly installed (some sort of disposing the adapted knowledge). A small amount of noise addition at point II can be seen as a randomly selected rounding technique, which could offer a positive result (see section 6.2). The points of interest which remain to add a noise signal are points III and IV. The following experiments will offer a pseudo random signal to the noise points III and IV to investigate the influences. In order to suggest a smaller representations of components, or the capability to react in real-time.

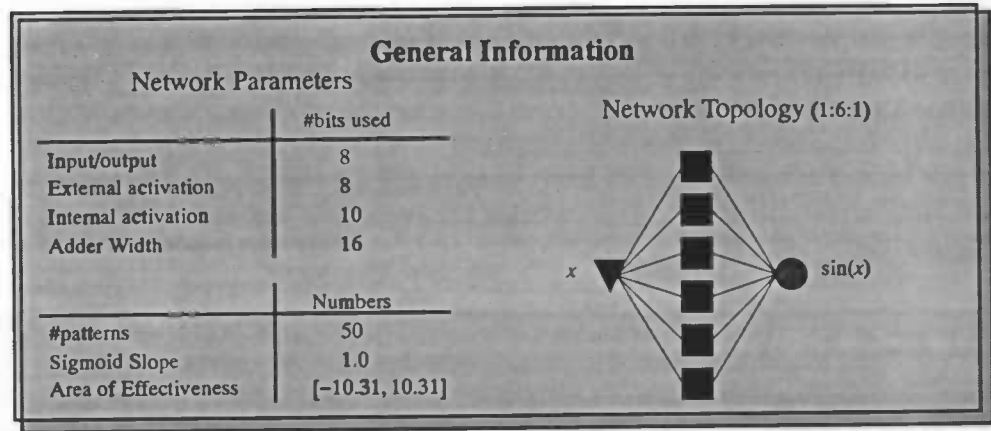
#### 6.3.1 The Experiments

The experiment which is used for this research is the sinewave function. For the generation of pseudo random noise a VHDL-decription is obtained from [16], which generates an 8 bit number where only the lowest-order bit are used. This sequence of numbers is repeated after a certain time period. But the random property is preserved, while the multiplication time depends on the amount of one's in a term (which is independent of the noise being generated). The addition of the noise signal is performed by chopping the  $N$  lowest-order bits of the signal, where the  $N$  lowest order bits are conducted for the  $N$  lowest-order bit of the generated pseudo random number. The

first experiments will show noise addition to point IV; after that it will be repeated for point III. For both experiments the general information is shown in sidebar 6-3.

### Sidebar 6-3;

The general information about the systems parameters, and its network topology.



#### 6.3.1.1 Results by Addition of Noise at point IV

This experiment performs the addition of white noise to the result of the adder (point IV). The used word width on this point is 16 bits while the internal activation level is discriminated by 10 bit, the lower-order bits are truncated. The addition of noise is performed on the 10 remaining bits. By entering noise to a signal means that for each pattern another answer is possible, thus each pattern must be offered several times to the system to see the influence of the random bits.

#### Input and Output behavior by Noise Addition

Figure 6-5;

The input output behavior of the sine-wave function by several amounts of noise added.

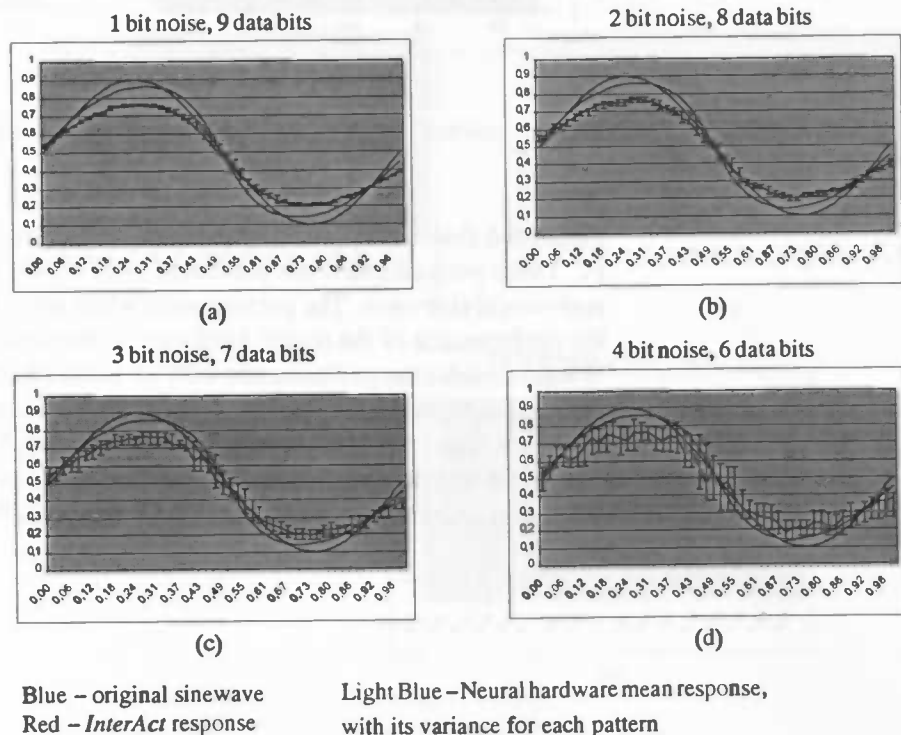


Figure 6-5 shows the input/output behavior of the neural hardware system where noise is applied to point IV. An observation which can be made directly is the change of the amplitude of the network response with regard to the response of the neural system without noise. The variance of each pattern becomes larger when the amount of noise becomes larger. Another observation of the variance is that it is the largest where the sinewave crosses the line at .5. Even when the 4 bit noise is added the sinewave is recognizable, while the original signal (10 bit) is reduced to 6 data bit with 4 noise bits.

**Figure 6-6;**

The influence of noise in relation to the performance. Figure (a) and (b) are showing the performance of the hardware system in relation to the InterAct response. While figure (c) and (d) shows it with regard to the real-world sinewave.

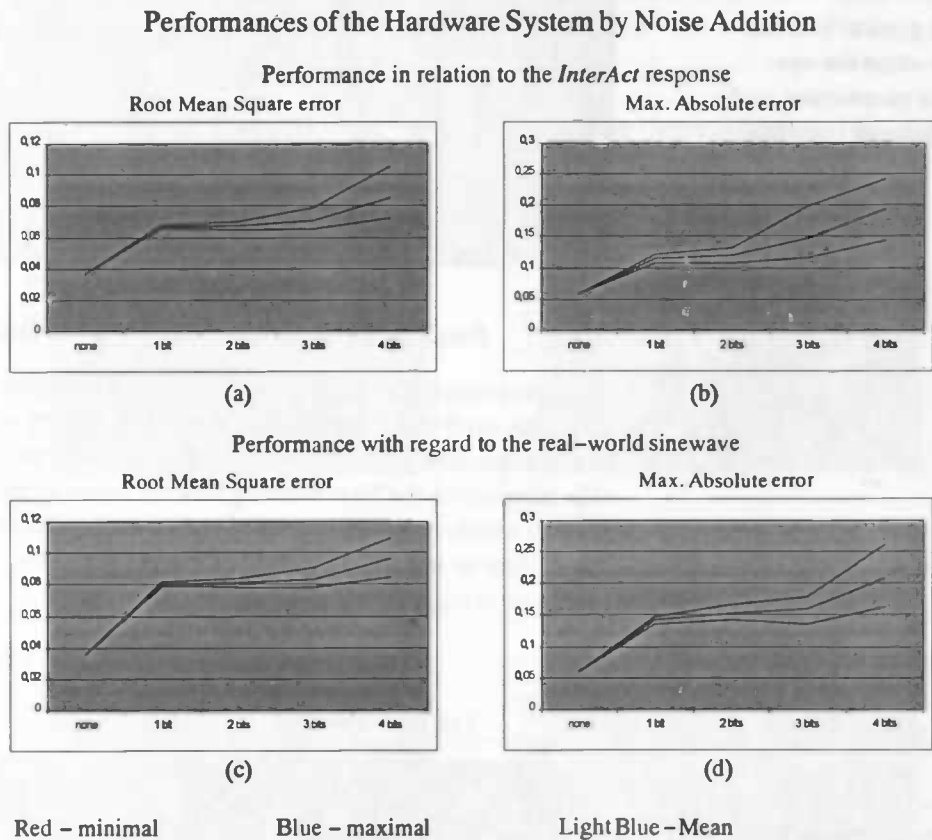


Figure 6-6 shows the performances of the hardware system by applying noise to point IV. These performances are presented with regard to the InterAct response and the real-world sinewave. The performance when noise is added to the signal at point IV, the performance of the neural hardware is decreasing dramatically, the performance at least doubles the performance with no noise addition. In all cases the performance decreases (exponentially) when 3 and 4 bits noise are added to the signal. Another observation is that the variance of the RMS and Max.Abs errors is becoming larger. The following section presents the results when noise is added to the signals on point III. After that presentation a discussion follows about the results.

### 6.3.1.2 Results by Addition of Noise at point III

The addition of noise at this point can be performed in two different ways. The first is to replace the  $N$  lowest order bits by a random value of the 16 bits multiplier result. The second way is to remove the 6 lowest-order bits, and to replace the signal bit from that point by random bits. One experiment with the first way of applying noise, is performed; the results are shown in table 6-6. These results are worse in relation to the performance obtained in the second way. The experiment results, by adding 4 random bits to the lowest-order bits of the 16 bits multiplier result, is almost equal to the performance of 8 data bits with 2 bits of noise (total 10 bits). Thus, the second way of noise addition uses less bits to perform better.

**Table 6-6;**

Performance by adding 4 bits noise to a 16 bits number at point III.

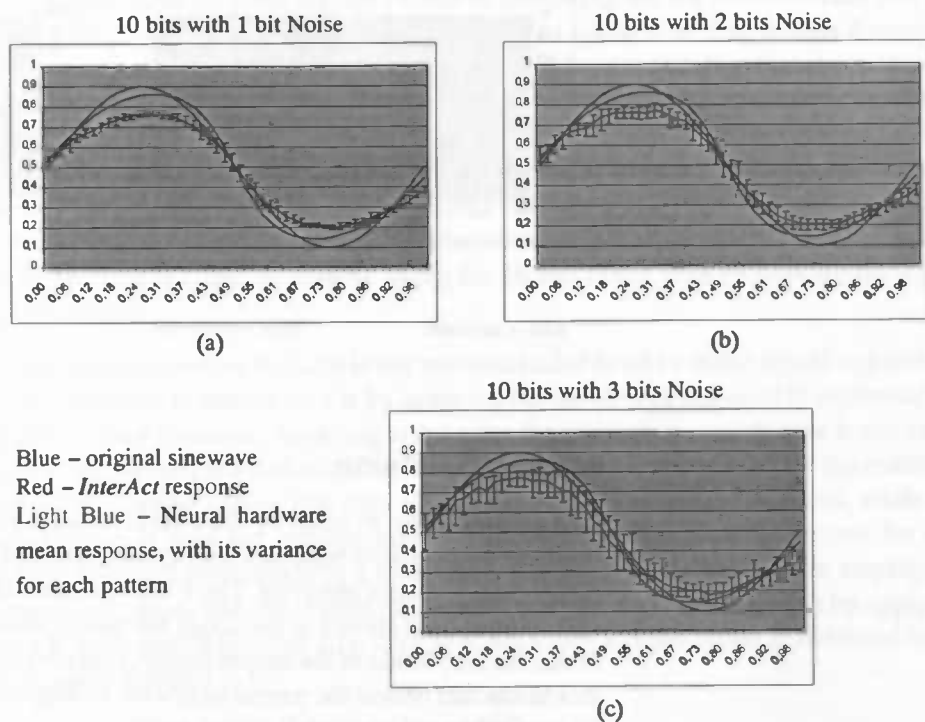
Hardware in relation to	MSE			RMS error			Max. Abs error		
	Min.	Mean	Max.	Min.	Mean	Max.	Min.	Mean	Max.
InterAct	4.3 E-3	4.4 E-3	4.6 E-3	65.5 E-3	66.6 E-3	67.7 E-3	0.108	0.115	0.119
Real-world data	6.3 E-3	6.4 E-3	6.6 E-3	79.3 E-3	80.1 E-3	81.1 E-3	0.135	0.138	0.142

Figure 6-7 shows the results by applying noise to point III, in such a way that the influence of the lowest-order bits are eliminated. The remaining 10 bits represent the information for the adder. These 10 bits of information will be used to apply the noise signal, for respectively 1, 2 and 3 bits of noise.

### Input - Output Behavior by applying Noise

**Figure 6-7;**

The input output behavior of the sine-wave function by several amounts of noise added.

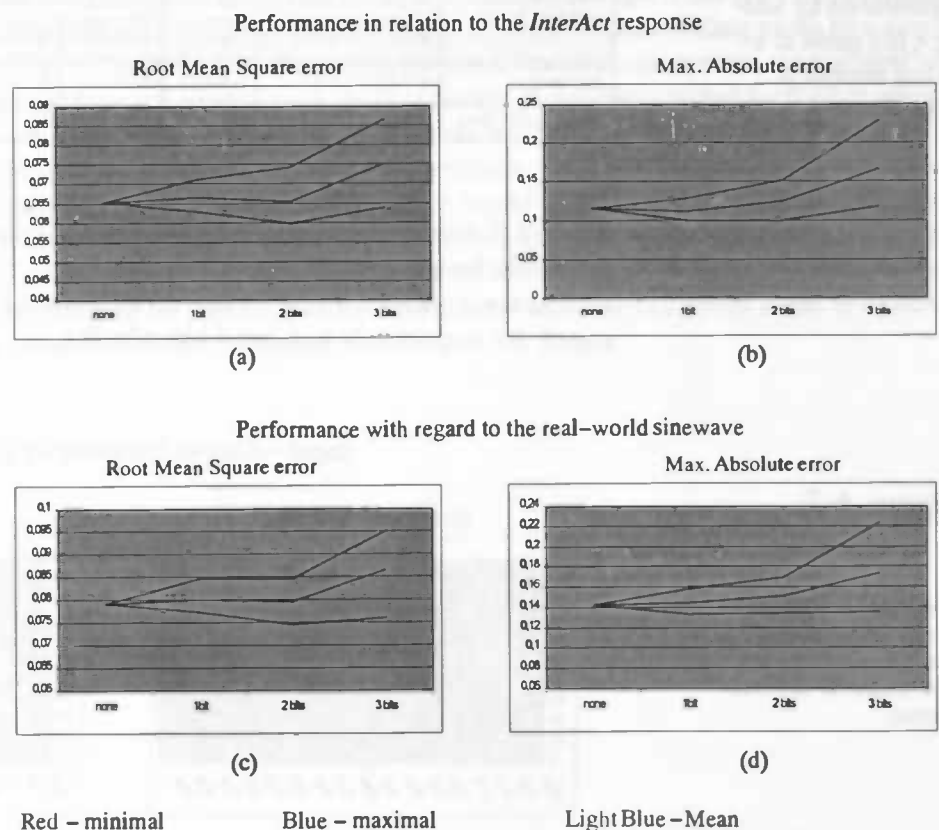


Also figure 6-7 shows the same results as when the noise is applied on point IV: loss in amplitude, variance increase by increasing the noise. The following figure 6-8 shows the performance in relation to the response of the InterAct simulation and of the real-world sinewave. From this figure the conclusion follows that applying 1 or 2 bits of noise has no influence on the mean performance of the hardware system, while the variance is increasing. By applying more than 3 bits of noise, the neural hardware systems performance is showing a dramatical decrease of the performance and its variance is out of proportions. The fit and the performance of the sinewave function are in some cases (especially by applying 2 bit noise) better than when applying no noise signal (figure 6-7a and c).

### Performances of the Hardware System by Noise Addition

**Figure 6-8;**

The influence of noise in relation to the performance. Figure (a) and (b) are showing the performance of the hardware system in relation to the *InterAct* response. While figure (c) and (d) shows it with regard to the real-world sine-wave.



### 6.3.2 Discussion

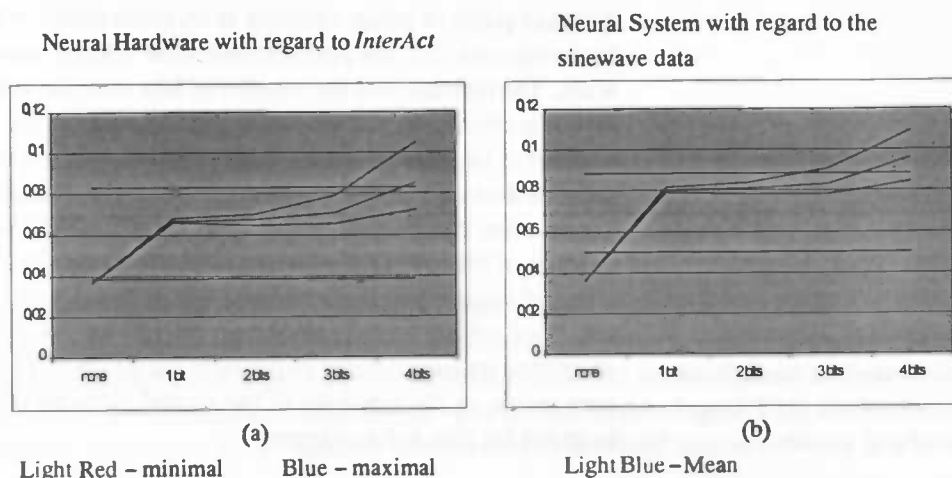
By applying noise to point IV the overall result is that the performance as well as the fit of a neural network becomes worse; this is concluded from the variance of the errors of the input – output behavior. The 6 bits that are truncated from the 16-bits adder result, is an optimal choice to minimise the use of silicon area. Another point of concern is that the amplitude of the output signal is smaller when noise is applied to the signals; this holds that beside the neural hardware an amplifier, or even a low-pass filter must be applied to get an usefull signal again.

In chapter 5, experiments are performed on the number of bits used to represent the discriminatory function. By combining them with the results obtained by applying noise, figure 6-9 can be obtained. Applying only 1 random bit to a 10 bits word performs a little better than a discriminatory function on 6 bits. From this it can be concluded that by applying noise to point IV is not recommended, while better performances and fits can be obtained by using smaller amounts of bits to represent a discriminatory function (so that a smaller area can be used).

#### Relation between applying Noise and reducing of the Discriminatory function

**Figure 6-9;**

The relation between applying noise at point IV and the reduction of the discrimination function. The two vertical line shows the performance a discriminatory function represented by 6 bit (green) and 7 bits (red).



By applying noise to point III, the choice of removing the six lowest-order bits has improved the performance by a smaller number of signal bits. This makes it possible to stop the DigiLog multiplier earlier which means for a realization that the multiplier as well as the recursive adder can be smaller in wordwidth. Another important thing is that the speed of the multiplier will be increased. By applying two bits of random generated numbers to the result of the multiplier it can be seen that it is possible to get a better performance than by applying none random signal. But that performance is no better than applying no random but also no reduction of the 6 bits. In other words, the RMS error is closer to zero by using the 16 bits result than by passing through a 10 bits signal.

The overall conclusion is that it is not recommended to add a noise signal to point IV, a better solution to reduce area is by lowering the number of bits used to represent the discriminatory function. Applying noise after the multiplier module is in some cases interesting, namely if a real-time response is needed. This holds also that the multiplier must be stopped earlier. But only then is it the question what is required, while applying noise is than not recommended because it holds various solutions for one pattern. For each error made in a neuron, the topology of the network is amplifying it each layer again, and the system holds error on error. This is happening by applying noise to both points (IV and III); only in some cases a better result is obtained but it is not under control.

## 6.4 Conclusions and Recommendations

The applied reduction technique for the input vectors and the synaptic weights are performing the best, when rounding is applied to the synaptic weights. The rounding technique applied to the inputs vectors is not of concern, because the influence is small. A practical recommendation is, that it could be useful to check the influences of rounding techniques, especially for the synaptic weights. For realization of a network in hardware, in some cases the mapping and the performance will increase by a proper choice of rounding techniques.

The addition of noise is not recommended after the adder, because a better solution can be obtained when another representation is chosen for the discriminatory function. Another point of noise addition is recommended when the systems requires a real-time response. But the performance after adding noise is not the best for a neural network. The influence of the number of bits after the multiplier has got a lot of influence on the performance of the entire system. Therefore when adding noise to a smaller number of bit than the used 16, the response of the hardware system can perform beter than for the same representation without noise. It is recommended for further research to see when the multiplier can be stopped, as well as to inspect the influence of the number of bits used the represent the products. On the other hand, applying noise to a classifying system is not researched yet, but it is recommended, as these applications are more robust than function approximating systems. Other research can also give an answer on a minimally required word width of a neuron, without changing the neurons result, as the topology of the neural network amplifies errors made in a neuron, in order for this not to happen.

---

# Chapter 7

## *Conclusions and recommendations*

---

Artificial neural networks are a promising new generation of information processing systems. They have proven to be good at tasks such as pattern matching, vector quantization, and data clustering, while traditional computers are inefficient at these tasks. Therefore, a dedicated neural hardware system must be developed. This work concentrates to investigate the possibility to represent an artificial neural network in a hardware architecture.

The design of the neural hardware system (chapter 3) described within a hardware description language (VHDL) is verified. From this verification it is known that the functionality of the hardware description compared to an artificial neural network is essentially the same. Another point is that the behavior of an artificial neural network is given by its characteristics. The translation and generation of those network characteristics gives the hardware system its final purpose, the behavior of a neural network. How to extract and generate the proper characteristics is researched in several experiments, hence to get a proper neural behavior and dimensioning the final architecture.

The research done to create a proper discriminatory function for the neural hardware system (chapter 5) shows us that a representation of the discriminatory function is dependent on the application of a neural network. For function approximation there is a optimum for the performance and the goodness of fit by a certain area of effectiveness. For classifying applications, the performance is increasing for large areas of effectiveness, but the goodness of fit becomes worse. Another conclusion can be made for classifying systems. By increasing the area of effectiveness the distance between the winner and the runner-up will also increase, so that the outputs of a network are better separable.

Another experiment shows, that the number of bits used to present the discriminatory function is directly related to the performance as well as the fit of a neural network. By increasing the number of bits the performance and the goodness of fit will increase. But the performance and the goodness of fit are asymptotically limited. From the combination of both experiments it can be concluded that the performance to the real world data will be improved by classifying systems by a wide area of effectiveness. But it

can not be improved by using more or less bits to represent the discriminatory function. For the function approximation the performance is bounded to an optimum for the area of effectiveness, but also the number of bits used to represent the discriminatory function is bounded by an optimum.

From the investigation of the influence of rounding techniques applied to the input as well as to the synaptic weights of the network the following can be concluded (chapter 6). The best performance is achieved by applying the rounding method for the synaptic weights. While the applied rounding technique for the input vector is mostly independent of the performance. By applying jamming to the synaptic weights by any technique for the input vectors, the performance becomes worse. The technique used to round the synaptic weights creates the largest influence on the performance and the goodness of fit.

The addition of random noise to the system after the adder in order to reduce the word width use by the system is not recommendable it is better to reduce the number of bits to represent a discriminatory function. Only then a better performances are obtained. The influence of adding noise after the multiplier, shows that in some cases a better performance is obtained, than by the same number of bits without noise. But the probability that is result is better is rather large. By an optimized hardware system, addition of noise is certainly not recommended.

The conclusion to the main objective is that representations of artificial neural networks, especially feedforward multi layered Perceptron networks in recall mode, is a fact. This is concluded from all performed experiments which show that the description is very flexible and very stable within simulations.

Further research on this topic may concentrate on extending this neural hardware description to a hardware device. The neural hardware system's realization can be optimized for several properties which will be demanded by real-world applications, such as: high speed, high accuracy, small area, real-time response, embeddable within others and, on-chip learnable, ect.

Other research in the area is the representation of the discriminatory function and the relations between the number of bits used to represent the function, and the area of effectiveness. Furthermore, research on the DigiLog multiplier shows that stopping the multiplication earlier can increase speed. Another issue is to stop early and add random noise to the product in order to save time, and maybe produce a more accurate answer.

A more practical recommendation for doing simulation within the simulation environment *V-System*, is that heavy computer equipment is recommended. Simulating a network of 10 neurons and 118 input vectors takes (on a 133 MHz machine, *Pentium*) approximately 20 min. and is increasing exponential (Valve experiment  $\rightarrow$  21 neurons, 1000 pattern approx. 2.5h). For the simulations of large system such as the system described in this work (for large look-up tables,  $<2^{14}$ ), another version (5.1) is needed else it will simply crash.

---

# *Acknowledgement*

---

This thesis was not yet finished without the support of many special people. Therefore I would like to thank everyone who has contributed to this work in one way or another. Specially I would like to thank my supervisors Prof. Dr. Ir. L. Spaanenburg and Dr. Ir. J. A. G. Nijhuis, for their comments, suggestions and support. Further, I would like to thank E. D. Gillissen for her help on mathematical statistics, and to all my friends and fellow students for our conversations which has inspired and motivated me to finish this work. And last but certainly not least I would like to thank my parents for all their support.

---

# References

---

- [1] M. Diepenhorst, W. J. Jansen, J. A. G. Nijhuis, L. Spaanenburg, and A. Ypma, "Using GREMLIN for Digital FIR Networks" *Proceedings MircoNeuro'96*, pp. 341–346, 1996.
- [2] S. Kaykin, "Neural Networks – A Comprehensive Foundation", *Macmillan College Pub. Comp. Inc.*, 1995.
- [3] D. B. Fogel, "Evolutionary Computation – Toward a New Philosophy of Machine Intelligence", *IEEE press*, 1995.
- [4] J. M. Zurada, "Introduction to Artificial Neural Systems", *West Publishing Company*, 1992.
- [5] D. D. Olmsted, "History and Principle of Neural Networks", at <http://neurocomputing.org/history.html>.
- [6] M. Hoehfeld, S. E. Fahlman, "Learning with Limited Numerical Precision Using the Cascade-Correlation Algorithm", tech. Report CMU-CS-91-130, Siemens AG, Munich, 1991.
- [7] S. I. Gallant, "Perceptron-based Learning algorithms", *IEEE Trans. Neural Networks*, No. 1, Vol. 2, pp. 179 – 191, 1990.
- [8] D. Chen, C. L. Giles, Z. Sun, et al. , "Constructive Learning of recurrent Neural Networks", *Proc. IEEE Int. Conf. Neural Networks*, Vol. 3, pp. 1196–1201, San Francisco, 1993.
- [9] G. F. Meijering, "Parallelism versus Accuracy in Error Back-Propagation Learning", Thesis M. Sc, 1996.
- [10] B. Spaanenburg, "Softwarearchitecten zijn onmisbaar", *Automatisering Gids*, No 40., pp 21, oct-2-1998.

- [11] D. Garlan, and M. Shaw, "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering*, Vol 1, World Scientific Pub. Comp., 1993.
- [12] F. J. Hill, and G. R. Peterson, "Digital Systems – Hardware Organization and Design", *John Wiley & Sons*, third edition, 1987.
- [13] R. vanDrunen, L. Spaanenburg, P. Lucassen, J.A.G. Nijhuis, and J.T. Udding, "Arithmetic for Relative Accuracy", *proc.IEEE symp. on computer Arithmetic*. Bath(UK), july 1995, pp. 239–250.
- [14] M. Diepenhorst, J.A.G. Nijhuis, and L. Spaanenburg, "A Time-multiplexed Multiplying Adder for Neural Signal Processing", *proceedings of the IEEE Benelux Signal Processing Symposium SPS 98*, pp. 83–86.
- [15] H. M. G. Ter Haseborg, "Een vermenigvuldiger", Scriptie voor technologie mapping en VLSI ontwerpen, 1998.
- [16] B. Molenkamp, "VHDL, VHDL '87/'93 en voorbeelden", *universiteit Twente – informatica*, Enschede, 1995.
- [17] K. Skahill, "VHDL for Programmable Logic", *Addison–Wesley Pub. Comp. Inc.*, 1996.
- [18] B. W. Lindgren, "Statistical Theory", *Chapman & Hall Inc.*, 4th ed., New York, 1993.
- [19] M. B. Priestley, "Spectral Analysis and Time Series – Probability and mathematical statistics", *Academic Press Limited*, London, 1981.
- [20] P. J. Brockwell, R. A. Davis, "Time Series: Theory and Methods", *Springer Series in Statistic*, 1991.
- [21] A. F. Murray, "Silicon Implementations of neural networks", *IEE Proceedings–F*, Vol 138, No. 1, Feb., 1991, pp. 3 – 12.
- [22] R. A. Fisher, "The Use of multiple measurements in taxonomic problems", *Annual Eugenics*, Vol. 7, Part II, pp. 179–188, 1936; also in "Contributions to Mathematical Statistics" (John Wiley, NY, 1950).
- [23] R. O. Duda, and P. E. Hart, "Pattern Classification and Scene Analysis", *John Wiley & Sons*, ISBN 0 – 471–22361–1. pp. 218, 1973.
- [24] S. W. Piché, "The Selection of Weight Accuracies for Madalines", *IEEE Transactions on Neural Networks*, Vol. 6, No. 2, 1995.
- [25] G. Dünder, and K. Rose, "The Effects of Quantization on Multilayer Neural Networks", *IEEE Transactions on Neural Networks*, Vol. 6, No. 6, 1995.
- [26] J. L. Holt, and J–N. Hwang, "Finite Precision Error Analysis of Neural Network Hardware Implementations", *IEEE Transactions on Computers*, Vol. 42, No. 3, 1993.
- [27] K. E. Atkinson, "An Introduction to Numerical Analysis", *John Wiley & Sons*, Second Edition, 1989.

# Appendix A

## *The arithmetic principles of Logarithms*

---



John Napier (1550 – 1617)

The idea of the logarithm could have had its source in the use of certain trigonometric formulas that transform multiplication into addition or subtraction. Recall that if one needs to solve a triangle using the law of sinus, a multiplication and a division are required. Astronomers realized that it would be simpler and reduce the number of errors if the multiplications and divisions can be replaced by additions and subtractions.

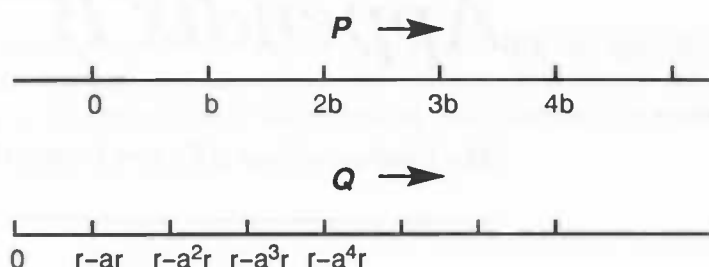
A second, more obvious, source of the idea of a logarithm can be probably found in the work of algebraists such as *Stifel* and *Chuquet*, who both displayed tables relating the powers of 2 to exponents and show that multiplication in one table corresponded to addition in the other. These tables have increasingly large gaps; they can not be used for necessary calculations. Around the turn of the seventeenth century, however, two men working independently, the Scot *John Napier* (1550–1617) and the Swiss *Jobst Bürgi* (1552–1632) come up with the idea of producing an extensive table that does allow one to multiply any number together (not just powers of 2) by performing additions.

### A.1 The idea of the logarithm

For the definition of the logarithm, *Napier* conceives two number lines. On one line an increasing arithmetic sequence,  $0, b, 2b, 3b, \dots$  is represented, and on the other a sequence whose distances from the right endpoint from a decreasing geometric sequence,  $ar, a^2r, a^3r, \dots$ , where  $r$  is the length of the second line, see figure A-1. The points on this line can be marked  $o, r-ar, r-a^2r, r-a^3r, \dots$ . For *Napier*, these points generally represent sines of certain angles.

Figure A-1;

Napier's moving points.



Napier now considers points P and Q moving to the right on each line as follows: P moves on the upper line "in an arithmetical sequence" (that is, with constant velocity). Thus P covers equal interval  $[0, b]$ ,  $[b, 2b]$ ,  $[2b, 3b]$ , ... in the same time. Q moves on the lower line, the geometrical sequence. Its velocity changes so that it too covers each (decreasing) interval  $[0, r-ar]$ ,  $[r-ar, r-a^2r]$ ,  $[r-a^2r, r-a^3r]$ , ... in the same time. The distances traveled in each interval form a decreasing geometric sequence  $r(1-ar)$ ,  $ar(1-a)$ ,  $a^2r(1-a)$ , ... each member of the sequence being the same multiple of the distance of the left endpoint of the interval to the right end of the line. Because distances covered in equal times have the same ratios as velocities, it follows that the point's velocity over each interval is proportional to the distance of the beginning of that interval from the right end of the line. It appears that *Napier* initially thought of the velocity of the lower point as changing abruptly when it passed each marked point, remaining constant in each of the given intervals. In his definition of logarithm, however, *Napier* smoothed out these changes by considering the second point's velocity as changing continuously. Thus a point moves geometrically if its velocity is always proportional to its distance from the right end of the line. In other words, if the upper point P begins moving from 0 with a constant velocity equal to that with which the lower point Q also begins moving (geometrically) from 0, and if P has reached y when Q has reached a point whose distance from the right endpoint (radius) is x, then y is said to be the logarithm of x.

Sidebar A-1;

#### Napier's, idea in a modern calculus notation

Napier's idea is reflected in the following differential equations

$$\frac{dx}{dt} = -x \quad x(0) = r \quad \frac{dy}{dt} = r \quad y(0) = 0$$

The solution to the first equation is:

$$\ln x = -t + \ln r \quad \text{or} \quad t = \ln \frac{r}{x}$$

Combining this with the solution  $y = rt$  of the second shows that Napier's logarithm y, may be expressed in terms of the modern natural logarithm as:

$$y = N \log x = r \ln \frac{r}{x}$$

---

# Appendix B

## The Hardware Neural System Description

---

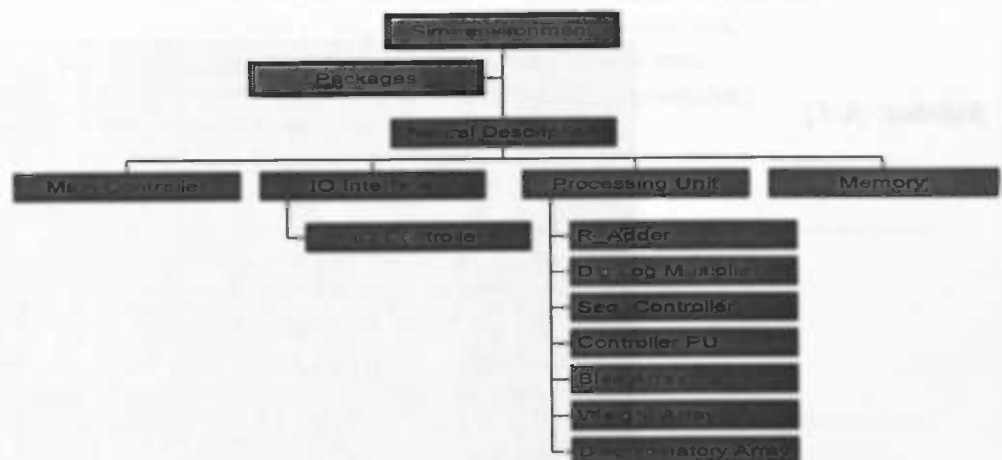
### B.1 Neural System description

The description of the neural hardware system is divided over several modules. The relations between the modules are shown in figure B-1. Now the whole system is viewed in this figure, the four main parts of the neural description, namely: main controller, the IO interface, the processing unit, and last the memory module, and there dependent module. The packages and the `sim_environment`, there VHDL descriptions can be found in appendix C. The following sections will show the description for each module.

Structure of the modules dependency

Figure B-1;

The structure of dependency of the modules described in the VHDL language.



### B.1.1 Interface module

The Interface module description coded by using the VHDL language, takes care of the communication between the neural hardware systems environment and the internal organization of the system it self. The following code describes the sturcture of the interface module.

```
-----
--
-- VERSION : 1.0.1
-- DATE : 11-04-1999
-- BY : H.M.G. Ter Haseborg
--
-- DESCRIPTION : The structure of the interface module
--
-----

library ieee;
use ieee.std_logic_1164.all;

entity IO_interface is
    generic ( d_buswidth : Integer := 8;
              a_buswidth : integer := 8);

    port (
        start          : In          bit;
        inputGo         : in          bit;
        outputGo        : In          bit;
        not_reset       : In          bit;           -- asynchrone reset
        clk             : In          bit;           -- clock signal
        ce              : In          bit;           -- chip enable
        ReleaseOut       : In          bit;           -- release output data
        data_portin      : In          Std_Logic_vector(d_buswidth-1 downto 0);
        data_portout     : out         Std_Logic_vector(d_buswidth-1 downto 0);
        data_busI        : In          bit_vector(d_buswidth-1 downto 0);
        data_busO        : out         bit_vector(d_buswidth-1 downto 0);
        strobe           : out         bit;
        ReadyIO          : out         bit;           -- interface ready signal
        Read             : out         bit;
        Write            : out         bit;           -- read/write controle line
        address_bus      : out         bit_vector(a_buswidth-1 downto 0)
    );

end IO_interface;

-----

use work.general.all;
use work.constants.all;

architecture Struct of IO_interface is
```

```

-----
--                                     Declaration of signals                                     --
-----

```

```

    signal valid_data : bit;

```

```

-----
--                                     Definition of components                                     --
-----

```

```

component IO_controller
    generic ( a_buswidth : integer := 8);

    port (
        start      : In      bit;
        InputGo    : In      bit;           -- start input sequence
        outputGo   : In      bit;           -- start output sequence
        not_reset  : In      bit;           -- a-synchrone reset
        clk        : In      bit;
        ReleaseOut  : in      bit;           -- release output data
        strobe     : out      bit;           -- timing signal chip
        ReadyIO    : out      bit;           -- timing signal chip
        Read       : out      bit;           -- read for memory
        Write      : out      bit;           -- write in memory
        address_bus : out      bit_vector(a_buswidth-1 downto 0)
    );

```

```

end component;

```

```

begin

```

```

u0 : IO_controller
    generic map (a_busWidth => a_busWidth)
    port map ( start => start, inputGo => inputGo, outputGo => outputGo,
        not_reset => not_reset, clk => clk, strobe => valid_data,
        ReadyIO => ReadyIO, Read => read, Write => Write,
        address_bus => address_bus, releaseout=> releaseOut );

```

```

-----
--                                     Combinational Logic                                     --
-----

```

```

    data_busO <= to_bitvector(data_portIn) when valid_data = '1'
        else ( others => '0');
    strobe    <= valid_data;

```

```

-----
--                                     three-state control of the output                                     --
-----

```

```

three_state: process (ce, data_bus)
begin

```

```

    if ce = '1' then
        data_portOut <= to_StdLogicVector(data_busI);
    else
        data_portOut <= (others => 'Z');
    end if;
end process;

end Struct;

```

### B.1.1.1 The Interface Controller

The controller of the interface module, is separated of the description of the interface module. The following description shows the logical behavior of the interface module, using a state-machine as controller mechanism.

```

-----
--
--
--  VERSION: 1.0.1
--  DATE   : 11-04-1999
--  BY     : H.M.G. Ter Haseborg
--
--  DESCRIPTION : The logical behavior of the interface controller.
--
-----

library ieee;
use ieee.std_logic_1164.all;

entity IO_controller is
    generic ( a_buswidth : integer := 8);

    port (
        start      : in  bit;
        inputGo     : in  bit;  -- start input sequence
        outputGo    : in  bit;  -- start output sequence
        not_reset   : in  bit;  -- a-synchrone reset
        clk         : in  bit;
        ReleaseOut  : in  bit;  -- release output data
        strobe      : out bit;
        ReadyIO     : out bit;
        Read        : out bit;
        Write       : out bit;
        address_bus : out bit_vector(a_buswidth-1 downto 0)
    );

end IO_controller;

-----

use work.general.all;
use work.constants.all;

```

architecture behaviour of IO\_controller is

-----  
-- *Declaration of signals* --  
-----

signal Address : bit\_vector(a\_buswidth-1 downto 0);

-----  
-- *state machine for data input/output* --  
-----

signal StateIO : bit\_vector(2 downto 0);  
constant Idle\_IO : bit\_vector(2 downto 0) := "000";  
constant Address\_IO : bit\_vector(2 downto 0) := "001";  
constant data\_IO : bit\_vector(2 downto 0) := "011";  
constant ready\_IO : bit\_vector(2 downto 0) := "100";

alias rdyIO : bit is StateIO(2);  
alias ReadWrite : bit is StateIO(1);  
alias AddressIO : bit is StateIO(0);

begin

-----  
-- *Combinational Logic* --  
-----

strobe <= AddressIO;  
address\_bus <= Address;  
readyIO <= rdyIO;  
  
Write <= '1' when inputGO = '1' and ReadWrite = '1' else '0';  
read <= '1' when outputGO = '1' and ReadWrite = '1' else '0';

-----  
-- *Load data state machine behavior* --  
-----

dataIO : process (not\_reset, clk, start, inputGO, outputGO)  
variable counter : integer;  
begin  
    If not\_reset = '0' then  
        StateIO <= Idle\_IO;  
    elsif clk = '1' and clk'event then  
        case StateIO is  
            when Idle\_IO =>  
                If start = '1' and inputGO = '1' then  
                    counter := 0;  
                    StateIO <= Address\_IO;  
                elsif outputGo = '1' and inputGO = '0' and ReleaseOut = '1' then  
                    counter := (network(0) + network(1));

```

        StateIO <= Address_IO;
    end if;
    when Address_IO =>
        StateIO <= data_IO;
    when data_IO =>
        counter := counter + 1;
        If inputGo = '1' then
            If counter /= network(0) then
                StateIO <= Address_IO;
            else
                StateIO <= ready_IO;
            end if;
        elsif outputGo = '1' then
            If counter /= (network(0) + network(1) + network(2)) then
                StateIO <= Address_IO;
            else
                StateIO <= ready_IO;
            end if;
        end if;
    when ready_IO =>
        if start = '0' then
            StateIO <= Idle_IO;
        end if;
    end case;
end if;
Address <= i2bvd(counter,a_busWidth);
end process;

end behaviour;

```

## B.1.2 Memory module

The systems memory, which stores and retrieves the results of earlier calculations of the network, is a RAM (Random Access Memory). The behavior of this module can be described as follows;

```

-----
--
--  VERSION: 1.0.1
--  DATE    : 11-04-1999
--  BY      : H.M.G. Ter Haseborg
--
--  DESCRIPTION : The behavior of a RAM-memory module
--
-----

library IEEE;
use IEEE.std_logic_1164.all;

```

```

entity memory is
  generic (
    tpd           : time := 1 ns;
    d_busWidth    : integer := 16;
    a_busWidth    : integer := 10;
    mem_locations : integer := 1024
  );

  port (
    d_busout : out    std_Logic_vector(d_busWidth-1 downto 0);
    d_busin  : in     std_Logic_vector(d_busWidth-1 downto 0);
    a_bus    : in     std_Logic_vector(a_busWidth-1 downto 0);
    write    : in     bit;
    read     : in     bit;
    ready    : out    bit
  );
end memory;

-----

use work.utilities.all;

architecture behaviour of memory is
begin
  process
    constant low_address : natural := 0;
    constant high_address: natural := mem_locations;
    type memory_array is array (natural range low_address to high_address) of integer;
    variable mem: memory_array := ( others => 0 );
    variable address : natural;
    variable data_out : std_Logic_vector(d_busWidth-1 downto 0);
    constant unknown : std_Logic_vector(d_busWidth-1 downto 0) := ( others => '-' );

    begin
      ready <= '0' after tpd;
      --
      -- wait for a command
      --
      wait until (read = '1') or (write = '1');
      address := bitv2nat(a_bus);
      assert (address >= low_address) and (address <= high_address)
        report "Out of memory range" severity warning;
      if write = '1' then
        mem(address) := bitv2int(d_busin);
        ready <= '1' after tpd;
        wait until write = '0';      -- wait until end of write cycle
      else
        -- read = '1'
        int2bitv(mem(address),data_out);
        d_busout <= data_out;
        ready <= '1' after tpd;
        wait until read = '0';
        d_busout <= unknown;
      end if;
    end process;
  end architecture;

```

```

end if;
end process;
end behaviour;

```

### B.1.3 Processing unit

The processing unit of the system hold a variety of modules, the structure of all those module together is the processing unit. This module describes the construction of all the modules use and the way in which they are connected.

```

-----
--
--  VERSION: 1.0.1
--  DATE   : 11-04-1999
--  BY     : H.M.G. Ter Haseborg
--
--  DESCRIPTION : The structure of the processing unit
--
-----

library ieee;
use ieee.std_logic_1164.all;

entity Processing_unit is
  generic (
    d_busWidth  : integer := 8;
    d_busNorm   : integer := 3;      -- 2log(d_busWidth)
    a_busWidth  : integer := 8;
    act         : integer := 8;      -- width input activation function
    internalWidth : integer := 16
  );

  port(
    d_busin  : in    bit_vector(d_busWidth-1 downto 0);
    d_busout : out   bit_vector(d_busWidth-1 downto 0);
    a_bus    : out   bit_vector(a_busWidth-1 downto 0);
    LdMem    : out   bit;          -- load data from memory
    rdyAdder : out   bit;          -- adder ready
    not_reset : in    bit;         -- a-synchrone reset
    clk      : in    bit;
    start    : in    bit;          -- start determining neuron
    ready    : out   bit;          -- determination ready
  );

end Processing_unit;

-----
use work.general.all;

```

```

use work.utilities.all;
architecture Struct of Processing_unit is

```

```

-----
--                                     Definition of components
-----

```

```

component R_adder

```

```

  generic (

```

```

    busWidth      : integer := 8;
    internalWidth  : integer := 16
  );

```

```

  port(

```

```

    d_input  : in    bit_vector(busWidth-1 downto 0);
    not_reset : in    bit;           -- a-synchrone reset
    clk      : in    bit;
    new_data : in    bit;           -- valid input data
    start    : in    bit;           -- start adder_repeat
    ready    : out   bit;           -- adder ready
    bias     : out   bit;           -- next bias call
    d_output : out   bit_vector(internalWidth-1 downto 0)
  );

```

```

end component;

```

```

component DigiLog

```

```

  generic (

```

```

    d_busWidth      : integer := 8;
    d_busNorm       : integer := 3
  );

```

```

  port(

```

```

    data_input : in    bit_vector(d_busWidth-1 downto 0);
    sign       : in    bit;           -- sign data term
    clk        : in    bit;           -- clock signal
    not_reset  : in    bit;           -- an a-synchrone reset
    start      : in    bit;           -- start signal
    data_output : out   bit_vector((2*d_busWidth)-1 downto 0);
    ready      : out   bit;           -- digilog ready
  );

```

```

end component;

```

```

component Seq_Controller

```

```

  generic ( a_busWidth : integer := 8 );

```

```

  port(

```

```

    not_reset : in    bit;           -- a-synchrone reset
    start     : in    bit;           -- start PU_controller
    clk       : in    bit;           -- clock signal for timing
    a_bus     : out   bit_vector(a_busWidth-1 downto 0);
    LdMem     : out   bit;           -- load from memory.
  );

```

```

        stAdd : out bit -- start the adder
    );
end component;

component PU_Controller
port(
    not_reset : in bit; -- a--synchron reset
    start : in bit; -- start PU_controller
    RdyDigiL : in bit; -- DigiLog multiplier ready
    BuzyAdd : in bit; -- Adder still expect data
    RdyAdd : in bit; -- adder is ready
    clk : in bit; -- clock signal for timing
    StartSeq : out bit; -- start sequence machine
    NextW : out bit; -- call next synaptic weight
    StartDigi : out bit; -- start digiLog multiplier
    Ready : out bit; -- calculation done
);
end component;

component biasArray
generic ( d_busWidth : integer := 16);

port (
    follower : in bit; -- next weight
    data_bus : out bit_vector(d_busWidth-1 downto 0)
);
end component;

component weightArr
generic ( d_busWidth : integer := 8 );

port (
    follower : in bit; -- next synaptic weight
    sign : out bit;
    data_bus : out bit_vector(d_busWidth-1 downto 0)
);
end component;

component discriminatoryFunction
port (
    input : in bit_vector(9 downto 0);
    output : out bit_vector(7 downto 0)
);
end component;

-----
-- Declaration of signals --
-----

signal rdyAdd : bit; -- adder ready
signal rdyMulti : bit; -- multiplier ready

```

```

signal startAdd    : bit;      -- start adder
signal startMulti  : bit;      -- start multiplier
signal startSeq    : bit;      -- start sequence machine
signal nextB       : bit;      -- next bias
signal nextW       : bit;      -- next weight
signal sign        : bit;      -- sign of the synaptic weight
signal inputAdd    : bit_vector(2*d_busWidth-1 downto 0);
signal multiout    : bit_vector(2*d_busWidth-1 downto 0);
signal outAdd      : bit_vector(internalWidth-1 downto 0);
signal inAct       : bit_vector(act-1 downto 0);
signal outmulti    : bit_vector(d_busWidth-1 downto 0);
signal biasOut     : bit_vector(2*d_busWidth-1 downto 0);
signal WeightOut   : bit_vector(d_busWidth-1 downto 0);
signal data_bus    : bit_vector(d_busWidth-1 downto 0);
signal cpuOut      : bit_vector(d_busWidth-1 downto 0);

begin
u0: R_adder
    generic map ( busWidth => 2*d_busWidth , internalWidth => internalWidth )

    port map ( d_input => inputAdd, not_reset => not_reset, clk => clk,
               new_data => rdyMulti, start => startAdd, ready => rdyAdd ,
               bias => nextB, d_output => outAdd);

u1: DigiLog
    generic map ( d_busWidth => d_busWidth, d_busNorm => d_busNorm )

    port map ( data_input => data_bus, sign => sign , clk => clk,
               not_reset => not_reset, start => startMulti,
               data_output => multiout, ready => rdyMulti );

u2: Seq_Controller
    generic map ( a_busWidth => a_busWidth )

    port map ( not_reset => not_reset, start => startSeq, clk => clk,
               a_bus => a_bus, LdMem => LdMem, stAdd => startAdd );

u4: PU_Controller

    port map( not_reset => not_reset, start => start, RdyDigiL => rdyMulti,
               BuzyAdd => startAdd, RdyAdd => rdyAdd, clk => clk,
               StartSeq => startSeq,
               NextW => nextW, StartDigi => startMulti, Ready => Ready );

u5: biasArray
    generic map ( d_busWidth => 2*d_busWidth)
    port map ( follower => nextB, data_bus => biasOut );

u6: weightArr
    generic map ( d_busWidth => d_busWidth)
    port map ( follower => nextW, sign => sign, data_bus => weightOut);

```

```

u7: DiscriminatoryFunction
    port map ( input => inAct, output => cpuOut);      -- outADD too large

```

```

-----
--                               Combinational Logic                               --
-----

```

```

    data_bus <= weightOut when nextW = '1' else d_busin;
    inputadd <= biasOut when nextB = '1' else multiout;
    d_busout <= cpuOut;
    inAct    <= OutAdd(OutAdd'left downto OutAdd'right+6);
    rdyAdder <= rdyAdd;
end Struct;

```

### B.1.3.1 Repeated Adder

The adder use in our design of a neural hardware system is a repeated adder. This adder has to calculated sums of products, where the products are determined by a DigiLog multiplier. The number of terms which will be added together, depends on the signal of the controller within the processing unit.

```

-----
--                               --
--  VERSION: 1.0.1              --
--  DATE    : 11-04-1999        --
--  BY      : H.M.G. Ter Haseborg --
--                               --
--  DESCRIPTION : A two's complement adder, repeated adder --
--                               --
-----

```

```

library ieee;
use ieee.std_logic_1164.all;

```

```

entity R_adder is

```

```

    generic (
        busWidth : integer := 8;
        internalWidth : integer := 16
    );

```

```

    port(
        d_input  : In   bit_vector(busWidth-1 downto 0);
        not_reset : In   bit;           -- a-synchrone reset
        clk       : In   bit;
        new_data  : In   bit;           -- valid input data
        start     : In   bit;           -- start adder_repeat
        ready     : out  bit;           -- adder ready
        bias      : out  bit;           -- next bias call
        d_output  : out  bit_vector(internalWidth-1 downto 0)
    );

```

```
);
end R_adder;
```

```
-----
use work.general.all;
use work.utilities.all;
architecture behaviour of R_adder is
```

```
-----
--                                     the functionality of a full adder
-----
```

```
function full_add (a, b : bit_vector; carry_in: bit) return bit_vector is
variable result, carry_array : bit_vector(a'left+1 downto 0);
begin
    result(0):=a(0) xor b(0) xor carry_in;
    carry_array(0):=((a(0) xor b(0)) and carry_in) or (a(0) and b(0));
    for i in 1 to a'left loop
        result(i):=a(i) xor b(i) xor carry_array(i-1);
        carry_array(i):=((a(i) xor b(i)) and carry_array(i-1)) or (a(i) and b(i));
    end loop;
    result(a'left+1):= carry_array(a'left);
    return result;
end full_add;
```

```
-----
--                                     converts a signed representation into a two's compl.
-----
```

```
function mag2twos (q : bit_vector; width : integer) return bit_vector is
variable twos_q: bit_vector (width downto 0);
begin
    twos_q := full_add( (not q), i2bvd(0, width), '1');      -- (not q) + 1
    return '1' & twos_q(twos_q'left-2 downto 0);
end mag2twos;
```

```
-----
--                                     Declaration of signals
-----
```

```
signal  input      : bit_vector (internalWidth-1 downto 0);
signal  add_reg    : bit_vector (internalWidth-1 downto 0);
signal  add_res     : bit_vector (internalWidth  downto 0);
alias result       : bit_vector(add_res'left-1 downto add_res'right)
                    is add_res(add_res'left-1 downto add_res'right);
```

```
-----
--                                     signal def. state machine
-----
```

```
signal  State      : bit_vector(2 downto 0);
constant Idle      : bit_vector(2 downto 0) := "011";
```

```

constant NextTerm : bit_vector(2 downto 0) := "001";
constant AddEnd   : bit_vector(2 downto 0) := "000";
constant AdderRdy : bit_vector(2 downto 0) := "110";
alias CallBias : bit is State(1);
alias addrdy   : bit is State(2);

begin

-----
--                               Combinational Logic                               --
-----

ready    <= addrdy;
bias     <= CallBias;
d_output <= result when result(result'left) = '0' else mag2twos(result,internalWidth);

-----
--                               State machine definition                               --
-----

stateMachine : process (not_reset, clk)
    variable number : integer;
    variable New_input : bit_vector (internalWidth-1 downto 0);
begin
    if not_reset = '0' then
        State <= Idle;
    elsif clk = '1' and clk'event then
        case State is
            when Idle =>
                add_reg <= input;
                if start = '1' then
                    State <= NextTerm;
                end if;
            when NextTerm =>
                if new_data = '1' then
                    add_res <= full_add(input, add_reg,'0');
                    State <= AddEnd;
                end if;
            when AddEnd =>
                add_reg <= result;
                if start = '0' and new_data = '1' then
                    State <= AdderRdy;
                else
                    State <= NextTerm;
                end if;
            when AdderRdy =>
                State <= Idle;
            end case;
        end if;
        number := bitv2int(d_input);
        int2bitv(number, New_input);
    end if;
end process;

```

```

        input <= new_input;
    end process;
end behaviour;

```

### B.1.3.2 Digilog Multiplier

A multiplier is one of the main components of a neural hardware implementation. The multiplier use in this design is a digital logarithmic multiplier. The product terms a offered one by one to the system, another property is that both terms are positive and are in signed magnitude representation. The answer of the multiplier is only in two's complement representation. This to get a better and easier way to connect the multiplier to the repeated adder. The following description shows the behavior of the DigiLog multiplier.

```

-----
--
--  VERSION: 1.0.1
--  DATE    : 11-04-1999
--  BY      : H.M.G. Ter Haseborg
--
--  DESCRIPTION : DigiLog multiplier for signed data
--                as result two's complement
--
-----

library ieee;
use ieee.std_logic_1164.all;

entity DigiLog is
    generic (
        d_busWidth : integer := 8;
        d_busNorm   : integer := 3
    );

    port(
        data_input  : in  bit_vector(d_busWidth-1 downto 0);
        sign        : in  bit;        -- sign data term
        clk         : in  bit;        -- clock signal
        not_reset   : in  bit;        -- an a-synchrone reset
        start       : in  bit;        -- start signal
        data_output  : out bit_vector((2*d_busWidth)-1 downto 0);
        ready       : out bit         -- digilog ready
    );
end DigiLog;

-----

use work.general.all;
architecture Struct of DigiLog is

```

---

-- the functionality of a full adder --

---

```

function full_add (a,b:bit_vector; carry_in: bit) return bit_vector is
variable result, carry_array : bit_vector(a'left+1 downto 0);
begin
    result(0):=a(0) xor b(0) xor carry_in;
    carry_array(0):=((a(0) xor b(0)) and carry_in) or (a(0) and b(0));
    for i in 1 to a'left loop
        result(i):=a(i) xor b(i) xor carry_array(i-1);
        carry_array(i):=((a(i) xor b(i)) and carry_array(i-1)) or (a(i) and b(i));
    end loop;
    result(a'left+1):= carry_array(a'left);
    return result;
end full_add;

```

---

-- A barrel shifter --

---

```

function barrel_Shifter (vector: bit_vector; shift: bit_vector) return bit_vector is
variable shift_int : integer := 0;
variable shift_counter : integer range 0 to 2**(shift'high+1) := 1;
begin
    shift_int := bit2int(vector);
    if orvec(shift) = '1' then
        for j in shift'low to shift'high loop
            if (shift(j) /= '0') then
                for l in 1 to shift_counter loop
                    shift_int := shift_int * 2;
                end loop;
            end if;
            shift_counter := shift_counter * 2;
        end loop;
    end if;
    return i2bvd(shift_int, 2*d_busWidth);
end barrel_Shifter;

```

---

-- the function 'stripbit' removes the  
-- highest order bit of a data vector --

---

```

function stripbit (vector: bit_vector) return bit_vector is
variable bitfound : bit := '0';
variable norm_vector : bit_vector(d_busWidth-1 downto 0);
begin
    for i in vector'high downto vector'low loop
        if vector(i) = '1' and bitfound = '0' then
            norm_vector(i) := '0';

```

```

        bitfound := '1';
    else
        norm_vector(i) := vector(i);
    end if;
end loop;
return norm_vector;
end stripbit;

```

---

```

--                                     This function checks if a data vector is zero                                     --

```

---

```

function not_Zero(vector: bit_vector) return bit is
variable bitFound: bit := '0';
begin
    for i in vector'high downto vector'low loop
        if vector(i) = '1' then
            bitFound := '1';
        end if;
    end loop;
    return bitFound;
end not_Zero;

```

---

```

--                                     normalize determines the index of the MSB in a vector                                     --

```

---

```

function normalize (vector:bit_vector)return bit_vector is
variable bitFound : bit := '0';
variable norm_vector : bit_vector(d_busNorm-1 downto 0);
begin
    for i in vector'high downto vector'low loop
        if vector(i) = '1' and bitFound = '0' then
            norm_vector := i2bvd(i-vector'low, d_busNorm);
            bitFound := '1';
        end if;
    end loop;
    return norm_vector;
end normalize;

```

---

```

--                                     converts a signed representation into a two's compl.                                     --

```

---

```

function mag2twos (q:bit_vector; width : integer) return bit_vector is
variable twos_q: bit_vector (width downto 0);
begin
    twos_q := full_add( (not q), i2bvd(0,width), '1');      -- (not q) + 1
    return '1' & twos_q(twos_q'left-2 downto 0);
end mag2twos;

```

```

-----
--                                     Declaration of signals                                     --
-----

signal norm_r      : bit_vector(d_busNorm-1 downto 0);  -- normalize register
signal ar          : bit_vector(d_busWidth-1 downto 0);  -- data register a
signal br          : bit_vector(d_busWidth-1 downto 0);  -- data register b
signal data        : bit_vector(d_busWidth-1 downto 0);  -- data path of data_input or
data_arbr
signal data_arbr   : bit_vector(d_busWidth-1 downto 0);  -- data path of ar and br
signal o_reg       : bit_vector(2*d_busWidth-1 downto 0); -- output register
signal shifter_O   : bit_vector(2*d_busWidth-1 downto 0); -- shifter output
signal norm_data   : bit_vector(d_busNorm-1 downto 0);  -- the norm of the data
signal adder_O     : bit_vector(2*d_busWidth downto 0);  -- adder output
signal loadRa      : bit;
alias carry        : bit is adder_O(adder_O'left);
alias r            : bit_vector(adder_O'left-1 downto adder_O'right)
                    is adder_O(adder_O'left-1 downto adder_O'right);

```

```

-----
--                                     signal def. state machine                                     --
-----

signal State       : bit_vector(4 downto 0);
constant End_Start : bit_vector(4 downto 0) := "01000";
constant InitA     : bit_vector(4 downto 0) := "00011";
constant InitB     : bit_vector(4 downto 0) := "00101";
constant ProcesA   : bit_vector(4 downto 0) := "00110";
constant ProcesB   : bit_vector(4 downto 0) := "00100";
alias switch_data  : bit is State(0);
alias switch_arbr  : bit is State(1);
alias rdy          : bit is State(3);

```

begin

```

-----
--                                     Combinational Logic                                     --
-----

data_output <= o_reg when o_reg(o_reg'left)='0' else mag2twos(o_reg, 2*d_busWidth);
shifter_O   <= barrel_shifter(data, norm_r);
data        <= data_input when switch_data = '1' else data_arbr;
data_arbr   <= ar when switch_arbr='1' else br;
adder_O     <= full_add(shifter_O, o_reg, '0');
ready       <= rdy;
norm_data   <= normalize(data);
loadra      <= switch_arbr;
norm_r      <= norm_data when clk='1' and clk'event else norm_r;

```

```

state_tr:process(not_reset,clk)
begin
    if not_reset = '0' then
        State <= End_Start;
    elsif clk = '1' and clk'event then
        case State is
            when End_Start =>
                if start = '1' then
                    State <= InitA;
                end if;
            when InitA =>
                o_reg <= (others => '0');
                if not_Zero(data) = '1' then
                    ar <= stripbit(data);
                    State <= InitB;
                else
                    State <= End_start;
                end if;
            when InitB =>
                if not_Zero(data) = '1' then
                    br <= stripbit(data);
                    o_reg <= r;
                    State <= ProcesA;
                else
                    State <= End_Start;
                end if;
            when ProcesA =>
                o_reg <= r;
                ar <= stripbit(data);
                if not_Zero(data) = '1' then
                    State <= ProcesB;
                else
                    o_reg(o_reg'high) <= sign;
                    State <= End_Start;
                end if;
            when ProcesB =>
                o_reg <= r;
                br <= stripbit(data);
                if not_Zero(data) = '1' then
                    State <= ProcesA;
                else
                    o_reg(o_reg'high) <= sign;
                    State <= End_Start;
                end if;
        end case;
    end if;
end process;

```

```

        end if;
    end process state_tr;

end Struct;

```

### B.1.3.3 Sequence Controller

This controller description, takes care of the determine sequence, given a network topology. The only output is the address of a memory location where the data is stored which must be use by the next calculations.

```

-----
--
--
--  VERSION: 1.0.1
--  DATE   : 11-04-1999
--  BY     : H.M.G. Ter Haseborg
--
--  DESCRIPTION : The determination sequence machine
--                it det. the addres of the following memory
--                location to recall
--
-----

library ieee;
use ieee.std_logic_1164.all;

entity Seq_Controller is
    generic ( a_busWidth : integer := 8 );

    port(
        not_reset : in    bit;           -- a-synchrone reset
        start      : in    bit;           -- start PU_controller
        clk        : in    bit;           -- clock signal for timing
        a_bus      : out   bit_vector(a_busWidth-1 downto 0);
        LdMem      : out   bit;           -- load from memory.
        stAdd      : out   bit;           -- start the adder
    );
end Seq_Controller;

-----

use work.general.all;
use work.constants.all;
architecture behaviour of Seq_Controller is

    signal address : bit_vector (a_busWidth-1 downto 0);

    -----
    --                                     signal def. state machine
    -----

```

```

signal    State    : bit_vector(1 downto 0);
constant Idle      : bit_vector(1 downto 0) := "10";
constant waiting   : bit_vector(1 downto 0) := "01";
alias      Read    : bit is State(0);

begin

-----
--                                     Combinational Logic                                     --
-----

    a_bus    <= address;
    LdMem    <= Read;

-----
--                                     The calculation sequence machine's behavior                                     --
-----

sequence : process (not_reset, clk, start)
variable counter, layer, neuron, next_site : integer;
begin
    if not_reset = '0' then
        layer := 1; neuron := 0; next_site := 0; counter := 0;
        State <= Idle;
    elsif clk = '1' and clk'event then
        case State is
            when Idle =>
                if start = '1' then
                    address <= l2bvd((counter + next_site), a_busWidth);
                    next_site := next_site + 1;           -- next input site
                    State <= waiting;
                end if;
            when waiting =>
                if next_site >= network(layer-1) then
                    next_site := 0;
                    neuron := neuron + 1;                 -- next neuron
                    if neuron > network(layer)-1 then
                        neuron := 0;
                        counter := counter + network(layer-1);
                        layer := layer + 1;
                        if layer > (Nr_layers-1) then
                            counter := 0; layer := 1;
                        end if;
                    end if;
                end if;
            end if;
            State <= Idle;
        end case;
    end if;
    if next_site > 0 then
        stAdd <= '1';           -- Start adder
    else
        stAdd <= '0';
    end if;
end process;

```

```

        end if;
    end process;

    end behaviour;

```

#### B.1.3.4 Processing Units Controller

This description is the main controller of the processing unit. It controls all the ins and outs of the processing unit. All the modules controlling line are coming together in this controller to make sure that every action is controlled and synchronized. The state machine which performs all the actions is described in the following VHDL-code.

```

-----
--
--  VERSION: 1.0.1
--  DATE   : 11-04-1999
--  BY     : H.M.G. Ter Haseborg
--
--  DESCRIPTION : Calculation machine that is capable of
--                controlling the synaptic and neuron calculations
--
-----

library ieee;
use ieee.std_logic_1164.all;

entity PU_Controller is
    port(
        not_reset : in  bit;           -- a-synchrone reset
        start      : in  bit;           -- start PU_controller
        RdyDigiL   : in  bit;           -- DigiLog multiplier ready
        BuzyAdd    : in  bit;           -- Adder still expect data
        RdyAdd     : in  bit;           -- adder is ready
        clk        : in  bit;           -- clock signal for timing
        StartSeq   : out bit;           -- start sequence machine
        NextW      : out bit;           -- call next synaptic weight
        StartDigi  : out bit;           -- start digiLog multiplier
        Ready      : out bit;           -- calculation done
    );
end PU_Controller;

-----
use work.general.all;
use work.utilities.all;
architecture behaviour of PU_Controller is
    -----
    --
    --                signal def. state machine
    --
    -----

```

```

signal    State      : bit_vector(6 downto 0);
constant Idle       : bit_vector(6 downto 0) := "1000000";
constant LdInpandB : bit_vector(6 downto 0) := "0000011";
constant StDigi     : bit_vector(6 downto 0) := "0000010";
constant LdW        : bit_vector(6 downto 0) := "0001100";
constant Waiting    : bit_vector(6 downto 0) := "0000000";
constant WriteOut   : bit_vector(6 downto 0) := "0100000";
constant Rdy        : bit_vector(6 downto 0) := "0010000";
alias    NextGo     : bit is State(0);      -- determine next addres to call
alias    StartM      : bit is State(1);     -- start multiplier
alias    nextWeight  : bit is State(2);     -- call next synapic weight
alias    calRdy      : bit is State(4);     -- calculation done

```

```

begin

```

```

-----
--                               Combinational Logic                               --
-----

```

```

    startSeq <= NextGo;
    NextW    <= NextWeight;
    Ready    <= calRdy;
    StartDigi <= StartM;

```

```

-----
--                               State machine definition                               --
-----

```

```

calculation : process (not_reset, clk, start)

```

```

begin

```

```

    if not_reset = '0' then
        State <= Idle;
    elsif clk = '1' and clk'event then
        case State is
            when Idle =>
                if start = '1' then
                    State <= LdInpandB;
                end if;
            when LdInpandB =>
                State <= StDigi;
            when StDigi =>
                State <= LdW;
            when LdW =>
                State <= Waiting;
            when Waiting =>
                if RdyDigiL = '1' and BuzyAdd = '0' then
                    State <= WriteOut;
                elsif RdyDigiL = '1' and BuzyAdd = '1' then
                    State <= LdInpandB;
                end if;
            when WriteOut =>
                if RdyAdd = '1' then

```

```

        State <= Rdy;
    end if;
    when Rdy =>
        State <= Idle;
    end case;
end if;
end process;

end behaviour;

```

### B.1.3.5 Bias Array

The biases of the neural network are stored in a look-up table in such an order that they can be accessed one by one. The output of this module represents the values of the table in two's complement, and with a word width twice as large as the use data bus. The output is directly connected to the repeated adder. The behavioral description of the working of this module is given by the following code.

```

-----
--
--  VERSION: 1.0.1
--  DATE   : 11-04-1999
--  BY     : H.M.G. Ter Haseborg
--
--  DESCRIPTION : The behavior of the bias-ROM memory
--
-----

library ieee;
use ieee.std_logic_1164.all;

entity biasArray is
    generic ( d_busWidth : integer := 16);

    port (
        follower : in    bit;           -- next weight
        data_bus : out   bit_vector(d_busWidth-1 downto 0)
    );
end biasArray;

-----

use work.general.all;
use work.constants.all;

architecture behavior of biasArray is

```

```

-----
--                                     the functionality of a full adder                                     --
-----

```

```

function full_add (a,b:bit_vector; carry_in: bit) return bit_vector is
variable result, carry_array : bit_vector(a'left+1 downto 0);
begin
    result(0):=a(0) xor b(0) xor carry_in;
    carry_array(0):=((a(0) xor b(0)) and carry_in) or (a(0) and b(0));
    for i in 1 to a'left loop
        result(i):=a(i) xor b(i) xor carry_array(i-1);
        carry_array(i):=((a(i) xor b(i)) and carry_array(i-1)) or (a(i) and b(i));
    end loop;
    result(a'left+1):= carry_array(a'left);
    return result;
end full_add;

```

```

-----
--                                     from signed to two's complement representation                                     --
-----

```

```

function mag2twos (q:bit_vector) return bit_vector is
variable twos : bit_vector(d_busWidth downto 0);
begin
    twos := full_add((not q), i2bvd(0, d_busWidth), '1');
    return '1' & twos(twos'left-2 downto 0);
end mag2twos;

```

```

-----
--                                     Declaration of signals                                     --
-----

```

```

    signal value   : bit_vector(23 downto 0);
    signal valTbl  : bit_vector(d_busWidth-1 downto 0);
    signal pre_out : bit_vector(d_busWidth-1 downto 0);
    signal count   : integer :=maxBias;
    alias valueTbl : bit_vector(d_busWidth-1 downto 0) is
        value(value'left downto value'right+(23-d_busWidth));

```

```

begin

```

```

-----
--                                     This process represents the table values one bye one                                     --
--                                     in only one sequence. Namely from index 0 to max index.                                     --
-----

```

```

counter: process (follower)
begin
    if follower = '1' and follower'event then
        if count < maxBias then
            count <= count + 1;
        else
            count <= 0;

```

```

        end if;
    end if;
end process;

-----
--                               Combinational Logic                               --
-----

-- translates the integer values of the weight table into
-- a binary representation and if necessary puts it in two's
-- complement.

value    <= i2bvd(BiasTbl(count), 24);           -- coding bias in 24 bit
valTbl   <= valueTbl;
pre_out  <= valTbl when BiasTbl(count) >= 0 else mag2twos(valTbl);
data_bus <= pre_out;

end behavior;

```

### B.1.3.6 Synaptic Weight Array

This module describes the interaction between the generated look-up table for the synaptic weights, and the processing unit it self. The weights are stored in a table in such an order that they can be recall one by one. These synaptic weights are represented in signed magnitude representation, to preserve the DigiLog multipliers properties. Two functions are described, to make use of rounding techniques, as mentioned in chapter 6. Those functions are round-off for the rounding the result up and down, and the jamming operator for some sort of ceiling operation. The following description shows how the tables are accessed and the way of converting the values to signed magnitude representation.

```

-----
--                               --
--  VERSION: 1.0.1              --
--  DATE   : 11-04-1999        --
--  BY     : H.M.G. Ter Haseborg --
--                               --
--  DESCRIPTION : The behavior of the weight rom module --
--                               --
-----

use work.general.all;
use work.constants.all;
library ieee;
use ieee.std_logic_1164.all;

entity weightArr is
    generic ( d_busWidth : integer := 8);

```

```

port (
    follower : In      bit;      -- next weight
    sign      : out     bit;
    data_bus : out      bit_vector(d_busWidth-1 downto 0));
end weightArr;

```

architecture behavior of weightArr is

---

Function Definition

---

function roundoff(Nmax: bit\_vector; N : integer) return bit\_vector is

variable p : bit\_vector(N-1 downto 0);

variable q,klad: integer;

begin

for j in 0 to N-1 loop

p(N-1-j):= Nmax(11-j);

end loop;

q := bit2int(Nmax(Nmax'high-N downto 0));

if q >= (2\*\*(12-N-1)) then

klad := bit2int(p)+1;

p := i2bvd(klad,N);

end if;

return p;

end roundoff;

function jamming(Nmax: bit\_vector; N : integer) return bit\_vector is

variable p: bit\_vector(N-1 downto 0);

variable q,klad: integer;

begin

for j in 0 to N-1 loop

p(N-1-j):= Nmax(11-j);

end loop;

q:=bit2int(Nmax(Nmax'high-N downto 0));

if q /= 0 then

p(p'low):='1';

end if;

return p;

end jamming;

---

Declaration of signals

---

signal value : bit\_vector(11 downto 0);

signal valTbl : bit\_vector(d\_busWidth-1 downto 0);

signal count : integer :=maxweight;

alias valueTbl : bit\_vector(d\_busWidth-1 downto 0) is

value(value'left downto value'right+(11-d\_busWidth));

```
begin
```

```
-----
--          This process represents the table values one bye one          --
--          in only one sequence. Namely from index 0 to max index.      --
-----
```

```
counter: process (follower)
```

```
begin
```

```
    if follower = '1' and follower'event then
```

```
        if count < maxweight then
```

```
            count <= count + 1;
```

```
        else
```

```
            count <= 0;
```

```
        end if;
```

```
    end if;
```

```
end process;
```

```
-----
--          translates the integer values of the weight table into      --
--          a binary representation and if nessecery in                  --
--          a signed representation                                       --
-----
```

```
value    <= i2bvd(weightsTbl(count),12);
```

```
-- max coding 12 bits
```

```
valTbl   <= valueTbl;
```

```
-- valTbl <= roundoff(value, d_busWidth);
```

```
-- for the round-off operator
```

```
-- valTbl <= jamming(value, d_busWidth);
```

```
-- for the jamming operator
```

```
sign     <= '0' when weightsTbl(count) >= 0 else '1';
```

```
data_bus <= valTbl;
```

```
end behavior;
```

### B.1.3.7 Discriminatory Function

The discriminatory function use is the sigmoid function. The function results are stored in a look-up table. The VHDL description of this module is generated by using a tool, therefore it can be found in the following appendix C.

### B.1.4 Main Controller

The module main controller takes care of a good communication between all components in the neural hardware system. Those main three components are; (1) the interface module, (2) the RAM module, and (3) the Processing unit. The behavior of the main controller is described in the following VHDL-code.

```

-----
--
-- VERSION: 1.0.1
-- DATE : 11-04-1999
-- BY : H.M.G. Ter Haseborg
--
-- DESCRIPTION : The Main controller
--
-----

```

```

library ieee;
use ieee.std_logic_1164.all;

```

```

entity Main_Controller is
  generic ( a_busWidth : integer := 8);

```

```

  port(
    not_reset : in bit;      -- a-synchrone reset
    clk       : in bit;
    RdyIO     : in bit;      -- IO ready
    RdyCal    : in bit;      -- calculations done
    StLd      : out bit;      -- Start Load process
    StCal     : out bit;      -- Start calculations
    StOut     : out bit;      -- Start output process
    a_bus     : out bit_vector(a_busWidth-1 downto 0)
  );

```

```

end Main_Controller;

```

```

use work.general.all;
use work.utilities.all;
use work.constants.all;

```

```

architecture behaviour of Main_Controller is

```

```

-----
--                               signal def. state machine
-----

```

```

signal State : bit_vector(2 downto 0);
constant Load : bit_vector(2 downto 0) := "001";
constant Cal : bit_vector(2 downto 0) := "010";
constant Output : bit_vector(2 downto 0) := "100";
alias LoadGo : bit is State(0);
alias CalGo : bit is State(1);
alias OutGo : bit is State(2);

```

```

begin

```

```

-----
--                               Combinational Logic
-----

```

```

  StLd <= LoadGo;
  StCal <= CalGo;
  StOut <= OutGo;

```

```

-----
--                               State machine main behavior                               --
-----

MainProcess : process (not_reset, clk )
variable StoreAddress: integer;
variable number      : integer := 0;
begin
    if not_reset = '0' then
        StoreAddress := network(0);
        State <= Load;
    elsif clk = '1' and clk'event then
        case State is
            when Load =>
                StoreAddress := network(0);
                if rdyIO = '1' then
                    State <= Cal;
                end if;
            when Cal =>
                if RdyCal = '1' and StoreAddress >= Nr_Neurons-1 then
                    if number > network(2)-1 then
                        State <= Output;
                    end if;
                elsif RdyCal = '1' and StoreAddress < Nr_Neurons-1 then
                    StoreAddress := StoreAddress + 1;
                end if;
                number := number + 1;
            when Output =>
                if rdyIO = '1' then
                    State <= Load;
                end if;
            end case;
        end if;
        a_bus <= i2bvd(StoreAddress,a_busWidth);
    end process;
end Behaviour;

```

---

# Appendix C

## *The Generation Tools and Simulation Environments*

---

### C.1 Characteristic Extraction Tool

The extraction of the network characteristics is performed by using a *InterAct* application. This application is also capable of translating the pattern database file, which is used to train and test the network in *InterAct*. Not all the characteristics are converted to the resulting VHDL-file, in chapter 4 those missing statements are reported there. The code missing is the translation of the networks topology, and is in the hardware neural system very important (for the calculation sequence). The *InterAct* application is written in the program language C. Using this program can only be satisfying as the properties known and used, as mentioned in chapter 4. The two VHDL-files generated by this tool are, the file named constant and the file name suitable. The first file (constant.vhd) shows the extracted parameters of the network which are mentioned in two different look-up tables, namely the weight table and the bias table. The second file (Simtbl.vhd) is showing a translation of the input and target vectors from the pattern database file. The following C-code shows the source code of this *InterAct* application.

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "interact.h"

#define TRUE (1==1)
#define FALSE (1==0)
#define DATA_PWD "/rug103/home2/users/csg/csg890/data/"
#define MAX_BITS 14

typedef struct {
    Int NrBits;
```

```

        int createSim;
        int createTbl;
        int manual;
        float UpperRange, LowerRange;
    } Options;

void CheckStatus (status_St status)
/* _____ */
/* ACTION : Checks the status fields after processing a InterAct call */
/* _____ */
/* INPUT : status_St status -> a record with status fields */
/* _____ */
/* OUTPUT : FUNCTION -> by a successful InterAct call the program resumes */
/* else the application will stop */
/* _____ */
{
    if(status.all != status_Sok)
    {
        error_Sn_print(status);
        interact_Sterminate(init_mode_Sstop, &status);
        exit(1); }
}

void setDefault(Options *option)
/* _____ */
/* ACTION : Set the default options is the data structure option */
/* _____ */
/* INPUT : Options *Option -> a struct with data field for the options */
/* _____ */
/* OUTPUT : FUNCTION -> The option field filled with default values */
/* _____ */
{
    option->NrBits = 12;
    option->createSim = TRUE;
    option->createTbl = TRUE;
    option->manual = FALSE;
    option->UpperRange = 10.0;
    option->LowerRange = -10.0;
}

void getOptions(int argc, char **argv[], Options *option)
/* _____ */
/* ACTION : Checks which options are given as parameters to the application */
/* _____ */
/* INPUT : int argc -> a counter and gives the number of given arguments */
/* char argv[] -> an array with the arguments of the application */
/* Options *Option -> the Options received from the environment */
/* _____ */
/* OUTPUT : FUNCTION -> in the struct option, the arguments from the input */
/* _____ */

```

```

{   int i;
    for (i=2; i<=argc; i++) {
        switch(**argv[i]) {
            case 's' : case 'S' :
                option->createSim = TRUE;
                break;
            case 't' : case 'T' :
                option->createTbl = TRUE;
                break;
            case 'm' : case 'M' :
                option->manual = TRUE;
                break;
            case 'u' : case 'U' :
                option->UpperRange = atof(*argv[i+1]);
                break;
            case 'l' : case 'L' :
                option->LowerRange = atof(*argv[i+1]);
                break;
            default:
                break;
        }
    }
}

void MyScalePatterns (pattern_Slist_id_t list_id,
                     float input_min, float input_max, float target_min, float target_max)
/*
/* ACTION : Scales the pattern database
/*
/* INPUT : float input_min -> lowerbound of the input range to scale
/* float input_max -> upperbound of the input range to scale
/* float target_min -> lowerbound of the target range to scale
/* float target_max -> upperbound of the target range to scale
/*
/* OUTPUT : FUNCTION -> a fully scaled database between the given bounds
/*
{
    status_St          status;
    pattern_Slist_info_t info;
    pattern_Slist_column_statistics_t statistics;
    long               i;

    pattern_Slist_get_info (list_id, &info, &status);
    /* Scale the input-patterns */
    for (i = 1; i <= info.nr_inputs; i++)
    {
        pattern_Slist_get_statistics (list_id, selection_Sinput_column, i, &statistics, &status);
        pattern_Slist_scale (list_id, selection_Sinput_column, i, statistics.min, statistics.max,
                             input_min, input_max, &status);
    }
}

```

```

/* Scale the target-patterns */
for (i = 1; i <= info.nr_targets; i++)
{
    pattern_Slist_get_statistics(list_id, selection_Starget_column, i, &statistics, &status);
    pattern_Slist_scale(list_id, selection_Starget_column, i, statistics.min, statistics.max,
        target_min, target_max, &status);
}
}

void ExamineConstants(Int *nrBias, Int *nrWeights, float *max)
/*
/* ACTION : searching the largest absolute weight or bias
/*
/* INPUT : int nrBias -> the total number of biases in the network
/*          int nrWeights-> the total number of synaptic weights in the system
/*
/* OUTPUT : FUNCTION -> the maximal absolute value of the weights as biases
/*
{
    get_Sneuron_info_t neuron_info;
    get_Sinput_info_t *Info;
    status_St status;
    long nr_inputs;
    int i;
    float largest=0.0, smallest=0.0;
    bias_St bias;

    get_Sid_neuron(get_Soption_first, 0L, hidden_Slist, NULL, &neuron_info.id, &status);
    CheckStatus(status);
    while(status.all==status_Sok)
    {
        get_Sneuron_bias(neuron_info.id, &bias, &status); CheckStatus(status);
        If (largest < bias) largest = bias;
        If (smallest > bias) smallest = bias;
        *nrBias += 1;

        get_Sinfo_neuron(&neuron_info, &status); CheckStatus(status);
        /* Allocate room to store the results */
        info = (get_Sinput_info_t *) malloc(neuron_info.nr_inputs * sizeof(get_Sinput_info_t));
        get_Sneuron_inputs(neuron_info.id, info, &nr_inputs, &status);
        for(i=0; i<=(neuron_info.nr_inputs-1); i++)
        {
            If (largest < info[i].weight) largest=info[i].weight;
            If (smallest > info[i].weight) smallest=info[i].weight;
            *nrWeights += 1;
        }
        get_Sid_neuron(get_Soption_larger, neuron_info.id, neuron_Slist, NULL,
            &neuron_info.id, &status);
    }
    If (fabs(largest) < fabs(smallest))

```

```

        *max = fabs(smallest);
    else
        *max = fabs(largest);
}

float scaleTo(float value, float FromLow, float FromHigh, float ToLow, float ToHigh)
/*
/* ACTION : scales a value of an range to another range
/*
/* INPUT : float    value -> the value that must be scaled
/*          float    Formlow -> The lowest interval of the value given
/*          float    Formhigh -> The highest interval of the value given
/*          float    Tolow -> The lowest interval of the new value given
/*          float    Tohigh -> The highest interval of the new value given
/*
/* OUTPUT : FUNCTION -> a new value in another range
/*
{
    float intervalFrom, intervalTo;
    intervalFrom = FromHigh - FromLow;
    intervalTo = ToHigh - ToLow;

    return ((value-FromLow)*intervalTo) / intervalFrom + ToLow;
}

void createBiasTbl(float range, FILE *ptr, int NrBits, int nrbias)
/*
/* ACTION : writes to file the translated bias values in N bits
/*
/* INPUT : float    range -> the bias interval
/*          FILE     prt -> a filepointer to a file
/*          int      nrbits -> the number of bits to represent the biases
/*          int      nrbias -> the total number of biases in the network
/*
/* OUTPUT : FUNCTION -> a file holding the translated biases of the network
/*
{
    get_Sneuron_info_t neuron_info;
    status_St          status;
    bias_St            bias;
    int                i=1;

    get_Sid_neuron(get_Soption_first, 0L, hidden_Slist, NULL, &neuron_info.id, &status);
    CheckStatus(status);
    while(status.all==status_Sok)
    {
        get_Sneuron_bias(neuron_info.id, &bias, &status); CheckStatus(status);
        If (i < nrbias)
            fprintf(ptr,"%d, ",(int)scaleTo(bias, -range, range, -pow(2.0,(NrBits*2.0)-1.0)
                ,pow(2.0,(NrBits*2.0)-1.0)));
    }
}

```

```

else
    fprintf(ptr,"%d\n\t", (Int)scaleTo(bias, -range, range, -pow(2.0,(NrBits*2.0)-1.0)
        ,pow(2.0,(NrBits*2.0)-1.0)));
    if (i%10 == 0)        fprintf(ptr,"\n\t");
    get_Sid_neuron(get_Soption_larger, neuron_info.id, neuron_Slist, NULL,
        &neuron_info.id,&status);
    i+=1;
}
}

void createWeightTbl(float range, FILE *ptr, Int NrBits, Int nrweights)
/*
/* ACTION : writes to file the translated weights in N bits
/*
/* INPUT : float      range -> the weights interval
/*          FILE      prt -> a filepointer to a file
/*          int       nrbits -> the number of bits to represent the weights
/*          int       nrbias -> the total number of weights in the network
/*
/* OUTPUT : FUNCTION -> a file holding the translated weights of the network
/*
{
    get_Sneuron_info_t    neuron_info;
    get_Sinput_info_t     *info;
    status_St             status;
    long                  nr_inputs;
    Int                    i, j=1;

    get_Sid_neuron(get_Soption_first, 0L, hidden_Slist, NULL, &neuron_info.id, &status);
    CheckStatus(status);
    while(status.all==status_Sok)
    {
        get_Sinfo_neuron(&neuron_info,&status);    CheckStatus(status);
        /* Allocate room to store the results */
        info = (get_Sinput_info_t *)malloc(neuron_info.nr_inputs *sizeof(get_Sinput_info_t));
        get_Sneuron_inputs(neuron_info.id, info, &nr_inputs, &status);
        for(i=0; i<=(neuron_info.nr_inputs-1); i++)
        {
            if (j< nrweights)
                fprintf(ptr,"%d", (Int)scaleTo(info[i].weight, -range, range, -pow(2.0,(NrBits)-1.0)
                    ,pow(2.0,(NrBits)-1.0)));
            else
                fprintf(ptr,"%d\n", (Int)scaleTo(info[i].weight, -range, range, -pow(2.0,(NrBits)-1.0)
                    ,pow(2.0,(NrBits)-1.0)));
            if (j%10 == 0)
                fprintf(ptr,"\n\t");
            j+=1;
        }
    }
}

```

```

        get_Sid_neuron(get_Soption_larger, neuron_info.id, neuron_Slist, NULL,
                        &neuron_info.id,&status);
    }
}

void createFile(float range, Int nrBias, Int nrWeights,Int nrbits, char *filename)
/*
/* ----- */
/* ACTION : Creates the body of the VHDL file */
/*
/* INPUT : float      range -> the range of all the characteristics */
/*         int         nrBias -> The number of biases in the network */
/*         int         nrWeights -> The number of weights in the network */
/*         int         nrbits -> The number of bits used for representation */
/*         char        filename -> the name of the VHDL file */
/*
/* OUTPUT : FUNCTION  -> a file with a body */
/* ----- */
{
    FILE *filePtr;

    if ((filePtr = fopen("constant.vhd", "w")) == NULL)
    {
        printf("FATAL ERROR -> VHDL File could NOT be GENERATED !!!\n");
        return;
    }

    fprintf(filePtr,"-----}\n");
    fprintf(filePtr,"--                                VERSION : 1.0    --\n");
    fprintf(filePtr,"-- FILE      : constant.vhd                        --\n");
    fprintf(filePtr,"-- DATE       : may 3rd, 1999                                --\n");
    fprintf(filePtr,"-- GENERATED by : Generate                                    --\n");
    fprintf(filePtr,"-- DESCRIPTION  : The biases and weight tables of            --\n");
    fprintf(filePtr,"--              a MLP-network.                                --\n");
    fprintf(filePtr,"--              %s      --\n", filename );
    fprintf(filePtr,"--                                --\n");
    fprintf(filePtr,"-----\n\n");
    fprintf(filePtr,"library ieee;\nuse ieee.std_logic_1164.all;\n");
    fprintf(filePtr,"package constants is\n");
    fprintf(filePtr,"type tableBias is array (0 to %d) of integer;\n",nrBias-1);
    fprintf(filePtr,"constant BiasTbl: tableBias := (\n");
    /* Fill bias Table */

    createBiasTbl(range, filePtr, nrbits, nrBias);

    fprintf(filePtr,")\n\nconstant maxBias : integer := %d;\n",nrBias-1);
    fprintf(filePtr,"type tableWeights is array (0 to %d) of integer;\n",nrWeights-1);
    fprintf(filePtr,"constant weightsTbl: tableWeights := (\n");
    /* Fill weights Table */

    createWeightTbl(range, filePtr, nrbits, nrWeights);
    fprintf(filePtr,")\n\nconstant maxWeight : integer := %d;\n",nrWeights-1);

```

```

        fprintf(filePtr, "end constants;\n");
        fclose(filePtr);
    }

    void createTables(Options *option, char *filename)
    /* ----- */
    /* ACTION : decide which file to create and how */
    /* ----- */
    /* INPUT : Options      option -> the argument record */
    /*          char        Filename -> The name of the file */
    /* ----- */
    /* OUTPUT : FUNCTION   -> a decision how to create the file */
    /* ----- */
    {
        Int nrBias=0, nrWeights=0;
        float max=0.0;

        If(option->manual)
            createFile(option->UpperRange-option->LowerRange, nrBias, nrWeights, option->NrBits,
                        filename);
        else
        {
            /* determine abs. lower or upper range */
            ExamineConstants(&nrBias, &nrWeights, &max);
            createFile(max, nrBias, nrWeights, option->NrBits, filename);
        }
    }

    void createSimFile(pattern_Slist_id_t list_id, char *filename)
    /* ----- */
    /* ACTION : Creates the simulation files (input and target patterns) */
    /* ----- */
    /* INPUT : A Pattern list -> a pattern database */
    /*          char          Filename -> The name of the file */
    /* ----- */
    /* OUTPUT : FUNCTION   -> a file containing several input and target patterns */
    /* ----- */
    {
        status_St      status;
        out_St          inputs[INTERACT_MAX_PATTERN_INPUTS];
        out_St          targets[INTERACT_MAX_PATTERN_INPUTS];
        long            nr_inputs, nr_targets, nr_patterns, pattern_id;
        pattern_Slist_Info_t info;
        FILE            *filePtr;
        Int             i,j=1;

        pattern_Slist_get_info(list_id, &info, &status);
        nr_patterns = info.nr_patterns;
        If ((filePtr = fopen("SimTbl.vhd", "w")) == NULL) {

```

```

printf("FATAL ERROR -> VHDL File could NOT be GENERATED !!!\n");
return;
}
fprintf(filePtr,"-----}\n"
fprintf(filePtr,"--                                VERSION : 1.0    --\n");
fprintf(filePtr,"-- FILE                        : SimTbl.vhd      --\n");
fprintf(filePtr,"-- DATE                          : may 3rd, 1999      --\n");
fprintf(filePtr,"-- GENERATED by                     : Generate              --\n");
fprintf(filePtr,"-- DESCRIPTION                       : The Input and Target signal of --\n");
fprintf(filePtr,"--          %s                      --\n", filename);
fprintf(filePtr,"--          in a Max-bit representation %d          --\n",MAX_BITS);
fprintf(filePtr,"--                                          --\n");
fprintf(filePtr,"-----\n\n\n");
fprintf(filePtr,"\\nlibrary ieee;\\nuse ieee.std_logic_1164.all;\\n");
fprintf(filePtr,"package simtbl is \\n\\n");
fprintf(filePtr,"type tablesim is array (0 to %d) of integer\\n",nr_patterns-1);
fprintf(filePtr,"\\nconstant maxpatterns : integer := %d;\\n",nr_patterns-1);

pattern_Slist_get_pattern (list_id, pattern_id, inputs, &nr_inputs, targets, &nr_targets, &status);
for (i=1; i<=nr_inputs; i++) {
    fprintf(filePtr,"\\n\\nconstant inputTbl%d: tablesim := (\\n\\t",i);
    for (pattern_id = 1; pattern_id <= nr_patterns; pattern_id++)
    {
        pattern_Slist_get_pattern (list_id, pattern_id, inputs, &nr_inputs,
            targets, &nr_targets, &status);
        if (j< nr_patterns)
            fprintf(filePtr,"%d, ",(int)scaleTo(inputs[i-1],0.0,1.0,0.0,pow(2.0,MAX_BITS)-1.0));
        else
            fprintf(filePtr,"%d\\n", (int)scaleTo(inputs[i-1],0.0,1.0,0.0,pow(2.0,MAX_BITS)-1.0));
        if (j%8 == 0)
            fprintf(filePtr,"\\n\\t");
        j+=1;
    }
    j=1;
    fprintf(filePtr,"\\t\\t);\\n");
}
for (i=1; i<= nr_targets; i++)
{
    fprintf(filePtr,"\\n\\nconstant targetTbl%d: tablesim := (\\n\\t",i);
    for (pattern_id = 1; pattern_id <= nr_patterns; pattern_id++)
    {
        pattern_Slist_get_pattern (list_id, pattern_id, inputs, &nr_inputs,
            targets, &nr_targets, &status);
        if (j< nr_patterns)
            fprintf(filePtr,"%d, ",(int)scaleTo(targets[i-1],0.0,1.0,0.0,pow(2.0,MAX_BITS)));
        else
            fprintf(filePtr,"%d\\n", (int)scaleTo(targets[i-1],0.0,1.0,0.0,pow(2.0,MAX_BITS)));
        if (j%8 == 0)
            fprintf(filePtr,"\\n\\t");
        j+=1;
    }
}

```

```

    }
    j=1;
    fprintf(filePtr, "\t\t");
}
fclose(filePtr);
}

void main (Int argc, char *argv[])
/*
ACTION : The main program
INPUT : int      argc -> number of arguments
        char     argv[] -> An array of give arguments
OUTPUT : FUNCTION -> Two vhdI files, one with test and train data,
                the other with the network characteristics
{
    status_St      status;
    Options        option;
    char           *filename;
    pattern_Slist_id_t patterns;
    net_Sid_t      network_id;

    setDefault(&option);
    if (argc >= 1)
    {
        getOptions(argc, &argv, &option);
        filename=(char*)malloc(sizeof(char)*(INTERACT_MAX_CHARS_FILE_NAME+1));
        sprintf(filename, "%s%s", DATA_PWD, argv[1]);
        /* create a workspace for an INTERACT application */
        interact_Sinit(init_mode_Sno_network, "", &status);
        if (option.createSim)
        {
            /* load a pattern database file */
            printf("PatternList Loading !!!\n");
            pattern_Slist_load (filename, &patterns, &status); CheckStatus(status);
            createSimFile(patterns,filename);
        }
        if (option.createTbl)
        {
            /* load a feedforward neural network */
            printf("Network structure loading !!!\n");
            store_Sload_net(filename, store_Scontinue, &network_id, &status);
            CheckStatus(status);
            createTables(&option, filename);
        }
        interact_Sterminate(init_mode_Sstop, &status);
    }
}

```

### C.1.1 An Example

The following VHDL packages are generated by the *InterAct* application shown above. The extraction of the network characteristics do by this application can be found in the VHDL file constants.vhd. This file shows the biases and weights of the network. The example given in this section shows both files, first the characteristics of the network, and secondly the simulation tables. The following description shows the tables of the network parameters, where the synaptic weights are coded by 12 bits and the biases by 24 bits. The neural system can use rounding techniques to create shorter word width. The file genrreated which holds these number is shown below.

```
-----  
--                                     VERSION : 1.0                                     --  
-- FILE   : constant.vhd                                           --  
-- DATE    : may 3rd, 1999                                           --  
-- GENERATED by : Generate                                           --  
-- DESCRIPTION : The biases and weight tables of                     --  
--               a MLP-network.                                       --  
--               /rug103/home2/users/csg/csg890/data/sinustest        --  
--               -----  
--  
library ieee;  
use ieee.std_logic_1164.all;  
package constants is  
  
    type tableBias is array (0 to 6) of integer;  
    constant BiasTbl: tableBias := (  
        539889, -622154, -137814, 3306048, 4176247, -1233781, 2231473  
    );  
  
    constant maxBias : integer := 6;  
  
    type tableWeights is array (0 to 11) of integer;  
    constant weightsTbl: tableWeights := (  
        -557, -326, -926, -880, -2048, -109, -741, -322, -1005, -1180, 1709, -48 );  
  
    constant maxWeight : integer := 11;  
  
    -- These lines are put here by hand, as told in chapter 4.  
    constant Nr_layers : integer := 3;  
    type topology is array (0 to Nr_layers-1) of integer;  
    constant network: topology := ( 1,6,1);  
    constant Nr_Neurons : integer := 8;  
  
end constants;
```

The following VHDL decription shows the simulation tables in this case the translated network has only one input and an output. The application which belongs to these files is the sine function. The simualtion values for the input as well as the target of the system are translated into 14 bit representation, the use of rounding techniques the num-

ber of bit can be made smaller. The following description show us the SimTbl.vhd for the sine function application.

```
-----  
--          VERSION : 1.0                      --  
-- FILE       : SimTbl.vhd                      --  
-- DATE       : may 3rd, 1999                  --  
-- GENERATED by: Generate                      --  
-- DESCRIPTION : The Input and Target signal of --  
--              /rug103/home2/users/csg/csg890/data/sinustest --  
--              in a Max-bit representation 14  --  
-----
```

```
library ieee;  
use ieee.std_logic_1164.all;  
package simtbl is  
  
    type tablesim is array (0 to 50) of integer;  
  
    constant maxpatterns : integer := 50;  
  
    constant inputTbl: tablesim := (  
        0, 334, 668, 1003, 1337, 1671, 2006, 2340,  
        2674, 3009, 3343, 3678, 4012, 4346, 4681, 5015,  
        5349, 5684, 6018, 6352, 6687, 7021, 7356, 7690,  
        8024, 8359, 8693, 9027, 9362, 9696, 10031, 10365,  
        10699, 11034, 11368, 11702, 12037, 12371, 12705, 13040,  
        13374, 13709, 14043, 14377, 14712, 15046, 15380, 15715,  
        16049, 16383, 16383  
    );  
  
    constant targetTbl1: tablesim := (  
        8192, 9030, 9855, 10652, 11409, 12113, 12753, 13318,  
        13799, 14187, 14477, 14664, 14745, 14718, 14584, 14345,  
        14005, 13569, 13045, 12442, 11769, 11036, 10258, 9445,  
        8612, 7771, 6938, 6125, 5347, 4614, 3941, 3338,  
        2814, 2378, 2038, 1799, 1665, 1638, 1719, 1906,  
        2196, 2584, 3065, 3630, 4270, 4974, 5731, 6528, 7353, 8191, 8191  
    );  
  
end Simtbl;
```

## C.2 An Example of a Simulation Environment

The simulations can be done by hand but for automatically performance of the neural hardware system, a simulation environment is developed. This carries out the communication between the internal system by using the IO-ports of the system and the input which are offered to the system(or the systems environment). The input vectors of the file SimTbl.vhd are offer one by ine to the system, and the systems response is written to a file. The following simulation environment is an example of the sine function. This decription change by using other types of networks, especially as the number of inputs and/or outputs are changing.

```
-----
--
--  VERSION: 1.0.1
--  DATE   : 11-04-1999
--  BY     : H.M.G. Ter Haseborg
--
--  DESCRIPTION : The total system in the description chip
--
-----

library ieee;
use ieee.std_logic_1164.all;
library std;
use std.textio.all;

entity sim_environment is
  generic (
    d_busWidth      : integer := 8;
    a_busWidth      : integer := 8;
    tpd              : time    := 0 ns;           -- Time propargation delay
    mem_locations    : integer := 1024;
    d_busNorm        : integer := 3;             -- 2log(d_busWidth)
    act              : integer := 14;            -- width input activation function
    internalWidth    : integer := 16             -- Adder width
  );

  port(
    startS      : in  bit;           -- start the simulation environment
    not_reset   : in  bit;           -- an asynchrone reset
    readyS      : out bit
  );

end sim_environment;

-----

use work.general.all;
use work.utilities.all;
use work.simtbl.all;                -- Table with the input and target patterns
```

```

use work.io_utils.all;           -- File IO sub-routines
architecture struct of sim_environment is

```

---

```

--                               Function Definition
--

```

---

```

function roundoff(Nmax: bit_vector; N : integer) return bit_vector is
variable p      : bit_vector(N-1 downto 0);
variable q,klad: integer;
begin
  for j in 0 to N-1 loop
    p(N-1-j):= Nmax(11-j);
  end loop;
  q := bit2int(Nmax(Nmax'high-N downto 0));
  if q >= (2**(12-N-1)) then
    klad := bit2int(p)+1;
    p := i2bvd(klad,N);
  end if;
  return p;
end roundoff;

```

```

function jamming(Nmax: bit_vector; N : integer) return bit_vector is
variable p: bit_vector(N-1 downto 0);
variable q,klad: integer;
begin
  for j in 0 to N-1 loop
    p(N-1-j):= Nmax(11-j);
  end loop;
  q:=bit2int(Nmax(Nmax'high-N downto 0));
  if q /= 0 then
    p(p'low):='1';
  end if;
  return p;
end jamming;

```

---

```

--                               Definition of components
--

```

---

```

component neuralSystem

```

```

  generic (

```

```

    d_busWidth   : integer := 8;
    a_busWidth   : integer := 8;
    tpd          : time    := 0 ns;
    mem_locations : integer := 1024;
    d_busNorm    : integer := 3;           -- 2log(d_busWidth)
    act          : integer := 10;         -- width input activation function
    internalWidth : integer := 16 );

```

```

port(
    d_in      : In      Std_logic_vector(d_busWidth-1 downto 0);
    d_out     : out     Std_logic_vector(d_busWidth-1 downto 0);
    not_reset : In      bit;                                -- a-synchrone reset
    ce        : In      bit;                                -- chip enable
    start     : In      bit;                                -- start determining neuron
    clk       : In      bit;
    releaseout : In      bit;
    R_W       : out     bit;
    inputRdy  : out     bit;
    ready     : out     bit                                -- determination ready
);
end component;

```

```

--                                     Declaration of signals                                     --

```

```

signal clk      : bit;
signal input    : Integer :=0;
signal address  : Integer :=0;
signal dataInput : std_logic_Vector(13 downto 0);
signal d_in     : std_logic_Vector(d_busWidth-1 downto 0);
alias dataIn    : std_logic_Vector(d_busWidth-1 downto 0) is
    dataInput(dataInput'left downto dataInput'right+(13-d_busWidth));
signal data     : std_logic_Vector(d_busWidth-1 downto 0);

signal datatarget : std_logic_Vector(13 downto 0);
signal d_out     : std_logic_Vector(d_busWidth-1 downto 0);
alias dataout    : std_logic_Vector(d_busWidth-1 downto 0) is
    datatarget(datatarget'left downto datatarget'right+(13-d_busWidth));

signal StartNetwork : bit;
signal inpRdy       : bit;
signal R_W          : bit;
signal ce           : bit;
signal rdy          : bit;
signal memRdy       : bit;
signal release      : bit;

```

```

begin

```

```

u2: neuralSystem

```

```

    generic map ( d_busWidth => d_busWidth, a_busWidth => a_busWidth,
        tpd => tpd, mem_locations => mem_locations,
        d_busNorm => d_busNorm, act => act, internalWidth => internalWidth )

```

```

    port map ( d_in => d_in, d_out => data, not_reset => not_reset,
        ce => release, start => StartNetwork, releaseout => release,
        R_W => R_W, inputRdy => inpRdy , ready => rdy, clk => clk );

```

```

dataInInput <= to_stdLogicVector(i2bvd(inputTbl(Input),14));
dataTarget<= to_stdLogicVector(i2bvd(targetTbl1(Input),14));
d_out      <= dataout;

simulation: process                                -- process for only one system input.
    constant header : string := "-----output file sinus.out -----";
    constant firstLine : string := " input      targeted      output ";
    file output_file : text is out "sinus.out";
    variable stringVec : line;
begin
    write(stringVec, header); writeline(output_file, stringVec);
    write(stringVec, firstLine); writeline(output_file, stringVec);
    wait until StartS = '1';
    for i in 0 to Maxpatterns loop
        -- for the roundoff operator
        -- d_in <= to_stdLogicVector(roundoff(to_bitVector(dataIn), d_busWidth));
        -- for the jamming operator
        -- d_in <= to_stdLogicVector(jamming(to_bitVector(dataIn), d_busWidth));
        d_in <= dataIn;                                -- set input data
        startNetwork <= '1' after 1 ns;                -- start neural hardware
        write_string(stringVec," ");
        write(stringVec, bit2int(to_bitVector(d_in)));
        write_string(stringVec," ");
        write(stringVec, bit2int(to_bitVector(d_out)));
        wait until inpRdy = '1';                        -- wait for ack of the system
        startNetwork <= '0';
        wait until rdy = '1';                            -- wait for calculation rdy;
        release <= '1';
        wait until inpRdy = '1';                        -- wait for ack of the system
        wait until r_w = '1';
        wait until inpRdy = '0';                        -- wait for ack of the system
        write_string(stringVec," ");
        write(stringVec, bit2int(to_bitVector(data)));
        release <= '0';
        input <= input + 1 ;                            -- next input vector
        if input > Maxpatterns+1 then
            input <= 0;
        end if;
        wait until rdy = '0';
        writeline(output_file,stringVec);
    end loop;
    readyS <= '1';
end process;

```

---

```
--                               Clock signal generator                               --
```

---

```
clock : process (clk)
begin
    clk <= not(clk) after 5 ns;
end process;

end struct;
```

### C.3 Generation Tool Discriminatory Function

For the generation of the discriminatory function, a software tool is designed to accomplish that task. This program is written in the C language, and has several properties to influence the result. This tool is cable of producing the discriminatory function in the VHDL language, and a file which holds the same properties as the VHDL-file but is compatible for the MatLab program. All options of the sigmoid function can be changed, so that it makes a good tool for experimental usage. Those options are: the slope of the sigmoid function, the range of effectiveness, and the number of bit coding for the internal activation and external activation. All these parameter can be changed from there default value by using the command line options parameters. The following C code shows the generation tool for the discriminatory function.

```
#include<stdio.h>
#include<math.h>
#define TRUE                (1==1)
#define FALSE               (1==0)
#define INTERNAL_ACTIVATION 10
#define EXTERNAL_ACTIVATION 8
#define SLOPE                1.0
#define LOWER_RANGE          -10.31
#define UPPER_RANGE          10.31
#define VHDL_FILE            "sigmoid.vhd"
#define MATLAB_FILE          "sigmoid.m"

typedef struct {
    long int_act, ext_act;
    float slope, lw_range, up_range;
    int create_matlab;
} Options;

void set_default(Options *option)
{
    option->int_act = INTERNAL_ACTIVATION;
    option->ext_act = EXTERNAL_ACTIVATION;
    option->slope = SLOPE;
    option->lw_range= LOWER_RANGE;
    option->up_range= UPPER_RANGE;
```

```

        option->create_matlab = FALSE;
    }

    void create_matlab_file(Options *option)
    {
        FILE *fileptr;
        Int i;
        Int sigmoid;
        float step,tmp;

        If ((fileptr = fopen(MATLAB_FILE, "w")) == NULL)
        {
            printf("FATAL ERROR -> VHDL file could not be generated!!\n");
            return;
        }

        fprintf(fileptr,"%s-----\n",
        fprintf(fileptr,"%s--                               VER. 1.0 --\n");
        fprintf(fileptr,"%s--                               --\n");
        fprintf(fileptr,"%s-- THIS FILE IS GENERATED BY: sigmoid.exe --\n");
        fprintf(fileptr,"%s-- WITH THE OPTIONS,                --\n");
        fprintf(fileptr,"%s--                               --\n");
        fprintf(fileptr,"%s-- NR OF BITS INTERNAL ACTIVATION : %d --\n",option->int_act);
        fprintf(fileptr,"%s-- EXTERNAL ACTIVATION : %d         --\n",option->ext_act);
        fprintf(fileptr,"%s--                               --\n");
        fprintf(fileptr,"%s-- THE LOWER RANGE : %f             --\n",
            option->lw_range );
        fprintf(fileptr,"%s-- UPPER RANGE : %f                 --\n",
            option->up_range );
        fprintf(fileptr,"%s-- SLOPE : %f                       --\n", option->slope);
        fprintf(fileptr,"%s-----\n");

        step = (option->up_range - option->lw_range) / (pow(2.0,option->int_act));
        for(i = 0; i <= pow(2.0,option->int_act); i++) {
            tmp = (option->lw_range + (i * step) ) * option->slope;
            sigmoid = (Int) ( (( pow( 2.0, option->ext_act ) ) / (1.0 + exp(-tmp) ) ));
            fprintf(fileptr,"y(%d)=%d; ", i + 1, sigmoid);
            fprintf(fileptr,"x(%d)=%d;\n", i + 1,(Int) (option->lw_range + ( i * step)) );
        }
        fprintf(fileptr,"bar(y);\n");
        fprintf(fileptr,"y=./%d\n",(Int)( pow(2.0, option->ext_act) - 1.0 ) );
        fprintf(fileptr,"figure;\n plot(x,y);\n");
        fprintf(fileptr,"\n\nrealx=(%f:%f:%f);\n",option->lw_range, step, option->up_range);
        fprintf(fileptr,"realy=1./(1+exp(-realx*%f));",option->slope);
        fprintf(fileptr,"\ngrid; hold;\n plot(realx,realy,'r');\n");
        fprintf(fileptr,"hold;\n");
        fclose(fileptr);
    }

    void get_settings(Int argc, char **argv[ ], Options *option)
    {
        Int i;

```

```

for(i=2; i<=argc; i++)
{
switch(*argv[i]){
case '-l': case '-L' :
option->lw_range=atof(argv[i+1]);
break;
case '-u': case '-U' :
option->up_range=atof(argv[i+1]);
break;
case '-i': case '-I' :
option->int_act=atol(argv[i+1]);
break;
case '-e': case '-E' :
option->ext_act=atol(argv[i+1]);
break;
case '-s': case '-S' :
option->slope=atof(argv[i+1]);
break;
case '-m': case '-M' :
option->create_matlab=TRUE;
break;
default:
break;
}
}
}

void create_vhdl(Options *option)
{
FILE *fileptr;
long i;
Int sigmoid;
float step, tmp;

If ((fileptr = fopen(VHDL_FILE, "w")) == NULL)
{
printf("FATAL ERROR -> VHDL file could not be generated!!\n");
return;
}

fprintf(fileptr,"-----\n");
fprintf(fileptr,"-- VER. 1.0 --\n");
fprintf(fileptr,"--\n");
fprintf(fileptr,"-- THIS FILE IS GENERATED BY: sigmoid.exe --\n");
fprintf(fileptr,"-- WITH THE OPTIONS, --\n");
fprintf(fileptr,"--\n");
fprintf(fileptr,"-- NR OF BITS INTERNAL ACTIVATION : %d --\n", option->int_act);
fprintf(fileptr,"-- EXTERNAL ACTIVATION : %d --\n", option->ext_act);
fprintf(fileptr,"--\n");
fprintf(fileptr,"-- THE LOWER RANGE : %f --\n", option->lw_range);
}

```

```

fprintf(fileptr,"-- UPPER RANGE : %f                --\n", option->up_range);
fprintf(fileptr,"-- SLOPE : %f                      --\n", option->slope);
fprintf(fileptr,"-----)\n"

fprintf(fileptr,"\n\nuse work_general.all;\n");
fprintf(fileptr,"entity DiscriminatoryFunction is\n");
fprintf(fileptr,"\tport ( input: in bit_vector(%d downto 0);\n",option->int_act-1);
fprintf(fileptr,"      output: out bit_vector(%d downto 0));\n",option->ext_act-1);
fprintf(fileptr,"end DiscriminatoryFunction;\n\n");
fprintf(fileptr,"Architecture Sigmoid of ActivationFunction is\n");

fprintf(fileptr,"type vector is array (0 to %ld) of integer;\n"
            (long)pow(2.0,(option->int_act-1.0)));
fprintf(fileptr,"constant SigmoidTbl: vector := (\n\t");

step=(option->up_range - option->lw_range) / (pow(2.0,option->int_act));
for(i:=0; i <= (long)pow(2.0,(option->int_act-1.0)); i++)
{
    tmp=i*step*option->slope;
    sigmoid=(Int)ceil((pow(2.0, option->ext_act)-1.0) / (1.0 + exp(-tmp)));
    if(i /= pow(2.0,(option->int_act-1.0)))
        fprintf(fileptr,"%d, ",sigmoid);
    else
        fprintf(fileptr,"%d ",sigmoid);
    if(i>9 && i%10 == 0)
        fprintf(fileptr,"\n\t");
}

fprintf(fileptr,");\n");
fprintf(fileptr,"nsignal klad:integer:= 0;\n");
fprintf(fileptr,"signal pre_out:bit_vector(%d downto 0);\n",option->ext_act-1);
fprintf(fileptr,"\n\nbegin\n");
fprintf(fileptr,"\tklad <= bit2int(input(%d downto 0));\n",option->int_act-2);
fprintf(fileptr,"    tpre_out <= i2bvd(SigmoidTbl(klad),%d);\n",option->ext_act);
fprintf(fileptr,"    toutput<=pre_outwheninput(%d)='0'elsenotpre_out;\n",option->int_act-1);
fprintf(fileptr,"end Sigmoid;");
fclose(fileptr);
}

void main (Int argc, char *argv[])
{
    Options option;
    set_default(&option);
    if (argc >=2 )
        get_settings(argc, &argv, &option);
    create_vhdl(&option);
    if (option.create_matlab)
        create_matlab_file(&option);
}

```

### C.3.1 A generated Discriminatory function

This section shows a result of the discriminatory function generation tool. Here the code is viewed of a sigmoid discriminatory function, with the options; a Slope of 1.0, upper range 10.31, lower range -10.31, and the coding happens all in 8 bits.

```
-----
--                               VER. 1.0                               --
--                               --                                     --
--   THIS FILE IS GENERATED BY: sigmoid.exe                           --
--   WITH THE OPTIONS,                                                 --
--                               --                                     --
--   NUMBER OF BITS INTERNAL ACTIVATION : 8                             --
--   EXTERNAL ACTIVATION : 8                                           --
--                               --                                     --
--   THE LOWER RANGE : -10.310000                                       --
--   UPPER RANGE : 10.310000                                           --
--   SLOPE : 1.000000                                                  --
-----

use work.general.all;
entity DiscriminatoryFunction Is
    port ( input    : In      bit_vector(7 downto 0);
          output    : out    bit_vector(7 downto 0));
end DiscriminatoryFunction;

Architecture Sigmoid of DiscriminatoryFunction Is
    type vector Is array (0 to 128) of Integer;
    constant SigmoidTbl: vector := (
        128, 133, 138, 143, 148, 153, 158, 163, 168, 172, 177,
        181, 185, 189, 193, 197, 200, 204, 207, 210, 213,
        216, 218, 221, 223, 225, 228, 229, 231, 233, 235,
        236, 237, 239, 240, 241, 242, 243, 244, 245, 246,
        246, 247, 248, 248, 249, 249, 250, 250, 251, 251,
        251, 252, 252, 252, 252, 253, 253, 253, 253, 253,
        254, 254, 254, 254, 254, 254, 254, 254, 255, 255,
        255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
        255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
        255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
        255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
        255, 255, 255, 255, 255, 255, 255, 255 );

    signal klad:integer:= 0;
    signal pre_out:bit_vector(7 downto 0);

    begin
        klad <= bit2int(input(6 downto 0));
        pre_out <= i2bvd(SigmoidTbl(klad),8);
        output <= pre_out when input(7) = '0' else not pre_out;
    end Sigmoid;
```



# Appendix D

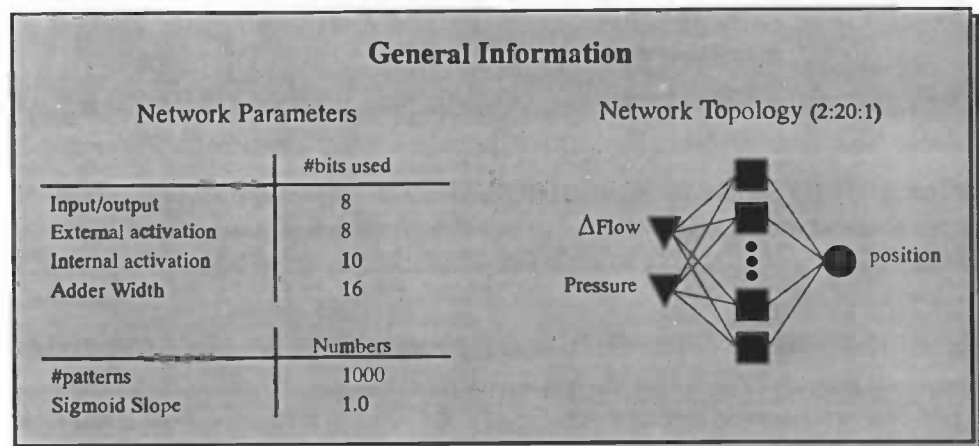
## *The experimental result of the Valve and Iris problem*

### D.1 The experimental Results of the Valve problem

The following experiment uses the results obtained from simulations performed by *InterAct* using the infinite set of numbers, while *V-System* uses a representation of the same network only with a finite set of numbers. The *V-System* simulator is using the neural hardware as described in chapter 4. This experiment uses a trained multi-layer Perceptron network which contains 2 inputs, 20 hidden neurons and an output neuron. Sidebar D-1, shows under which conditions the results are obtained.

#### *Sidebar D-1;*

General information on the Valve experiments. It contains the network parameters and the network topology.

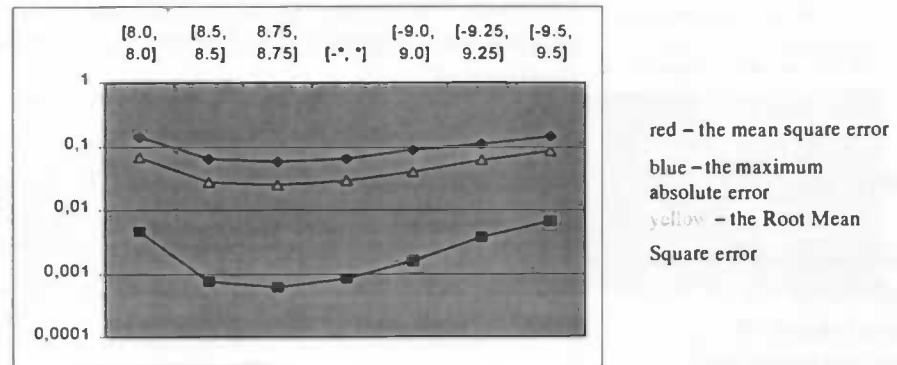


In the following figures the results are presented of the Valve problem. Figure D-1 shows the relationship between the two system one with a finite and the one with a infinite number representation, whereby several area's of effectiveness are chosen for the system that uses the finite set of numbers. For the presentation of the performance, errors are calculated so that both systems can be compared. The referencing network is the system which uses the infinite set of numbers.

**Figure D-1;**

This graph shows the relationship between the system which uses finite and infinite number representations. Whereby the finite system, uses several effective intervals for the sigmoid function.

**The errors between infinite and finite system**  
For several ranges of the sigmoid's effectiveness.

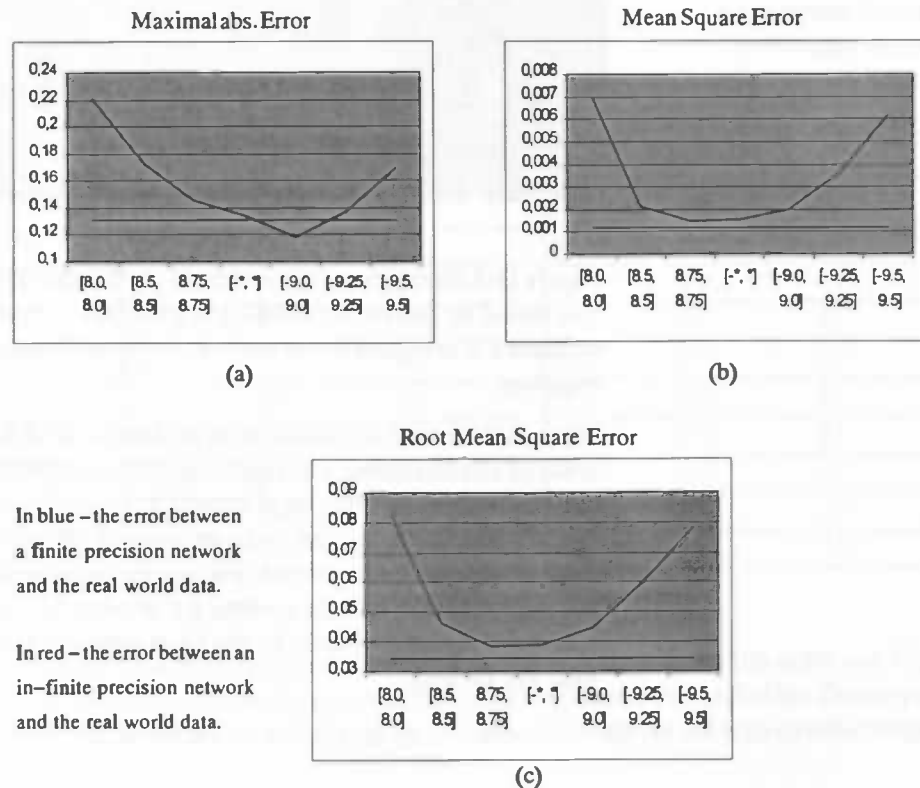


Another view of the performance is shown in figure D-2. Here the performance of both systems is compared with the real Valve data: the infinite system (red) which is constant, while the other system (blue) uses different areas of effectiveness from the sigmoid function. The presentation of these performances shows three types of error; the maximal absolute error, the mean square error (MSE), and the root mean square error (RMS). The real valve data will be used as the reference, such that these errors can be calculated.

**The performance of both system in regard to the real valve data**  
For several ranges of the sigmoid's effectiveness.

**Figure D-2;**

The performance of both system in regard to the real valve data is shown here, under the constrained values for b. Figure a shows the max.abs.error of both systems, (b) and (c) respectively the mean square error and the RMS error.



## D.2 Iris classifier results

This classifier uses four input neurons to support our neural network with the data, so that the following Iris plants can be classified: Iris Setosa, Iris Versicolour, and Iris Virginica. That mean that the network has for each Iris class an output. Eight hidden neurons are configured in such a way that the network response in *InterAct* has a performance of:

**Table D-1;**

The table shows the performance of a trained neural network with infinite precision, done by *InterAct*.

	Iris Setosa	Iris Versicolour	Iris Virginica
Max. Abs Error	775.43 E-3	824.26 E-3	495.75 E-3
MSE	5.42 E-3	14.21 E-3	15.236 E-3
RMS error	73.62 E-3	119.20 E-3	123.43 E-3

The following results are obtain from the simulation performed by using the digital neural hardware system. The properties of the system are shown in Sidebar D-2.

**Sidebar D-2;**

General information on the Iris experiments. It contains the network parameters and the network topology.

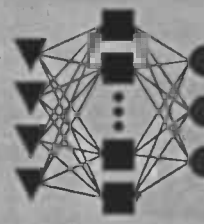
General Information			
Network Parameters		Network Topology (4:8:3)	
	#bits used		
Input/output	8	sepal length	
External activation	8	sepal width	
Internal activation	10	petal length	
Adder Width	16	petal width	
	Numbers		
#patterns	118		
Sigmoid Slope	1.0		

Figure D-3 shows the performance of the digital network in relation to the one trained and tested by *InterAct*. Secondly it provides us with the information to see what the influence is in regard to our main goal, the influence of a limited range of the sigmoid function.

Table D-2 shows the classification performance of both system, with respect to several areas of effectiveness. The rejection criteria is that the difference between the winner and the runner-up must be at least 0.5. The infinite systems misclassify only one of the 118 input patterns, while large areas of effectiveness ( $> [-10.0, 10.0]$ ) the finite system misclassify 2 patterns. The number of classified patterns and the rejected ones is constant for the infinite system, it classifies 107 and rejects 11. The number of classified and rejected patterns by the finite system changes when other areas of effectiveness are chosen.

# The performance of the finite system in comparison to infinite system For several ranges of the sigmoid's effectiveness.

Figure D-3;

For each class of Iris the performance is show in relation with other intervals for the sigmoid representation. As referencing system is taken the infinite system.

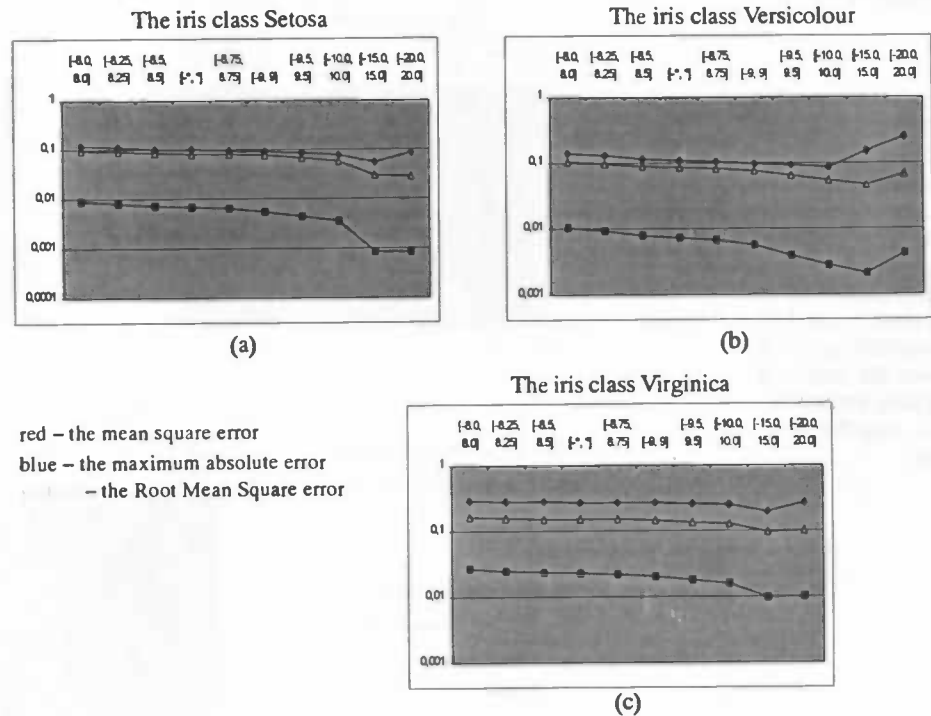


Table D-2;

The classifying performance of the finite and infinite system. The total numbers of patterns are 118. The used rejection criteria is that the difference between the winner and the runner-up must be at least 0.5.

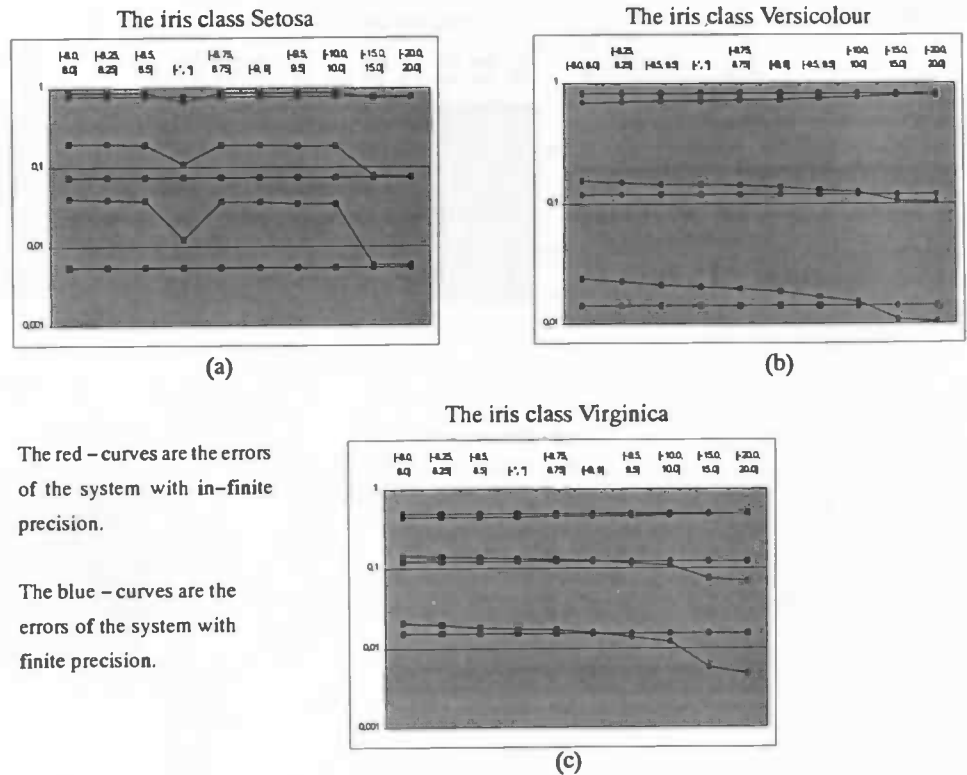
	Infinite System			Finite System		
	classified	rejected	misclassified	classified	rejected	misclassified
[-8.0, 8.0]	107	11	1	73	44	1
[-8.25, 8.25]	107	11	1	80	37	1
[-8.5, 8.5]	107	11	1	89	28	1
[-*, *]	107	11	1	90	27	1
[-8.75, 8.75]	107	11	1	93	24	1
[-9.0, 9.0]	107	11	1	98	19	1
[-9.5, 9.5]	107	11	1	102	15	1
[-10.0, 10.0]	107	11	1	104	13	1
[-15.0, 15.0]	107	11	1	112	4	2
[-20.0, 20.0]	107	11	1	113	3	2

The last result is the response of both systems compared with the Iris database. Figure D-4 shows for each class the errors of the infinite system in red and of the finite system in blue. These errors are calculated by different intervals for the area of effectiveness.

**The performance of both systems compared to the real world**  
For several ranges of the sigmoid's effectiveness.

**Figure D-4;**

This view shows the performance of both systems (infinite and finite) with regard to the real Iris data. Each graph shows the error which can be calculated, as the real data is used as referencing point. It shows the maximal absolute error, the MSE, and the RMS error.



The red - curves are the errors of the system with in-finite precision.

The blue - curves are the errors of the system with finite precision.

NOTE : The upper lines are the maximal absolute errors, the middle is the RMS error, and the lowest lines represents the MSE.