GLOBAL OPTIMIZATION VIA
NEURAL NETWORKS AND D.C. PROGRAMMING

by

Nicolas Roddier

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY
WITH A MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1994

Abstract

The ultimate goal of this work is to provide a general global optimization method. Due to the difficulty of the problem, the complete task is divided into several sections that can be summarized as a modeling phase followed by a global minimization phase. Each of the various sections draws from different engineering fields. What this work suggests is an interface and common grounds between these.

The modeling phase of the procedure consists of converting a general problem into a given formulation using a particular type of neural network. The architecture required for the neural network forms a new class: the pseudo multilayer neural network. It is introduced and compared to more classical neural network architectures such as the regular multilayer neural network. However, a key difference between these is that the behavior of the usual multilayer network has to be programmed, while an extremely efficient procedure is given here to synthesize the pseudo multilayer neural network. Therefore any initial problem can be systematically converted into a pseudo multilayer network without going through the undesired programming steps such as the backpropagation rule.

The second phase of the work consists of translating the initial global optimization problem into the global minimization of a target function related to the neural network model. Systematic procedures are again given here.

The last phase consists of globally minimizing the target function. This is done via the so-called DC programming technique where DC stands for "Difference of Convex". The pseudo multilayer was created such that it can systematically be converted into a DC formulation, and therefore be compatible with DC programming. A translation procedure to

go from the pseudo multilayer neural network model to the DC formulation is given. When a DC program is applied to this last formulation, the resulting solution can be directly mapped to the global minimum of the target function previously defined, thereby producing the global optimal solution of the neural network modeling the initial problem. Therefore, the optimal solution of the original problem is known as well.

Chapter 1: Introduction

It has been said that equality has been a driving concept for mankind for centuries: equality of every citizen, equality of rights, equality of opportunity, and others. If there has been so much attention toward reaching equality, it is probably because nature by itself tends to provide inequality. In fact inequalities are omnipresent, from abstract ideas to concrete measurements. Life is based and surrounded by inequalities and measures which define an order.

The inequality is always modified to introduce competition. For example, consider a very simple inequality: $a-b<0$. This equation is automatically translated into $a<b$, and immediately interpreted as "a is less than b", or "a is not as good as b", or "b is better than a", or "between a and b, b is the better, or the best". When introducing another inequality $b<c$, b becomes "not as good as c", while it was "best" in the previous case. c happens to become the "very best". It is clear that a situation x (x may be an idea, a fact, or a data) can be compared only to its environment, and depending on this environment the same x can be called good, bad, better, or best.

It occurs everywhere all the time that people want to find the "very best" among a set of various circumstances. The cook looks toward finding the best recipe to prepare some dishes, the politician tries best to convince the people about his program, the teacher organizes his lectures so that students understand best, the army general positions best his army to fight the enemy, the manager integrates his employees in a manner that makes the company the most productive. In more theoretical areas, the chemist finds the best components to perform a given reaction, the optician uses filters to achieve best desired interferences, the mathematician wants to find the best method to solve some equations. In short, everywhere, it is desired to find the best of something. The field of study for finding

the best is called the field of optimization. Where is the difficulty of this field? There are several problems, from the difficulty of defining the goals to having to deal with trade-offs. If there are such difficulties, how do people search for the "best"?

A "best" is only such in a given environment. It is therefore clear that the process which finds it must perfectly know where to search. The search domain consists of every possible situation in an environment which the process has full knowledge of. However, is having complete knowledge of the environment a sufficient condition for finding the "best" within the search domain. The knowledge of the search domain and the problem itself enable to finding the "best". However, there are problems so complex that this solution can't be found without an infinite amount of time using infinite resources. What happens in these situations?

Following Descartes' remarks, according to whom intelligent people separate complex problems into several simpler ones, each of them being solvable, a problem that wouldn't be directly solvable might be solvable after the decomposition.

In different engineering fields, people have recently imagined that they could create machines that would be able to solve problems humans can't. An application is for example the field of optimization: finding the "best".

These suggestions have often been considered unrealistic, and the little current progress seems to confirm this. A machine seems unable to perform something a human can't do by himself. If the human can do it, the machine can be programmed to do it as well, quicker, and in larger quantities, but trying to have the machine do something humans can't is going in a wrong direction.

Therefore, if it is desired for a machine to perform an optimization task, then it must follow a route that has before been proven successful to humans. The machine can improve by the fact that it is able to process faster than humans and in larger quantities.

Such a cartesian approach is suggested in this dissertation. There exist optimization algorithms that are successful in finding solutions for some particular problems formulated in a certain manner. Similarly there exist techniques to transform a problem into one such formulation. Hence, the task is to create an interface between the two. A given problem is transformed in such a manner that it becomes amenable to the optimization algorithm.

To be specific, in this project, a problem is modeled, and the model is optimized. The process of modeling is performed by a neural network, and the optimization is performed by DC programming.

First, the modeling section is presented. Chapter 2 is a general review of neural networks, and of function approximations in particular. Chapter 3 presents an introduction to a class of neural networks: the pseudo multilayer feedforward networks, along a method to synthesize them. This creates the desired model.

Then, the optimization section is presented. Chapter 4 is a review of global optimization techniques, in particular deterministic methods. Chapter 5 presents an implementation of DC programming that allows to solve problems formulated in the manner previously mentioned.

Finally, the two sections are interfaced together. Chapter 6 is the actual implementation of the modeling followed by optimization: A given problem is first modeled by a pseudo multilayer network the output of which is in turn minimized using DC programming. At last, some concrete example are presented.

Chapter 2:
Review of Neural Networks

In this chapter neural networks in general are reviewed, along more specifically those used for continuous mapping and function approximations. These are the kinds of networks that are later used for interfacing with the global optimization algorithms. As it was stated in the introduction, the neural network performs the modeling phase, that precedes the optimization phase.

2.1 General information

Before dealing with neural networks, it is imperative to first describe a neuron by itself. What exactly is a neuron? It must be made clear that there are two major classes of neurons: the biological neurons and the engineering neurons, also called artificial neurons.

The brain of any animal is made up of a huge quantity of very similar cells, interconnected in a very extensive and complex manner. Each of these cells is called a biological neuron [Block, 1964]. They each have a very simple behavior, but due to the large number and the interconnections, what is called intelligence can exist. The number of neurons in a human brain is about 10^{11}.

The cell is composed of several parts. An aquisition section consists of receiving signals from neighbor neurons. These signals are electrical pulses. They are accumulated in another area until a threshold level is reached. This storage area clearly acts like an electrical capacitor. Once the threshold level is reached, the neuron discharges itself and "fires". This means that all the stored energy is released to the environment: to all

surrounding neurons in the neighborhood. At this point there is no longer any energy left in the neuron [Cotterill, 1988].

What makes a neuron different from its neighbor is the way it is connected. During what was just described as aquisition and firing phases, the quantity of travelling energy is not identical in every direction. Therefore, for a given neuron that fires, all its neighbors receive a different amount of energy, which implies that they in turn fire at different times since they don't recharge all at the same rate.

This was the description of a biological neuron. The artificial neuron is a mathematical model of what was just described, to be used by engineers [Koch, 1989]. The connections are modeled by resistors or matrix weights, the threshold firing process is modeled by a nonlinear activation function, and the output propagation is modeled again by matrix weights. In the coming section, complete descriptions of artificial neurons are given. From now on the biological neuron is no longer mentioned and what is refered as 'neuron' indicates an artificial neuron. Now that the very basic of neurons have been presented, what is a neural network?

The simplest definition of a neural network is a "set of neurons combined in a particular manner". It is therefore clear that both the neurons themselves and the connections determine the network response. A neuron can be modeled as an extremely simple multi input single output (MISO) subsystem.

There are several different models for neurons, which all have common characteristics. The general formulation for a neuron input-output behavior is provided by the following equation: [Wasserman, 1989]

$$y = f(x) \quad \text{and} \quad \dot{x} = -\alpha x + \underline{w} \cdot \underline{u} \tag{2.1}$$

where $\underline{u}$ is the input vector, $\underline{w}$ the input weight vector, y the scalar output, x the scalar state, $f(.)$ the so-called non-linear activation function, and α the self feedback connection. With no special assumption on any of its parameters, this neuron is called a dynamic neuron, or a dynamic node of a neural net. Its block diagram representation is shown in Figure 2.1.

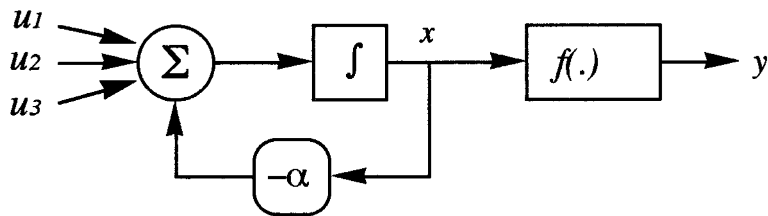

Figure 2.1. A model for a dynamic neuron.

The Σ symbol doesn't stand for the simple sum of the inputs u_i, but for a sum of the inputs u_i associated with weights w_i. Some restrictions may be applied which make the model described by equation (2.1) to become a member of another class: the static class instead of the previous dynamic class. Considering now only a stable system (i.e. $\alpha>0$) and its steady state, it is clear that the input-output relationship of the neuron becomes:

$$y = f(x) = f\left(\underline{w} \cdot \underline{u} / \alpha \right) \tag{2.2}$$

This model is called a static neuron, or a static node of a neural net [Simpson, 1990]. The input vector is multiplied (dot product) by the weight vector to produce a state that in turn is passed through the non-linear activation function to produce the output. A block diagram is shown in figure 2.2:

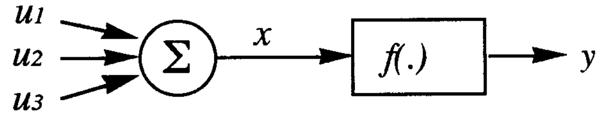

Figure 2.2: A model for a static neuron

The two models just presented define the two major classes of neurons: static and dynamic.

The activation function is the only non-linear element in a neuron, independent of its type [DARPA, 1988]. It is a scalar real-valued function of a scalar real-valued argument:

$$f(.):\quad \Re \rightarrow \Re; \quad x \rightarrow f(x) \tag{2.3}$$

There are several kinds of these functions but once again, most of the time, they have common characteristics and properties. These are:

- Odd symmetry about the $(0,f(0))$ point: $f(-x) = 2f(0)-f(x)$

- Monotonicity: $\forall x \in \Re, f'(x) \geq 0$

-Saturation: $f(\infty) = f(0) + Sat$ and $f(-\infty) = f(0) - Sat$

Among the functions mostly used are: threshold function, piece-wise linear function, sigmoid type function. The analytical formula for a sigmoid type function can be, but is not limited to *Atan(x), Tanh(x),* and $1/(1+e^{-x})$. A few of these activation functions are plotted in Figure 2.3:

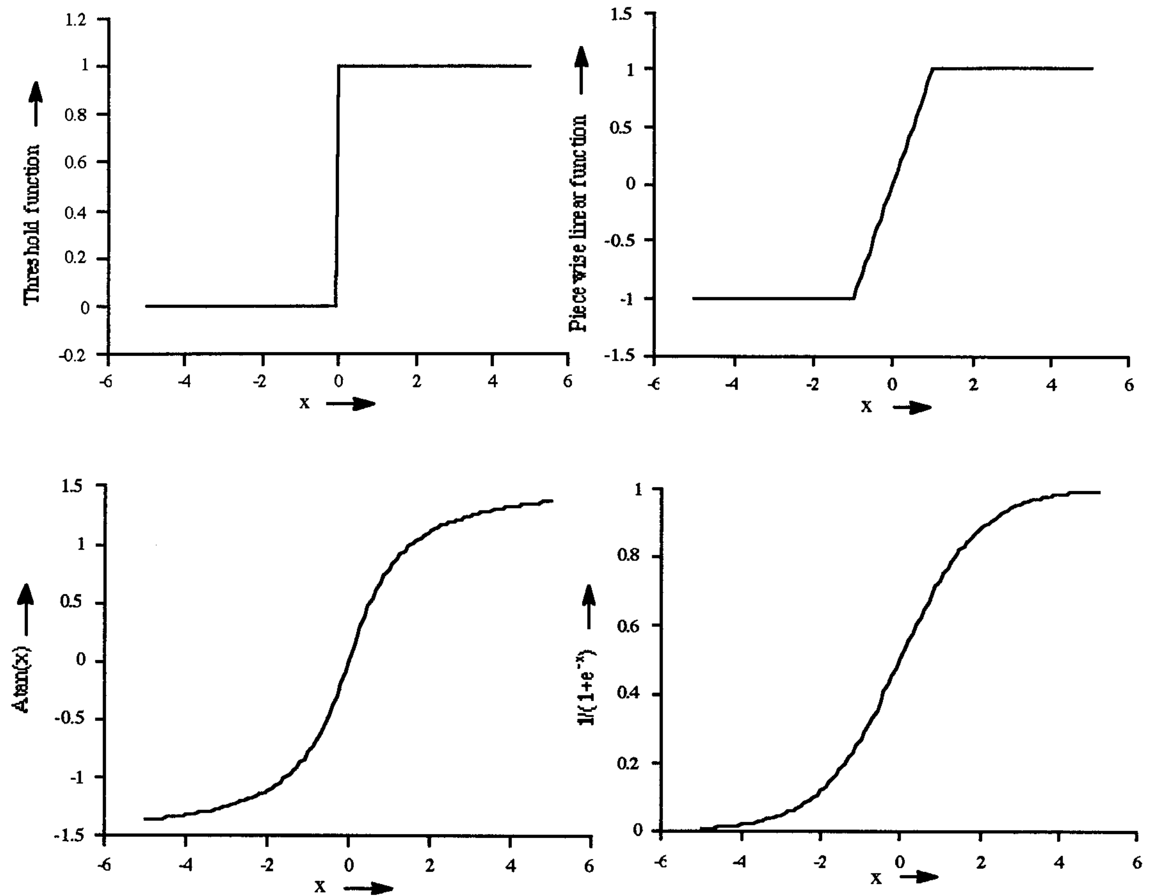

Figure 2.3: Plots of several common neuron activation functions.

In addition to the characteristics and properties given in (2.3), the following statements hold in addition:

- Convexity for a negative argument: $f''(x) \geq 0$ for $x < 0$
- Concavity for a positive argument: $f''(x) \leq 0$ for $x > 0$

These last two properties play a critical role for the optimization analysis presented later in this project.

Names have been given to some particular neuron configurations. For example, a static neuron with a threshold function is called a perceptron [Rosenblatt, 1962]. As it was stated earlier, a set of interconnected neurons is called a neural network. Similarly to the neurons themselves, there are several types of neural network architectures. The most common of these are feedforward and recurrent networks.

In a feedforward network, all the connections between the different neurons must be laid in a forward manner, no feedback is allowed. Most of the time this network is composed of static nodes. However, in some special cases, such as for example to produce delays, it may be desired to include a dynamic node. In that case, the self feedback may be allowed. It is therefore clear that there is no stability problem for a feedforward network.

In the special case that all the neurons in the network have their inputs connected only to system external inputs and not to the output of other neurons, the neural network is called a single layer net.

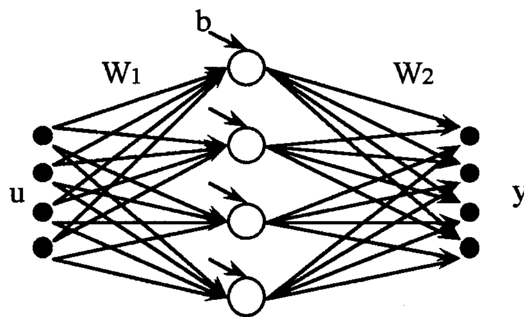

Figure 2.4: A 4-neuron, 4-input, 4-output single layer neural network

If neurons are assembled into subsets such that every neuron from a given subset has its input connected to the output of neurons of a subsequent subset, then the neural network is called a multilayer feedforward network. Each subset is called a layer.

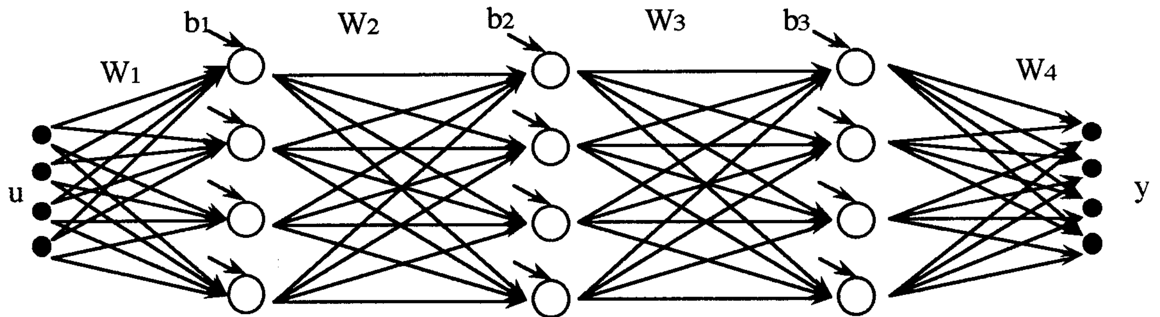

Figure 2.5: A 12-neuron, 4-input, 4-output 3-layer neural network

Later in this project a "pseudo-multilayer" architecture will be introduced and its characteristics, properties and advantages presented in due course.

Contrary to feedforward networks, the recurrent networks have feedback connections. In order for them to work it is clear that any possible closed loop from any starting point must include at least one dynamic node. Otherwise for the same node at the same instant, its ending value would be different from the initial one, which is clearly an error. Similarly to the feedforward networks, elements of the recurrent networks are allowed several types of activation functions.

The recurrent neural networks are used for modeling dynamical systems, and any phenomenon that requires "memory". However, in this project this kind of network is not used because only static and memoryless systems are considered.

Before completing the presentation on neural networks in general, it is necessary to quickly mention yet another class. It is a feedforward network in terms of "instantaneous behavior" and overall architecture, but it has quite different rules: the counter-propagation network [Hecht-Nielsen, 1987b]. It looks like the single layer network of figure 2.4, with all the neurons having a perceptron type activation function. Furthermore, the bias vector (which controls the threshold) is actively controlled so that at every instant one and only one neuron is active.

It is clearly a pattern classifier network [Hecht-Nielsen, 1990]. There are as many neurons as there are patterns to be classified. They are positioned by the input weight matrix in an equidistant manner in the space so that the network information is optimized and any noise perturbation minimized. The output weight matrix is a direct mapping of the desired output for each situation.

When a noisy pattern is presented to the input, its distance toward all the valid patterns is computed (one by every neuron) and the smallest one is selected (by means of the active bias control). Only the corresponding neuron is fired which produces the noise free pattern on the output.

The main drawback of counter-propagation networks is that the number of neurons becomes extremely large, even for small systems. Another disadvantage is that a master must look after all the neurons all the time in order to actively control the bias level. This is clearly in contradiction with a fully parallel architecture which is the basic structure of neural networks.

A very important consideration in neural network theory is the question of programming the network. There are quite many ways to look at the problem and its implementation. Among the various studies, [Montana, 1989], [Green, 1989], [Holland, 1975], and [Werbos, 1988] can be mentioned. However, this list is far from being a complete review of all the various ways for programming a neural network since there are

virtually dozens of such studies. It can be viewed as an optimization problem where parameters of the network such as weights and biases are optimized so that the difference between the desired behavior of the network with its actual behavior is minimized.

There are several different approaches for the programming task. Because the problem is fairly complex, the minimization does not have a single solution. In other words, the optimization finds minima among which only one is the desired global minimum. Therefore, the parameters can't just be updated using a steepest descent algorithm because depending on the starting point they would reach various local minima, and probably never the desired global minimum. Over the years, several methods have been developed that are usually successful.

The most popular one is the so-called back-propagation [Rumelhart, 1986]. Although it is globally a gradient descent algorithm, it has some hill-climbing capabilities. However, it has several problems. On a theoretical aspect, even though it is able to occasionally climb, it has no guarantee to always converge toward the global minimum. It may be trapped in a local node. Only a trial and error approach with different starting points can drive to a desired solution. On a practical point of view, it is extremely slow to run, and furthermore the speed depends on some gains and parameters that must be carefully chosen, but for which there is no general rule. If the gain is chosen too small, the convergence takes forever, if it is chosen too large, the overall stability of the algorithm is at risk.

Assuming a multilayer neural network of the type depicted in figure 2.5. It is described by the following set of equations:

$$\underline{x_1} = W_1 \cdot \underline{u} + \underline{b_1}$$
$$\underline{x_2} = W_2 \cdot f(\underline{x_1}) + \underline{b_2}$$
$$\underline{x_3} = W_3 \cdot f(\underline{x_2}) + \underline{b_3} \qquad (2.4)$$
$$\underline{y} = W_4 \cdot f(\underline{x_3})$$

where the various *Ws* and *bs* are weights and biases as shown in the figure.

During the programing process a set of desired (u, y_{des}) pairs is given and the algorithm must figure out the best *Ws* and *Bs* so that the network behavior (u, y_{act}) matches the desired (u, y_{des}).

Backpropagation is an iterative algorithm with a two step process at each iteration. A forward pass computes the overall error signal $y_{des} - y_{act}$. A backward pass updates the parameters for every layer. It starts by the last one, and propagates the correction back toward the input layer. The gradient of the activation function is used for backpropagating the error signal. At a given iteration the network behavior (u, y_{act}) for a given input $\underline{u}$ is computed using equation (2.3).

Then the overall error signal $y_{des} - y_{act}$, and the backpropagated error signals for each layer are computed as:

$$error_4 = y_{des} - y_{act}$$
$$error_3 = W_4 * error_4 * \frac{\partial f}{\partial x}(x_3)$$
$$error_2 = W_3 * error_3 * \frac{\partial f}{\partial x}(x_2) \qquad (2.5)$$
$$error_1 = W_2 * error_2 * \frac{\partial f}{\partial x}(x_1)$$

Finally, the parameters of every layer are updated following the rule:

$$\begin{aligned} W_1 &= W_1 + g \cdot error_1 \cdot u \\ W_2 &= W_2 + g \cdot error_2 \cdot x_1 \\ W_3 &= W_3 + g \cdot error_3 \cdot x_2 \\ W_4 &= W_4 + g \cdot error_4 \cdot x_3 \end{aligned} \tag{2.6}$$

where g is the gain to be carefully adjusted.

Proceeding in this manner makes the actual behavior of the network progressively match its desired one. Notice that the bias vectors are combined with the weight vectors: a bias is equivalent to a weighted unit signal.

This was an overwiew of the backpropagation algorithm. For practical implementation, and improvements, further reading is available [Korn, 1991].

To this date, backpropagation is the most popular algorithm for programming a neural network in spite of its lack of global convergence and its speed. There doesn't exist a universal algorithm that guarantees global convergence to the "best" solution.

2.1.2 Continuous function approximation

There are many applications for neural networks. Among the most popular ones are speech and pattern recognition [Grossberg, 1968] [Naylor, 1988], speech processing [Sejnowski, 1987], associative memory [Kohonen, 1989], robotics [Brooks, 1986], and system identification [Narendra, 1990]. In this project the facet of neural networks of interest is function approximation [Lippmann, 1987].

A function approximation can be interpreted as an expansion of a given arbitrary function into a set of other functions. For example the Fourier approximation of a function is its expansion in terms of $Sin(nx)$ and $Cos(nx)$ where n corresponds to various harmonics. A Fourier series can be interpreted as a single layer neural network where the

activation function is a Cosine instead of the regular sigmoid shaped activation function. The first layer weights are the harmonic coefficients and the biases correspond to the phases that match the *Cos/Sin* proportions. The second layer weights are nothing but the Fourier coefficients. From the Fourier theory, a periodic function $y=f(x)$ can be decomposed with no error using an infinite number of terms, or approximated with some truncation error using N terms as:

$$\begin{aligned} y &= y_0 + \sum_{i=1}^{\infty} a_i Cos(i\omega x) + b_i Sin(i\omega x) \\ y &\approx y_0 + \sum_{i=1}^{N} a_i Cos(i\omega x) + b_i Sin(i\omega x) \\ y &\approx y_0 + \sum_{i=1}^{N} c_i Cos(i\omega x + \varphi_i) \end{aligned} \tag{2.7}$$

where

$$\begin{aligned} c_i &= \sqrt{a_i^2 + b_i^2} \\ \varphi_i &= -Tan^{-1}\left({b_i}/{a_i} \right) \end{aligned} \tag{2.7a}$$

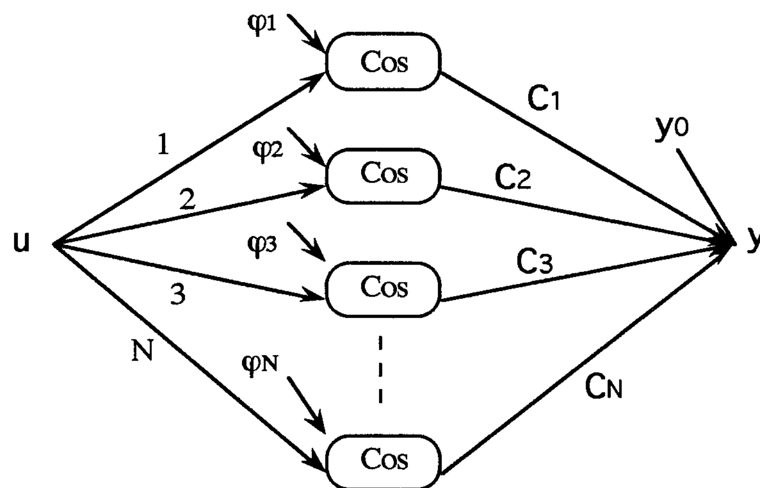

Figure 2.6: A Fourier Series represented as a single layer neural network

As stated earlier, the activation functions usually are not allowed to be Cosine shaped. The main reason being that they must be monotonic and go to saturation at both ends. This is why the particular function decomposition shown in Figure 2.6 is not of great interest for neural network studies.

There have been several sets of basic functions used for approximation with neural nets. From the Fourier example, it follows that the basic functions are used as activation functions for the neurons. Among the most popular of these, in addition to usual sigmoid shaped functions, radial basis functions must be mentioned, as well as hyper-ridge functions [Flacke, 1993]. However, the later are fairly close to the sigmoid type even though they are not monotonic for the simple reason that a hyper-ridge function is nothing but the difference between two threshold functions. At this point it must be noticed that both feedforward networks and recurrent networks can be used for the purpose of function approximation. However, throughout this project, only feedforward networks are considered for the reason stated before that only static and memoryless systems are considered.

To conclude this introduction on function approximation by neural networks, several important background works must be mentioned. The whole function approximation theory started when Hecht-Nielsen related it to the Kolmogoroff theory [Hecht-Nielsen, 1987a]. At that point the existence and feasibility of the function approximation was proven: For a given continuous function, it is always prossible to find a neural network that approximates this function with as much accuracy as desired. Several works then proceeded in order to refine those statements, and provided some methods for the construction of the expansion. Among the numerous researchers, Koiran worked on the complexity of the problem [Koiran, 1993]; Cybenko [Cybenko, 1989], Funahashi [Funahashi, 1989], and Ito [Ito, 1991] worked on construction techniques using sigmoidal shaped activation functions; and Hornik [Hornik, 1989] [Hornik, 1990]

showed that the original function to be approximated need not be continuous. Funahashi's and Hornik's works will be described in detail in due course since their work is closely related to some aspects of this project.

Chapter 3: Neural Networks as Multi Input Single Output Function Approximators

This chapter is the description of the study on the approximation of a wide class of MISO (Multi Input Single Output) functions by a neural network. First theoretical background to support the feasibility of this model is presented, then an object oriented approach to build the corresponding neural network is introduced. In other words, considering a MISO function from a particular class described later, there exists a neural network for approximating it; a few theorems that prove its existence are recalled. However, even though the network exists, its construction may be very difficult in practice. The purpose of the object oriented approach is to make it easier.

The interest of this study has several aspects. First, using neural networks for function approximations has already been studied, and as it is shown in a comparative study near the end of this chapter, it seems that the here suggested approach presents several advantages. A second reason is that the approximating neural network output can be 'minimized' using the procedure described in the chapter 4, thereby 'minimizing' or 'optimizing' the initial target function. A last interest of approximating functions by a neural network lies in the computer architecture area: Such a neural network is particularly well suited for the so popular new RISC (Reduced Instruction Set Computer) architecture.

3.1. Theoretical background.

3.1.1 SISO case.

This study begins with a look at neural networks as SISO (single input single output) function approximators. Considering a single layer network with N neurons, where the activation function is piecewise linear, and where each neuron is allowed a bias input, the system equation that describes this network can be written as,

$$y = y_0 + \sum_{i=1}^{N} c_i f(x_i) = y_0 + \sum_{i=1}^{N} c_i f(a_i u + b_i) \tag{3.1}$$

where u is the single network input, y the single network output, x_i the state of the ith neuron, a_i the input weight of the ith neuron, b_i the bias of the ith neuron, c_i the output weight of the ith neuron, and y_0 the output bias. In matrix notation, the system equation becomes

$$y = y_0 + \underline{c}^T \cdot f(\underline{a}u + \underline{b}) \tag{3.2}$$

where $\underline{a}$, $\underline{b}$, and $\underline{c}$ are vectors of components a_i, b_i, c_i and respectively. The piecewise linear activation function $f(x)$, figure 3.1, is defined as,

$$\begin{cases} f(x) = 0, & \text{for } x \le 0 \\ f(x) = x, & \text{for } 0 < x < 1 \\ f(x) = 1, & \text{for } x \ge 1 \end{cases} \tag{3.3}$$

Note that this activation function, on the contrary to the usual activation functions, does not pass through the origin and is not odd. However, without loss of generality it could be shifted as desired along both axis since this transformation would occur only on the linear part of the network. Using this function simplifies the equations and algorithms presented later.

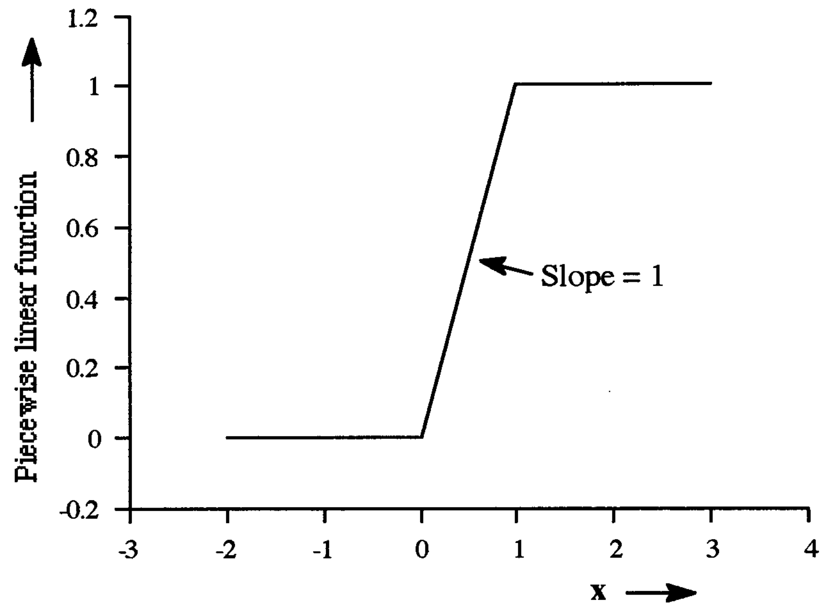

Figure 3.1: The piecewise linear activation function $f(x)$.

This neural network is depicted in the following figure 3.2:

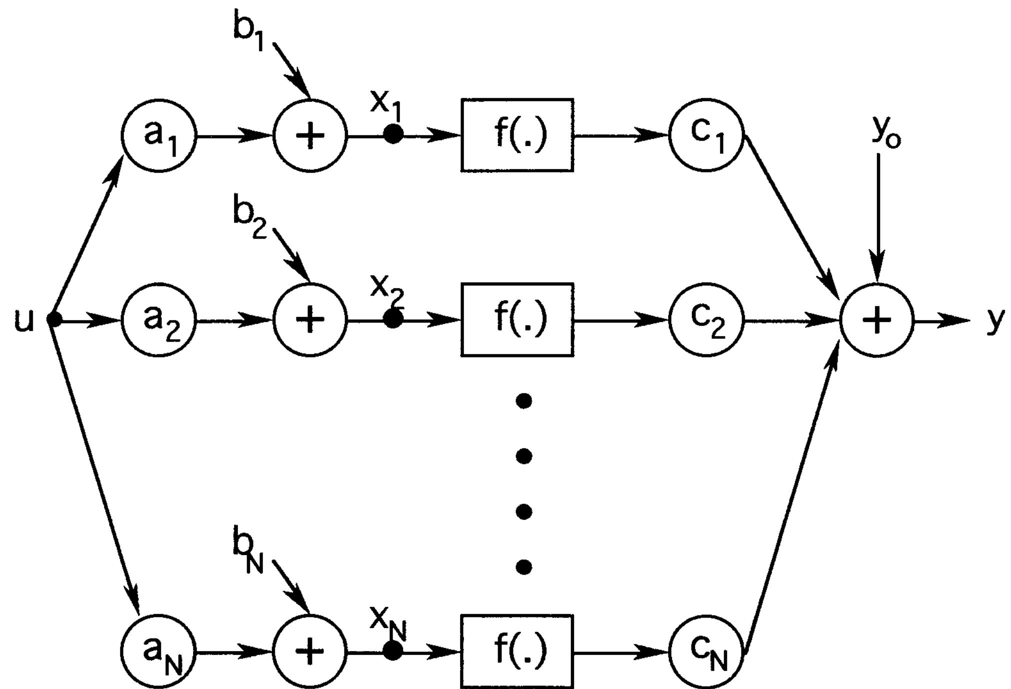

Figure 3.2: Single hidden layer SISO neural network.

In the remainder of this section, it is shown that this kind of network can approximate a wide class of SISO functions. Furthermore, an algorithm to find the required values of the coefficients a_i, b_i, and c_i for performing the correct approximation is introduced.

A SISO function $y = g(u)$ is considered, where both u and y are scalar real variables. Furthermore, g is assumed to be continuous over a compact domain $u \in [\lambda_1, \lambda_2]$, and the 1st derivative exists and is continuous everywhere over that domain.

Theorem 3.1: Considering a function *g(u)* that follows these assumptions, then for every point $u_0 \in [\lambda_1, \lambda_2]$, and for every tolerance τ, there exists a real number ε, and there exists an affine function $w(u) = l_1 u + l_2$ with constraints $w(u_0) = g(u_0)$ and

$w(u_0+\varepsilon)=g(u_0+\varepsilon)$ so that over the sub-domain $[u_0,u_0+\varepsilon]$ the difference between the original function and the affine function is less than τ.

Mathematically, it can be written:

$$\begin{aligned}&\forall u_0 \in [\lambda_1,\lambda_2], \forall \tau \in \Re,\\ &\exists \varepsilon \in \Re, \exists w(u) = l_1 u + l_2,\\ &\text{where} \quad w(u_0)=g(u_0), w(u_0+\varepsilon)=g(u_0+\varepsilon)\\ &\text{so that } \forall u_1 \in [u_0,u_0+\varepsilon], \|g(u_1)-w(u_1)\| < \tau\end{aligned} \tag{3.4}$$

<u>Proof 3.1:</u> Because it is required that $w(u_0)=g(u_0)$ and that $w(u_0+\varepsilon)=g(u_0+\varepsilon)$ the two parameters l_1 and l_2 are actually directly related to ε, and there really is only one degree of freedom: ε. In other words, it is known that for all ε, the original function and the affine function will intersect at both u_0 and $u_0+\varepsilon$. Since one and only one straight line can pass through two distinct given points, for every ε there is therefore only one possible $w(u)$. Figure 3.3 illustrates these relations.

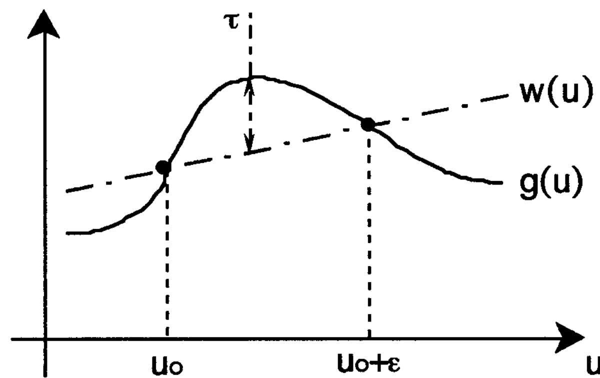

Figure 3.3: Illustration of the relation between *g(u)* and *w(u)* for a given u_0 and ε.

Therefore it can be noticed that the theorem is equivalent to saying that a function from C_1 can always be locally approximated by a piece a straight line, and that is a well known result the proof of which can easily be found in math books. This concludes the proof 3.1.

It is along these lines that a single layer neural network for approximating any SISO function that belongs to C_1 over a finite domain $\left[\lambda_1,\lambda_2\right]$ can be constructed: The neural network intersects the original function at a number of points, and has pieces of straight line in between. Each piece corresponds to a neuron being activated.

At this point another function $\Psi(\bullet)$ needs to be introduced. It is used in the algorithm for constructing the neural network. In the theorem above, the existence of ε and of the corresponding $w(u)$ were stated. However, these are certainly not unique. The purpose of this new function $\Psi(\bullet)$ is to find among all the possible $\{\varepsilon, w(u)\}$ the set of most interest: the largest ε, and the corresponding $w(u)$. The larger the value of ε, the larger the range covered by each neuron. Using the notation above, for every u_0, this new function returns the largest ε such that an affine function the distance of which from the original function is less than τ indeed exists:

$$\varepsilon = \Psi(g(u), u_0, \tau) \tag{3.5}$$

There is no major difficulty coding this function $\Psi(\bullet)$ in a computer program. The procedure is to increase ε, compute the corresponding $w(u)$ and check the distance between $w(u)$ and $g(u)$. As long as this distance is less than τ, ε is kept increasing. Once this distance is equal to τ, the corresponding ε is assumed to be the desired solution.

At this point it can be questioned how good this simple algorithm is.

Some computations can be performed on $g(u)$ and $w(u)$. A 2nd order Taylor series of these can be written as:

$$g(u) = g(u_0) + (u - u_0)g'(u_0) + \frac{(u-u_0)^2}{2} g''(u_0) + O^3 \tag{3.6a}$$

$$\begin{aligned} w(u) &= \frac{g(u_0+\varepsilon) - g(u_0)}{\varepsilon}(u - u_0) + g(u_0) \\ &= \frac{g(u_0) + \varepsilon g'(u_0) + \frac{\varepsilon^2}{2} g''(u_0) - g(u_0)}{\varepsilon}(u - u_0) + g(u_0) + O^3 \\ &= g'(u_0)(u - u_0) + \frac{\varepsilon}{2} g''(u_0)(u - u_0) + g(u_0) + O^3 \end{aligned} \tag{3.6b}$$

The distance between $g(u)$ and $w(u)$ is now computed. Notice that the common terms $g(u_0) + (u - u_0)g'(u_0)$ in both equations cancel out. The equation becomes:

$$\begin{aligned} \|g(u) - w(u)\| &= \left\| \frac{(u-u_0)^2}{2} g''(u_0) - \frac{\varepsilon}{2} g''(u_0)(u - u_0) + O^3 \right\| \\ &= \left\| g''(u_0)\left(\frac{(u-u_0)^2}{2} - \frac{\varepsilon(u-u_0)}{2} \right) + O^3 \right\| \\ &= \left\| g''(u_0)\left(\frac{(u-u_0)}{2} \right)\left((u-u_0) - \varepsilon \right) + O^3 \right\| \end{aligned} \tag{3.7}$$

Note that $0 \le \|u - u_0\| \le \varepsilon$. Therefore the distance can be bounded by:

$$\|g(u) - w(u)\| \le \left\| g''(u_0)\left(\frac{\varepsilon}{2}\right)(\varepsilon) \right\| + \|O^3\| \le \left\| \frac{\varepsilon^2}{2} g''(u_0) \right\| + \|O^3\| \tag{3.8}$$

Functions for which higher order terms can be neglected are first considered. Under such an assumption, the maximum distance is bounded by $\left\| \frac{\varepsilon^2}{2} g''(u_0) \right\|$. It is clear that this distance increases monotonically with ε. This confirms that the simple algorithm that slowly increases the value of ε indeed computes its correct value: there can't be a larger ε that would produce a smaller distance than the one that is found.

Considering functions for which higher order terms cannot necessarily be neglected, it can be realized that the algorithm may sometimes fail. A function for which $\|O^3\|$ is large for small ε, but decreases as ε increases is considered. At that point it can be seen that the bound on the distance doesn't necessarily increase with ε since a decreasing term and an increasing one are summed up. This situation can be also described on figure 3.4 where it happens that as ε is increased, the distance between the function and the affine line becomes larger than τ, but if ε were to be increased further the distance would decrease and become again smaller than the tolerance. The algorithm produces the value ε_1 while the value ε_2 is the desired value. It is clearly seen in this example that the function *g(u)* has locally a strong cubic coefficient which is the same as saying that $\|O^3\|$ can't be neglected.

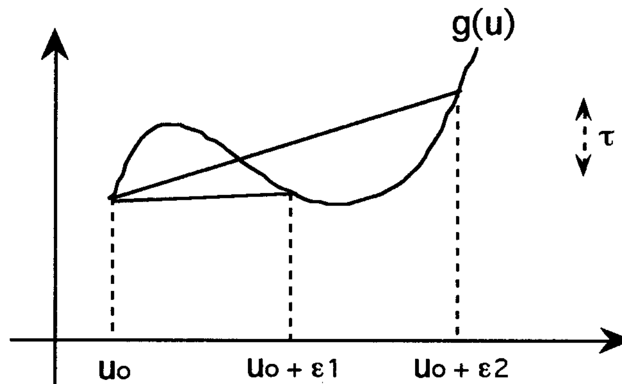

Figure 3.4, example where the 'get epsilon' algorithm would fail.

It can be noticed that for most functions the higher order terms can indeed be neglected, and therefore the cases where the algorithm fails are very rare. In addition it has to be realized that even when the algorithm fails, this has no major consequences. The only thing that happens is that two neurons are required while in theory one would have

been sufficient. Since a typical neural network deals with overall several hundred or thousand neurons, one more or one less certainly shouldn't be much of a worry.

The code for this algorithm is shown in the Appendix. It is called `GetEps()`. The input arguments are a pointer to the function *g(u)*, a real value for u_0, and a real value for τ. It returns a real value for ε.

All the required elements have been presented and therefore an algorithm for constructing a network of the type of equation (3.1) for approximating a SISO function from C_1, with a tolerance τ can be introduced. Usually such a construction is performed using algorithms for programming single hidden layer neural networks such as the so-called backpropagation rule. However, the algorithm about to be presented is extremely simple to implement, and perfectly adapted to this application: the input vector is a set of continuous values. The backpropagation rule requires any arbitrary set of values; with this algorithm that continuity fact is made use of, and the overall programming is therefore more efficient.

Algorithm for approximating a SISO function *y=g(u)* over an interval $[\lambda_1, \lambda_2]$.

Step 1. The output bias y_0 is set equal to $g(\lambda_1)$, $N=0$, $u_{work} = \lambda_1$

Step 2. Find $\varepsilon = \Psi(g(u), u_{work}, \tau)$

Step 3. $N=N+1$: A neuron is added to the network for that sub-domain.

Step 4-1. $a_N = 1/\varepsilon$

Step 4-2. $b_N = -u_{work} \cdot a_N$

Step 4-3. $c_N = g(u_{work} + \varepsilon) - g(u_{work})$

Step 5. $u_{work} = u_{work} + \varepsilon$

Step 6. If $u_{work} \geq \lambda_2$, exit, the construction is finished, N neurons have been programmed. The weights and biases a_i, b_i, and c_i are known. Otherwise, go back to Step 2 and continue putting more neurons.

The code for this algorithm is shown in the Appendix. The routine is called `ProgSISO()`. Input arguments are a pointer to the *g(u)* function, real values for τ, λ_1, and λ_2, and pointers to the a_i,b_i, and c_i arrays. It fills up the arrays and returns the values y_0 for the output bias and N for the number of neurons required by the network.

Here is an example of approximation over a domain of a SISO function by a single layer neural network using this implementation.

The approximation of the function $y = g(u) = 2u + Cos(2\pi u)$ over the domain $[-0.2,1.2]$ is required. The maximum tolerance is set to $\tau = 10^{-3}$. After completion, the programming requires 80 neurons. The approximation is then checked by computing both the original function and the neural network approximation, and plotting them against one another. In between the bounds, the result is indeed excellent. Outside the bounds, no neurons are active, i.e. they are either saturated or at rest, and therefore the network approximation is constant. The code for this example is in the Appendix, it is called `ChkSISO.c`.

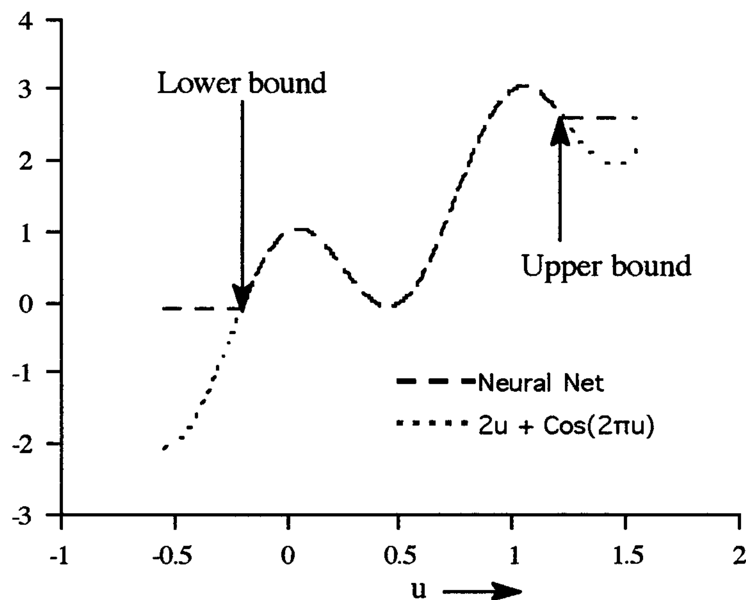

Figure 3.5, Example of approximation of a SISO function by a neural network.

It has just be shown that any SISO function that belongs to C_1 over a finite domain can be approximated by a single layer network with as much accuracy as desired. The more accurate the model, the larger the number of required neurons. Similarly, if the domain is increased, the number of neurons has to be increased if the overall accuracy has to stay the same. Furthermore, an algorithm for programming this single layer network was presented, i.e. for the approximation of a given function, a set of values for a_i, b_i, and c_i was found.

3.1.2 MISO case.

3.1.2.a Previous approaches.

It was shown in the previous section how a SISO function can be modeled or approximated with a single layer neural network. This coming section summarizes a similar study, except that now the input is a vector instead of being a scalar. Therefore, the function is a so-called multi input function, or MISO function. There are theorems which state that MISO functions belonging to some classes can be approximated by neural networks. However, construction techniques similar in some sense to the ones described in the previous section for SISO functions are unfortunately not as explicit.

Some results have been reported relating to the existence of such models. The next paragraph summarizes two different approaches for approximating a MISO function by a neural network.

First, Ken-Ichi Funahashi considers a continuous (for each variable) MISO function being a continuous mapping from the n-dimensional space $\Re^n$ to the single

dimensional space $\Re$. He shows that this mapping can be approximated by a single layer neural network over a finite compact subset of $\Re^n$ with as much accuracy as desired.

However, this work considers only the existence of the realization, and a couple of problems arise. The first problem is related to the topology of the network. Because all the layers are on the same single layer, they are all acting in a similar manner. This produces a rather inefficient realization where the number of required neurons becomes extremely large. The second problem is related to the programming. Indeed, there is no systematic and efficient way to program the network. Once again backpropagation may have to be called.

In contrast to Funahashi's work, K. Hornik et al. show that this approximation can be realized using a neural network with several neuron layers (multilayer) rather than with a single layer. Proceeding in this manner improves tremendously the efficiency of the approximation. The number of required neurons drops by orders of magnitude compared to Funahashi's architecture. There is no longer a huge single layer, but several smaller ones instead. However, it must be noticed that an accurate topology of this multilayer network is unknown. The multilayer feedforward network is not a simple extension of the single layer network, it is not just a "scale-up". There is currently no definite approach to optimize, or even determine the overall number of required neurons, and worse, there is no rule to determine the number of hidden layers, and the size of each of these layers.

With respect to programming the network, the problem has unfortunately gone even worse than it was in the previous case. As stated before, it is difficult to program a single layer network. It can be stated now that it is much worse to program a multilayer network since it is extremely difficult to observe, understand, and control effects of parameters associated with neurons that lie on deep hidden layers.

In the coming section, an alternate approach for programming a neural network for approximating a MISO function is presented. From the previous results, it is clear that having several layers is advantageous in terms of efficiency. However, it must be kept in mind that programming must still be feasible. What is suggested here is a multilayer network where each layer is actually directly related to a SISO function, for which a systematic programming algorithm was presented. This is a trade-off where the multilayer efficiency is preserved and the programming is kept within firm grounds. The cost for that trade-off is that the MISO function to be approximated must have some particular characteristics. In other words, it will not be possible for all MISO functions to be approximated using this scheme, but only for a class of them. However, this class includes almost all the functions that are "used" in engineering applications.

3.1.2.b The unary$^+$ decomposable functions.

This section starts with the recall and the introduction of a few terms.

A 'unary' function is a real scalar function of a real scalar variable. It is a mapping from $\Re$ to $\Re$. It actually is what was called so far a SISO function. For example *Cos(x)* is a 'unary' function.

A 'binary' function is a real scalar function of a pair of real scalar variables. It is a mapping from $\Re \times \Re$ to $\Re$. It is a MISO function where there are 2 inputs. For example *Cos(x+y)* is a 'binary' function.

The term 'unary$^+$ decomposable' function is now introduced:.

Definition 3.1: A unary$^+$ decomposable function is an n-input MISO function (from $\Re^n$ to $\Re$) that can be decomposed into a set of unary functions interconnected in a particular manner: The elements of this set can be cascaded one to the next one, and their

outputs can only be combined by linear combinations to feed another input. In addition, only feedforward is allowed, which is equivalent to saying that absolutely no signal can go back through feedback.

A mathematical representation of unary+ decomposable functions is the following:

$y = f(u_1, u_2, u_3, \dots, u_n)$ is a unary+ decomposable function

$\Leftrightarrow$

$\exists\{\tilde{f}_1(.), \tilde{f}_2(.), \dots \tilde{f}_p(.)\}$ a set of p unary functions

$\exists\{x_1, x_2 \dots x_k\}$ a set of k state variables

$\exists\{\varsigma\}, \exists\{\gamma\}, \exists\{\delta\}, y_0$ all real coefficients

such that

$$y = y_0 + \sum_{i=1}^{k} \varsigma_i x_i \quad \text{and} \quad x_i = \sum_{j=1}^{i-1} \gamma_{ij} \tilde{f}_{*_1}(x_j) + \sum_{j=1}^{n} \delta_{ij} \tilde{f}_{*_2}(u_j)$$

Even though this definition seems somewhat restrictive, these unary+ decomposable functions are very general, and almost all the MISO functions used by engineers are actually of that type.

Notice that constants can always be entered using a constant $\tilde{f}_*(.)$ function.

It can also be noticed that any function that can be decomposed into unary+ decomposable blocks, interconnected only by linear combination, is also a unary+ decomposable function, i.e. the set of unary+ decomposable functions is closed under decomposition.

As an example, the unary+ decomposition of a very fancy MISO function (3-input) is shown. The function is

$$f(u_1, u_2, u_3) = u_1 + \sqrt{Log(Cos(u_3)) + Tan\left(u_1 + \sqrt{10^{u_1} + u_2}\right)} \tag{3.9}$$

This particular function has probably no interest whatsoever except to show how a rather complex mathematical expression can be decomposed into unary functions, interconnected by the binary operator 'addition'. At this point it can be noticed that for this application a 'binary operator' or 'binary function' are absolutely equivalent terms. From now on, it will be refered to multiplication or addition as 'binary functions'.

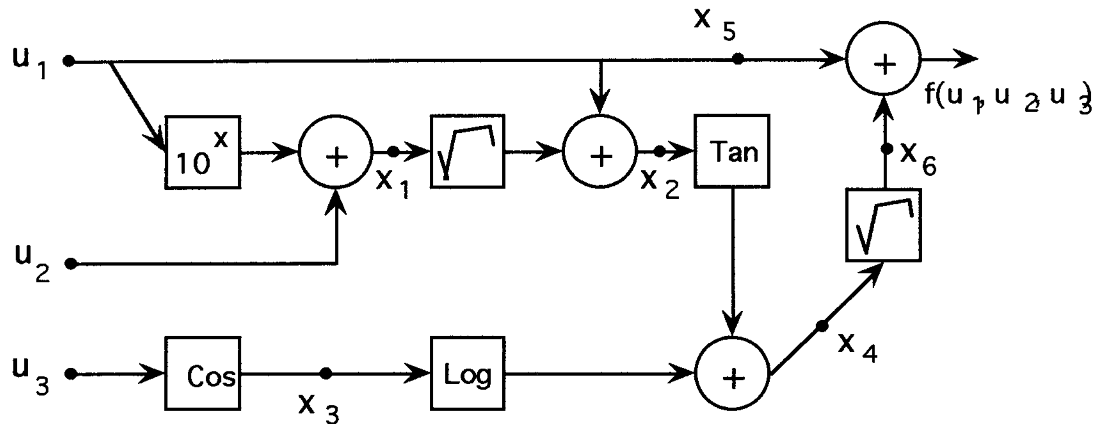

Figure 3.6, example of the unary+ decomposition of a MISO (3ISO) function.

The nomenclature of the definition 3.1 above can be illustrated after this example. Six unary functions are introduced. Using the notation above, they are:

$$\tilde{f}_1(x) = Cos(x),\ \tilde{f}_2(x) = Tan(x),\ \tilde{f}_3(x) = Log(x)$$
$$\tilde{f}_4(x) = 10^x,\ \tilde{f}_5(x) = \sqrt{x}, \text{ and } \tilde{f}_0(x) = x \tag{3.10}$$

Six state variables x_1, x_2, x_3, x_4, x_5, and x_6 as shown in the figure (3.6) are also created. The system equation can be written as:

$$\begin{aligned} x_1 &= \tilde{f}_4(u_1) + \tilde{f}_0(u_2) \\ x_2 &= \tilde{f}_5(x_1) + \tilde{f}_0(u_1) \\ x_3 &= \tilde{f}_1(u_3) \\ x_4 &= \tilde{f}_2(x_2) + \tilde{f}_3(x_3) \\ x_5 &= \tilde{f}_0(u_1) \\ x_6 &= \tilde{f}_5(x_4) \end{aligned} \tag{3.11}$$

It is straight forward to check that $y = f(u_1,u_2,u_3) = x_5 + x_6$

The decomposition of a unary$^+$ decomposable function using the correct procedure was just presented. Notice that the dependence of any state variable x_i to another state variable x_j occurs indeed only for $j < i$ as required by the definition.

As a verbal argument to show that most functions used by engineers are indeed unary$^+$ decomposable, a small HP calculator is considered. It happens that functions that are realizable using an HP calculator are unary$^+$ decomposable. It is pretty unusual to be stuck with a function that cannot be realizable with an HP. It is therefore very unusual to be stuck with a function that is not unary$^+$ decomposable!

It is fairly straightforward to show that an HP computes only unary$^+$ decomposable functions. The HP works with reverse polish notation, which means that data are stored into the stack, operations affect only the first register, or the first two registers. Functions using only the first register such as *cos(x), sqrt(x), or 1/x* are clearly unary functions. Functions using the first two registers are addition, subtraction, and a few others. Addition is the basic binary element, subtraction is nothing but an addition where the second argument has been applied the unary function 'times -1', and the other few binary functions such as multiplication, division, and y^x (*y* to the *x*) are unary$^+$ decomposable too, as it will be shown in due course. Therefore, any function realizable on an HP calculator is indeed unary$^+$ decomposable.

To conclude this section on unary$^+$ decomposable functions, one has to be fair and show a few examples of functions that are not unary$^+$ decomposable. These are basically from 3 different classes. The first class includes functions that can't be computed as a one step process, such as a function that involves an integral operator. The second class includes the non-continuous functions. However, these are rarely used to describe a real life engineering process. Lastly, the third class includes the functions that are given through a tabulation rather than through an analytic expression. Such a tabulated SISO function is no problem, but for such a tabulated MISO function, it is very difficult to perform the decomposition into smaller SISO blocks.

3.1.2.c The 'pseudo multilayer' neural network.

A new feedforward-only neural network architecture is now introduced and defined: it is called the 'pseudo multilayer' topology. It has first to be recalled that a regular multilayer network is a set of neurons that are grouped into subsets, the layers, where connections occur only from one layer to the next.

Definition 3.2: A 'pseudo multilayer' network is a multilayer network with the usual grouping configuration, i.e. the 'layers', and 2 additional properties:

1. Each layer is actually a SISO single hidden layer network.

2. A finite number of discrete direct connections, or connections by means of linear combinations, are allowed between the input and output element (there's only one of each per layer since they are SISO layers) of layers that are not necessarily consecutive.

The next figure (3.7) is a representation of such a 'pseudo multilayer' network. For the simplicity of the drawing, neither the linear weights between layers, nor the input or

output weights and biases of the SISO stages are shown. However, every connection is weighted by a real constant. This example is a 4 input, single output network with a pseudo multilayer architecture. It is clear how to locate the single layer sub-networks, along the linear combinations of their outputs to be fed into the input of another stage, not necessarily the very next one, contrary to regular multilayer networks.

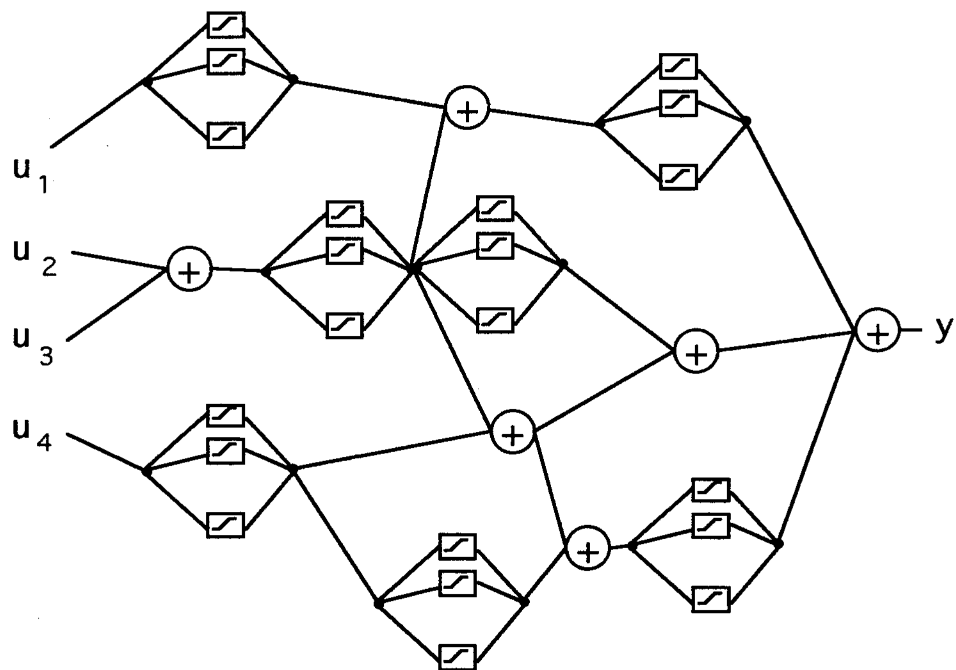

Figure 3.7, example of a 'pseudo multilayer' neural network.

The mathematical representation of a pseudo multilayer network can now be considered. Similarly to every other network, state variables can be assigned to different elements of the network. Two distinct types of state variables are provided. A "macro" state variable, and a "micro" state variable. The macro state variables are used to describe the interconnections among the inputs and the SISO sub networks that constitute the

network; this section of the network is linear. Notice that there must be 2 kinds of macro state variables: those that correspond to the inputs of the sub networks and those that correspond to their outputs. Each of these SISO sub networks can be described on a smaller scale by means of the micro state variables. This description includes the nonlinear elements.

The following assigments are made: the inputs are called u_i, the output y, the micro state variables x_i, the macro state variables corresponding to the SISO inputs ξ_i, and the macro state variables corresponding to the SISO outputs ζ_i. The following statements can be written:

Macro state variables: each input macro variable is a linear combination of the outputs of previous stages, along with the network inputs:

$$\underline{\xi} = M_u \cdot \underline{u} + M_\zeta \cdot \underline{\zeta} \tag{3.12}$$

Notice that the M_ζ matrix is a stricktly lower triangular square matrix: no feedback is allowed, not even self feedback.

The next step is to look at the kth subnetwork: $SISO_k$. Its output macro state variable is ζ_k and its sole input is ξ_k. The description of a SISO single layer network, equations (3.1) and (3.2), that was presented at the very beginning of this chapter can therefore be used here:

$$\zeta_k = \zeta_{k_0} + \underline{c_k}^T \cdot f(\underline{x_k}) \tag{3.13}$$

where $\underline{x_k}$ is a state vector corresponding to the micro state variables for the description of the kth subnetwork $SISO_k$. The micro state variables are connected to the SISO input as:

$$\underline{x_k} = \underline{a_k} \cdot \xi_k + \underline{b_k} \tag{3.14}$$

The global output is a linear combination of the macro states with weights Ω:

Output description: $y = y_0 + \Omega^T \cdot \underline{\zeta}$ (3.15)

At this point the pseudo multilayer network can be summarized by the following set of equations:

$$\begin{cases} \underline{\xi} = M_u \cdot \underline{u} + M_\zeta \cdot \underline{\zeta} \\ \underline{x} = A \cdot \underline{\xi} + \underline{b} \\ \underline{\zeta} = \underline{\zeta}_0 + C^T \cdot f(\underline{x}) \\ y = y_0 + \Omega^T \cdot \underline{\zeta} \end{cases} \tag{3.16}$$

There is now a single vector of micro state variables $\underline{x}$ which is the vertical concatenation of all the previous $\underline{x_k}$ vectors, each of them being the micro state variable vector of the $SISO_k$ sub network. A and C are now block diagonal matrices where each block k is the previous vector $\underline{a_k}$ or $\underline{c_k}$. Similarly to $\underline{x}$, the $\underline{b}$ vector is constructed as the vertical concatenation of the previous individual $\underline{b_k}$ vectors.

3.1.2.d The neural network for MISO approximation.

At this point the 2 concepts introduced earlier need to be related to each other: the 'pseudo multilayer' feedforward neural network, and the 'unary+ decomposable' functions. The following theorem can be stated:

Theorem 3.2: Every unary^{+} decomposable function can be approximated with as much accuracy as desired over a finite compact domain by a pseudo multilayer feedforward neural network. The complexity of the function determines the required number of pseudo layers (i.e. SISO blocks) in the network, and the required accuracy determines the size of these layers.

Proof 3.2: In order to prove this theorem, it is necessary to have a look at the definition of a unary^{+} decomposable function. The required unary functions can be directly mapped into the SISO sub networks of the pseudo multilayer network. Similarly, the required function state variables can be directly mapped into the input macro state variables of the pseudo multilayer network.

Depending on the complexity of the original function, its unary^{+} decomposition includes more or less unary functions, hence the network includes more or less pseudo layers.

Depending on the required resolution, each of the SISO blocks is programmed differently, hence determining the number of neurons in each pseudo layer. This concludes the proof 3.2.

A unary^{+} decomposable function which is decomposed according to the definition (3.1) can be immediately converted into a pseudo multilayer network using the mapping that was just presented. The state variable mapping is quite simple. The mapping for the unary functions involves the conversion of a unary function into a single hidden layer neural network: A method for systematically programming such a network with as much accuracy as desired was introduced earlier. It is therefore suggested to follow this route here.

It was shown so far that a wide class of MISO functions can be approximated by a neural network with as much accuracy as desired. It was also shown how it is possible to actually construct the corresponding network.

3.2. Object oriented approach for building the desired network.

In the previous section he existence of the approximation of a MISO function by a neural network, and a method for constructing the network were shown. However, even if this method works well, it is sometimes cumbersome to use; for example it is still necessary to program single layer subnetworks for approximating the unary functions. The purpose of this section is to show an implementation using an object oriented methodology of that construction method in order to perform the network synthesis more easily and efficiently.

3.2.1 Objects and SISO functions.

What is really desired is to avoid the programming phase when it is required to convert a unary function into a single layer subnetwork. The alternative that is suggested here is to have a list of very basic unary functions that are already preprogrammed, from which more complex functions are constructed. By means of further combinations, every possible unary function can eventually be assembled.

What was said in the SISO section is that every unary function could be converted into a subnetwork using the algorithm presented. What is now said is that these functions must first be decomposed into simpler unary functions that are chosen from a given list. This list contains functions that were programmed once for all using the

algorithm presented, and the result was stored for further use. Proceeding in this manner allows to 'synthesize' single layer subnetworks, rather than 'program' them.

These preprogrammed subnetworks are called 'objects' and are chosen from what is called the 'basic element list'.

The next figure (3.8) presents the whole picture of the process of decomposition of a unary$^+$ MISO function not only into subnetworks, but also into elements from the basic element list. Once that decomposition has been performed, a pseudo multilayer network for approximating that function can be directly synthesized instead of being programmed.

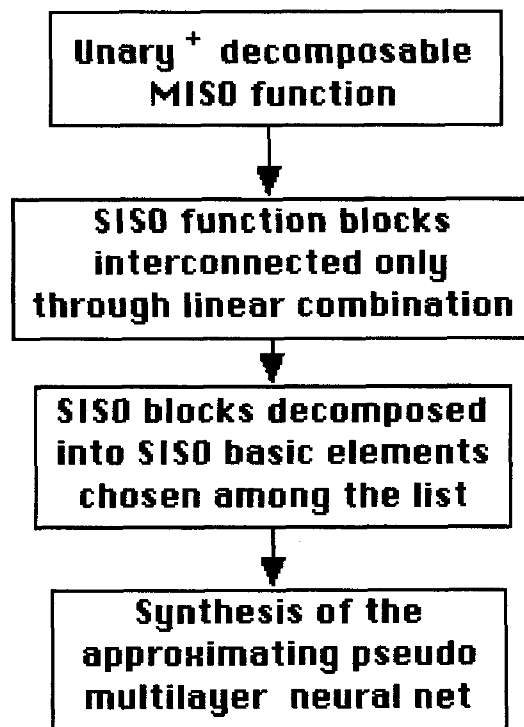

Figure 3.8, Complete decomposition of a MISO function.

More specification and information of the 'basic element list' is now needed. How should these elements be selected? They certainly need to be basic enough so that other SISO functions can be decomposed into them. They also need to be the functions that are used most often. The operations that are already by default directly possible through the network architecture without any 'list' are the addition and multiplication by a scalar. These are linear operations and are performed on what was defined earlier as the macro state variables, for which the network is linear. The list of non-linear basic elements needs to be constructed.

Due to lack of real and strong guidelines, it is clear that the list is certainly not unique. It requires a minimum of about half a dozen elements, but could include many more. These non-required elements could possibly be decomposed into the required elements, but are however desirable to include if they are used often.

A few SISO basic functions are required. There are two families: 'trigonometric' and 'exponential'.

With respect to the 'exponential' family, 2 basic elements are needed: the natural logarithm *Ln(x),* and the natural exponential function e^x. Considering these basic elements along with already legal linear operators, the following SISO functions can be constructed:

$$
\begin{aligned}
x^2 &= e^{2Ln(x)} \\
\sqrt{x} &= e^{Ln(x)/2} \\
1/x &= e^{-Ln(x)} \\
x^p &= e^{pLn(x)} \\
10^x &= e^{xLn(10)} \\
Log_{10}(x) &= Ln(x)/Ln(10)
\end{aligned}
\qquad (3.17)
$$

Similarly, the whole 'hyperbolic' family can be constructed as well. It includes functions such as *Cosh(x), Sinh(x), Tanh(x),* and their inverses.

In addition, it is possible to construct the very important binary multiplication operator, *, and the binary division operator, /, according to:

$$x * y = e^{(Ln(x)+Ln(y))} \quad \text{and} \quad \frac{x}{y} = e^{(Ln(x)-Ln(y))} \tag{3.18}$$

However, at that point it must be noticed that the *Ln(x)* function is defined only for positive numbers, and therefore the multiplication by a negative number would be erroneous. A better alternative for computing multiplication of 2 numbers is the following: $x \cdot y = \frac{1}{2}\left((x+y)^2 - x^2 - y^2\right)$. Notice that now the multiplication operator is based around the 'square' function, which itself in turn is built around the exponential family. It is therefore needed to find a way to program the square function independently from the exponential. There will be more details on this particular problem later.

After the 'exponential' family comes the 'trigonometric' family. Assuming that an approximation of *Cos(x)* is known, the approximation for *Sin(x)* is readily available. The formula $Sin(x) = Cos\left(\pi/2 - x\right)$ can be used. The 'other' well known relation $Cos^2(x) + Sin^2(x) = 1$ could have been used instead, but a sign uncertainty would have appeared in $Sin(x) = \sqrt{1 - Cos^2(x)}$. Besides, it is easier to just change a sign rather than computing a square followed by a square root! With known approximations for *Cos(x)* and *Sin(x)*, the construction of *Tan(x), Sec(x), Csc(x),* and *Cot(x)* is straightforward. Therefore, the construction of all the 'forward trigonometric' functions can be built around the single *Cos(x)* approximation.

To approximate *Cos(x)* the general method described earlier for approximating a SISO function can be used. However *Cos(x)* is periodic, and it is desired to try to use that fact in order to make the approximation more efficient. Assuming that for a given accuracy N_C neurons are required for the approximation of *Cos(x)* over one period, it is clear that using the general method, $n*N_C$ neurons will be required for the same approximation over n periods. In order to improve the efficiency, the solution is to transform *Cos(x)* over n periods, into a function that 'takes care' of the periodicity (compute the residual modulo 2π), followed by the approximation of *Cos(x)* over a single period. Each of these 2 functions is a SISO function. Therefore *Cos(x)* is approximated by a 2-layer neural network. The function that removes the periodicity in theory could be a tooth saw function from 0 to 2π. It is shown in figure (3.9). However, it is clear that this function is not continuous. Hence it absolutely can't be approximated by a single layer neural network.

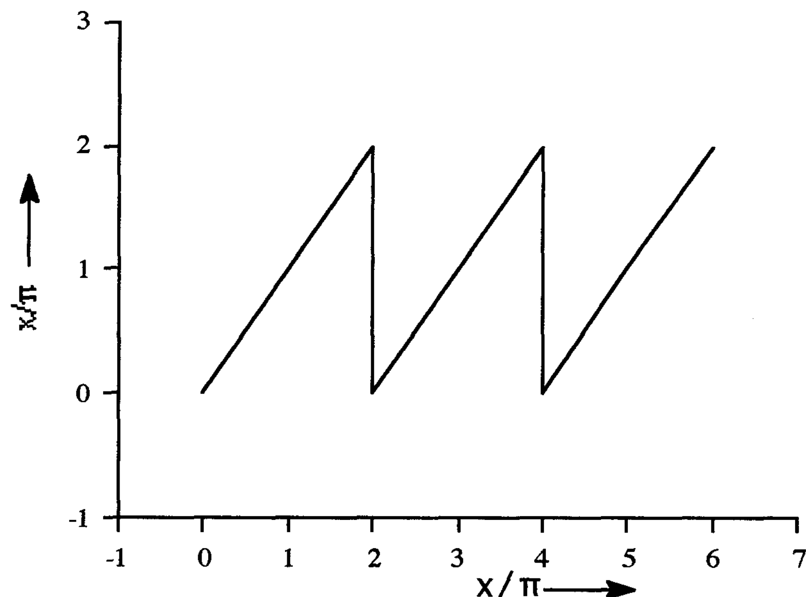

Figure (3.9), Plot of the tooth saw function to remove the 2π periodicity.

Due to the fact that *Cos(x)* is periodic and even, the relation *Cos(x)=Cos(π-x)* can be used. For an argument the residual of which is in the interval [0,π] the function to be used is *Cos(x)*. For an argument the residual of which residual is in the interval [π,2π] the function to be used is *Cos(π-x)*. With these assumptions, the original tooth saw function to remove the periodicity can be replaced by the new tooth saw function shown on figure (3.10).

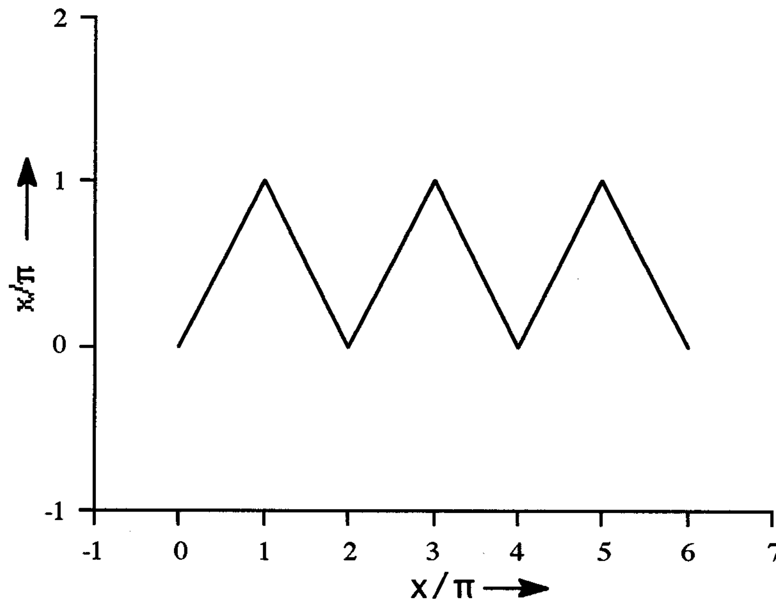

Figure (3.10), Plot of the improved tooth saw function.

This function is now continuous, it can therefore be approximated by a single layer neural net. In terms of 'cost', every period on that function needs 2 neurons. Therefore the total cost for approximating *Cos(x)* over n periods is $2*n+N_C$, which is much smaller than the original cost $n*N_C$, especially when n becomes large.

With respect to the inverse functions, both *ACos(x)* and *ASin(x)* are needed for uncertainty concern. $ASin(x) = ACos\left(\sqrt{1-x^2}\right)$ or vice versa is a valid expression only for results in the 1st quadrant. Otherwise there is an uncertainty between the 2nd and 4th quadrants. However, with some trigonometric manipulations, it is easy to show that *Atan(x)* to *ASin(x)* in all situation can be related by the always true expression:

$$ATan(x) = ASin\left(x \Big/ \sqrt{1+x^2} \right) \tag{3.19}$$

Notice however, that *ATan(x)* is a possible neuron activation function. This function can therefore be realized using a single element neural network, i.e. a single neuron. That short alternative may be desirable since it is an efficient model.

Several functions that must be included in the basic element list were presented, as well as some more that could be included, if desired. Once the basic element list is determined, these functions must be converted to single layer subnetworks by means of the SISO algorithm. Once this is performed, the subnetworks must be saved, i.e. for each one, the number of required neurons, the weight vectors and biases are saved. When a MISO function has to be approximated by a neural network, it is first decomposed in a correct manner, then synthesized using these preprogrammed and stored elements.

3.2.2 Improvements of the set of primary elements.

The object oriented approach that was just presented is so far missing a few important elements that deserve more attention. A few key functions that do not exist as such, but only as 'built ups' out of other functions were noticed. In this coming section an efficient construction of these missing functions is presented.

3.2.2.a Synthesis of the "square" unary function.

In the previous section the importance of the "square" function was shown . This unary function is used not only as-is, but also, for example, for computing the magnitude of an error signal. However, its most important aspect is that it is the basic element to construct the fundamental binary "multiplication" operator, so far missing in the network architecture.

Because of its importance and its frequent use, it is most desired to have a very efficient synthesis for it, rather than just a usual program as for other unary functions, such as 'cosine' or 'square root'. It happens, as it is shown in the coming paragraphs, that with only 2 neurons the 'x square' function can be synthesized with an accuracy almost only limited by the numerical precision of the machine, roughly 10^{-6} in single precision. Note that a "regular" programming (following the SISO algorithm) could never be that accurate, and would require at least 32 neurons for a 10^{-3} accuracy over the domain [-1.0,1.0].

As previously said, there is no real reason for having a completely homogenous network (all neurons having the same activation function), and for a reason that will become clear in due course, these 2 neurons have the sigmoid activation function $s(x) = \frac{1}{1+e^{-x}}$ rather than the piecewise linear activation function mostly used so far.

The function $t_\gamma(x)$ is introduced and constructed as:

$$t_\gamma(x) = s(x-\gamma) - s(x+\gamma) = \frac{1}{1+e^{-(x-\gamma)}} - \frac{1}{1+e^{-(x+\gamma)}} \tag{3.20}$$

x is considered as being the only variable, and γ being a fixed parameter. From a previous section the symmetry relation $s(-x) = 1 - s(x)$ stands.

Using the two previous equation, it can be computed:

$$\begin{aligned} t_\gamma(-x) &= s(-x-\gamma) - s(-x+\gamma) = (1 - s(x+\gamma)) - (1 - s(x-\gamma)) \\ &= -s(x+\gamma) + s(x-\gamma) = t_\gamma(x) \end{aligned} \tag{3.21}$$

This clearly implies that $t_\gamma(x)$ is an even function.

$t_\gamma(x)$ can now be expanded into a 5th order Taylor series around the origin. Since $t_\gamma(x)$ is even all the odd power coefficients are equal to 0. Hence,

$$t_\gamma(x) = t_\gamma(0) + t_\gamma^{(2)}(0)\frac{x^2}{2} + t_\gamma^{(4)}(0)\frac{x^4}{24} + O(x^6) \tag{3.22}$$

It is straight forward to compute $t_\gamma(0) = \frac{1}{1+e^\gamma} - \frac{1}{1+e^{-\gamma}}$

However, computing the second derivative of $t_\gamma(x)$, keeping along the γ parameter involves longer computation, and computing the 4th derivative is even worse. In order to perform these symbolic manipulations, a mathematical software package can be used.

Mathematica finds that

$$t_\gamma^{(2)}(0) = \frac{e^{-\gamma}}{\left(1+e^{-\gamma}\right)^2} - \frac{2e^{-2\gamma}}{\left(1+e^{-\gamma}\right)^3} + \frac{2e^{2\gamma}}{\left(1+e^{\gamma}\right)^3} - \frac{e^{\gamma}}{\left(1+e^{\gamma}\right)^2} \tag{3.23a}$$

and that,

$$t_\gamma^{(4)}(0) = \frac{e^{-\gamma}}{\left(1+e^{-\gamma}\right)^2} - \frac{14e^{-2\gamma}}{\left(1+e^{-\gamma}\right)^3} + \frac{36e^{-3\gamma}}{\left(1+e^{-\gamma}\right)^4} - \frac{24e^{-4\gamma}}{\left(1+e^{-\gamma}\right)^5}$$
$$- \frac{e^{\gamma}}{\left(1+e^{\gamma}\right)^2} + \frac{14e^{2\gamma}}{\left(1+e^{\gamma}\right)^3} - \frac{36e^{3\gamma}}{\left(1+e^{\gamma}\right)^4} + \frac{24e^{4\gamma}}{\left(1+e^{\gamma}\right)^5} \tag{3.23b}$$

To this point, the γ parameter is still free. It can be assigned a value of particular interest for the rest of the study. Indeed if $t_\gamma^{(4)}(0)$ were equal to 0, that would imply that the Taylor series of $t_\gamma(x)$ becomes:

$t_\gamma(x) = t_\gamma(0) + t_\gamma^{(2)}(0)\dfrac{x^2}{2} + O(x^6)$, hence $t_\gamma(x)$ would approximate a parabola, the error being only of 6th order.

$t_\gamma^{(4)}(0)$ can be looked at assuming γ as the single variable. There is interest in finding the values of γ, so that $t_\gamma^{(4)}(0) = 0$. $t_\gamma^{(4)}(0)$ can be plotted as a function of γ

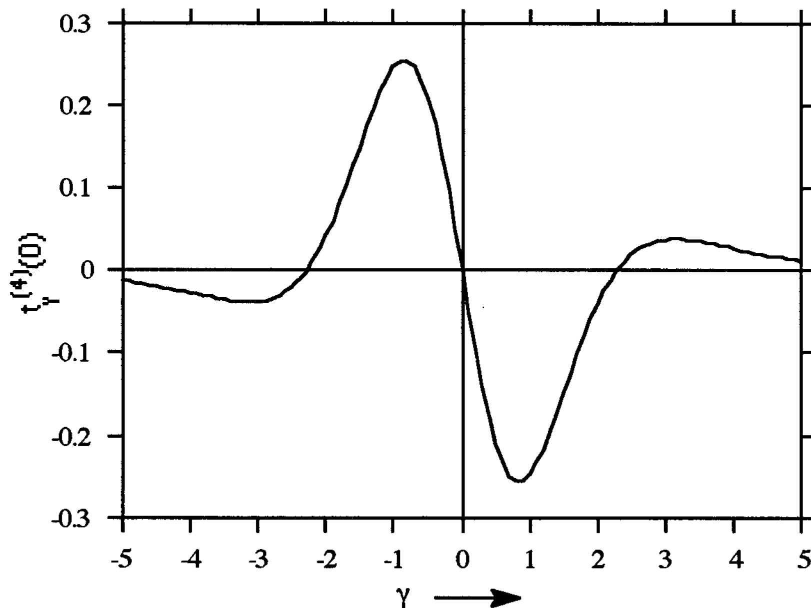

Figure (3.10), Plot of $t_{\gamma}^{(4)}(0)$ as a function of the variable γ

This function is odd. It passes through the origin, crosses the x-axis at 2 more points, and goes to 0 at infinity. Clearly the point of current interest is the x-axis crossing for γ between 2 and 3. Once again using mathematica, it is found that for $\zeta = Log\left(\frac{10+\sqrt{96}}{2}\right) \approx 2.2924$, $t_{\zeta}^{(4)}(0) = 0$. For this value,

$$t_{\zeta}(0) = \frac{1}{1+e^{\zeta}} - \frac{1}{1+e^{-\zeta}} \approx -0.8165 \tag{3.24a}$$

and,

$$t_{\zeta}^{(2)}(0) = \frac{e^{-\zeta}}{\left(1+e^{-\zeta}\right)^2} - \frac{2e^{-2\zeta}}{\left(1+e^{-\zeta}\right)^3} + \frac{2e^{2\zeta}}{\left(1+e^{\zeta}\right)^3} - \frac{e^{\zeta}}{\left(1+e^{\zeta}\right)^2} \approx 0.1361 \tag{3.24b}$$

The function $square(x)$ is introduced and constructed as:

$$\begin{aligned} square(x) &= \frac{s(x-\zeta) - s(x+\zeta) - t_{\zeta}(0)}{t_{\zeta}^{(2)}(0)/2} \\ &\approx \frac{s(x-2.2924) - s(x+2.2924) + 0.8165}{0.0680} \end{aligned} \tag{3.25}$$

It can be noticed that this function is perfectly "legal" in terms of being built using the pseudo multilayer neural network architecture. It can be rewritten as,

$$square(x) = \left(t_\xi(x) - t_\xi(0)\right) \Big/ \left(t_\xi^{(2)}(0) \Big/ 2\right) \tag{3.26}$$

Performing a 5th order Taylor expansion produces:

$$\begin{aligned} square(x) &= \left(\left(t_\xi(0) + t_\xi^{(2)}(0)\frac{x^2}{2} + O(x^6)\right) - t_\xi(0)\right) \Big/ \left(t_\xi^{(2)}(0) \Big/ 2\right) \\ &= \left(t_\xi^{(2)}(0)\frac{x^2}{2} + O(x^6)\right) \Big/ \left(t_\xi^{(2)}(0) \Big/ 2\right) = x^2 + O(x^6) \end{aligned} \tag{3.27}$$

The function $square(x)$ that has just been synthesized is approximating x^2 with only a 6th order error.

It can now be pondered up to what point that small error can be neglected. In order to investigate this, for an increasing value of x both x^2 and $square(x)$ are computed. The error between the two can be computed, and therefore an unacceptability bound can be determined. These are summarized in the plot and table below, figures (3.11) and (3.12). Because both x^2 and $square(x)$ are even functions, only the positive real axis is studied.

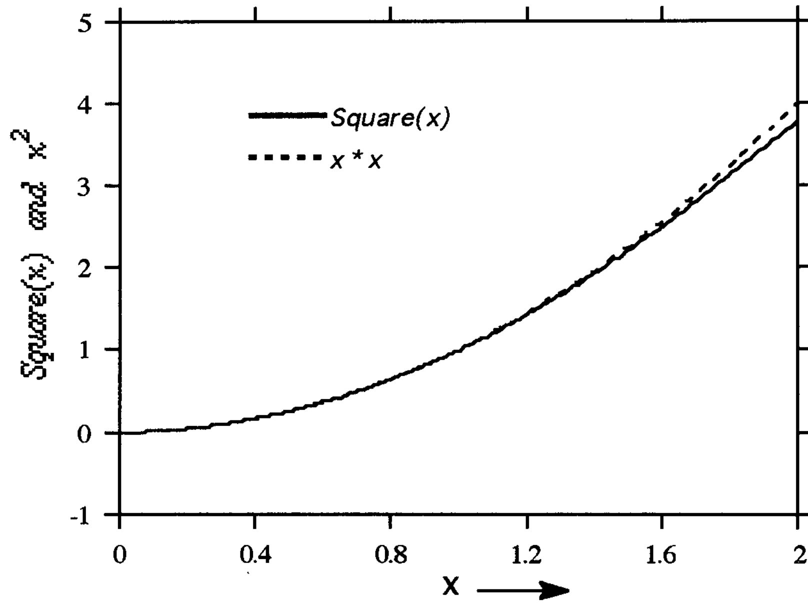

Figure (3.11). Comparison of synthesized *square(x)* against original x^2

x	Error
0.10	0.0000
0.20	0.0000
0.30	0.0000
0.40	0.0000
0.50	-0.0001
0.60	-0.0002
0.70	-0.0005
0.80	-0.0011
0.90	-0.0021

1.00	-0.0040
1.10	-0.0070
1.20	-0.0117
1.30	-0.0186
1.40	-0.0287
1.50	-0.0427

Figure (3.12). Tabulated error of the synthesized *square(x)* function

It is no surprise to notice that the error magnitude increases as the function is evaluated further away from the origin. The 6th order error term can no longer be ignored when x is around unity. At that point both x^2 and x^6 are unity, hence the signal to noise ratio is close to 1. It is clear that this $square(x)$ function can be used only when the argument is less than 0.80 if a 10^{-3} accuracy is required, or less than 0.50 if a 10^{-4} accuracy is required. However, this limitation is not a problem for computing squares of larger numbers. Indeed it is always possible to scale down the input by a constant factor α, and scale up the output by a constant factor α^2. Scaling by a constant factor is again a "legal" operation. However, it is not recommended to scale down by a too large factor, the reason being that the computation would be made with numbers too close to the machine numerical resolution.

The $square(x)$ function just designed is indeed accurate enough for the current needs, and is simple enough in terms of neural network topology to be included in the basic element list for further use.

3.2.2.b Synthesis of the "multiplication" binary operator, *.

When the neural network topology and "legal" operators were defined, it was stressed that the very basic and fundamental multiplication operator was missing. It was shown why it would be unwise to build a multiplication around addition, logarithm, and exponential, although all these are legal, and an alternate approach was suggested, namely that $x \cdot y = \frac{1}{2}\left((x+y)^2 - x^2 - y^2\right)$. It is clear that this method involves another unary operator, the square function. Meanwhile, in the previous section an efficient model using only 2 neurons to compute the square of a number was developed. This $square(x)$ function is used for synthesizing the "multiplication" operator.

From the previous section,

$$square(x) = \frac{s(x-\zeta) - s(x+\zeta) - t_\zeta(0)}{t_\zeta^{(2)}(0)/2} \tag{3.28}$$

where ζ, $t_\zeta(0)$, and $t_\zeta^{(2)}(0)$ are constants defined earlier. The multiplication operator $mult(x,y)$ is introduced and defined as,

$$mult(x,y) = \frac{1}{2}\left(square(x+y) - square(x) - square(y)\right) \tag{3.29}$$

Combining the two previous equations, it can be obtained,

$$mult(x,y) = \frac{1}{2}\left(\begin{array}{l} \dfrac{s(x+y-\zeta)-s(x+y+\zeta)-t_\zeta(0)}{t_\zeta^{(2)}(0)/2} \\ -\dfrac{s(x-\zeta)-s(x+\zeta)-t_\zeta(0)}{t_\zeta^{(2)}(0)/2} - \dfrac{s(y-\zeta)-s(y+\zeta)-t_\zeta(0)}{t_\zeta^{(2)}(0)/2} \end{array} \right)$$

$$= \frac{1}{t_\zeta^{(2)}(0)}\left(\begin{array}{l} s(x+y-\zeta)-s(x+y+\zeta) \\ -s(x-\zeta)+s(x+\zeta)-s(y-\zeta)+s(y+\zeta)+t_\zeta(0) \end{array} \right) \tag{3.30}$$

It has to be remembered that the $square(x)$ function produces a valid result only when the argument has a small magnitude. Otherwise a scaling is required. It is obvious that some care is required when computing the multiplication of 2 numbers using the $mult(x,y)$ operator as well.

Figure (3.13) presents some examples of the usage of $mult(x,y)$, along with the errors. Note that the scaling factor was 100.

x	y	x*y	mult(x,y)	| Error |
2	4	8	8.0000	0.00000
3	6	18	18.0000	0.00001
-1	5	-5	-5.0000	0.00000
5	9	45	44.9999	0.00015
-7	-11	77	76.9993	0.00067
9	12	108	107.9983	0.00171
10	-14	-140	-139.9998	0.00018
13	13	169	168.9938	0.00622

Figure 3.13, examples of the use of the *mult(x,y)* operator, along the error.

The synthesis of the $mult(x,y)$ operator is acceptable. The errors are of small magnitude. Similarly to the $square(x)$ function, it can be included in the basic element list to be used when assembling more complex functions. It can be noticed at this point that $mult(x,y)$ requires only 6 neurons.

3.3. A case study for comparison with classical methods.

In this section, the object oriented method that was presented for constructing a neural network for function approximation is compared against more classical methods for solving the same problem. As it was said before, it is possible to approximate a MISO function by a single layer neural network, following Funahashi, or by a multilayer neural network following Hornik. Both of these use the backpropagation rule as a mean for programming the networks.

In order to duplicate those results, backpropagation has also to be used. The algorithms are implemented using Granino Korn's package Neunet/Desire presented in 'Neural Network Experiments on Personal Computers and Workstations'. This package is run on a Sun-4 station, in single user mode. In this way, the elapsed CPU time can be tracked more easily. As shown later, the deal is days of CPU time...

The case study is an arbitrary function. It is desired that it has a nice 'look' and be easily presentable. Therefore a dual input function is welcome, to be plotted in three dimension. The arbitrary function to be approximated by a neural network is chosen to be:

$$
\begin{aligned}
y &= f_{case}(u_1,u_2) \\
&= \left[u_1 + \cos(5u_1 + 2)\right]\left[u_2 + \sin(2u_2)\right] + 2e^{-\left((u_1-1)^2+(u_2-1)^2\right)}
\end{aligned}
\tag{3.30}
$$

The approximation is to be performed over the domain $[0,3]\times[0,3]$. Figure (3.14) is a Matlab plot of f_{case}:

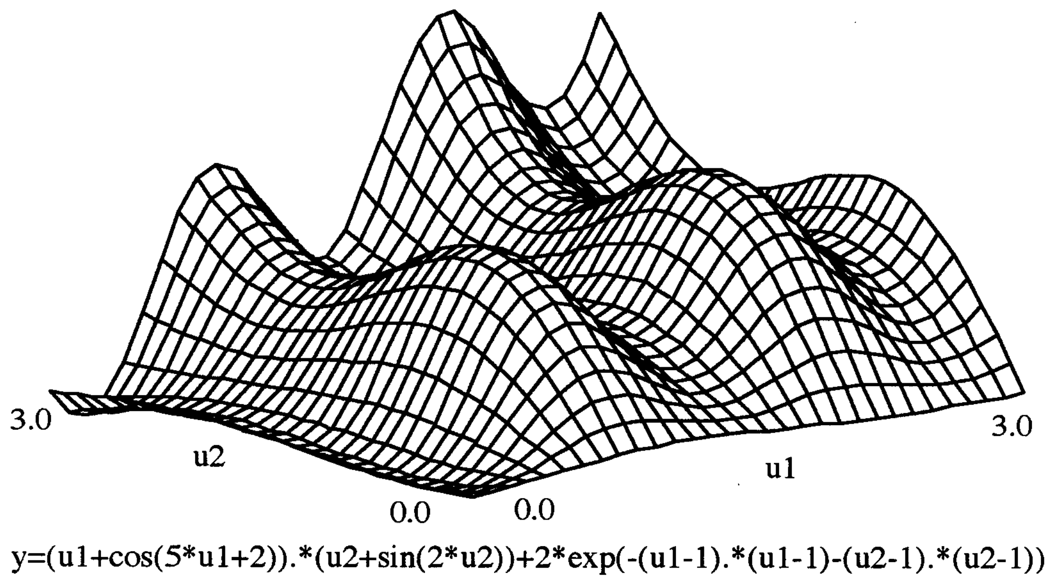

Figure 3.14, Plot of the case study function.

The case study consists of approximating this function with different network architectures. In order to perform a meaningful comparison there must be some common ground between all the cases. For every case the network has the same number of neurons, and the corresponding residual errors are compared. Another indicator is the time it takes to perform the programming, and these are compared as well. With respect to this last point the number of times f_{case} has to be evaluated is the base of the comparison, which is a better indication that just the CPU time.

3.3. 1 Approximation by classical feedforward networks.

In order to perform the programming of a feedforward neural network for approximating a function, Korn's example in his section 7.2 is followed. This example has to be modified a bit in order to accommodate two inputs instead of one, but the overall programming scheme is the same. Throughout the analysis, a different number of hidden layers are considered. Overall, the approximating capabilities of feedforward networks having one to six hidden layers in addition to the input and to the output layers are studied. Similarly to Korn's example, and more importantly, similarly to the pseudo multilayer network architecture to be compared to, the output layer neuron doesn't have an activation function, but is a purely linear element.

In order for the backpropagation to run, the activation function needs to be continuous and have a derivative. Similarly to Korn, *tanh(x)* is used. With respect to the backpropagation rule, Korn's improved version (faster convergence) is used as shown in the next paragraph. With respect to neuron biases, each layer is allowed to be biased, before the activation function, i.e. on the linear part of the neuron.

As just said, the back propagation rule used here is Korn's improved version. For each layer i, both the weight matrix W_i with the previous layers and the bias vector b_i have to be updated.

The algorithm works in 3 steps:

- For a given input, the information is propagated forward, up to a value for the output. For each layer, the neuron states are computed as:

$$x_i = \tanh(W_i x_{i-1} + b_i) \tag{3.31}$$

- The error between the target output and the obtained output is backpropagated through the network, according to the rule:

$$delta_i = W_{i+1}^T \cdot delta_{i+1} \cdot tri(x_i) \tag{3.32}$$

where tri(.) is the triangle function, this is Korn's improvement: it replaces the derivative used usually.

- The weight matrices and the biases are updated according to the rules:

$$\begin{aligned} W_i &= W_i + gain \cdot delta_i \cdot x_{i-1} \\ b_i &= b_i + gain \cdot delta_i \end{aligned} \tag{3.33}$$

This describes the backpropagation algorithm used for the case study. A sample code is shown in appendix A. This sample code is for the 4 hidden layer network case.

The complete process of running all the cases is fairly long. The results are summarized in the coming section. The common ground for all the different networks is that they all have an overall number of neurons equal to 100 (or closest to 100 as possible). Every layer within the network has the same number of neurons. Figure (3.14) below summarizes these results. The residual error is computed as the average over the domain of the norm of the difference between the original function and the approximating network:

$$Error = \left\langle \left\| f_{case} - NN_{approx} \right\| \right\rangle_{[0,3]\times[0,3]} \tag{3.34}$$

Hidden layers	# Neurons	Architecture	Residual Error	# f_{case} Evaluation
1	100	100	0.238	10^6
2	100	2•50	0.031	$2 \cdot 10^6$
3	99	3•33	0.030	$6 \cdot 10^6$
4	100	4•25	0.022	$10 \cdot 10^6$
5	100	5•20	0.017	$13 \cdot 10^6$

Figure 3.14, Summary of the case study for backpropagation networks

It is clear from these results that any network with more than one hidden layer does a fairly good job of approximating the function. However, it can be noticed that the number of function evaluations grows much faster than the error gain when increasing the architecture complexity.

The reason that it was decided to use a common ground of only 100 neurons, and that examples with more hidden layers than 5 were not run is just a pure technological constraint: The last example, 100 neurons over 5 hidden layers, 20 neurons each, took more than 4 days CPU time on a Sun-4 workstation in single user mode! The study could have been pursued further, but the computations were getting out of hand.

3.3. 2 Approximation by the suggested network.

The suggested approach works in two steps. First the MISO function, which is unary$^+$ decomposable, is decomposed following the description from definition 3.1, or the example that was shown thereafter. The second step consists of converting this

decomposition into a pseudo-multilayer neural network following what was said in section 3.1.2.d. Therefore this has to be done for f_{case} too.

The unary+ decomposition of f_{case} is shown in figure (3.16)

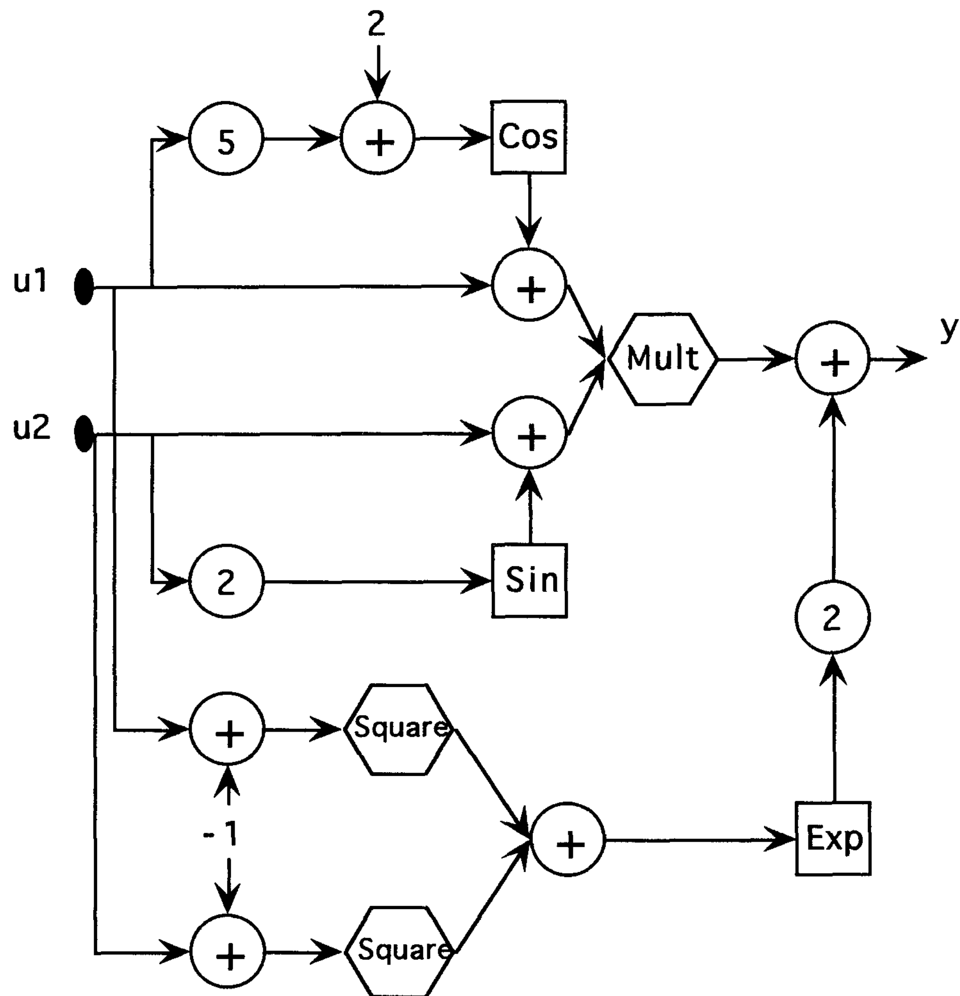

Figure 3.16, Unary+ decomposition of the case function f_{case}.

At this point this decomposition can be converted into a pseudo-multilayer network, where the SISO subnetworks correspond to the unary functions *cos(x)*, *sin(x)*, and *exp(x)*. The square and multiplication operators required in this example can be

directly synthesized using the *square(x)* and *mult(x,y)* elementary functions described in detail in sections 3.2.2.a and 3.2.2.b respectively.

In order to program this network, it can be remembered that there are 2 possibilities. The direct approach calls for programming individually the 3 SISO subnetworks according to the algorithm described in section 3.1.1. Once these are programmed, the complete network can be assembled. The object oriented approach assumes that the 3 SISO subnetworks represent 'basic' functions that have already been programmed and are stored in memory somewhere. In both cases, the square and multiplication are synthesized into the 2 neuron and 6 neuron special sub networks as it was said in the previous paragraph.

The number of neurons required for programming cos() is called N_{cos}, N_{sin} for sin(), and N_{exp} for exp(). The overall number of neurons of the approximating network, including the square and multiplication subnetworks, therefore is

$$\#_{neurons} = N_{\cos} + N_{\sin} + N_{\exp} + 2_{square_{a1}} + 2_{square_{a2}} + 6_{mult} \qquad (3.35)$$

Figure (3.16) shows the pseudo-multilayer network that performs the approximation of f_{case}.

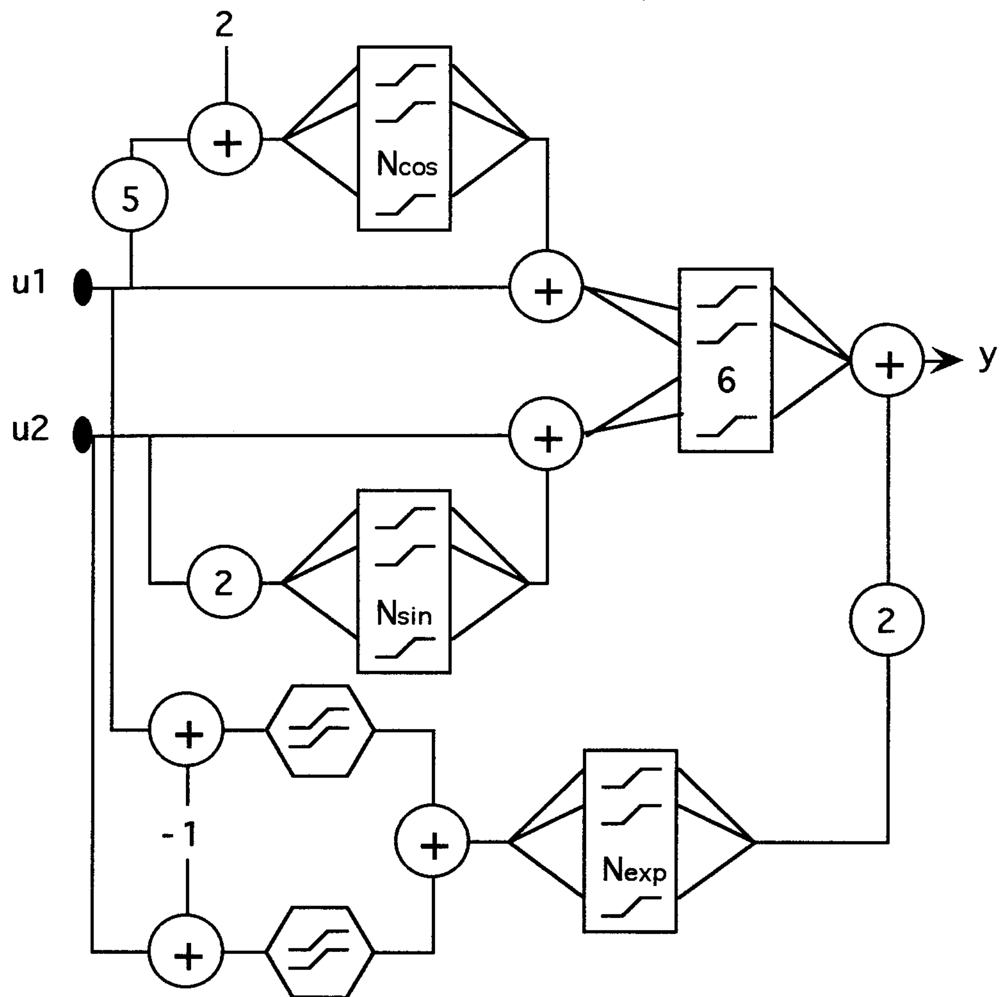

Figure 3.16, Pseudo multilayer network for approximating f_{case}.

Similarly as in the previous study, the residual error (defined as before) and the number of times a function must be evaluated are recorded. For this last parameter, the function to be evaluated is not f_{case} but the three unary functions instead. Furthermore, this is for the direct approach only, since for the object oriented approach the number of function evaluations is 0.

Since it is desired to know more about this network and since the programming is very easy and fast, the study is not limited to only 100 neurons. Other cases are run as

well to perform a more complete study. However, the data for 100 neurons will be the basis for the comparison against the straight feedforward networks that were studied in the previous section. The table shown in figure (3.17) summarizes the results.

# of Neurons	Residual Error	# Unary Function Evaluations
73	0.023	3039
75	0.021	3129
77	0.019	3151
81	0.017	2992
84	0.016	3128
91	0.014	3137
97	0.011	3014
105	0.009	3023
119	0.008	3088
134	0.006	3062
162	0.004	3121
226	0.002	3122

Figure 3.17, Result for approximating f_{case} with the suggested network.

From this study it can be concluded that a small number of neurons does a very nice job in approximating the function. It can also be noticed that the number of function evaluations for the direct approach is very small, hence the programming is very fast.

Once again, with the object oriented approach the number of function evaluations is 0, hence the programming is instantaneous!

An interesting fact is that the number of function evaluations seems unrelated to the required accuracy, or to the number of neurons. This is due to the way the SISO programming algorithm works. The number of function evaluations is related to the step size by which ε is increased in the $\Psi(\bullet)$ function as described in section 3.1.1., equation (3.4). What happens is that the whole function domain is scanned with this step size resolution, independently of the required tolerance.

3.3. 3 Conclusion of the comparison.

An extensive test on both the classical approach and the suggested approach for approximating the case study function f_{case} has been run. All elements are present for concluding the comparison.

Considering 100 neurons as test bench as said earlier, it is noticed that the suggested network, in terms of residual error, performs twice as good as a feedforward network with 4 hidden layers, and performs 35% better than a feedforward network with 5 hidden layers. In terms of function evaluations, and therefore speed of programming, the comparison shows a terrific improvement: more than 4000 times for a similar accuracy! Furthermore, it terms of real time for programming, the difference is much more than that since one single evaluation of f_{case} is much more CPU intensive than one single evaluation of any unary function. Therefore the gain in terms of CPU time is much more than 4000.

Chapter 4:
Review of Global Optimization

This chapter is a review of current global optimization techniques. First, optimization and minimization are defined and related to one another. Then, several methods for various optimization problems are presented, and those of interest for this project, including D.C. programming are reviewed, where 'D.C.' stands for 'Difference of Convex'.

4.1 General information

Before considering global optimization, it is necessary to first look at optimization by itself, and then the difference between the two. A problem is said to be "optimized" when it is in a "best" situation in some sense. In order to measure how good the situation is, a performance function (also called cost function) is created. Optimizing a problem is equivalent to minimizing (or maximazing) its associated performance function. A general problem deals with a number of variables from several classes: the input variables which are independent to the system, the state variables of the system which depend on both themselves and the inputs, and the output variables that depend on the state variables and the inputs. When the general problem is translated into a performance function, it is composed of two parts: static and dynamic. The static part corresponds to a snapshot look at the system, the dynamic section corresponds to what was involved to drive the system to the actual state. The cost function is often represented as:

$$J(T) = f\left(\underline{u}(T), \underline{x}(T), y(T)\right) + \int_0^T g\left(\underline{u}(t), \underline{x}(t), y(t)\right)dt \qquad (4.1)$$

By the fact that the state variables considered here are possibly depending on themselves, the system is implicitly assumed to be recurrent. For a feedforward system (memoryless), the state variables depend only on the input; similarly to the previous case, the outputs depend on both inputs and state variables. However, because there is no feedback present, a given input results in a unique possible output. The system cost function no longer involves what it took to drive the system to the present state since that task is direct. It can therefore be measured instantaneously, and be represented as:

$$J(T) = f\big(\underline{u}(T), \underline{x}(T), y(T)\big) \tag{4.2}$$

Only this later class will be considered from now on. The cost function may depend on output variables which depend in turn on state variables that depend further on input variables. Therefore the cost function ultimately depends only on input variables. A complete feedforward system along the cost function may be written as:

$$J(T) = f\big(\underline{u}(T), \underline{x}(\underline{u}(T)), y(\underline{u}(T), \underline{x}(\underline{u}(T)))\big) = \tilde{f}\big(\underline{u}(T)\big) \tag{4.3}$$

which clearly shows the dependency of the cost only on the inputs.

As it was stated earlier, the system is optimized when the performance function is minimized. Equation (4.3) clearly shows that the cost function is a MISO function. Assuming it belongs to C_2, a given input vector $\underline{u}$ that satisfies the equations (4.4) is called a minimizing set of J, and J is said to be minimized when these conditions hold:

$$\begin{cases} \underline{\nabla J} = \underline{0} \\ \underline{\nabla^2 J} \geq \underline{0} \end{cases} \Leftrightarrow \begin{cases} \forall i, \partial J / \partial u_i = 0 \\ \forall i, \partial^2 J / \partial u_i^2 \geq 0 \end{cases} \tag{4.4}$$

The first equation, sometimes referred as Fermat's rule, states that J is "flat" in every direction. However it might be a maximum, or a minimum, or a flat section, independently in every direction. If it is of the same type in every direction and not of the flat type, then this point it is what is called an extremum. The second equation states that J is convex in every direction, implying that it is overall a minimum.

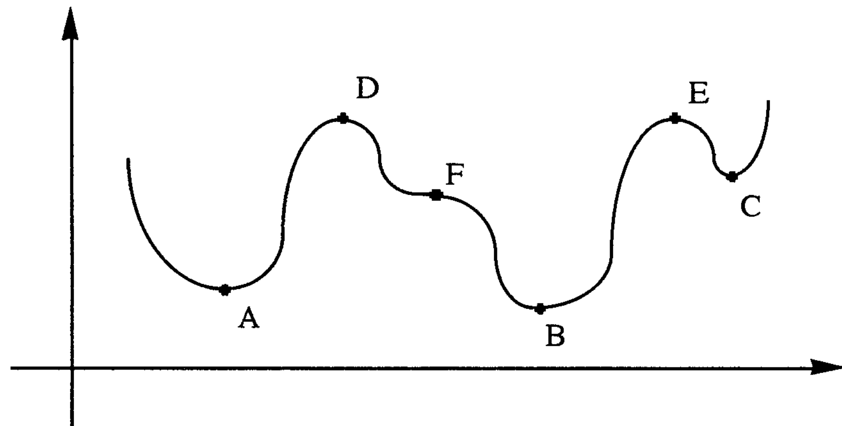

Figure 4.1: Plot of an arbitrary function that shows "zero derivative" points

In Figure 4.1 a sample function is shown. Points A, B, C are minima, D, E are maxima and F is a flat point. All of these points satisfy the first equation of (4.4), but only points A, B, and C satisfy also the second equation. From this example it is clear that a function doesn't necessarily have only a single minimum. Therefore a cost function that has several minima produces the same number of optimal solutions for the system it represents.

Up to now, optimization has been related to minimization of a cost function. How about global optimization? As it was just illustrated a system may have several optimal solutions. A system is said to be globally optimized when among all the possible optimal solutions, the very best one is determined: it is the global optimal solution.

It is straightforward to conclude that the global optimization of a system corresponds to the global minimum of the performance function *J*.

The equation set (4.4) provides conditions for a point to be a minimum, but in no sense does it give information with respect to global performance. At this point the contradiction between a local study and a global result becomes obvious: a derivative and a curvature used in equation (4.4) are local quantities:

$$\begin{aligned} \partial f / \partial u &\approx \left(f(u+h) - f(u-h)\right) / 2h \\ \partial^2 f / \partial u^2 &\approx \left(f(u+h) - 2f(u) + f(u-h)\right) / h^2 \end{aligned} \tag{4.5}$$

where h is a "very small" number. Clearly only the close neighborhood of a point is explored when determining whether it is a minimum. In contrast, a global minimum has to take into account the whole search space. When is a point u satisfying (4.4) a global minimum?

If it can be determined that a function has a single minimum over the search domain, then for sure there is only a unique solution to (4.4), and it corresponds to the global minimum. For example a convex function is of this type. This is why it is simple to globally minimize it. However, for an arbitrary function there is no such rule.

As a matter of fact there is no general method to determine the global minimum of an arbitrary function. On a pessimistic but realistic note, Press [Press, 1991] can be quoted as saying: "It is not hard to see why very likely there never will be any good,

general methods"! Well, although there might never be general methods, several methods which work in particular cases have been developed.

Before summarizing these methods for some particular problems, it is necessary to present the two general classes of global optimization methods: deterministic and stochastic. A deterministic algorithm proceeds step by step following straight rules. Starting from a given initial state, the trajectory is always identical from one experiment to another. In contrast, a stochastic algorithm proceeds in a manner that is disturbed by noise. It is therefore clear that different trajectories are produced for different experiments, even if all the initial conditions are alike.

When searching for a global minimum, an algorithm may try to "go down" in order to reach the minimum. However, if it happens that the search proceeds into a valley that doesn't include the global minimum, the deterministic algorithm stops when the bottom of the valley is reached, without a chance to find the global minimum. In contrast, the statistical algorithm may have a chance to "shake up" the system and have the search continue in another valley. This situation describes the main advantage of a statistical algorithm versus a deterministic one. However, there are disadvantages to this kind of algorithms as well. For example, the amount of noise to be added to the system is system dependent. Therefore, an a priori knowledge of the system is required. However, the purpose of this introduction is not to do a complete comparative study of the two classes of algorithms for global optimizations, but just a summary. There are good reference texts for this task [Christofides, 1979]. Before finishing with statistical algorithms, the most popular of these must be mentioned: the Simulated Annealing method [Kirkpatrick, 1983], [Sejnowski, 1986]. The system is "heated up" (i.e. a large quantity of noise is added), and the minimum is looked for. As the iterations progress, the system is progressively "cooled down" (i.e. the noise level decreases through time). Eventually the system is "cold" (no noise at all) and the estimate of the minimum is supposed to be the

global minimum. The reason for this temperature change is the following: As the noise level is extreme, the small details of the target function are ignored and only the main structures are apparent to the algorithm. As the noise level decreases, the search may move from one valley to a lower one, but hardly vice versa: more energy is required to go up than down, therefore there exists a temperature when there is enough energy to go climb the mountain from the upper side, but not enough from the lower side. Hence, progressively the lowest valley, the one containing the global minimum is selected for the final search where the noise is eliminated.

Throughout the rest of the project, deterministic methods are used. This therefore concludes the presentation of statistical methods for global optimization.

4.2 Deterministic methods for global optimization

Contrary to the previously mentioned statistical methods, deterministic methods are organized in such a manner that at every instant, for a given situation, the search toward the global minimum continues in a direction determined by some strong guidelines and rules. This implies that the algorithms must be able to deal with many different events. They must also be designed in such a way that their efficiency to move in the right direction is strong: From the knowledge of the problem accumulated through the search so far, they must have good induction to where to search next.

It is clear that a deterministic algorithm for global optimization is a set of rules to be continuously applied according to the performance of the search. This is why these deterministic methods are called "programs". From the initial condition until the success of finding the global minimum, or the failure not to do so, every event and decision is "programmed". A deterministic method for global optimization is indeed fully deterministic.

As previously mentioned, the task of finding the global minimum of an objective function may be very simple, or quite difficult depending on the complexity of the problem. In order to achieve best success for various types of problems, several methods have been developed for solving a particular problem. As the problem becomes more complex, so does the method.

How is it possible to classify a problem as being easy or difficult? What are the important parameters in a problem that may simplify the task? An optimization task is made of essentially two components: the objective function to be minimized and the domain over which the function must be minimized. Both of these are equally important: For example a very simple objective function such as a linear function has no solution if the considered search domain is not bounded. On the other hand, minimizing a periodic function over single period domain is very easy, even though the function by itself has an infinite number of minima. These two extreme examples illustrate that both the domain and the function determine the overall complexity of the problem.

In addition, it can be shown that they are both related: it is always possible through change of variables to simplify the domain at the expense of a more complex cost function, or vice versa. This is why every algorithm is designed for solving a problem presented as a pair: the domain and the function.

Before presenting a brief summary of various problems, some mathematical definitions have to be recalled.

The definition of a convex function over a domain Ω is the following:

$$\begin{aligned} &f(.):\Omega \rightarrow \Re \text{ is a convex function } \Leftrightarrow \\ &\qquad \forall u_1, u_2 \in \Omega, \forall t \in [0,1], \\ &f(tu_1 + (1-t)u_2) \leq tf(u_1) + (1-t)f(u_2) \end{aligned} \tag{4.6}$$

In other words this means that every point on the curve is always below the line segment connecting any other two points.

A more practical definition of a convex function is to say that its curvature is "upward" or equivalently that its second derivative is non-negative in every direction. Obviously, the second definition applies only to functions from C_2 while the first one is valid even for non-continuous functions. For example the function $y=x^2$ is convex.

A convex domain is one for which the line segment connecting any two points belonging to the domain is fully included in the domain. Figure 4.2 illustrates a convex domain and a non-convex domain.

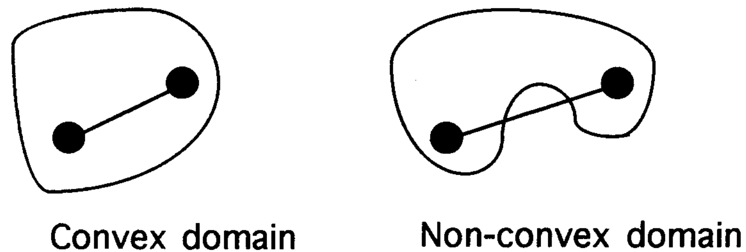

Figure 4.2. Example of convex and non-convex domains.

The next term to be recalled is a concave function. It is closely related to a convex function:

$$\{f(\underline{u}) \text{ is concave}\} \Leftrightarrow \{-f(\underline{u}) \text{ is convex}\} \quad (4.7)$$

Therefore a function from C_2 is concave if its second derivative is non-positive.

A convex constraint is an inequality involving a convex function:

$f(\underline{u}) \leq 0$ where f is convex is a called a convex constraint.

A reverse convex constraint is also an inequality involving a convex function:

$g(\underline{u}) \geq 0$ where g is convex is a called a reverse convex constraint.

Now that some definitions have been stated, a few common optimization problems can be summarized.

The simplest problem in global optimization is a convex problem. Both the domain and the objective function are convex. Therefore, there exists a unique minimum. This implies that any descent method such as a simple Newton method can solve the problem.

The next problem is called concave minimization. A concave objective function has to be minimized over a convex domain. For this problem, it is known that the solution has to be on the boundary of the domain. It is easy to observe that any point located in the inside of the domain cannot be a minimum, otherwise the objective function wouldn't be concave. It must be said that this problem has been studied very extensively for centuries and that there exist several very efficient algorithms to solve it. However, concave functions are of a very particular type and concave minimization lacks generalization to other problems.

The next problem, which is more general, is called reverse convex. A convex objective function is minimized over a domain which is the intersection of a convex domain with the complement of another convex domain. Figure 4.3 illustrates such a domain.

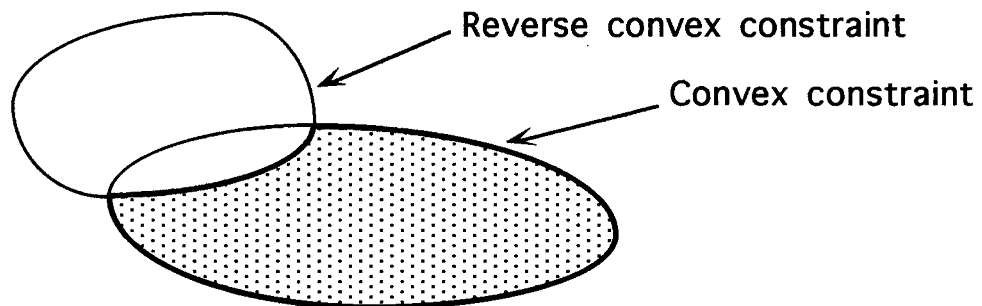

Figure 4.3. Search domain for a reverse convex and canonical DC problem.

Finally, the last problem recalled here, which is the most general in comparison with the previous ones is called a DC problem. "D.C." stands for Difference of Convex. In a DC problem a DC function f, along a set of DC constraints $\{g_i\}$ is minimized over a convex domain Ω. A function or a constraint f is DC if it can be written $f=f'-f''$ where both f' and f'' are convex functions. A DC problem is written in the following manner:

$$\begin{aligned} &\text{Minimize } f(x) \\ &\text{s.t. } x \in \Omega,\ g_i(x) \leq 0 \\ &\text{where } \Omega \text{ is convex, } f \,\&\, g: \Re^n \rightarrow \Re \text{ are DC.} \end{aligned} \tag{4.8}$$

Due to the duality principle between the domain and the objective function as stated earlier using variable transformations, a DC problem can be transformed into a canonical DC problem that is easier to solve [Tuy, 1985]. A canonical DC problem can be stated as the minimization of a linear objective function over a domain made up of two constraints: one convex, the other reverse convex. It is written in the following manner:

$$\begin{aligned} &\text{Minimize } cx \\ &\text{s.t. } h(x) \leq 0, g(x) \geq 0 \\ &\text{where } c \in \Re^n,\ h \,\&\, g: \Re^n \rightarrow \Re \text{ are convex.} \end{aligned} \tag{4.9}$$

It must be specified at this point that the general solution of DC problems is a very important result in global optimization since DC functions actually form a wide class of functions. Hiriart-Urruty reviewed their importance, and discussed their generality [Hiriart-Urruty, 1985]. It must be stated that a function from C_2 is a DC function, and

therefore most of the functions used in engineering are DC functions. However, even if it is known that a given function f is a DC function, it is not necessary that its DC decomposition (finding f' and f'' so that $f=f'-f''$) is easily known. This is one important drawback of DC functions since this decomposition must be known in order to use the minimization algorithms.

Assuming that the DC decompositions of both the objective function and of all the constraints are known, through a transformation of variables, and the addition of additional variables, a general DC problem can be converted into a canonical DC problem. As stated in the definition, this problem is equivalent to solving a linear minimization over the intersection of a convex domain with the complement of a convex domain: the convex constraint corresponds to the convex domain, while the reverse convex constraint corresponds to the complement of the convex domain. Therefore, the search domain for a canonical DC problem is of the type represented on Figure 4.3.

Since the objective function is linear, it is certain that the minimum has to be on the boundary of the overall domain. However, two situations can occur. Solving the convex problem while ignoring the reverse constraint results in a solution S on the boundary of the convex domain corresponding to the convex constraint. If this point S satisfies also the reverse convex constraint, then the later is said to be a nonessential reverse convex constraint. On the other hand, if the point S does not satisfy the reverse convex constraint, then the constraint is called an essential reverse convex constraint. In the latter situation, it is clear that the point S is not the solution of the original problem. However, it can be said that the solution belongs to the intersection of the boundaries of the constraints.

The canonical DC problem solution S can be summarized as:

-If $g(\underline{u}) \geq 0$ is a nonessential reverse convex constraint, then $h(S)=0$.

-If $g(\underline{u}) \geq 0$ is an essential reverse convex constraint, then $h(S)=g(S)=0$.

In any case, the solution must be on the boundary of the convex constraint. It is not possible that $h(S) \neq 0$. The proof of this statement can be found in [Horst, 1993]. Figure 4.4 illustrates the different possibilities.

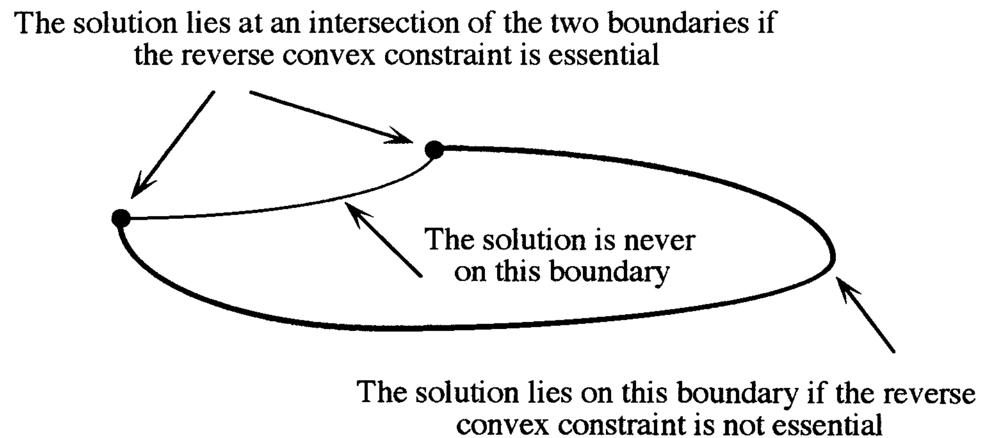

Figure 4.4. Various possibilities for the solution of a canonical DC problem.

Now that various global optimization problems have been presented, methods for solving them must be mentioned as well. The common principle for every method is that the search space for the global solution is modified following some rules and facts. The modification can be an enlargement of the space, or its reduction. It just depends on the algorithm and the problem itself. Notice that there are a priori no reductions on the search domain: it can be convex, concave, reverse convex, or nothing specific.

The first method mentioned here is called <u>branch and bound</u>. The search domain is relaxed and subsequently split into parts (branching) over which lower and upper bounds of the objective function value can be determined (bounding). Depending upon the respective bounds, certain subsets can be dropped, and others be expanded further for refining the solution. A known disadvantage of this method, similarly to every branch and

bound method, is that the correct solution might be known for quite some time before it is confirmed that it is indeed the global solution.

The second method, which has many variations, is based around what is called a "cut". A cut is nothing but the introduction of an additional inequality constraint during the iterations of an algorithm [Tuy, 1964]. A constraint can be viewed as a hyperplane *(dim=n-1)* in the search space *(dim=n)*. Therefore the constraint creates a boundary and splits the search space in two sections. This boundary is referred to as the "cut". Since the constraint is an inequality, this implies that one of the two sections is dropped or cut off from the search domain, while the other one must contain the solution.

The most popular among the cutting methods is the outer approximation algorithm. It starts with a large and simple feasible set D that includes the search domain Ω. The feasible domain is relaxed to a smaller set still containing Ω. The objective function is minimized over the newer set. This solution clearly belongs to D and is the global solution if and only if it also belongs to Ω. Otherwise, an appropriate section of $D \backslash \Omega$ is determined by a cut and dropped off, yielding to a new relaxed set that is a better approximation of the search domain. The algorithm keeps iterating in this manner until the solution is found. Figure 4.5 illustrates the evolution of the search domain as the iterations progress using the outer approximation algorithm.

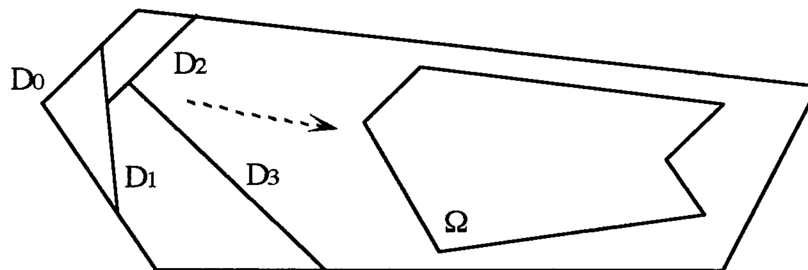

Figure 4.5. The domain reduction for the outer approximation algorithm

The advantage of this method is that it allows to use a very simple working set for which it is easier to determine the minimum than for the original domain. Furthermore, at each iteration, only a part of the set is modified which helps tracking and finding the new minimum if the previous one was known.

The drawback is that if the original domain Ω is very fancy, then it may require quite many cuts to approximate Ω with the required level of accuracy. As the number of cuts grows, the complexity of the set does too. This involves a lot of record keeping and makes it more computationally intensive to find the minimum at each iteration. In order to minimize the amount of book keeping a constraint dropping strategy is recommended. This involves checking for constraints that may have become redundant and can therefore be dropped off. Removing redundant constraints does not affect the set itself, but simplifies its representation.

Another algorithm which is not based on cutting is the inner approximation method. However, it can easily and quickly be described as the dual of the previously mentioned cutting method. In this case the search domain Ω is approximated from inside by a sequence of nested sets, while before it was coming from outside. The cuts have been mapped to annexations. Figure 4.6 illustrates the evolution of the search domain as the iterations progress in the inner approximation algorithm.

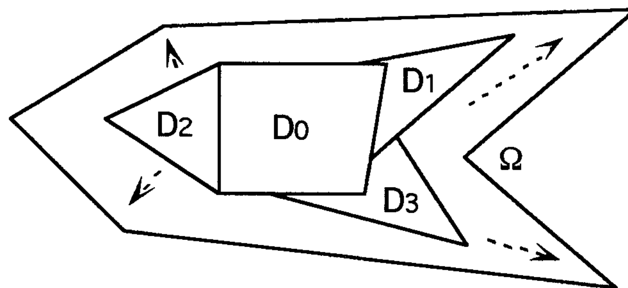

Figure 4.6. The domain expansion for the inner approximation algorithm

There are a few other popular algorithms such as convex underestimation, concave polyhedral underestimation, and conical and simplical algorithms. However, there are good books reviewing these [Horst, 1993], and therefore they will not be discussed further in this presentation.

Before finishing this section on global optimization, it is necessary to present which of these problems and methods are later used in this project.

For reasons that become apparent in due course, the problem of interest in this project is to solve a DC optimization, which is transformed into a simpler canonical DC problem using the variable transformations as stated earlier. In order to solve this problem, the algorithm of choice is an outer approximation cutting algorithm developed by Thoai [Thoai, 1988]. It must be said that this algorithm is an improvement over a similar one developed by Tuy [Tuy, 1987] and that Pham Dinh and Bernoussi developed a good possible alternative [Pham Dinh, 1989]. Thoai's algorithm is dealing only with and is specialized for DC problems. Following what was said earlier, in order to maintain a reasonable number of constraints, a constraint dropping strategy is also used. This part of the algorithm follows what was developed by Horst [Horst, 1988]. Both Thoai's and Horst's algorithms have been slightly modified to match this project's requirements. These changes are exposed in Chapter 5. Here, a quick review of the original algorithms is presented.

Once again, the definition of a few terms must first be recalled.

A polytope is a bounded polyhedral convex set. In other words it is a closed domain that is bounded by linear inequalities. It can be viewed as a convex domain bounded by hyperplanes.

A vertex (pl. vertices) is a point located at the intersection of n constraints of a polytope (assuming that the space dimension is n).

Figure 4.7 illustrates a polytope along its vertices. In this example the inequalities making up the polytope are:

$$\begin{aligned} &-x_2 \leq 0 && (1) \\ &3x_1 - x_2 - 9 \leq 0 && (2) \\ &-x_1 + 4x_2 - 8 \leq 0 && (3) \\ &-x_1 \leq 0 && (4) \end{aligned} \tag{4.10}$$

The vertices can be found to be:

$$\begin{aligned} &(A)\text{:} && (0,0) && \{(1)\&(4)\} \\ &(B)\text{:} && (3,0) && \{(2)\&(1)\} \\ &(C)\text{:} && (4,3) && \{(3)\&(2)\} \\ &(D)\text{:} && (0,2) && \{(4)\&(3)\} \end{aligned} \tag{4.11}$$

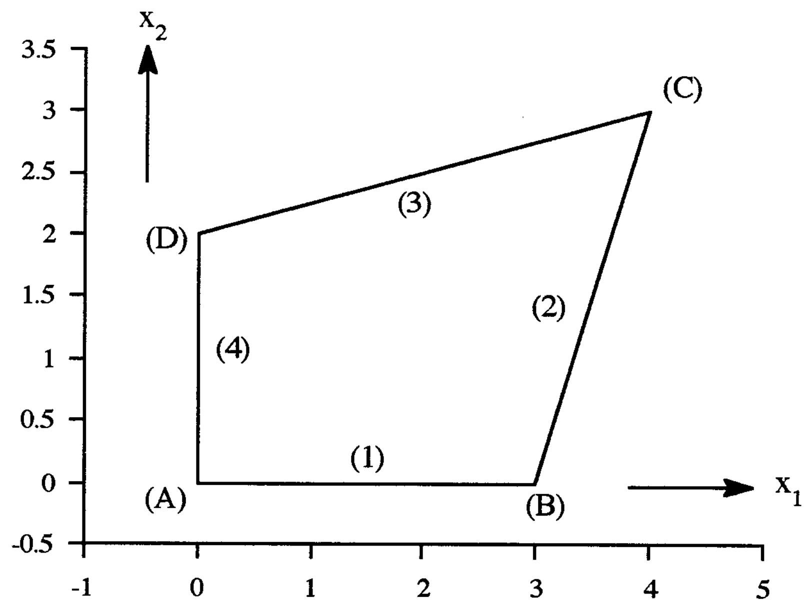

Figure 4.7. Sample polytope showing constraints and vertices.

In this particular example, it can be noticed that the number of constraints and the number of vertices are identical. This is true when the dimension of the space is 2. However, for higher dimensions, there is no direct relation between the number of constraints and the number of vertices.

As stated earlier, Thoai's algorithm solves a canonical DC problem. This means that a linear objective function is minimized over the intersection of a convex domain Ω_1 and the complement of another convex domain Ω_2 as shown on Figures 4.3 and 4.4. It uses what was described as an outer approximation algorithm.

Thoai starts with a large polytope P_0 containing entirely the convex domain Ω_1. Its vertices are computed. The objective function is minimized over P_0. Because P_0 is a polytope, it is known that the minimum S must be located at one of the vertices. Therefore, looking for it involves a search among a known finite number of points.

If S belongs to both Ω_1 and the complement of Ω_2 then it is the global solution of the initial problem.

If S belongs only to Ω_2 then a cut parallel to the boundary of Ω_1 at the closet distance between S and Ω_1 is created. A smaller polytope and its vertices are then generated. The procedure starts again. This cut allows the working polytope to approximate Ω_1 better.

Similarly, if S belongs only to Ω_1 then a cut is generated for approximating Ω_2. This particular cut is more complex to compute and is not discussed here.

The algorithm keeps iterating along these lines until it finds the global minimum. It may happen that the solution is an accumulation point in which case the algorithm must be stopped when the required accuracy is achieved.

For a complete discussion, Thoai's paper should be consulted. However, it can be stated at this point that the bottom line and strongest point of this algorithm is that the search for a minimum is always performed among a list of a finite number of points. This

list is updated at each iteration as the working polytope is modified by successive cuts for better approximation of the desired domain.

It is clear that as the number of iterations grows, so does the number of constraints, hence the number of vertices. It is very desirable to reduce these as much as possible. A constraint dropping strategy is recommended.

The algorithm presented next was developed by Horst. It must be applied at every iteration, first thing after a new cut $C: h_c(x) \leq 0$ has been created.

Consider the set *V(P)* of all the vertices of the polytope *P* and create a subset $V^-(P)$ of those which strictly satisfy h_c:

$$V^-(P) = \{u \in V(P) : h_c(u) < 0\} \tag{4.12}$$

A previous constraint $h(x) \leq 0$ is said to be redundant and can therefore be dropped off if:

$$\forall u \in V^-(P), \quad h(u) < 0 \tag{4.13}$$

Once a constraint is dropped off, its corresponding vertices are also removed from the list of vertices. Proceeding in this manner at every iteration is the most efficient way to reduce the number of active constraints and vertices.

The last point to be discussed here is how to expand the list of vertices once a new cut is performed. Several approaches have been introduced [Hoffman, 1981], [Thieu, 1983], and [Falk, 1976]. The manner chosen here for doing this was developed by Horst [Horst, 1988]. Once a new cut is introduced, the first thing to do, as just previously mentioned, is to check for redundant existing constraints and drop off what can be

deleted. Once that is performed, the new constraint has to be appended to the list of valid constraints, and new vertices have to be created.

Once again the subset $V^{-}(P)$ is created. Every element of this subset is a vertex corresponding to the intersection of n constraints (assuming total space dimension is n). Each element is therefore the solution of a set of n linear equations.

For each element $u \in V^{-}(P)$ consider the set $S_u = \left\{h_{u_i}(x) = 0, \quad i = 1,2,\ldots n\right\}$ of the constraints intersecting at u. Build n successive sets of equations where every equation of S_u is successively replaced by the new constraint $h_c(x) = 0$. For each new set of equations $S_{u_j} = \left\{h_c(x) = 0\right\} \cup S_u \setminus \left\{h_{u_j}(x) = 0\right\}$, the corresponding solution is computed. If it belongs to the polytope (every constraint must be satisfied), then it is a new vertex to be appended to the list of existing vertices. Otherwise, it can be forgotten and the algorithm continues.

Proceeding in this manner at every iteration of Thoai's algorithm (check for redundant constraints and build the new vertices) allows to have the minimum sized but correct set of vertices for computing the required successive minima.

This concludes for the moment the presentation on global optimization algorithms. In chapter 5, some improvements of Thoai's and Horst's algorithms are presented, as well as how they can be used in relation with neural networks.

Chapter 5:
Global Optimization via D.C. Programming

This chapter is the complete description of one method to solve a DC problem. It is based around Thoai's algorithm [Thoai, 1988]. A few subtasks are also presented such as problem formulation which must be able to accept neural network models, and a constraint dropping strategy and computation of vertices which follow the work by Horst [Horst, 1988].

The preliminary materials are first presented which include the required changes of variables for converting a DC problem into a canonical DC problem [Tuy, 1985], along with some properties of convex topology that are used. Then Thoai's algorithm is stated and described, followed by Horst's complementary methods. It must be said that all the material through this section is similar to what was presented during the introduction on optimization in Chapter 4. However, at this point, the definitions and results are no longer the original definitions by Tuy, Thoai, or Horst, but are formulated again, now in a manner that matches the requirements for the case of minimizing a neural network output, later presented in Chapter 6.

5.1 Preliminary material.

A result concerning convex functions must first be recalled.

Result 5.1: Consider a set $\{f_i(x), i = 1,2,\ldots,N\}$ of N convex functions. Consider the function $f(x)$ defined by:

$$f(x) = \underset{i=1}{\overset{N}{Max}}(f_i(x)) \tag{5.1}$$

Then, the function *f(x)* is convex too.

Proof 5.1: Consider two convex functions $f_1(x)$ and $f_2(x)$, and the function *f(x)* defined by $f(x) = Max(f_1(x), f_2(x))$. *f(x)* can be represented as successive pieces of either $f_1(x)$ and $f_2(x)$. Within a section, *f(x)* is uniformely equal to one of the two. It is therefore convex over that domain. At the intersection between two successive sections, Figure 5.1 shows that the definition (2.10) can be directly applied to show that *f(x)* is convex there too. Therefore, *f(x)* is a convex function.

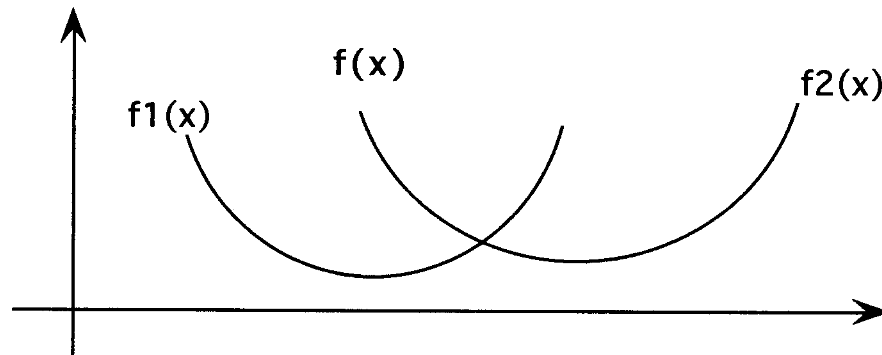

Figure 5.1: The intersection of two convex functions $f_1(x)$ and $f_2(x)$.

Taking the maximum of a set $\{f_i(x), i = 1,2,\ldots,N\}$ of *N* convex functions can be viewed as successively taking the maximum of *2* functions that make another convex function, which is in turn further combined. Therefore, the maximum of the whole set of convex functions is also a convex function.

Another similar result:

Result 5.2: Consider a set $\{g_i(x), i = 1,2,\ldots,N\}$ of N concave functions. Consider the function *g(x)* defined by:

$$g(x) = \underset{i=1}{\overset{N}{Min}}\left(g_i(x)\right) \tag{5.2}$$

Then, the function *g(x)* is concave too.

Proof 5.2: Consider the function $h(x) = -g(x)$. It can then be related to the definition of *g(x)* as:

$$h(x) = -g(x) = -\underset{i=1}{\overset{N}{Min}}\left(g_i(x)\right) = \underset{i=1}{\overset{N}{Max}}\left(-g_i(x)\right) \tag{5.3}$$

Since $g_i(x)$ is concave, then, using equation (2.11), $-g_i(x)$ must be convex. Using the result 5.1, this implies that *h(x)* is convex too. Therefore *g(x)* is concave.

Now a result on DC constraints:

Result 5.3: A DC constraint of the form $f(x) = f_1(x) - f_2(x) \le 0$, where both $f_1(x)$ and $f_2(x)$ are convex functions, can always be transformed into a set of convex and reverse convex constraints as:

$$\begin{aligned} &f_1(x) - z \le 0 \\ &z - f_2(x) \le 0 \end{aligned} \tag{5.4}$$

after the introduction of an additional independent variable z.

This transformation can be viewed as a trade-off between the complexity of the constraint and the complexity of the search domain over which the optimization process is performed. By introducing the new variable *z*, each of the two new equations has a nicer

topological structure, but the overall dimension of the search space has increased by one. This however implies no overall major topological change: Assuming the search space dimension originally was *N*. Since there was one constraint, the number of degrees of freedom therefore was *N-1*. After the transformation, the space dimension has been increased to *N+1* and there are two constraints. Therefore, the number of degrees of freedom still is *N-1*.

Proof 5.3: The initial constraint *(C1)*, and the new constraint set *(C2)* are equivalent:

$(C1) \Rightarrow (C2)$: Because the optimization process has no requirement on z in the constraints, it is always possible to find a value that bounds both $f_1(x)$ and $f_2(x)$ as desired. Because z is linear, it does not affect the convexity of the new constraints, which therefore maintain the same convexity as $f_1(x)$ and $-f_2(x)$: convex and reverse convex.

Consider for example the DC constraint $f(x) = x^4 - 5x^2 + 4 \leq 0$. This constraint is plotted in figure 5.2. It is clear that its solution domain is $[-2,-1] \cup [1,2]$.

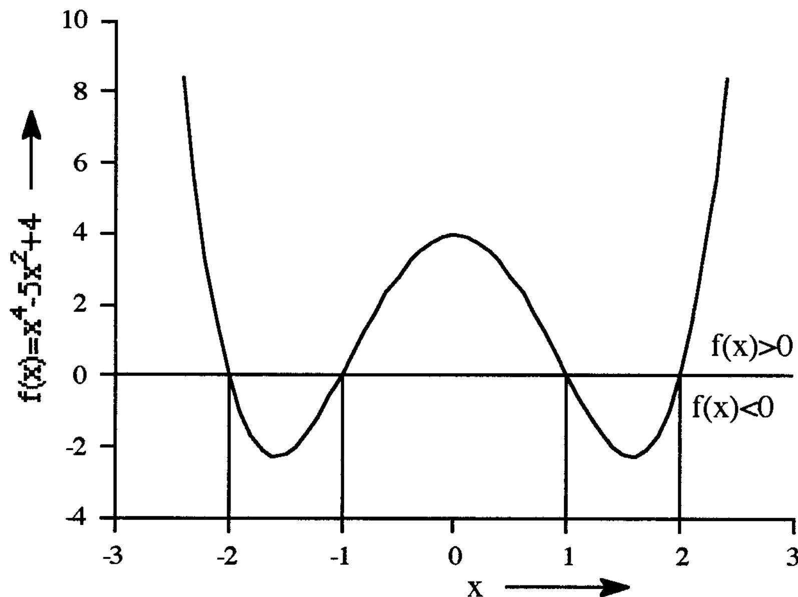

Figure 5.2: Sample DC constraint $f(x) = x^4 - 5x^2 + 4 \leq 0$

After the separation of the convex and concave parts, and the introduction of the additional variable z, the resulting two convex and reverse convex constraints are:

$$\begin{aligned} f_1(x) - z &= x^4 + 4 - z \leq 0 \\ z - f_2(x) &= z - 5x^2 \leq 0 \end{aligned} \tag{5.5}$$

Each of them defines a boundary in the *(x,z)* plane where solutions are or are not acceptable. The intersection of the two constraints determines the overall acceptable region. The *(x,z)* plots of the constraint set is shown in figure 5.3. It is clear that the solution domain is again $x \in [-2,-1] \cup [1,2]$ as it was determined for the original DC inequality. Once again, there are no restrictions on z which varies between various values, which are of no interest.

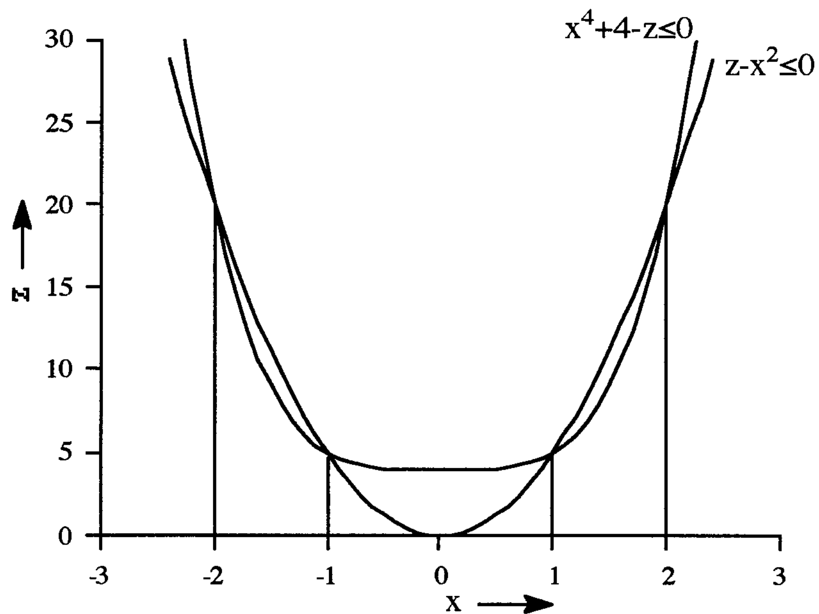

Figure 5.3. Two constraints in the *(x,z)* plane replace the original DC constraint.

$(C2) \Rightarrow (C1)$: Assuming both inequalities of *(C2)* are satisfied, the sum of the two is therefore satisfied, and this happens to be *(C1)*.

Now that these preliminary results have been stated, DC problems can be introduced. What is referred to as a DC programming problem is the minimization of a DC function over a convex domain with additional DC constraints:

Definition 5.1: A DC programming problem is a global optimization problem of the form:

$$\begin{aligned} &\text{minimize } \ f(x) \\ x \in \Omega, \ \ &\{g_i(x) \le 0, \ \ i = 1,2,\dots,m\} \end{aligned} \tag{5.6}$$

where Ω is a convex closed subset of $\Re^n$, *f(x)* and $\{g_i(x)\}$ are DC functions on $\Re^n$.

The first step is to perform a change of variable of this DC problem in order to transform it into a more desirable form: the canonical DC problem [Tuy, 85]. A canonical DC problem consists of the minimization of a linear objective function where all the constraints are convex functions except at most one which is concave. Notice that using the result 5.1, the set of all the convex constraints can be simplified and written as a single convex constraint.

Definition 5.2: A canonical DC problem is a global optimization problem of the form:

$$\begin{aligned} &\text{minimize } \ c^T x \\ &x \in \Omega, \ \ g(x) \le 0 \end{aligned} \tag{5.7}$$

where Ω is a convex closed subset of $\Re^n$, *c* is a constant vector of $\Re^n$, and *g(x)* is a concave function.

It may seem that the definition 5.2 drives more restrictions on the type of problems than definition 5.1. In other words it may seem that canonical DC problems form a subset within DC problems. However, this is not so. The next results states so:

Result 5.4: Any DC problem can be transformed into an equivalent canonical DC problem.

Proof 5.4: The problem described by (5.5) can be equivalently written in the following manner:

$$\begin{gathered} \text{minimize} \quad \omega \\ x \in \Omega, \ \ \{g_i(x) \le 0, \ \ i = 1,2,\dots,m\}, \ f(x) - \omega \le 0 \end{gathered} \tag{5.8}$$

which is the minimization of a linear target function with a set of convex constraints ($x \in \Omega$) and a set of $m+1$ DC constraints. Following Result 5.3, these $m+1$ DC constraints can be converted into a set of $m+1$ convex constraints, and a set of $m+1$ reverse convex constraints. Problem (5.6) is therefore transformed into the minimization of a linear target function with a set of convex constraints ($m+1$ & $x \in \Omega$) and a set of $m+1$ reverse convex constraints.

Following Result 5.1 all the convex constraints can be converted into a single convex constraint, and following Result 5.2 all the reverse convex constraints can be converted into a single reverse convex constraint.

Therefore, the original DC minimization problem (5.6) is transformed into the minimization of a linear target function with a single convex constraint and a single reverse convex constraint, which is stated as problem (5.7): the canonical DC problem.

The transformation can be viewed as a trade-off between the target function to be optimized and the domain over which the optimization is processed. The target from being a DC function is simplified into a linear function, but the search space is enlarged: its

dimension is increased by $m+1$ resulting from the introduction of the $m+1$ independent variables necessary to convert the $m+1$ DC constraints into a single pair of convex and reverse convex constraints.

5.2 Thoai's algorithm.

Consider a canonical DC problem of the form:

$$\begin{gathered}\text{minimize} \quad c^T x \\ x \in \Re^n, \;\; h(x) \le 0, \;\; g(x) \le 0\end{gathered} \tag{5.9}$$

where c is a vector of $\Re^n$, $h(x)$ a continuous convex function and $g(x)$ a continuous concave function. In the case there are more than one convex or concave contraints, they can be combined into a single one using Result 5.1 or Result 5.2.

Before all, the following notations need to be summarized:

- T is the convex set defined by the convex constraint: $T = \{x \in \Re^n | h(x) \le 0\}$.

- Consider a set of points $\aleph$ and a general function $\psi(x)$, then argmin$\{\psi(x) | x \in \aleph\}$ is the solution among all the points from $\aleph$ of the minimization of $\psi(x)$.

- $\partial h(x)$ is the gradient of the function $h(x)$ at the location x.

- $V(S)$ is the set of all vertices of a polyhedral convex set S.

- k is the iteration counter, and the corresponding index.

- Superscripts are indices: z^k is the value at iteration k of a point z.

Now Thoai's algorithm is presented. First a general flowchart is shown for laying out the program, then explanations are provided, and finally a precise step by step implementation is given.

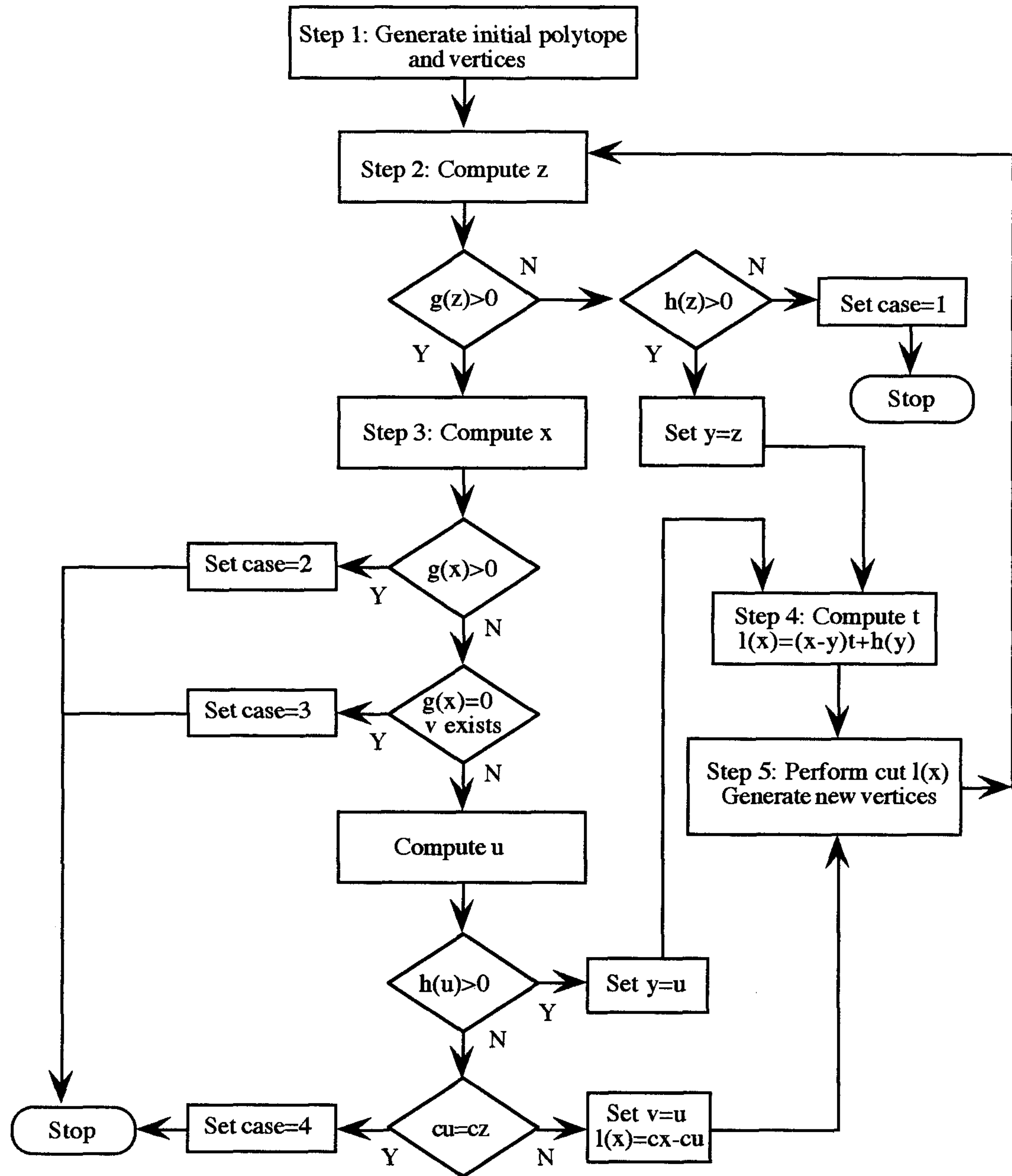

Figure 5.4: Flowchart of the minimization algorithm.

This algorithm has a simple structure in terms of program flow. Step 1 is an intialization step. Step 2 is a minimization step, the result of which indicates whether the process should be continued either along step 3 or step 4. Both of those determine a required cut that is performed by step 5, which in turn branches back to step 2.

Several situations can occur while running this program. It can either finish and exit by one of the *case* points, or it can get into an infinite loop. In the situation it finishes after a finite time, the *case* variable is set to various values the meanings of which are the following:

case=1: The solution is the current point z^k.

case=2: The problem has no solution.

case=3: The solution is the previously computed point v.

case=4: The solution is the current point u^k.

In the case the program gets itself into an infinite loop, there are two possible situations:

* This means that the solution is an accumulation point. In other words, the estimate of the solution is closer and closer to the true solution at each iteration without ever reaching it. However, by putting a required tolerance ε, the infinite loop may be detected by comparing two successive estimates of the solution, and the execution may then finish.

* The program cycles through all the possible solutions: Assuming there are more than a single global minimum (i,e, several minima that are not local), then a sequence $\{ z^k \}$ is generated by the algorithm. Everyone of these points which also satisfy $g(z^k)<0$ is one of the global solutions.

For more information, and proofs of these various situations, looking at the complete algorithm [Thoai, 1988] is recommended.

Now that the overall structure and flow of the algorithm have been presented, some of the individual tasks need more explanation.

Twice throughout the program a minimization has to be performed:

$$z^k \in \arg\min\{c^T x | x \in V(S^k)\} \tag{5.10a}$$

$$x^k \in \arg\min\{g(x) | x \in V(S^k)\} \tag{5.10b}$$

Either the linear objective function or the reverse convex constraint have to be minimized among the set of vertices. The most efficient way to achieve this task is to keep the finite set of vertices in a structure that stores also the values of both these functions for each vertex. In addition, both minima can be stored too. Whenever a new vertex is appended to the list, its corresponding function values can be computed and stored. They can be compared to the stored minima and if one of them is less than the current minimum it becomes the new minimum. Proceeding in this manner makes the minimization steps very simple: the algorithm just has to go and fetch the minimum value that is strored as a variable in a lookup table.

Another possible difficulty is the computation of u^k during step 3:

$$u^k \in \{x | g(x) = 0\} \cup [z^k, x^k] \tag{5.11}$$

However, this task is nothing more than solving a convex equation: The solution is the intersection of the line segment $[z^k, x^k]$ with the set of points $g(x)=0$. The line segment can be parameterized with a single variable λ and the equation is reduced to solving

$$g(\lambda z^k + (1-\lambda)x^k) = 0, \qquad 0 \leq \lambda \leq 1 \tag{5.12}$$

where λ is the single variable. Therefore, this reduces to solving a one-dimensional convex problem. A Newton or bisection method works perfectly well.

In step 4, the computation of $t^k \in \partial h(x)$ involves the gradient $\partial h(x)$ of *h(x)* and may be tricky. There are two solutions. If the analytical formulation of *h(x)* is known and if its derivative can be computed, then it should be implemented in the program, and finding the derivative is just a function evaluation. In the case that *h(x)* is not known analytically or that its derivative can't be directly implemented, then a numerical differentiation needs to be used. However, the required accuracy is such that this is not a major problem.

At last, step 5 presents the difficulty of performing a cut that involves a new constraint, resulting new vertices, possibly dropping redundant constraints and removing past vertices, and keeping track of the minimum of both the linear target function and the reverse convex constraint. This is the most difficult task: Horst's specific algorithm for that purpose, and more are presented in the next section.

A complete step by step implementation of the algorithm is now shown:

Step 1: Generate a polytope S^1 such that $S^1 \supset T$.
Compute $V(S^1)$.
Set index $k \leftarrow 1$

Step 2: Compute $z^k \in \arg\min\{c^T x | x \in V(S^k)\}$
If $g(z^k) > 0$, then go to Step 3.
Else,
If $h(z^k) \leq 0$, set $Case \leftarrow 1$, then stop.
Else, set $y^k \leftarrow z^k$, then go to Step 4.

Step 3: Compute $x^k \in \arg\min\{g(x) | x \in V(S^k)\}$.
If $g(x^k) > 0$, Set $Case \leftarrow 2$, then stop.

If $\{g(x^k)=0$ and $\exists v$, defined below, from a previous iteration $\}$,

set $Case \leftarrow 3$, then stop.

Otherwise,

Compute the point u^k of intersection of the set $\{x \mid g(x)=0\}$ and the line segment $[z^k, x^k]$. Notice that the solution is unique because $g(x)$ is convex. Any classical descent method can solve this step.

If $h(z^k) \leq 0$, set $y^k \leftarrow z^k$, then go to Step 4.

If $\{h(u^k) \leq 0$ and $c^T u^k = c^T z^k\}$, set $Case \leftarrow 4$, then stop.

Otherwise, set $v \leftarrow u^k$, $l(x) = c^T x - c^T u^k$, then go to Step 5.

Step 4: Compute a vector $t^k \in \partial h(y^k)$.

Generate a cutting plane: $l(x) = (x - y^k)t^k + h(y^k)$

Go to Step 5.

Step 5: Generate a new polytope $S^{k+1} = S^k \cap \{x \mid l(x) \leq 0\}$

Compute $V(S^{k+1})$

Set $k \leftarrow k+1$ and go to Step 2.

5.3 Horst's complementary material and keeping track of minima.

All the independent points in Thoai's algorithm have been developed except what was called the step 5. This section comes at the point when a required new cut has been determined. The purpose of step 5 is to perform this cut.

The state of the algorithm at the entrance of step 5 consists of the following:

- A set of linear constraints that form a convex polytope P.
- The set of all the vertices of this polytope P.
- The minimum among these vertices of the target function cx.
- The minimum among these vertices of the reverse convex constraint $g(x)$.
- An extra linear constraint that corresponds to the required new cut.

The desired state after the completion of step 5 is:

- An updated set of linear constraints that for a new smaller convex polytope P'.
- The updated set of all the vertices of this new polytope P'.
- The minimum among these new vertices of $g(x)$ and cx.

In order to achieve this transformation, the manner to proceed here is to perform the following tasks in this particular order:

- Determine which constraints become redundant due to the new cut and can be dropped out.

- Remove from the set of vertices the elements that were corresponding to these past dropped constraints.

- Create new vertices corresponding to the intersection of the new cut with previous constraints that are still valid.

- Keep track of the two minima as vertices are created.

Figure 5.5 shows an example that illustrates what is happening when a new cut $h(x) \leq 0$ is introduced.

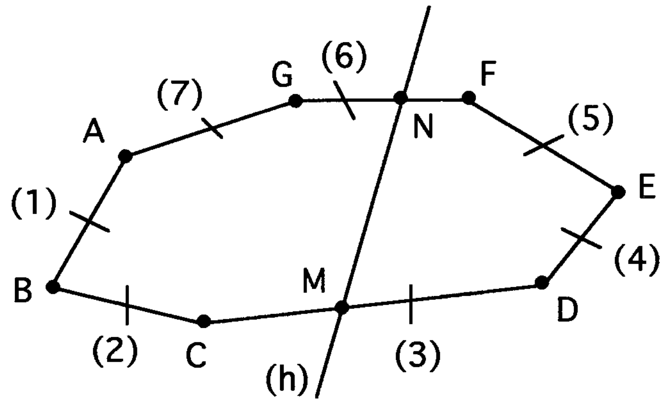

Figure 5.5. Example of a cut (h) over a polytope.

Assuming that the cut is toward the right, then the constraints (4) and (5) become what is called redundant and can be eliminated. Vertices D, E, and F are removed as well. The new boundary of the polytope with the new constraint (h) is defined by the two new vertices M and N that must be created. After the cut is performed, the polytope is defined by the constraints (1), (2), (3), (6), (7), and (h). Its vertices are A, B, C, G, M, and N.

The first part consists of checking for any redundant constraint. A redundant constraint can been seen as a "weaker" constraint than the new cut. It has therefore been superceded by the new cut and can be removed. How is it possible to measure if one constraint is weaker than another? Following Horst [Horst, 1988] the following method can be used:

- Consider $V(P)$ the set of all the vertices of the polytope P.
- Consider a cut generated by the constraint $h(x) \leq 0$.
- Construct the set $V^{-}(P) = \{u \in V(P) : h(u) < 0\}$.
- Then a constraint $g(x) \leq 0$ is redundant for P relative to $h(x)$ if and only if:

$$\forall u \in V^{-}(P), \qquad g(u) < 0 \tag{5.13}$$

The procedure to identify all the redundant vertices when a new cut $h(x)$ is performed is therefore the following:

- Construct the set $V^{-}(P) = \{u \in V(P): h(u) < 0\}$.

- Loop through all the existing constraints to check which ones are redundant. Notice that the set $V^{-}(P)$ depends only on the new cut and is therefore the same for all the constraints to be checked.

- All the constraints that have been determined to be redundant are dropped.

Once redundant constraints have been dropped, the corresponding vertices must be removed as well. This task is fairly easy if the vertex database has been constructed correctly. As stated before, the minima of two functions must be saved in order to recall them as needed without recomputing them. Similarly, once a vertex is created, its supporting vertices must be saved as well. There are as many supporting constraints as there are dimensions in the working space. The vertex database must therefore have the following data for each entry, assuming that the dimension of the space is N:

- Real valued vector of size N for the position of the vertex.

- Integer valued vector of size N for the index of supporting constraints.

- Value of the target function cx at the vertex location.

- Value of the reverse convex constraint $g(x)$ at the vertex location.

- Flag to indicate if the vertex is active, or was active and has been removed.

Assuming such a database is created and maintained, then when a constraint is dropped, the database has to be scanned for vertices that had this constraint as one of their supports, and their flag can be switched from active to inactive.

At this point, the half space on the "wrong" side of the hyperplane created by the new cut $h(x) \le 0$ has effectively been removed from the algorithm environment. The polytope containing the feasible set has been reduced in size. The next step is to create and classify the new boundaries between the previous polytope and the new cut. The constraint is known, but the vertices must be created.

Assuming a space of dimension *N*, a vertex can be viewed as the solution of *N* simultaneous linear equations, each being the boundary of a cut (i.e. if the cut constraint is $h(x) \le 0$, then the equation is $h(x)=0$). It can also be viewed as the intersection of edges of the polytope *P*. Every edge connects two vertices. These two vertices and all the points in between that lie on the edge can be each described by a set of *N* equations. However, all these points have *N-1* equations in common.

It can be assumed that a new cut always intersects every edge of the polytope. In the case that an edge is parallel to the cut, then the intersection is assumed at infinity. Because of the way cuts are created in Thoai's algorithms, an edge can't be included in the cut. Therefore, the intersection point always exists and is unique following the previous assumptions, and it is described by a set of *N* equations: *N-1* from the edge and one from the cut.

When a cut is introduced, the set of existing vertices can be split into two classes: the ones that satisfy the cut (previously called the set $V^-(P)$), and the ones that are on the "wrong" side of the cut (called the set $V^+(P)$). It is clear that the new boundary lies in between these two sets. In order to compute the vertices of the new polytope, it is sufficient to check all the edges which are defined by a couple of points one from each set, and compute the intersection with the new cut. As stated earlier, this defines a unique point that is a new vertex. The edges of interest can be found by checking vertices from $V^-(P)$ and $V^+(P)$ that have *N-1* constraints in common. As stated earlier, these *N-1* equations are the equations defining the edge.

Once all the new vertices have been found, the vertex database must be updated. This includes computing the two functions (target and reverse convex) for each vertex, check if they happen to be a new minimum, and filling all the entries in the database.

This concludes step 5 of the algorithm.

5.4 Conclusion of this section: Ready to use Thoai's algorithm.

The independent tasks for solving a DC problem have been presented, along with Thoai's algorithm. All the difficult points in that algorithm have been exposed and practical implementations have been suggested. The behavior of the algorithm in terms of termination points or infinite loop have been described. The algorithm can now successfully be implemented and used for the purpose of minimizing a neural network output.

Chapter 6:
Global Minimization of a Neural Network output via D.C. Programming

This chapter is the bridging chapter of this project. What was presented earlier is all put together here: the problem of globally minimizing the output of a neural network is studied. The outer approximation algorithm for solving a DC problem (Difference of Convex) introduced by Thoai is used. The target neural network is a MISO (Multi Input Single Output) pseudo multilayer feedforward network.

The pseudo multilayer architecture was introduced in Chapter 3. In Chapter 5 an algorithm for DC programming was presented. In this chapter an interface between this type of neural networks and DC programming is introduced. The neural network is programmed to approximate some function or model some phenomenon. The interface translates it into a DC problem that is solved using the optimization algorithm presented earlier. By minimizing the DC problem, the neural network is thereby optimized as well, which implies that the original problem modeled by the neural network is in turn optimized.

In this chapter, a review of previous studies related to both neural networks and global optimization is first presented.

Then, the relation between a single neuron and DC programming is introduced. As a direct application, this allows to globally minimize the output of a single layer neural network. First the single input case is considered, then the multi input case. The complete procedure is fully described.

The next presentation is an adaptation and enhancement of the previous single layer network case just mentioned to accommodate the pseudo multilayer architecture described in Chapter 3. Again, the complete procedure is described.

For each section, algorithms and methods are illustrated by concrete examples from various application areas. In particular the problem of programming a neural network using another neural network is developed in the final section of this chapter.

6.1 Neural networks and global optimization

This first section reviews the relationship between neural networks and global optimization. There are two aspects of this relationship: the neural network can be viewed either as the tool or as the object. In the case of the tool, the neural network is a model of the performance index of an object, or a model of the object itself to be optimized. By minimizing the neural network response, the performance index it represents is minimized, thereby optimizing the original object. On the other hand, in the case the neural network is viewed as an object, a global optimization algorithm can be used to perform the most difficult and delicate task of programming the neural network to match some desired behavior.

6.1.1 A neural network as an optimization tool

Neural networks have been used extensively to perform optimization tasks. However, it must be differentiated between various situations. Frequently, a particular neural network, modeling a particular system, is designed to perform a required optimization. In no way can this kind of situation be described as a generic approach for solving optimization problems: the reason is that the optimization algorithm chosen to perform the minimization depends on the network, which in turn depends on the problem itself. For example a robot arm may be modeled by a neural network and the optimization of the joint parameters may be performed by minimizing the network output.

However, in this present project, general solutions and algorithms to be used in various fields and applications are of larger interest. Unfortunately, only few results have been reported for such general approaches. Several studies have been performed towards this direction. A brief summary of those comes here.

The most complete work of a global optimization task solved by a neural network deals with quadratic minimization. The network used is a Hopfield network of the form [Hopfield, 1982]:

$$\dot{\underline{x}} = \underline{u} + W\underline{f}(\underline{x}) - A\underline{x} \tag{6.1}$$

where $\underline{x}$ is the network state vector, $\underline{u}$ the network external input vector, W the network interconnection matrix, and A a positive diagonal matrix representing the passive decay rate of the state vector. $\underline{f}(.)$ is a piecewise linear activation function as shown in figure 2.3. Clearly, this network is built around a recurrent architecture.

The Hopfield network is known for its ability to minimize an energy function E defined as [Hopfield, 1984]:

$$E = -\frac{1}{2}\sum_{i=1}^{i=N}\sum_{j=1}^{j=N} w_{ij} f(x_i) f(x_j) - \sum_{i=1}^{i=N} \alpha_i f(x_i) \tag{6.2}$$

where α_i is the threshold of the ith neuron. A quadratic minimization problem is written as:

$$J(\underline{v}) = \frac{1}{2}\underline{v}^T Q \underline{v} - \underline{v}^T \underline{u} \tag{6.3}$$

where Q is a positive definite matrix. By appropriate changes of variables and variable assignments, a network of the form (6.1) can be constructed so that the energy function E that it is being minimized corresponds to the quadratic minimization performance index J.

These variable transformations exist and the network was shown to solve quadratic minimization problems. Shi and Ward [Shi, 1990] were the first to write a procedure to perform this task. Sudharsanan and Sundareshan [Sudharsanan, 1991] then ameliorated the network in order to facilitate preconditioning, improve and bound the rate of convergence. Bouzerdoum and Pattison [Bouzerdoum, 1993] further generalized the procedure.

The main advantage of solving this problem is that it produces the solution of a global optimization problem in a finite amount of time, with known convergence rate. However, it falls short of being a general global optimization method for the simple reason that a quadratic minimization is an optimization task with particularly well behaved topological properties.

For example the solution of a linear matrix equation of the form $A\underline{x}=\underline{b}$ where A is a rectangular matrix can be turned into an optimization problem of the form:

$$\begin{aligned} J(\underline{x}) &= (A\underline{x}-\underline{b})^T(A\underline{x}-\underline{b}) \\ &= \underline{x}^T A^T A\underline{x} - \underline{x}^T A^T \underline{b} - \underline{b}^T A\underline{x} + \underline{b}^T\underline{b} \end{aligned} \tag{6.4}$$

This can easily be turned into a problem of the form (6.3) since the term $\underline{b}^T\underline{b}$ is a constant and therefore drops out from an optimization point of view. It must be realized that the initial problem $A\underline{x}=\underline{b}$ is known to be a very simple optimization problem.

Even though solving a quadratic minimization problem by a neural network is a very nice result, it is clear that it lacks generalization by the fact that the objective function is too constrained.

However, there are unfortunately very few other complete studies of a problem independent method for solving a global optimization task using a neural network.

6.1.2 Programming a neural network with an optimization method

As it was stated earlier, there are two ways to combine optimization and a neural network: either the network is the tool or it is the task. In the previous section the case where the neural network was the optimization tool was considered. In this section the neural network is "optimized" using an optimization method.

What does "optimizing a neural network" mean? In this particular context, a neural network is said to be optimized if its actual input-output behavior coincides with the desired one. This is clearly related to the "programming" of the network.

In the first section of this chapter, it was recalled that there exist several methods for programming a neural network and that the most popular among those is the so-called backpropagation. There are two major problems with backpropagation. First, the algorithm is very slow, the speed scales very poorly with the size increase of the network (the increase in term of layers is worse than in the increase in terms of neurons). The second reason is that there is no guarantee that the eventual solution is the "best" solution.

For these reasons people have first looked into improving the original algorithm. This has been performed with quite some success. There are now several known "tricks" to improve either the speed of convergence or the chance for reaching a better final point. These are well documented in Korn's book [Korn, 1991]. A brief summary may include adaptive gain values, added moments, multi random restarts, and modified derivatives for the activation functions.

The alternative to improving the backpropagation was to look into using global optimization algorithms to replace the backpropagation techniques as a whole. Three areas have been looked into: simulated annealing, genetic algorithms, and deterministic methods.

Good progress, although no perfect results yet, have been made in these three areas. To match the scope of this project, only deterministic methods are surveyed here. The most general and encouraging success was obtained using a "tunneling" method.

The tunneling methods were introduced by Levy and Montalvo [Levy, 1985]. They are composed of a sequence of cycles, each of which has two phases: a local minimization followed by tunneling. The local minimization searches for the closest local minimum. When found, a pole in the performance index is positioned at that location. Then starts the tunneling which consist of going "away" from that pole. The search then continues toward another local minimum which is lower than the previous one. Eventually, the global minimum is found. Among the various tunneling methods, the Terminal Repeller Unconstrained Subenergy Tunneling (TRUST) was introduced recently [Cetin, 1993a]. In this approach, the optimization is formulated as the solution of a deterministic dynamical system which incorporates a novel subenergy tunneling functional and terminal repellers. This algorithm guarantees to find the global minimum of a one-dimensional problem, and works "most of the time" in multidimensional cases.

The programming of a neural network can be put into the form of solving a global optimization problem. This is what Cetin and al. did [Cetin, 1993b]. They converted this task into a format well suited for TRUST, and solved the resulting optimization problem following the manner described earlier. The results are promising. However, the programming of a neural network is a multidimensional optimization process, and as it was stated earlier, TRUST only guarantees global convergence in the one-dimensional case. Therefore, the programming of a neural network only works "most of the time".

Because of this problem with the multidimensional search, it is not known at this point if tunneling is more efficient than multiple-random-restart improved back propagation methods after all. They may be more efficient than tunneling in the context of neural networks, because the tunneling phase itself is rather inefficient and close minima can be overlooked easily. The tunneling phase follows the steepest descent trajectory and must take quite small steps, otherwise minima lying on the trajectory are missed. Because the regions around a minimum are often very flat, the repeller effect in case of the TRUST algorithm is dominant and the trajectory approximates a straight line away from the minimum. The function evaluations used for tunneling may be better used for a random exploration of the minimum's neighborhood.

It can be concluded after looking at the TRUST analysis that there is interest for looking into global optimization methods for solving the problem of programming neural networks. Global optimization methods are a promising alternative to backpropagation since this algorithm will always lacks hill climbing capability and efficiency, even if partial improvements such as speed increase and global performance are found.

6.2 Minimizing a single layer neural network output via DC programming.

In this section, the study of minimizing the output of a single layer neural network is presented. First, the relation between neurons and DC problems is introduced: the nonlinear activation function is a DC function, and one DC decomposition of interest is shown. Once a neuron activation function is converted into a DC formulation, writing the neuron input output behavior in a DC form immediately follows. The next enhancement is to write a single layer neural network as a DC problem, first with a single input, then with multiple inputs. All these steps are presented in this order in the coming sections.

6.2.1 Neurons and DC functions.

The first task is to convert a neuron activation function into a DC form. It was recalled in Chapter 4 that any continuous function is a DC function. It is therefore no surprise to state that a neuron activation function is a DC function. However, it was also recalled that even though a function is known to be DC, it is not necessarily easy to find a decomposition for it. Its existence is known, but its construction is not. However, for the problem of decomposing a neuron activation function into a DC form, a construction technique is readily available. It is now presented.

It was recalled in Chapter 2 that a neuron activation function $f(x)$ has some given characteristics and properties: $f(x)$ has an odd symmetry about the point $(0,f(0))$, $f(x)$ is monotonic, and $f(x)$ saturates at infinity. These are well known properties. However, another property which is not well publicized was also stated: $f(x)$ is convex for a negative argument, and concave for a positive argument. The reason that this last property is not often used is that convexity theory and neural networks are very rarely put together. The change from convex to concave when x becomes positive implies that the origin $x=0$ is an inflexion point.

Considering $f(x)$ being an activation function. The two functions $f_1(x)$ and $f_2(x)$ are introduced and defined as:

$$
\begin{aligned}
f_1 := & \begin{cases} \forall x \le 0 & f_1(x) = f(x) \\ \forall x > 0 & f_1(x) = f'(0) \cdot x + f(0) \end{cases} \\
f_2 := & \begin{cases} \forall x \le 0 & f_2(x) = 0 \\ \forall x > 0 & f_2(x) = f'(0) \cdot x + f(0) - f(x) \end{cases}
\end{aligned}
\tag{6.5}
$$

where $f'(0)$ is the value of derivative of $f(x)$ at the origin.

Result 6.1: The pair of functions $\langle f_1(x), f_2(x)\rangle$ as defined above form a DC decomposition of the activation function *f(x)*.

Proof 6.1: It is necessary to prove that both functions $f_1(x)$ and $f_2(x)$ are convex, and that their difference is uniformely equal to the activation function *f(x)*.

* $f_1(x)$ is a convex function:

For a negative argument, $f_1(x)$ is uniformely equal to *f(x)* which is known to be convex for negative arguments.

For a positive argument, $f_1(x)$ is a linear function, therefore its curvature is zero, and hence it is convex.

At the origin, $f_1(x)$ as defined above is continuous $f_1(0^-) = f_1(0^+) = f(0)$, has a continuous first derivative $f_1'(0^-) = f_1'(0^+) = f'(0)$, and a zero second derivative.

Therefore $f_1(x)$ is a convex function over $\Re$.

* $f_2(x)$ is a convex function:

For a negative argument, $f_2(x)$ is a constant function, therefore its curvature is zero, and hence it is convex.

For a positive argument, $f_2(x)$ is uniformely equal to the difference between a linear function and *f(x)*. The linear function is known to have no curvature, *f(x)* is known to be concave for a positive argument, therefore *-f(x)* is convex which implies that $f_2(x)$ is convex too over the same domain..

At the origin, $f_2(x)$ as defined above is continuous and has a zero value: $f_2(0^-) = f_2(0^+) = 0$, has a continuous first derivative which is also zero: $f_2'(0^-) = f_2'(0^+) = 0$, and a zero second derivative.

Therefore $f_2(x)$ is a convex function over $\Re$.

* The difference between $f_1(x)$ and $f_2(x)$ is uniformely equal to *f(x)*.

Following the two definitions for $f_1(x)$ and $f_2(x)$, the difference between $f_1(x)$ and $f_2(x)$ can be written as:

$$f_1 - f_2 :$$
$$\begin{cases} \forall x \le 0 & f_1(x) - f_2(x) = f(x) - 0 = f(x) \\ \forall x > 0 & f_1(x) - f_2(x) = \left(f'(0) \cdot x + f(0)\right) - \left(f'(0) \cdot x + f(0) - f(x)\right) = f(x) \end{cases} \tag{6.6}$$

It is clear that the difference between $f_1(x)$ and $f_2(x)$ is indeed equal to *f(x)*.

Therefore, the pair $\langle f_1(x), f_2(x) \rangle$ indeed forms a DC decomposition of the activation function *f(x)*.

As an example, the DC decomposition of the *Atan(x)* activation function is shown in figure 6.1. In that particular example, the two convex componenents are:

$$\begin{aligned} f_1 := & \begin{cases} \forall x \le 0 & f_1(x) = \mathrm{Atan}(x) \\ \forall x > 0 & f_1(x) = x \end{cases} \\ f_2 := & \begin{cases} \forall x \le 0 & f_2(x) = 0 \\ \forall x > 0 & f_2(x) = x - \mathrm{Atan}(x) \end{cases} \end{aligned} \tag{6.7}$$

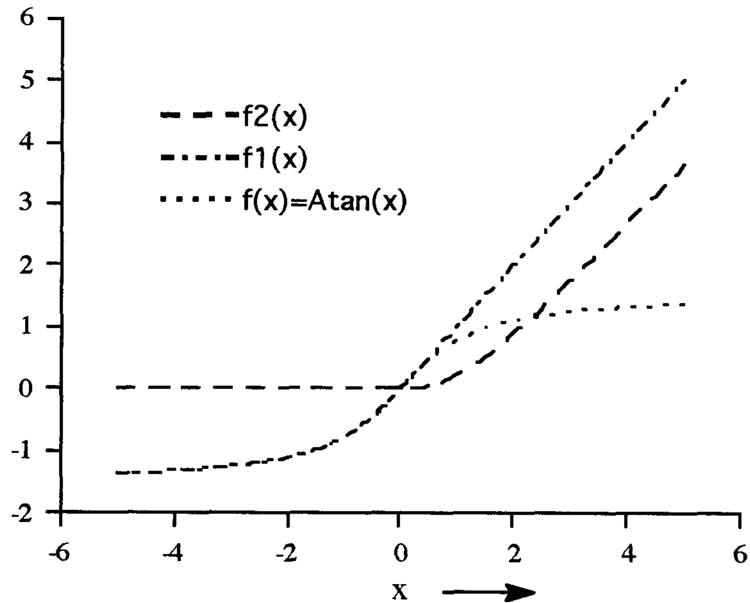

Figure 6.1. DC decomposition of the activation function *Atan(x)*.

In this manner other activation functions may be decomposed into DC components as well. However, the threshold activation function shown on figure 2.3 can't. The reason being that the threshold function is not continuous and that its derivative is not defined at the origin. However, the threshold function can be assumed to be the limit as ε goes to zero of the *Atan(x/ε)* function for which the DC decomposition exists. Therefore the DC decomposition of the threshold function is the limit of the DC decomposition of *Atan(x/ε)* as ε goes to zero.

Another activation function was introduced in Chapter 3. It was called the piecewise linear activation function $f^{pw}(x)$. Because this particular function is used extensively in what is coming up next, it is necessary to explicitely write its DC decomposition. The function was defined as:

$$f^{pw}(x) = \{(x < 0, 0 \le x < 1, 1 \le x) \rightarrow (0, x, 1)\} \tag{6.8}$$

Because it doesn't follow the regular assumptions for DC functions (symmetry and convexity with respect to the origin), it is not possible to use the DC decomposition rule (6.5) explained earlier. However, a DC decomposition for (6.8) exists nevertheless.

At this point the usual "ramp" function $r(x)$ must be recalled:

$$r(x) = \{(x < 0, 0 \le x) \rightarrow (0, x)\} \tag{6.9}$$

The DC decomposition of $f^{pw}(x)$ is now presented:

Result 6.2: The DC decomposition of $f^{pw}(x)$ is $\langle r(x), r(x-1) \rangle$.

Proof 6.2: Following the definition of $r(x)$ and of convexity, it is clear that $r(x)$ is a convex function. Because shifting doesn't affect convexity, this implies that $r(x-1)$ is a convex function as well.

Because $r(x-1)$ is a shift of $r(x)$ it can be written as:

$$r(x-1) = \{(x < 1, 1 \le x) \rightarrow (0, x-1)\} \tag{6.10}$$

The difference between $r(x)$ and $r(x-1)$ can be written as:

$$\begin{aligned} r(x) - r(x-1) &= \{(x < 0, 0 \le x) \rightarrow (0, x)\} - \{(x < 1, 1 \le x) \rightarrow (0, x-1)\} \\ &= \{(x < 0, 0 \le x < 1, 1 \le x) \rightarrow (0 - 0, x - 0, x - (x-1))\} \\ &= \{(x < 0, 0 \le x < 1, 1 \le x) \rightarrow (0, x, 1)\} \end{aligned} \tag{6.11}$$

This can be recognized as what was earlier called $f^{pw}(x)$.

The difference between *r(x)* and *r(x-1)* therefore is the desired $f^{pw}(x)$ function. Hence, this concludes the proof 6.2.

$f^{pw}(x)$ along with its DC decomposition $\langle r(x), r(x-1) \rangle$ are shown in figure 6.2.

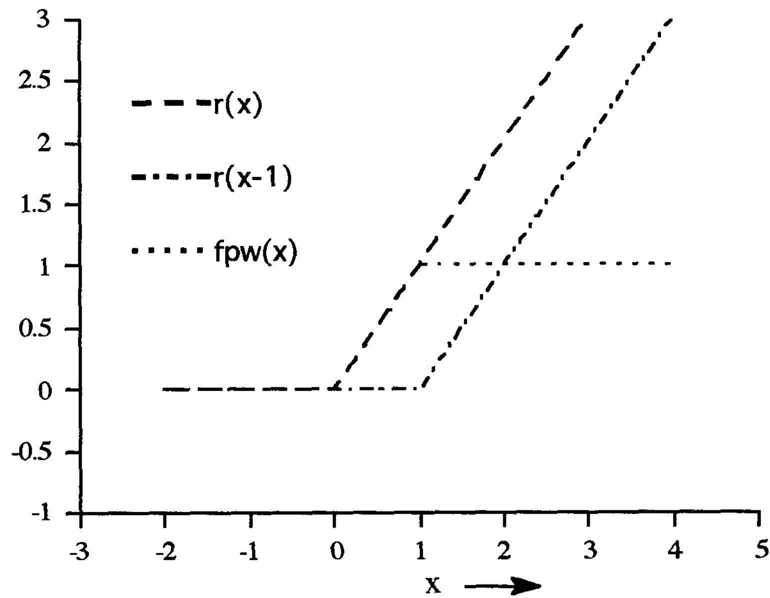

Figure 6.2. DC decomposition of the activation function $f^{pw}(x)$.

Up to this point, the nonlinear activation of a neuron has been shown to be expressable as a DC function. For two such functions of interest, namely *Atan(x)* and $f^{pw}(x)$, an explicit DC decomposition was also given. The next step consists of relating a whole neuron to DC programming, not only its activation function.

A static neuron input output relationship was given earlier in equation (2.2). With an additional bias and output weight, the relation becomes,

$$y = w_2 \cdot f(\underline{w_1} \cdot \underline{u} + b) \tag{6.12}$$

where y is the neuron output, $\underline{u}$ is the neuron input vector, b is the neuron bias, $\underline{w_1}$ the input weight vector, w_2 the output weight vector, and $f(.)$ the activation function.

Before relating this relation to DC programming, yet another result on convexity theory is needed.

Result 6.3: If $g(x \in \Re)$ is convex (or concave), then $\forall \underline{w} \in \Re^n, \forall b \in \Re, g(\underline{w} \cdot \underline{u} + b)$ is convex (or concave) along every direction of the vector $\underline{u}$.

Proof 6.3: Assuming a function $g(x)$ from C_2 to be convex (or concave) for a real scalar argument. Then following the definition, its second derivative $g''(x)$ is positive (or negative). The second derivative of $g(\underline{w} \cdot \underline{u} + b)$ with respect to any component u_i of the vector $\underline{u}$ is:

$$\frac{\partial^2}{\partial u_i^2}\left(g(\underline{w} \cdot \underline{u} + b)\right) = w_i^2 g''(\underline{w} \cdot \underline{u} + b) \tag{6.13}$$

This is clearly positive (or negative). Therefore if the scalar function $g(x)$ is convex (or concave) in the space $\Re$, then the vector function $g(\underline{w} \cdot \underline{u} + b)$ is convex (or concave) in the space $\Re^n$.

Earlier in this section it was shown that a neuron activation function $f(x)$ can be written $f(x) = f_1(x) - f_2(x)$ where both $f_1(x)$ and $f_2(x)$ are convex functions. Therfore the neuron input output relation (6.12) can be now written as

$$\begin{aligned} y &= w_2 \cdot \left(f_1(\underline{w_1} \cdot \underline{u} + b) - f_2(\underline{w_1} \cdot \underline{u} + b)\right) \\ &= w_2 \cdot f_1(\underline{w_1} \cdot \underline{u} + b) - w_2 \cdot f_2(\underline{w_1} \cdot \underline{u} + b) \end{aligned} \tag{6.14}$$

Following the Result 6.3, the functions $f_1(\underline{w_1} \cdot \underline{u} + b)$ and $f_2(\underline{w_1} \cdot \underline{u} + b)$ are both convex in the space $\Re^n$. If w_2 is positive, then $w_2 \cdot f_1(\underline{w_1} \cdot \underline{u} + b)$ is convex and

$-w_2 \cdot f_2(\underline{w_1} \cdot \underline{u} + b)$ is concave. If w_2 is negative, then $w_2 \cdot f_1(\underline{w_1} \cdot \underline{u} + b)$ is concave and $-w_2 \cdot f_2(\underline{w_1} \cdot \underline{u} + b)$ is convex. Therefore the DC decomposition of a neuron input output relation is:

$$\begin{aligned} &* \ \left\langle w_2 \cdot f_1(\underline{w_1} \cdot \underline{u} + b), w_2 \cdot f_2(\underline{w_1} \cdot \underline{u} + b)\right\rangle \text{ if } w_2 \text{ is positive.} \\ &* \ \left\langle |w_2| \cdot f_2(\underline{w_1} \cdot \underline{u} + b), |w_2| \cdot f_1(\underline{w_1} \cdot \underline{u} + b)\right\rangle \text{ if } w_2 \text{ is negative.} \end{aligned} \tag{6.15}$$

This completes the presentation of relating a single neuron to DC programming. The next step is to consider a set of neurons: a neural network.

6.2.2 Minimizing a SISO single layer neural network.

In the previous section, the translation of a neuron input output relation into DC formulation was presented. In this section, the translation of a single input single output neural network into DC formulation is first introduced, followed by several illustrative examples of actual minimizations of mathematical functions.

Consider a single layer neural network with a scalar input u and a scalar output y. The network is represented by the following equation set:

$$\begin{aligned} \underline{x} &= \underline{w_1} \cdot u + \underline{b} \\ y &= \underline{w_2} \cdot f(\underline{x}) \end{aligned} \tag{6.16}$$

assuming there are N neurons, the values of which are represented by the state vector $\underline{x}$. The vectors $\underline{w_1}$ and $\underline{w_2}$ are again the input and output weight vectors, while $\underline{b}$ is the bias vector. All the vectors are of length N.

$f(x)$ is the neuron activation function. It has a DC decomposition $\langle f_1(x), f_2(x)\rangle$ that can be explicitely computed as explained before. Introducing this DC decomposition, the neural network equation set becomes:

$$\begin{aligned} \underline{x} &= \underline{w_1} \cdot u + \underline{b} \\ y &= \underline{w_2} \cdot \left(f_1(\underline{x}) - f_2(\underline{x})\right) \end{aligned} \tag{6.17}$$

Following what was said in the previous section, the set of neurons must be split into two disjoint sets $\aleph_p$ and $\aleph_n$: the neurons that have a positive output weight ($w_2>0$), and the ones that have a negative output weight ($w_2<0$):

$$\begin{aligned} &\aleph_p := \left\{x_i : w_{2_i} > 0\right\} \quad Card(\aleph_p) = N_p \\ &\aleph_n := \left\{x_i : w_{2_i} < 0\right\} \quad Card(\aleph_n) = N_n \\ &N_p + N_n = N \end{aligned} \tag{6.18}$$

From this last relation separating the neurons, and relation (6.15) regarding the DC decomposition of a single neuron, the neural network equation set becomes:

$$\begin{aligned} \underline{x} &= \underline{w_1} \cdot u + \underline{b} \\ y &= \sum_{x_i \in \aleph_p} w_{2_i} \cdot \left(f_1(x_i) - f_2(x_i)\right) + \sum_{x_i \in \aleph_n} \left|w_{2_i}\right| \cdot \left(f_2(x_i) - f_1(x_i)\right) \end{aligned} \tag{6.19}$$

It can be further reduced to:

$$
\begin{aligned}
y &= \sum_{x_i \in \aleph_p} w_{2_i}\left(f_1(w_{1_i}u + b_i) - f_2(w_{1_i}u + b_i)\right) + \\
&\quad \sum_{x_i \in \aleph_n} \left|w_{2_i}\right|\left(f_2(w_{1_i}u + b_i) - f_1(w_{1_i}u + b_i)\right) \\
y &= \left[\sum_{x_i \in \aleph_p} w_{2_i} f_1(w_{1_i}u + b_i) + \sum_{x_i \in \aleph_n} \left|w_{2_i}\right| f_2(w_{1_i}u + b_i)\right] - \\
&\quad \left[\sum_{x_i \in \aleph_p} w_{2_i} f_2(w_{1_i}u + b_i) + \sum_{x_i \in \aleph_n} \left|w_{2_i}\right| f_1(w_{1_i}u + b_i)\right] \\
y &= F_1(u) - F_2(u)
\end{aligned}
\tag{6.20}
$$

where $F_1(u)$ and $F_1(u)$ are both convex functions. Therefore (6.20) is a DC decomposition of the neural network originally written (6.16).

The procedure for converting a SISO single layer neural network into a DC formulation has now been presented. This decomposition will subsequently be used for performing a global minimization on the neural network output.

Consider a SISO function $\varphi(u)$ being modelled by an approximating neural network $\psi(u)$. Because this approximation is valid only over a finite domain $u \in [u_{\min}, u_{\max}]$, the input u must be constrained to this domain. The neural network can be transformed into a DC format as it was just shown. Therefore, globally minimizing $\varphi(u)$ is equivalent to solving the DC problem:

$$
\begin{aligned}
&\text{minimize } \; F_1(u) - F_2(u) \\
&-u < -u_{\min}, \;\; u < u_{\max}
\end{aligned}
\tag{6.21}
$$

This is clearly a DC programming problem of the form (5.6). However, it is simpler in the sense that only the target function is DC, the constraints are not, they are linear. In order to use the algorithm developed in Chapter 5, this problem must first be converted into a canonical problem of the form (5.7).

This is performed following what was described in Chapter 5. A variable ω is first introduced to replace the target DC function, and another constraint $F_1(u) - F_2(u) - \omega \leq 0$ is added. The next step consists of replacing each DC constraint by a pair of convex and reverse convex constraints. In this case there is a single DC constraint. This is performed by introducing yet another independent variable z and decomposing the single DC constraint $F_1(u) - F_2(u) - \omega \leq 0$ into the set $F_1(u) - \omega - z \leq 0$ and $z - F_2(u) \leq 0$. The last step consists of regrouping convex and reverse convex constraints together. The linear constraints can go either way since they are both convex and reverse convex. Following this route, the problem (6.21) is now replaced by a canonical DC problem for which the algorithm of Chapter 5 is perfectly suited.

The complete process of minimizing a SISO function using a neural network model followed by performing a DC program on the network is now illustrated by two examples.

Example 6.1: Consider the function $y = \varphi(u) = 4 + 0.4\cdot(u-5)^2 + \cos(8\cdot u)$. It is desired to find with an accuracy τ=0.001 its global minimum over the interval $u \in [0,10]$. This function is plotted on figure 6.3.

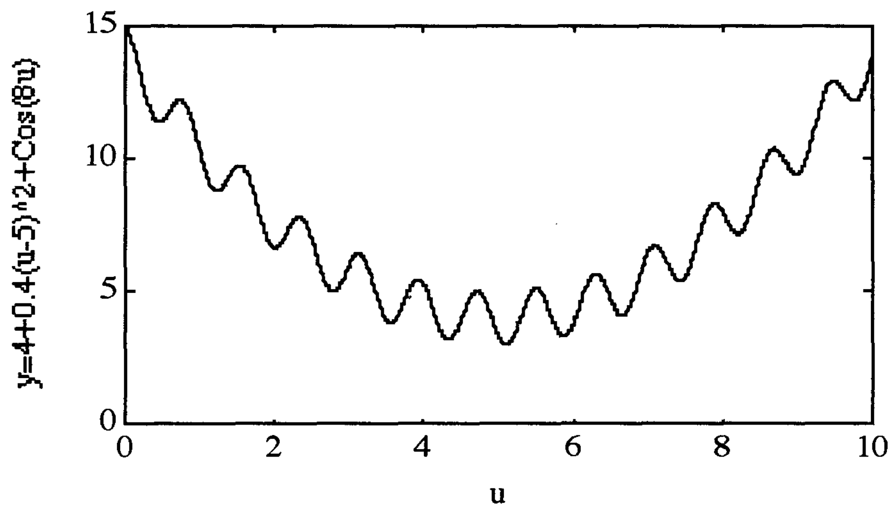

Figure 6.3. Target function for example 6.1.

The first step is to model $\varphi(u)$ by an approximating single layer neural network $\psi(u)$. This can be performed using the method described in Chapter 3. This function with the given accuracy and given interval requires 854 neurons for the approximation. Therefore the single layer neural network has 854 neurons.

The second step is to convert the neural network into a DC objective function. Following the method described earlier in this chapter, the neurons must be grouped into two sets: the ones with positive and those with negative output weights. This is done by looking at each neuron individually and building the two sets. In this example there are 421 neurons making up the "positive" set and 433 making up the "negative" set. Once the two sets are created, using formulation (6.20) the DC objective function is written.

The third step consists of converting the DC problem into a canonical DC problem. This part was just explained. Two extra variables ω and z are introduced. The convex and reverse convex constraints are grouped together.

The last part consists of running the algorithm which was explained in details in Chapter 5 and which is based on Thoai's algorithm. Once the algorithm has converged, the following is observed:

* The algorithm converged after 211 iterations.

* At the end there are 146 active vertices and 75 active constraints.

* The solution for the independent variables are: u=5.10, ω=3.005, and z=58.23.

It is clear that the solution of interest is u=5.10. This value is the solution of the minimization of the output of the neural network $\psi(u)$. Thereby it is the solution of the minimization of the initial function $\varphi(u)$. The algorithm was able to recognize the global minimum among the 13 local minima present over the interval of interest. This concludes the first example.

Example 6.2: This example is similar in nature to the first one. The target function is a little more complex. Consider the function $y = \varphi(u) = 4 - 0.1 \cdot e^{-(u-11.5)^2} + \cos(8 \cdot u)$. It is desired to find with an accuracy τ=0.001 its global minimum over the interval $u \in [0,15]$.

This function is plotted on figure 6.4.

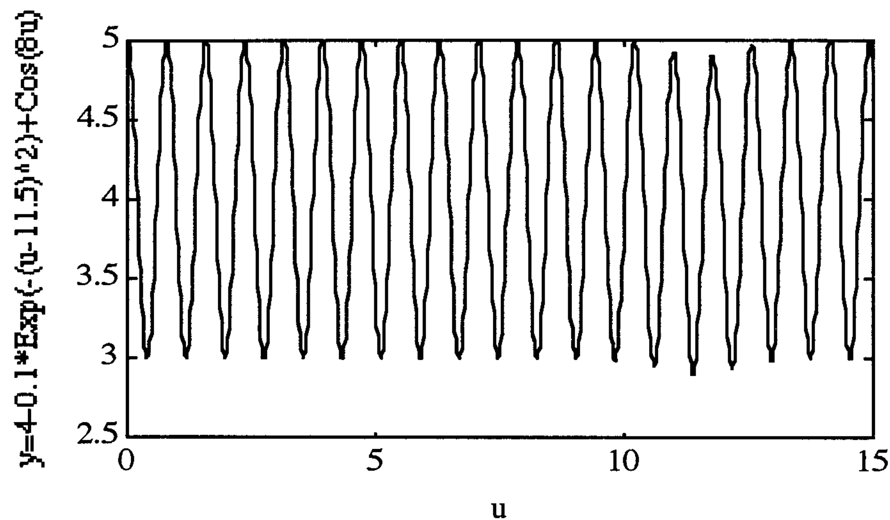

Figure 6.4. Target function for example 6.2

The procedure is the same as the one described in the first example. The approximating single layer neural network requires 1281 neurons for the desired approximation. Out of these 1281 neurons, 638 are "positive" and 643 "negative".

After convergence of the DC program, the following information is recorded:

* 608 iterations before convergence.

* 390 active vertices and 196 active constraints.

* Solution of the independent variables: u=11.39, ω=2.90, and z=255.27.

Therefore the minimization of the function over the desired interval is u=11.39. Once again, the algorithm was able to recognize the global minimum among 19 local minima which happen to be very close from one another.

This concludes the presentation of the minimization of the output of a SISO single layer neural network.

6.2.3 Minimizing a MISO single layer neural network.

In this section, an enhancement over the previous section is presented. In the previous section the minimization of a SISO single layer network was considered. In this section the input is no longer required to be a scalar u, but can be a vector $\underline{u}$. Hence, the type of networks considered here is the class of MISO single layer networks.

Such a network is represented by the system of equations:

$$\begin{aligned} \underline{x} &= W_1 \cdot \underline{u} + \underline{b} \\ y &= \underline{w_2} \cdot f(\underline{x}) \end{aligned} \qquad (6.22)$$

where $\underline{u}$ is the input vector of length p, and W_1 is a matrix of size $N \times p$. Similarly to what was previously done, the network must first be translated into a DC problem formulation, which is in turn converted into a canonical DC problem, and then solved.

The translation from (6.22) into DC formulation is extremely similar to the previous section. The fact that the input is multidimensional doesn't affect that the neurons have to be spilt into two sets $\aleph_p$ and $\aleph_n$, nor the manner in which this is accomplished. The same rules apply. Once the two sets are assembled, every neuron for each set is translated into a DC formulation. It must be noticed at this point that the result 6.3 was proven for the multidimensional case and that in the previous section it was used in a particular and simpler

case. Now, it is used in its general form. Following the analysis as before, the network (6.22) is transformed into the DC formulation:

$$
\begin{aligned}
y &= \left[\sum_{x_i \in \aleph_p} w_{2_i} f_1(\underline{w_{1_i}} \cdot \underline{u} + b_i) + \sum_{x_i \in \aleph_n} \left| w_{2_i} \right| f_2(\underline{w_{1_i}} \cdot \underline{u} + b_i) \right] - \\
& \left[\sum_{x_i \in \aleph_p} w_{2_i} f_2(\underline{w_{1_i}} \cdot \underline{u} + b_i) + \sum_{x_i \in \aleph_n} \left| w_{2_i} \right| f_1(\underline{w_{1_i}} \cdot \underline{u} + b_i) \right] \qquad (6.23) \\
y &= F_1(\underline{u}) - F_2(\underline{u})
\end{aligned}
$$

where $\underline{w_{1_i}}$ is a vector of length p. It is identical to the row i of the matrix W_1 and corresponds to the p input weights of the neuron i. It is clear that this formulation is indeed a DC form since both $F_1(\underline{u})$ and $F_2(\underline{u})$ are multidimensional convex functions.

Once the problem has been translated into DC, it must be converted into canonical DC form. This is performed in the exactly same manner as for the single dimension case. The fact that there are several input variables doesn't affect the creation of the two extra independent variables ω and z. In terms of the bounds for each input variable, two constraints are required for each dimension: every input variable must be bounded in an interval. At this point the algorithm for the multidimensional case has been exposed and can now be illustrated.

Example 6.3: Global minimization of the 2-dimension space. Consider the function:

$$
\begin{aligned}
y &= \varphi(u_1, u_2) \\
&= 10 \cdot \left(e^{-2u_1} + e^{-2u_2} + e^{-2(10-u_1)} + e^{-2(10-u_2)} \right) + \qquad (6.24) \\
& 4 \cdot \left(e^{-(u_1-4)^2} + e^{-(u_2-4)^2} + e^{-2(u_1-7)^2} + e^{-2(u_2-7)^2} \right)
\end{aligned}
$$

It is desired to find with a tolerance τ=0.001 its global minimum over the section of plane $\{u_1, u_2\} \in [0,10] \times [0,10]$. This function is shown in figures 6.5 and 6.6.

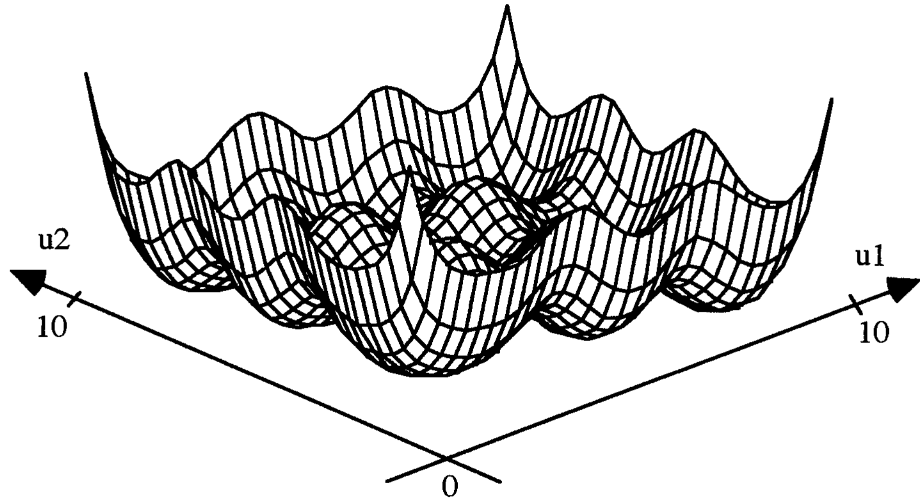

Figure 6.5. Mesh surface plot of the function for example 6.3.

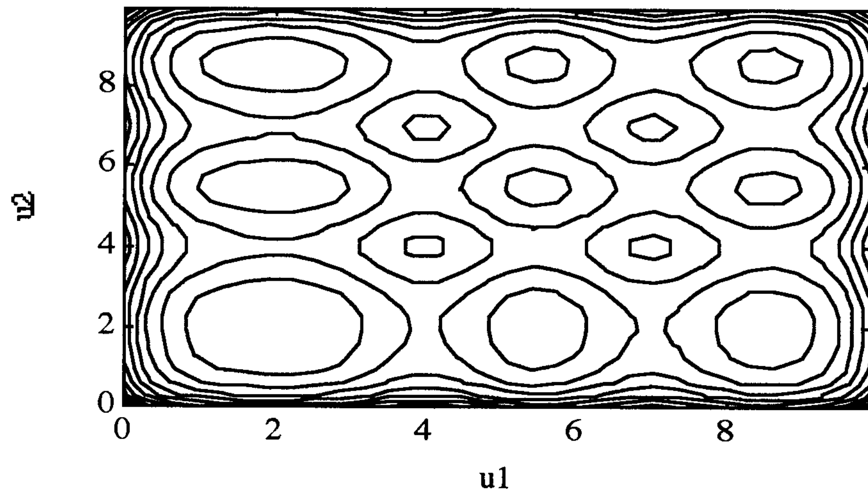

Figure 6.6. Contour plot of the function for example 6.3.

The first stage constists of modeling $\varphi(u_1,u_2)$ by a neural network $\psi(u_1,u_2)$. This is achieved using 640 neurons. Out of that number 324 are "positive" neurons and 316 are "negative" ones. The next stage consists of translating $\psi(u_1,u_2)$ into a DC form, and then into a canonical DC form. The last stage consists of running the canonical DC program.

After convergence, the following results may be recorded:

* Convergence reached after 355 iterations.

* At that point there are 803 active vertices and 204 active constraints.

* The solution is: $u_1 = 2.02$, $u_2 = 2.02$, ω=0.51, and z=23.83.

From this analysis it can be concluded that the global minimum over the required interval of the original function $\varphi(u_1,u_2)$ therefore is $u_1 = u_2 = 2.02$. Once again, this method succesfully found the correct global minimum among 9 local minima, which were very close from one another.

This concludes the presentation of the minimization of MISO functions using single layer neural networks and DC programming.

6.3 Minimizing a pseudo multilayer neural network output.

This section covers an "upgrade" of the method for globally minimizing the output of a single layer neural network that was described in the previous section. The enhancement consists of considering more general neural networks: the pseudo multilayer architecture that was introduced in Chapter 3. This section is made of three portions: first, a recall and introduction of theoretical results that are required in the analysis, then the implementation of an algorithm for globally minimizing the output of a pseudo multilayer neural network, either SISO or MISO, and lastly some examples and applications of this minimization procedure are shown.

6.3.1. Preliminary material.

The global minimization of the pseudo multilayer neural network is again performed using the DC programming technique. Due to the enhanced generality of the network, a few more results concerning DC programming are needed: All the results on DC formulation and operations presented so far are not sufficient for converting the network into a DC format. The missing results are now presented. These follow Hiriart-Urruty [Hiriart-Urruty, 1985].

Result 6.4: Considering a set $\{f_i\}$ of DC functions having decomposition $\{f_{i1} - f_{i2}\}$, the pointwise maximum function of the set $g = \max\{f_i\}$ is also DC, and its decomposition is given by (6.25):

$$g = \max_{i=1,\ldots,k}\{f_i\} = \max_{i=1,\ldots,k}\left\{f_{i1} + \sum_{j=1, j\neq i}^{k} f_{j2}\right\} - \sum_{i=1}^{k} f_{i2} \tag{6.25}$$

Proof 6.4: Since all f_i's (sub-1 and sub-2) are convex, the two parts on each side of the minus '-' sign are each convex (using the result 5.1). Therefore the equation (6.25) is indeed a DC form. Is it equal to the desired $g = \max\{f_i\}$?

Consider that the maximum of the set $\{f_i\}$ is f_δ having DC decomposition $f_{\delta_1} - f_{\delta_2}$.

$$
\begin{aligned}
f_\delta = \max_i\{f_i\} &\Rightarrow f_{\delta_1} - f_{\delta_2} = \max_i\{f_{i_1} - f_{i_2}\} \\
&\Rightarrow f_{\delta_1} - f_{\delta_2} + \sum_j f_{j_2} = \max_i\left\{f_{i_1} - f_{i_2} + \sum_j f_{j_2}\right\} = \max_i\left\{f_{i_1} + \sum_{j,j\neq i} f_{j_2}\right\} \\
&\Rightarrow \max_i\left\{f_{i_1} + \sum_{j,j\neq i} f_{j_2}\right\} - \sum_j f_{j_2} = f_{\delta_1} - f_{\delta_2} + \sum_j f_{j_2} - \sum_j f_{j_2} \\
&\qquad\qquad\qquad\qquad\qquad\quad = f_{\delta_1} - f_{\delta_2} = f_\delta
\end{aligned}
\tag{6.26}
$$

Result 6.5: Considering a DC function $f(x)$ having decomposition $f_1(x) - f_2(x)$, then the absolute value function $|f(x)|$ is also DC, and its decomposition is given by (6.27):

$$
|f(x)| = |f_1(x) - f_2(x)| = 2 \cdot \max(f_1(x), f_2(x)) - (f_1(x) + f_2(x)) \tag{6.27}
$$

Proof 6.5: Since both $f_1(x)$ and $f_2(x)$ are convex, the two parts on each side of the minus '-' sign are each convex (using the result 5.1). Therefore the equation (6.27) is indeed a DC form. Is it equal to the desired $|f(x)|$?

$$
\begin{aligned}
&\text{If } f(x) > 0 \Rightarrow f_1(x) > f_2(x) \Rightarrow \max(f_1(x), f_2(x)) = f_1(x) \\
&\qquad \Rightarrow 2 \cdot \max(f_1(x), f_2(x)) - (f_1(x) + f_2(x)) = 2 \cdot f_1(x) - (f_1(x) + f_2(x)) \\
&\qquad\qquad = f_1(x) - f_2(x) = f(x) = |f(x)| \\
&\text{If } f(x) < 0 \Rightarrow f_1(x) < f_2(x) \Rightarrow \max(f_1(x), f_2(x)) = f_2(x) \\
&\qquad \Rightarrow 2 \cdot \max(f_1(x), f_2(x)) - (f_1(x) + f_2(x)) = 2 \cdot f_2(x) - (f_1(x) + f_2(x)) \\
&\qquad\qquad = f_2(x) - f_1(x) = -f(x) = |f(x)|
\end{aligned}
\tag{6.28}
$$

This terminates the proof 6.5.

This concludes the presentation of preliminary material. The interface between DC programming and pseudo multilayer neural networks can now be introduced.

6.3.2 Pseudo multilayer networks and DC programming.

The pseudo multi layer neural network was introduced in Chapter 3. It is defined by the set of equations (3.16). It was noted that there are several types of state variable due to the internal architecture of the network. Macro state variables ξ and ζ are linearly interconnected, with the connections always going "forward", no feedback is allowed. The micro state variables describe the nonlinearity of the system and are grouped into clusters. Every cluster is connected to two macro variables: to one component of ξ and to one component of ζ. The network is constructed so that every of these clusters is actually a SISO single layer neural network. The pseudo multilayer neural network can therefore be viewed as a series of SISO single layer neural networks interconnected in a forward linear manner.

The idea for the procedure for converting such a network into DC formulation is to convert every cluster into a DC formulation following the previous section, and the interconnections being linear should not have a terrible effect on convexity. However, the path is not straightforward. There are actually two stages in the process. The first part consists of converting the network into DC formulation, and the second part is to translate this formulation into a formulation the global minimization algorithm presented before can understand and work on.

The procedure is to perform transformations into DC formulation in a backward motion manner: To start from the output and work backward to eventually reach the inputs. The network output node to be globally minimized is a linear combination of one or several SISO single layer network outputs. These single layer networks constitute the final stage of the pseudo multilayer network. They can be viewed as a MISO single layer network where the inputs are intermediate 'macro' variables. Following the previous section, this last stage

can be converted into a DC formulation. However, when doing so, the inputs are not the external inputs $\underline{u}$ but some of the linear macro state variables ξ and ζ. However, what was called the 'last stage' has effectively been converted into a DC problem. It has to be noticed that this DC problem is the target function for the desired global minimization.

The next sequence consists of relating the inputs of the last stage to other variables closer to the input, to eventually reach them. In a similar manner, each of these inputs can be related to the output of another MISO single layer neural network that can be called yet another stage. This is done following the construction of a pseudo multilayer neural network. This new stage is in turn converted into a DC formulation relying on input variables that are closer to the external inputs of the overall network than were the ones of the last stage. It is clear that proceeding in this manner allows to travel toward the external inputs and eventually reach them.

To illustrate this construction, an example is now presented.

Example 6.4: Consider the pseudo multilayer neural network depicted on figure (6.7). It has 4 external inputs and is composed of 7 internal SISO single layer networks numbered S_1 to S_7. Each of them has a scalar input I and a scalar output O. They are called I_1 to I_7 and O_1 to O_7 respectively. Following the previous section, it is known that each of these SISO network can be represented by a DC formulation $O_i = f_{i_1}(I_i) - f_{i_2}(I_i)$ where both $f_{i_1}(.)$ and $f_{i_2}(.)$ are convex functions. The numbers along the links represent the weights for the linear part of the network. For the segments where no number is shown, a unity gain is assumed.

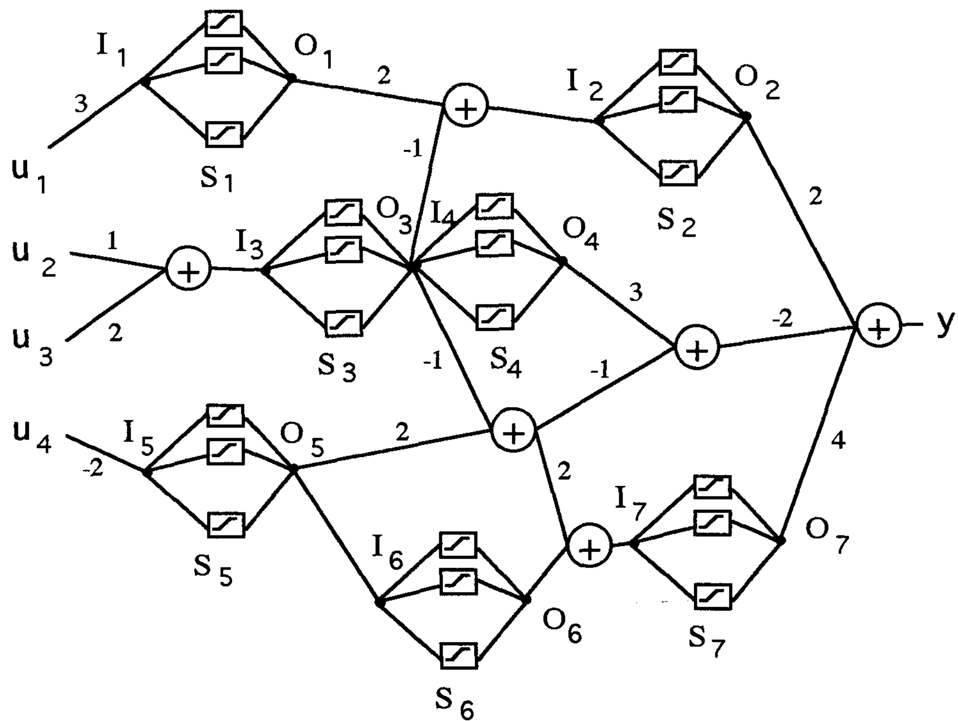

Figure 6.7. Pseudo multilayer neural network for example 6.4.

The pseudo multilayer network shown in figure (6.7) can be converted into DC formulation using the following route: Starting from the output y, write the internal structure as DC functions with the introduction of extra variables that are in turn represented as DC functions of other variables until external inputs are reached.

The network of figure (6.7) can be described by the set of equations:

$$
\begin{aligned}
&y = 2\cdot O_2 - 2\cdot O_3 - 6\cdot O_4 + 4\cdot O_5 + 4\cdot O_7 \\
&I_1 = 3\cdot u_1 \\
&I_2 = 2\cdot O_1 - O_3 \\
&I_3 = u_2 + 2\cdot u_3 \\
&I_4 = O_3 \\
&I_5 = -2\cdot u_4 \\
&I_6 = O_5 \\
&I_7 = -2\cdot O_3 + 4\cdot O_5 + O_6 \\
&O_i = f_{i_1}(I_i) - f_{i_2}(I_i)
\end{aligned}
\tag{6.29}
$$

In order to remove a few variables, some equations can be combined:

$$
\begin{aligned}
&y = 2\cdot f_{2_1}(I_2) - 2\cdot f_{2_2}(I_2) - 2\cdot f_{3_1}(u_2 + 2\cdot u_3) + 2\cdot f_{3_2}(u_2 + 2\cdot u_3) - 6\cdot f_{4_1}(I_4) \\
&\quad +6\cdot f_{4_2}(I_4) + 4\cdot f_{5_1}(-2\cdot u_4) - 4\cdot f_{5_2}(-2\cdot u_4) + 4\cdot f_{7_1}(I_7) - 4\cdot f_{7_2}(I_7) \\
&I_2 = 2\cdot f_{1_1}(3\cdot u_1) - 2\cdot f_{1_2}(3\cdot u_1) - f_{3_1}(u_2 + 2\cdot u_3) + f_{3_2}(u_2 + 2\cdot u_3) \\
&I_4 = f_{3_1}(u_2 + 2\cdot u_3) - f_{3_2}(u_2 + 2\cdot u_3) \\
&I_6 = f_{5_1}(-2\cdot u_4) - f_{5_2}(-2\cdot u_4) \\
&I_7 = -2\cdot f_{3_1}(u_2 + 2\cdot u_3) + 2\cdot f_{3_2}(u_2 + 2\cdot u_3) + 4\cdot f_{5_1}(-2\cdot u_4) - 4\cdot f_{5_2}(-2\cdot u_4) \\
&\quad + f_{6_1}(I_6) - f_{6_2}(I_6)
\end{aligned}
\tag{6.30}
$$

Each of these equations is a "first order" DC function. This means that the output is related to the input vector by successive known DC decompositions involving intermediate variables. If these extra variables were removed, the set of equations would become a single equation, but it would not explicitely show the DC decompostion since a composition of convex functions is not generally a convex function too. Therefore the equation set (6.30) can't be reduced further. However, it can be written as (6.31) to show better the convex and reverse convex parts:

$$
\begin{aligned}
y &= \left\{2 \cdot f_{2_1}(I_2) + 2 \cdot f_{3_2}(u_2 + 2 \cdot u_3) + 6 \cdot f_{4_2}(I_4) + 4 \cdot f_{5_1}(-2 \cdot u_4) + 4 \cdot f_{7_1}(I_7)\right\} \\
&\quad -\left\{2 \cdot f_{2_2}(I_2) + 2 \cdot f_{3_1}(u_2 + 2 \cdot u_3) + 6 \cdot f_{4_1}(I_4) + 4 \cdot f_{5_2}(-2 \cdot u_4) + 4 \cdot f_{7_2}(I_7)\right\} \\
I_2 &= \left\{2 \cdot f_{1_1}(3 \cdot u_1) + f_{3_2}(u_2 + 2 \cdot u_3)\right\} - \left\{2 \cdot f_{1_2}(3 \cdot u_1) + f_{3_1}(u_2 + 2 \cdot u_3)\right\} \\
I_4 &= \left\{f_{3_1}(u_2 + 2 \cdot u_3)\right\} - \left\{f_{3_2}(u_2 + 2 \cdot u_3)\right\} \\
I_6 &= \left\{f_{5_1}(-2 \cdot u_4)\right\} - \left\{f_{5_2}(-2 \cdot u_4)\right\} \\
I_7 &= \left\{2 \cdot f_{3_2}(u_2 + 2 \cdot u_3) + 4 \cdot f_{5_1}(-2 \cdot u_4) + f_{6_1}(I_6)\right\} \\
&\quad -\left\{2 \cdot f_{3_1}(u_2 + 2 \cdot u_3) + 4 \cdot f_{5_2}(-2 \cdot u_4) + f_{6_2}(I_6)\right\}
\end{aligned}
\tag{6.31}
$$

It can be noticed that the linear weights have no real effect on the DC decompositions except by swaping the convex and reverse convex parts whenever this coefficient is negative. At this point the pseudo multilayer neural network has been written into a DC formulation as it was desired. The next point consists of performing the global minimization on the output y.

As the goal is to globally minimize the external output y, following the DC decomposition just presented the problem can be written with an objective function and forcing constraints as:

$$
\begin{aligned}
&\textit{Minimize} \left\langle \begin{matrix} \left\{2 f_{2_1}(I_2) + 2 f_{3_2}(u_2 + 2 \cdot u_3) + 6 f_{4_2}(I_4) + 4 f_{5_1}(-2 \cdot u_4) + 4 f_{7_1}(I_7)\right\} \\ -\left\{2 f_{2_2}(I_2) + 2 f_{3_1}(u_2 + 2 \cdot u_3) + 6 f_{4_1}(I_4) + 4 f_{5_2}(-2 \cdot u_4) + 4 f_{7_2}(I_7)\right\} \end{matrix} \right\rangle \\
&\textit{Subject to } (u_1, u_2, u_3, u_4) \in \Omega \textit{ and} \\
&I_2 = \left\{2 \cdot f_{1_1}(3 \cdot u_1) + f_{3_2}(u_2 + 2 \cdot u_3)\right\} - \left\{2 \cdot f_{1_2}(3 \cdot u_1) + f_{3_1}(u_2 + 2 \cdot u_3)\right\} \\
&I_4 = \left\{f_{3_1}(u_2 + 2 \cdot u_3)\right\} - \left\{f_{3_2}(u_2 + 2 \cdot u_3)\right\} \\
&I_6 = \left\{f_{5_1}(-2 \cdot u_4)\right\} - \left\{f_{5_2}(-2 \cdot u_4)\right\} \\
&I_7 = \left\{2 \cdot f_{3_2}(u_2 + 2 \cdot u_3) + 4 \cdot f_{5_1}(-2 \cdot u_4) + f_{6_1}(I_6)\right\} \\
&\quad - \left\{2 \cdot f_{3_1}(u_2 + 2 \cdot u_3) + 4 \cdot f_{5_2}(-2 \cdot u_4) + f_{6_2}(I_6)\right\}
\end{aligned}
\tag{6.32}
$$

where Ω is a convex domain including all the valid values for the external input vector. This represents a convex inequality constraint.

It is clear that this problem is the minimization of a DC objective function along a set of DC inequality and equality constraints. In the original DC programming problem as shown in the section dealing with single layer networks, equality constraints are not allowed. Only inequality constraints can be used. However, the formulation of this new problem requires equalities in order to force the input of a cluster to match the linear combination of outputs of previous clusters as desired.

How can the DC algorithm be modified to handle these equality constraints? The whole optimization procedure is based around the algorithm developed by Thoai that deals exclusively with inequality constraints. The reason that is doesn't handle equality constraints is because "the algorithm works only on problems with a feasible set of full dimension. So, it can't be applied for solving problems with equality constraints" [Thoai, 1993]. Therefore, an alternative must be found since it seems impossible to modify the algorithm so that it handles this case.

The alternative suggested here is to modify the statement of the problem so that it can still be solved by the previous algorithm, rather than finding another algorithm. There are two ways to achieve this, and they are both presented next. Each of them has some advantages and some drawbacks, and depending on the problem itself, one may be desired over the other. In one case, the equality constraints are modified to be included into the target function, in the other case, the equality constraints are modified to become inequality constraints. In both cases, the resulting problem is the minimization of a DC objective function along DC inequality constraints. That problem can be handled by the previously used optimization algorithm.

As just mentioned, in the first suggested alternative, the equality constraints are modified and included in the objective function. An equality constraint of the type *a=b* can be written as *a-b=0*. Considering the function $f(a,b)=|a-b|$, this function has a unique minimum at the location *a=b* and the value of this minimum is *0*. Therefore solving an equation of the type *a=b* is equivalent to succesfully minimizing the function $f(a,b)=|a-b|$. Therefore the equality constraint *a=b* to be enforced can be replaced by an objective function to be minimized. Since the original equality constraint is DC, the new function is also DC and following the Result 6.5 mentioned earlier, the DC decomposition is directly known. It must be noticed however that the constraint has to be satisfied, therefore the corresponding part in the minimization process must be fully minimized, i.e. reach *0*. This can be simply achieved by putting a large weight on this part in the objective function. For example consider the DC problem:

$$\begin{gathered} \textit{Minimize } f_1 - f_2 \\ g_1 - g_2 = 0 \ \ \& \ \ \text{other DC inequalities} \end{gathered} \tag{6.33}$$

where f_1, f_2, g_1, and g_2 are all convex functions. This problem can be replaced by the equivalent new DC problem:

$$\begin{gathered} \left\{ \begin{matrix} \textit{Minimize } \ (f_1 - f_2) + M \cdot (|g_1 - g_2|) \\ \text{Other DC inequality constraints} \end{matrix} \right\} \equiv \\ \left\{ \begin{matrix} \textit{Minimize } \ (f_1 - f_2) + M \cdot (2 \cdot \max(g_1, g_2) - (g_1 + g_2)) \\ \text{Other DC inequality constraints} \end{matrix} \right\} \end{gathered} \tag{6.34}$$

where *M* is a Lagrange multiplier, i.e. a large weighting factor. In the case there are several equality constraints, they are successively transformed into terms in the objective function to be minimized. The fact that there is a large coefficient associated with each of them insures

that they are all satisfied. In some sense the minimization of the parts corresponding to the equality constraints have a higher priority than the minimization of the original objective function. That should be since the constraints are the strongest requirements in the whole optimization process. This concludes the presentation of the first suggested alternative.

In the second alternative, it is suggested that the cumbersome DC equality constraints be replaced by pseudo-equivalent DC inequality constraints. As it was mentioned all along, tolerance levels are omnipresent in the whole project, from the modelization into a neural network, to the synthetization of the pseudo multilayer neural network, to the threshold levels in determining redundant constraints and vertices in the optimization algorithm, and to the convergence of the algorithm itself. The bottom line of this comment is that there is a limit in the resolution achieved by the whole process. Assuming this limit is ε, then an equality of the type $a=b$, equivalently written as $a-b=0$ is actually no different than $|a-b| \leq \varepsilon$. This shows that an equality can be transformed into an equivalent inequality. Using again the result on the absolute value of a DC function (result 6.5), a DC equality has an equivalent DC inequality for a given resolution ε:

$$\begin{aligned} \{f_1(x) - f_2(x) = 0\} &\Leftrightarrow \{|f_1(x) - f_2(x)| \leq \varepsilon\} \\ &\Leftrightarrow \{2 \cdot \max\{f_1(x) - f_2(x)\} - (f_1(x) - f_2(x)) \leq \varepsilon\} \end{aligned} \tag{6.35}$$

Following this route, similarly to the first suggested alternative, this second method allows to transform a DC problem with DC equality constraints into a DC problem without DC equality constraints but only DC inequality constraints. The difference between the two approaches is that one performs the transformation by increasing the complexity of the objective function, while the other increses the complexity of the constraints. This is another example of the trade-off and duality there exists between objective functions and constraints as it was mentioned in Chapter 4.

To this point, the minimization of the output of the pseudo multilayer neural network has been successfully converted into the minimization of a DC objective function along with DC inequality constraints. The last step before executing the optimization algorithm is to convert all the DC constraints into a single DC constraint as it was explained in Chapter 5. This step was not necessary in the case of the single layer network for the reason that a single DC inequality constraint was present. The conversion of a set of DC inequality constraints into a single DC inequality constraints is performed easily when using the result 6.4. Considering a set of DC constraints $\{f_{i_1} - f_{i_2} \leq 0\}$, the resulting single constraint $\{\max(f_{i_1} - f_{i_2}) \leq 0\}$ is stronger than each of them individually, and its DC decomposition is immediately known using the result 6.4. Once the set of DC coinstraints has been converted into a single DC constraint, then the optimization algorithm can be run. The solution it produces corresponds to the global minimum of the output of the original pseudo multilayer neural network.

6.3.2. Examples and Applications.

In this section, some examples and applications of the global minimization of the output of a pseudo multilayer neural network are presented. The first example is a direct implementation of the method. An arbitrary mathematical function is modeled by a neural network that is synthesized using the procedure described in Chapter 3. The network equations are then converted into DC formulation, which in turn is translated into a problem the optimization algorithm can understand following what was said in the previous section. Solving the DC problem drives to the desired solution.

After showing this academic example, it is interesting to look at some actual engineering problems. Needless to say that there are numerous global optimization problems around in various engineering fields waiting to be solved. Following what was

said concerning the generality of the modeling of any physical phenomenon or system by a pseudo multilayer neural network, it is clear that the procedure described above can be used to solve these problems.

Example 6.5: This example is once again an academic example, i.e. it has no particular interest except for demonstrating that the method works, and to show the successive steps required for its implementation.

Consider the real function *f(.)* of the real variable *u* which is shown on figure 6.8 and which is defined by the equation (6.36). This is a *cosine* function the argument of which is modulated by a *Log* function. The *Exp* function (actually Gaussian function) is here to create a global minimum.

$$y = 3 + Cos(2 \cdot u + 10 \cdot Log(u+1)) - 0.2 \cdot e^{-(u-5)^2} \tag{6.36}$$

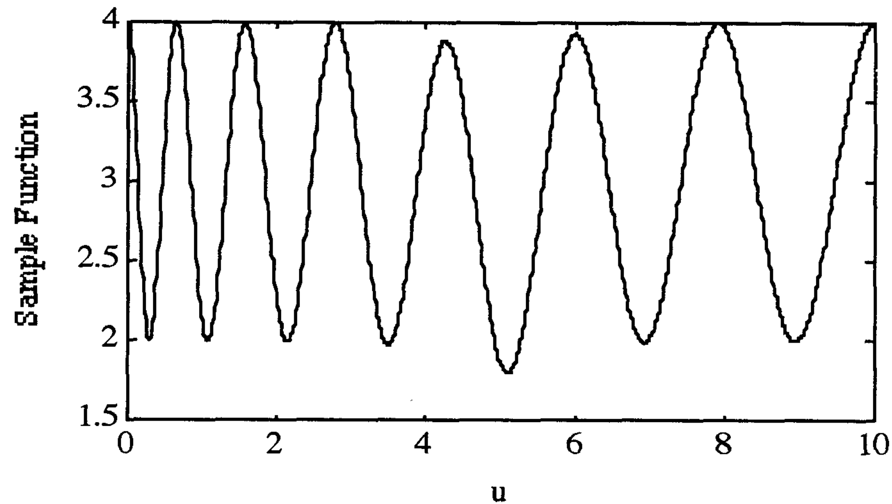

Figure 6.8. Sample function for example 6.5.

It is desired to find its global minimum over the interval [0,10] with a given tolerance of 0.01. It is clear that this function could be modeled by a single layer SISO neural network. It could therefore be minimized by the procedure shown in section 6.2. However at this point it is desired to illustrate the procedure for minimizing a pseudo multilayer neural network. Therefore the function *f(u)* is modeled by a pseudo multilayer neural network the output of which is globally minimized following the procedure described so far in section 6.3. A pseudo multilayer neural network modeling *f(u)* is shown on figure 6.9. This network clearly is of the right form since each nonlinear object is of the SISO type, and all the interconnections between objects are linear. Let's define the so called 'macro' state variables X_1 and X_2 as:

$$\begin{aligned} X_1 &= 2\cdot u + 10\cdot Log(u+1) \\ X_2 &= (u-5)^2 \end{aligned} \tag{6.37}$$

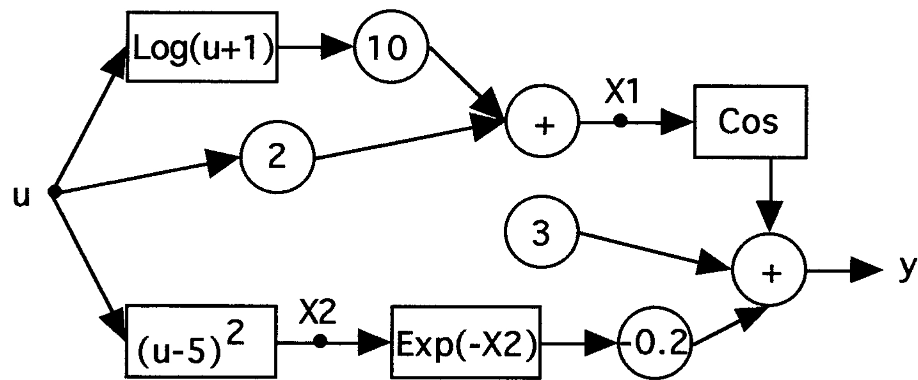

Figure 6.9. Model of the example function.

The complete equations describing the network are:

$$
\begin{aligned}
X_1 &= 2 \cdot u + 10 \cdot Log(u+1) \\
X_2 &= (u-5)^2 \\
y &= 3 + Cos(X_1) - 0.2 \cdot Exp(-X_2)
\end{aligned}
\tag{6.38}
$$

Following the procedure, each of the nonlinear objects has to be modeled by a SISO single layer neural network to be then converted into a DC formulation. However, it can be noticed that some of the objects are convex or concave: Both $(u-5)^2$ and $Exp(-X_2)$ are convex, while $Log(u+1)$ is concave. Therefore, there is no need to transform them first into a SISO single layer neural network. Hence, only the objects which have no 'obvious' DC decomposition are first transformed into a neural network. It is clear that the *Cosine* block is of this type. Since the input variable u varies from 0 to 10, it can be noticed that X_1 varies from 0 to 44 since the *Log* function is strictly monotonic. Therefore the SISO neural network modelling the *Cosine* block must be performing correctly over that domain. Following the synthesis procedure from Chapter 3, the *Cosine* object can be synthesized with the required tolerance over the domain [0,44] using 114 neurons. The modeling problem can be described by the following set of equations:

$$
\begin{aligned}
X_1 &= 2 \cdot u + 10 \cdot Log(u+1) \\
X_2 &= (u-5)^2 \\
y &= 3 + \left[y_0 + \sum_{i=1}^{114} w_{2_i} \cdot f(w_{1_i} \cdot X_1 + b_i) \right] - 0.2 \cdot Exp(-X_2)
\end{aligned}
\tag{6.39}
$$

The next step consists of converting these equations into DC format. As it was already stated some equations already are. However, the neural network part is not. It has to be converted following the usual procedure, separate the positive output weights from the negative ones. It so happens in this case that out of the 114 neurons, 56 are 'positive'

neurons and 58 are 'negative ones'. They form the sets $\aleph_p$ and $\aleph_n$ as defined before. Therefore the model can be written in DC format as:

$$
\begin{aligned}
X_1 &= 2\cdot u + 10\cdot Log(u+1) \\
X_2 &= (u-5)^2 \\
y &= 3 + y_0 + \left[\sum_{x_i \in \aleph_p} w_{2_i} f_1(w_{1_i}u + b_i) + \sum_{x_i \in \aleph_n} \left|w_{2_i}\right| f_2(w_{1_i}u + b_i)\right] \\
&\quad - \left[\sum_{x_i \in \aleph_p} w_{2_i} f_2(w_{1_i}u + b_i) + \sum_{x_i \in \aleph_n} \left|w_{2_i}\right| f_1(w_{1_i}u + b_i) + 0.2\cdot Exp(-X_2)\right]
\end{aligned} \tag{6.40}
$$

The next step consists of performing a global minimization of the DC model. As it was explained earlier, only the output equation has to be minimized, the other ones actually are DC equality constraints. This type of constraints is not allowed and must be converted once again. Following one of the routes that were suggested for solving this problem, each of the constraints is transformed into an absolute value form that is minimized using a large weighting factor. The two DC equality constraints are therefore converted according to the equations (6.41) and (6.42) that must be minimized as well as the original target output, but with larger weight.

$$
\begin{aligned}
&X_1 = 2\cdot u + 10\cdot Log(u+1) \\
&\rightarrow \left|X_1 - 2\cdot u - 10\cdot Log(u+1)\right| \\
&\rightarrow \left\langle 2\cdot Max\left\{0, X_1 - 2\cdot u - 10\cdot Log(u+1)\right\}, X_1 - 2\cdot u - 10\cdot Log(u+1)\right\rangle
\end{aligned} \tag{6.41}
$$

$$
\begin{aligned}
&X_2 = (u-5)^2 \\
&\rightarrow \left|(u-5)^2 - X_2\right| \\
&\rightarrow \left\langle 2\cdot Max\left\{0, (u-5)^2 - X_2\right\}, (u-5)^2 - X_2\right\rangle
\end{aligned} \tag{6.42}
$$

The problem has therefore been translated into solving the global minimization of the output y in the following DC problem:

$$
\begin{aligned}
y = 3 + y_0 &+ \left[\sum_{x_i \in X_p} w_{2_i} f_1(w_{1_i} u + b_i) + \sum_{x_i \in X_n} |w_{2_i}| f_2(w_{1_i} u + b_i) \right] \\
&- \left[\sum_{x_i \in X_p} w_{2_i} f_2(w_{1_i} u + b_i) + \sum_{x_i \in X_n} |w_{2_i}| f_1(w_{1_i} u + b_i) + 0.2 \cdot Exp(-X_2) \right] \\
&+ M \cdot \langle 2 \cdot Max\{0, X_1 - 2 \cdot u - 10 \cdot Log(u+1)\}, X_1 - 2 \cdot u - 10 \cdot Log(u+1) \rangle \\
&+ M \cdot \langle 2 \cdot Max\{0, (u-5)^2 - X_2\}, (u-5)^2 - X_2 \rangle \\
&\quad \text{subject to } \{u, X_1, X_2\} \in \Omega
\end{aligned} \tag{6.43}
$$

where M is a large weighting factor (chosen to be 20 in this particular example) and Ω is the convex domain containing the three independent variables u, X_1 and X_2. To show better the DC decomposition, (6.43) can be written again as:

$$
\begin{aligned}
y = &\left\{ \begin{aligned} &3 + y_0 + \left[\sum_{x_i \in X_p} w_{2_i} f_1(w_{1_i} u + b_i) + \sum_{x_i \in X_n} |w_{2_i}| f_2(w_{1_i} u + b_i) \right] \\ &+2 \cdot M \cdot \left[Max\{0, X_1 - 2 \cdot u - 10 \cdot Log(u+1)\} + Max\{0, (u-5)^2 - X_2\} \right] \end{aligned} \right\} \\
- &\left\{ \begin{aligned} &\left[\sum_{x_i \in X_p} w_{2_i} f_2(w_{1_i} u + b_i) + \sum_{x_i \in X_n} |w_{2_i}| f_1(w_{1_i} u + b_i) + 0.2 \cdot Exp(-X_2) \right] \\ &+ M \cdot \left[X_1 - 2 \cdot u - 10 \cdot Log(u+1) + (u-5)^2 - X_2 \right] \end{aligned} \right\} \\
&\quad \text{subject to } \{u, X_1, X_2\} \in \Omega
\end{aligned} \tag{6.44}
$$

The problem can then be converted to a canonical DC problem after the introduction of the two extra independent variables z and ω as it was explained before:

Minimize ω

Subject to $\{u, X_1, X_2\} \in \Omega$ (6.45)

$$\left\{\begin{array}{l} 2\cdot M\cdot\left[Max\{0, X_1 - 2\cdot u - 10\cdot Log(u+1)\} + Max\{0, (u-5)^2 - X_2\}\right] \\ +3 + y_0 + \left[\sum_{x_i \in X_p} w_{2_i} f_1(w_{1_i} u + b_i) + \sum_{x_i \in X_n} \left|w_{2_i}\right| f_2(w_{1_i} u + b_i)\right] - z \end{array}\right\} \le 0$$

$$\left\{\begin{array}{l} z - \omega - M\cdot\left[X_1 - 2\cdot u - 10\cdot Log(u+1) + (u-5)^2 - X_2\right] \\ -\left[\sum_{x_i \in X_p} w_{2_i} f_2(w_{1_i} u + b_i) + \sum_{x_i \in X_n} \left|w_{2_i}\right| f_1(w_{1_i} u + b_i) + 0.2\cdot Exp(-X_2)\right] \end{array}\right\} \le 0$$

where the second and third constraints are convex and reverse convex respectively.

This problem is now solvable by the DC minimization algorithm. After this algorithm is run, the convergence is reached with the following characteristics:

u=5.09, X_1=0.01, X_2=28.28, ω=1.80, and z=0.63. (6.46)

It is clear that the desired solution has indeed correctly been found. The value of the function at that location is ω which is correct. With respect to the DC equalities, they have indeed been fully satisfied since:

$$\begin{array}{lll} 2\cdot u + 10\cdot Log(u+1) = 28.25 & \text{and} & X_1 = 28.28 \\ (u-5)^2 = 0.01 & \text{and} & X_2 = 0.01 \end{array} \quad (6.47)$$

This concludes the presentation of the 'academic' example.

Application 6.1: Programming of a neural network.

It was recalled in Chapter 2 that the task of programming a neural network to behave according to some given directive is fairly complex. The method generally used for general feedforward networks is the so-called backpropagation. However it has several drawbacks such as lack of guarantee of global convergence to the "best" set of network gains and parameters. However, note that the task of programming a neural network can be put into a global optimization problem form. Once this is done, the problem can be modeled by yet another neural network. When the output of the model is globally minimized via DC programming, the desired solution for the initial problem then follows. This sounds close to "a neural network used to program another neural network".

A neural network desired response is described by a set of input-output pairs: when any of the vectors from the input set is applied to the network, it is desired to obtain the corresponding output. If the network is not programmed correctly or if the network can't achieve the desired behavior, then an error measure is available. For a given input, the measure of the error is the distance between the desired output with the actual output. This number is required to be as close to 0 as possible. Since there are usually a large number of input-output pairs, each of these error measures are summed up to create an overall performance index of the network. Ultimately the global error is equal to 0 which means that the network is programmed perfectly: for every input from the set, the actual network output is equal to the corresponding desired output.

It is clear that this formulation shows the relation between the programming of a neural network and a global optimization problem where a scalar performance index must be globally minimized. The input-output pairs represent parameters for the optimization problem, as well as the network architecture. The inputs of the optimization problem are the weights and biases of the neural network to be programmed: these quantities are the unknown variables to be determined.

Consider a single layer neural network with N neurons, with m inputs (given by the input vector $\underline{u}$), and with p outputs (given by the output vector $\underline{y}$). The desired behavior of the network is given by a set of k input output pairs $\{I_k, O_k\}$ to be satisfied as well as possible. The parameters to be determined are the input weights given by the matrix W_i, the neuron biases given by the vector $\underline{b}$, and the output weights given by the matrix W_o. The neural network equations are:

$$\left.\begin{array}{l}\underline{x} = f(W_i \cdot \underline{u} + \underline{b}) \\ \underline{y} = W_o \cdot \underline{x}\end{array}\right\} \Rightarrow \underline{y} = W_o \cdot f(W_i \cdot \underline{u} + \underline{b}) \tag{6.48}$$

For the pair ι, the error is defined as:

$$j_\iota(W_o, W_i, \underline{b}) = \left|O_\iota - W_o \cdot f(W_i \cdot I_\iota + \underline{b})\right| \tag{6.49}$$

It is therefore clear that the overall performance index of the neural network J when including all the pairs can be written as:

$$J(W_o, W_i, \underline{b}) = \sum_{\iota=1}^{k} j_\iota(W_o, W_i, \underline{b}) = \sum_{\iota=1}^{k} \left|O_\iota - W_o \cdot f(W_i \cdot I_\iota + \underline{b})\right| \tag{6.50}$$

This is a continuous MISO function. Therefore, it is possible to model it with a pseudo multilayer neural network, thereby using the procedure described earlier to convert it into a DC programming formulation and successively finding the optimal parameters after running a DC minimization algorithm.

This concludes the examples and applications. The method described in this chapter for globally minimizing the output of a neural network was shown to be successful for

performing global optimizations when the initial problem is first modeled by the neural network.

Chapter 7: Summary and Conclusions

This dissertation presented a bridge between modeling and global optimization. Following this route general global optimization problems can be solved. On the modeling side, this bridge is interfaced to the desired problem via a new class of neural networks: the pseudo multilayer neural network. On the global optimization side, the bridge is interfaced to the searched solution via the so-called DC programming.

7.1 Introduction.

In order to efficiently construct these interfaces, advanced knowledge of both neural networks and global optimization techniques are required. Neural networks in general were reviewedin Chapter 2. This involved various classes of neural networks, the programming aspect of the desired behavior of a neural network, as well as a literature review concerning neural networks in general and those dealing with function approximation in particular. Global optimization techniques in general were reviewed in Chpater 4. This involved a review of several families of methods including deterministic methods and Tuy's cuts. Again, a literature search was presented.

After the general reviews, Chapter 3 and Chapter 5 described the two interfaces. A new class of neural networks was introduced in Chapter 3 and a complete study of these was then described. This included presentation of the architecture of the networks, synthesis aspects of the network rather than backpropagation, followed by some concrete examples showing a comparison with more familiar architectures such as the classical multilayer feedforward neural network. In Chapter 5, a review and complete implementation of Thoai's algorithm for solving DC programming problems was presented. This included the

theoretical aspects of converting a DC problem into a canonical DC problem, as well as practical aspects such as constraint dropping strategies and computation of new vertices after a new cut is performed.

Finally, in Chapter 6 the bridge between pseudo multilayer neural networks and DC programming was introduced. This connection required several problem conversions and translations, as well as changes of variables in order to find general common grounds. Once these requirements were presented, actual implementations for practical examples were shown.

7.2 Contributions.

It is clear that the major contribution of this work has three aspects: showing the possibility of solving general optimization problems using neural networks and DC programming; the complete introduction of a new class of neural networks including a performance study and synthesis for construction; and finally the implementation of a modified version of Thoai's algorithm that can optimize problems formulated using pseudo multilayer neural networks.

7.2.1. Neural networks and DC programming.

The idea of mixing neural networks and global optimization techniques together was not new. In Chapter 2 several previous studies were mentioned. However, the approach shown in this thesis is more general. A larger class of problems can be considered, and the global convergence is guaranteed. The two phase process, modeling followed by minimization is probably where most was gained. On the one hand, the modeling is very general due to the wide capability of modeling various systems by the pseudo multilayer

neural network, and on the other hand, the minimization task is guaranteed to globally converge following the work by Thoai. It is therefore clear that each of the two phases is very promising by itself. Since it was shown in Chapter 6 that it was possible to put them together, and the complete procedure was derived and explained, the advocated method is therefore successful.

7.2.2. The pseudo multilayer neural network.

Although the pseudo multilayer architecture for neural networks was primarily introduced for an interface purpose with DC programming as described earlier, it so happens that it has a much larger application field. In many instances it can replace a regular multilayer feedforward neural network. It has the key advantage of being synthesized rather than programmed which implies that the procedure is systematic and fast compared to the classical backpropagation. In terms of actual implementation, the pseudo multilayer is a subset of the regular multilayer, i.e., the very same VLSI chips that are used for the hardware implementation of a feedforward multilayer network can be used for the pseudo multilayer neural network as well.

In order to model a mathematical function, a process, or a general system by a pseudo multilayer neural network, it must follow certain rules which were described, i.e. for a mathematical function, it must be unary^{+} decomposable. However, it was mentioned and proven that these requirements are not very stringent and that engineering systems almost always fall into these categories.

The strong point of the synthesis of a pseudo multilayer neural network is that the procedure is based around objects that are assembled and interconnected. The complete procedure was fully explained and its efficiency was illustarted by means of a few examples. A reason for this efficiency is that several "special situations" are implemented in

a direct manner. Those were carefully explained and developed. They include direct implementations of the "square" function, of the "multiplication" operator, and of the "modulo" computation used with periodic functions. Overall the pseudo multilayer architecture along with its synthesis procedure both described in Chapter 3 offer a promising alternative to the classical feedforward multilayer neural network and allow interfacing with DC programming and global optimization techniques.

7.2.3. Global optimization via DC programming of a problem formulated with a pseudo multilayer neural network.

Chapter 6 provided the completion of the bridge between neural networks and global optimization techniques. It was the chapter in which it was proven that it was possible to interface a pseudo multilayer neural network with DC programming. The procedure was shown by starting with a simple neuron; the DC decomposition of the input output behavior of the neuron was written. Then single layer neural networks were explored; such networks are sets of single neurons with their outputs linearly connected. It was henceforth shown and explained how to convert a single layer neural network into a DC formulation. The procedure was shown to be working with both SISO and MISO type networks. This was illustrated by means of examples for each kind. The step that followed was to expand the method to be able to deal also with pseudo multilayer neural networks. At every step through this enhancement process to increase generalization, theoretical results supporting the validity of the extension were recalled or introduced, and proven. At the end a systematic procedure had been derived that allows to convert any SISO pseudo multilayer neural network into a DC formulation such that the global minimization algorithm presented erlier could be successfully applied.

A practical implementation of this was shown by means of examples and applications. It was noticed that due to the generality of treatable problems, there are lots of possible applications.

7.3 Further research.

It is clear that any further research on any subject involves what has not been fully completed, or what could be improved on the original process. Therefore, the drawbacks and problems of the current method must be further explored and possibly improved. From what was said and shown concerning the wide possible use and guarantee of global convergence of the optimization process, it is clear that the theoretical aspects of this work are in fairly good shape. Problems lie more in its practical implementation.

There are two major areas that could be improved: on the one hand, there is the possibility of an implementation of automatic procedures to build a pseudo multilayer neural network from a given optimization problem, followed by an automatic implementation of the conversion of this neural network into DC formulation. On the other hand, improvements or alternative to Thoai's algorithm for solving the DC programming phase of the process can be looked for.

To begin with the practical implementation of automatic procedures, what could be envisioned is a compiler like process that would convert given problems into an object oriented structure. Once the object model is formed, what is left to do is to convert each individual object into a SISO single layer neural network as it was explained, and connect the objects through an external linear structure. The "special efficient cases" should be recognized and taken care of. Then, following the steps introduced in Chapter 6, the pseudo multilayer neural network can be converted into a DC formulation, then translated into a canonical DC formulation. What is suggested here is not any new work, it is just an

automatic implementation of all the steps described in the thesis that were manually taken during the various given examples. However, it must be said that implementing these automatic procedures is mandatory if the method is to be widely used.

With respect to a better and more efficient way of solving a DC problem, the question is more difficult. Starting with the current status, it can be said that it is actually working. It drives successfully the minimization process to the global minimum. However, the procedure is too slow for having much practical interest at this point. Why is the procedure so slow? There are two reasons. The first reason is that Thaoi's algorithm performs cuts and tracks vertices. The procedure is time consuming due to the large number of vertices created and eliminated after each cut. More efficient algorithms could probably be found by researchers in operations reasearch and optimization areas. The second reason however is more pessimistic. It is well known that global general optimization is an NP hard problem. It is therefore clear that DC programming which is able to solve global general optimization is also an NP hard process. No miracle can be found that will resolve this difficulty, and any DC programming algorithm will hence always be "slow". However, this does not mean that it always will be "useless" for that reason. With the advent of parallel computing and ever faster computers appearing on the market, a hardware implementation of Thoai's algortihm could for example be imagined, where there would be one simple process for each vertex, but due to the parallel architecture, all the vertices could be updated simultaneously. This would make DC programming problems solvable within an acceptable amount of time. However, the method described in this thesis addresses an NP hard problem, and therefore there is no possibility of finding a "very fast" algorithm.

References

[Block H.D., 1962] *The perceptron: A model for brain functioning*, Reviews of Modern Physics, vol. 34, pp. 123-135, 1962.

[Bouzerdoum A. Pattison T.R., 1993] *Neural network for quadratic optimization with bound constraints.* IEEE Transactions on Neural Networks, Vol 4, No 2, March 1993.

[Brooks, R. A., 1986] *A robust layered control system for a mobil robot,* IEEE Journal of Robotics and Automation, Vol. RA2, pp. 14-23.

[Cetin B.C., Barhen J., Burdick J.W., 1993a] *Terminal Repeller Unconstrained Subenergy Tunneling (TRUST) for fast global optimization,* Journal of Optimization Theory and Applications, vol.77,no.1,April 1993,p.97-126

[Cetin B.C., Burdin J.W., Barhen J., 1993b] *Global descent replaces gradient descent to avoid local minima problem in learning with artificial neural networks,* in Proc. of IEEE Conf. on Neural Networks, San Francisco, 1993, vol.II, p.836-842.

[Christofides N., Mingozzi A., Toth P., Sandi C., 1979] *Combinatorial Optimization,* Wiley-Interscience, London and New York, 1979.

[Cotterill R. M., 1988] *Computer simulation in brain science*, Cambridge University Press, Cambridge, Mass.

[Cybenko G., 1989] *Approximation by superpositions of a sigmoidal function,* Mathematics of Control, Signals and Systems, pp. 303-314, vol 2, 1989.

[DARPA, 1988] *D.A.R.P.A. Neural networks study report,* A.F.C.E.A. International Press, 1988.

[Falk J, Hoffman K.R., 1976] *A successive underestimation method for concave minimization problems*, Mathematical Operations Research, Vol. 1., pp. 251-259.

[Flake G.W., 1993] *Nonmonotonic activation functions in multilayer perceptrons,* Institute for Advance Computer Studies, Dept of Computer Sciene, University of Maryland, College Park.

[Funahashi K.I., 1989] *On the approximate realization of continuous mappings by neural networks,* Neural Networks, Vol 2, pp 183-192, 1989.

[Green H.S., Triffet T., 1989] *A zonal model of cortical functions*, Journal of Theoretical Biology, 136, pp. 87-116, 1989.

[Grossberg S., 1968] *Some nonlinear networks capable of learning a spatial pattern of arbitrary complexity,* Proceedings of the National Academy of Sciences, Vol. 59, pp. 368-372, 1968.

[Hecht-Nielsen R., 1987a] *Kolmogorov's mapping neural network existence theorem,* IEEE First International Conference on Neural Networks, pp. 11-14, vol 3, 1987.

[Hecht-Nielsen R., 1987b] *Counterpropagation networks,* Applied Optics, pp. 4979-4985, vol 26, 1987.

[Hecht-Nielsen R., 1990] *Neurocomputers,* Addison-Wesley, Reading, Mass.

[Hiriart-Urruty J.B., 1985] *Generalized differentiability, duality and optimization problems dealing with differences of convex functions.* Lecture Notes in Economics and Mathematical Systems, 256, 37-69, Springer-Verlag, Berlin.

[Hiriart-Urruty J.B., 1986] *When is a point x satisfying $\Delta f(x)=0$ a global minimum of f.* Amer. Math. Monthly, 93, 556-558.

[Hoffman K. R., 1981] *A method for globally minimizing concave functions over convex sets,* Mathematical Programming, Vol. 20, pp. 22-32, 1981.

[Holland J., 1975] *Adaptation in natural and artificial systems,* University of Michigan Press, Ann Arbor, Michigan, 1975.

[Hopfield J.J., 1982] *Neural networks and physical systems with emergent collective computational ability,* Proc. Natl. Acad. Sci. USA, vol 79, pp. 2554-2558, April 1982.

[Hopfield J.J., 1984] *Neurons with graded response have collective computational properties like those of two-state neurons,* Proc. Natl. Acad. Sci. USA, vol 79, pp. 3088-3092, 1984.

[Hornik K., Stinchcombe M., White H., 1989] *Multilayer feedforward networks are universal approximators,* Neural Networks, pp. 359-366, vol 2, 1989.

[Hornik K., Stinchcombe M., White H., 1989] *Universal approximation of unknown mapping and its dervatives using feedforward networks,* Neural Networks, pp. 551-560, vol 3, 1990.

[Horst R., Vries J., Thoai N.V., 1988] *On finding new vertices and redundant constraints in cutting plane algorithms for global optimization.* Operations Research Letters, Volume 7, Number 2, April 1988, 85-90.

[Horst R., Tuy H., 1993] *Global Optimization, Deterministic Approaches,* Springer-Verlag, Berlin, 1993.

[Ito Y., 1991] *Representation of functions by superposition of a step or sigmoidal functions and their applications to neural network theory,* Neural Networks, Vol 4, pp 385-394, 1991.

[Kirkpatrick S., Gelatt C., Vecchi M., 1983] *Optimization by simulated annealing,* Science, vol. 220, pp. 671-680, 1983.

[Koch C., Segev I., 1989] *Methods in neuronal modeling: From sysnapses to networks,* A Bradford Book, MIT Press, Cambridge, Mass.

[Kohonen T., 1989] *Self-organization and associative memory,* third edition, Series in Information Sciences, Vol 8, Springer Verlag, Berlin.

[Koiran P., 1993] *On the complexity of approximating mappings using feedforward networks,* Neural Networks, Vol 6, pp 649-653, 1993.

[Korn G.A., 1991] *Neural Network experiments on personal computers and workstations,* Bradford Book, MIT Press, Cambridge, Massachusetts, 1991.

[Levy A.V., Montalvo A., 1985] *The tunneling algorithm for the global minimization of functions,* SIAM Journal on Scientific and Statistical Computing, Vol 6, pp 15-29, 1985.

[Lippmann R.P., 1987] *An introduction to computing with neural networks,* IEEE ASSP Magazine, April 1987, pp. 4-22.

[Montana D.J., Davis L., 1989] *Training Feedforward neural networks using genetic algorithms*, Proceedings of International Joint Conference on Artificial Intelligence, I.J.C.A.I.-89, Vol. 1, Morgan Kaufmann, Palo Alto, Calif., pp. 762-767, 1989.

[Narendra K.S., Parthasarathy K., 1990] *Identification and control of dynamical systems using neural networks,* IEEE Transactions on Neural Networks, Vol. 1, pp. 4-27.

[Naylor J., Li, K., 1988] *Speaker recognition using Kohonen's self organizing feature map algorithm,* Neural Network Supplement: INNS Abstracts, 1988.

[Pham Dinh T., Bernoussi S., 1989] *Numerical methods for solving a class of global nonconvex optimization problems,* International Series of Numerical Mathematics, Vol 87, 1989 Birkhauser Verlag Basel.

[Press W., Flannery B., Teukolsky S., Vetterling W.,1990] *Numerical recipes in C, the art of scientific computing,* Cambridge University Press, 1990.

[Rosenblatt F., 1962] *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*, Spartan Books, Washington DC, 1962.

[Rumelhart D.E., Hinton G.E., Williams R.J., 1986] *Learning internal representations by error propagation,* Parallel Distributed Processing: Explorations in the Microstructure of Cognitions, Vol 1: Foundations. MIT Press, Cambridge, Mass., pp. 318-362.

[Sejnowski T., Hinton G., 1986] *Learning and relearning Boltzman machines,* Parallel Distributed Processing vol1., The MIT Press, Cambridge, MA, pp 282-317.

[Sejnowski T., Rosenberg C., 1987] *Parallel networks that learn to pronounce English text,* Complex systems, vol. 1, pp. 145-168, 1987.

[Shi P., Ward R.K., 1990] *The case for abandoning the biological resemblance restriction: An example of neural network solution of simultaneous equations,* IJCNN International Joint Conference on Neural Networks, San Diego, CA, 1990.

[Simpson P., 1990] *Artificial neuron systems*, New York, Pergamon Press, 1990.

[Sudharsanan S., Sundareshan M.K., 1991] *Exponential stability and a systematic synthesis of a neural network for quadratic minimization,* Neural Networks, Vol 4, pp 599-613, 1991.

[Thieu T.V., Tam B.T., Ban V.T., 1983] *An outer approximation method for globally minimizing a concave function over a compact convex set*, Acta Mathematica Vietnamica, Vol. 8, pp 21-40, 1983.

[Thoai N.V., 1988] *A modified version of Tuy's method for solving DC programming problems*. Optimization, 19, 665-674.

[Thoai N.V., 1993] Personnal Communication.

[Tuy H., 1964] *Concave programming under linear constraints.* Soviet Mathematics, 5, 1437-1440.

[Tuy H., 1985] *A general deterministic approach to global optimization via DC programming,* Fermat Days 1985. Mathematics for Optimization, Elsevier Science Publishers B.V., North Holland, 1986.

[Tuy H., 1987] *Global minimization of a difference of two convex functions.* Mathematical Programming Study, 30, 150-182.

[Wassermann P., 1989] *Neural Computing: Theory and practice,* Van Nostrand Reinhold, New York, 1989.

[Werbos P., 1988] *Generalization of backpropagation with application to a recurrent gas market model*, Neural Networks, Vol. 1, pp. 339-356, 1988.