APPLICATIONS OF NEURAL NETWORKS TO CONTROL SYSTEMS

by
António Eduardo de Barros Ruano

Thesis submitted to the
UNIVERSITY OF WALES

for the degree of
DOCTOR OF PHILOSOPHY

School of Electronic Engineering Science
University of Wales, Bangor

February, 1992

Statement of Originality

The work presented in this thesis was carried out by the candidate. It has not been previously presented for any degree, nor is at present under consideration by any other degree awarding body.

Candidate:-

Antonio E. B. Ruano

Directors of Studies:-

Dr. Dewi I. Jones

Prof. Peter J. Fleming

ACKNOWLEDGMENTS

I would like to express my sincere thanks and gratitude to Prof. Peter Fleming and Dr. Dewi Jones for their motivation, guidance and friendship throughout the course of this work.

Thanks are due to all the students and staff of the School of Electronic Engineering Science. Special thanks to the Head of Department, Prof. J.J. O'Reilly, for his encouragement; Siân Hope, Tom Crummey, Keith France and David Whitehead for their advice on software; Andrew Grace, for fruitful discussions in optimization; and Fabian Garcia, for his advice in parallel processing.

I am also very grateful to Prof. Anibal Duarte from the University of Aveiro, Portugal, whose advice was decisive to my coming to Bangor.

I would like to acknowledge the University of Aveiro, University College of North Wales and the 'Comissão Permanente Invotan' for their financial support.

I am also very grateful to all my friends in Bangor, who made my time here a most enjoyable one.

Special thanks and love to my parents, António Delfim and Maria Celeste, for their life-time care and encouragement. To my parents-in-law, for their continuous support.

Finally I thank my wife and colleague, Maria da Graça, for her love, patience and support and for the long hours of fruitful discussions; my daughter, Carina Alexandra, for cheering me up at those special moments.

À Carina e à Graça

ABSTRACT

This work investigates the applicability of artificial neural networks to control systems. The following properties of neural networks are identified as of major interest to this field: their ability to implement nonlinear mappings, their massively parallel structure and their capacity to adapt.

Exploiting the first feature, a new method is proposed for PID autotuning. Based on integral measures of the open or closed loop step response, multilayer perceptrons (MLPs) are used to supply PID parameter values to a standard PID controller. Before being used on-line, the MLPs are trained off-line, to provide PID parameter values based on integral performance criteria. Off-line simulations, where a plant with time-varying parameters and time varying transfer function is considered, show that well damped responses are obtained. The neural PID autotuner is subsequently implemented in real-time. Extensive experimentation confirms the good results obtained in the off-line simulations.

To reduce the training time incurred when using the error back-propagation algorithm, three possibilities are investigated. A comparative study of higher-order methods of optimization identifies the Levenberg-Marquardt (LM) algorithm as the best method. When used for function approximation purposes, the neurons in the output layer of the MLPs have a linear activation function. Exploiting this linearity, the standard training criterion can be replaced by a new, yet equivalent, criterion. Using the LM algorithm to minimize this new criterion, together with an alternative form of Jacobian matrix, a new learning algorithm is obtained. This algorithm is subsequently parallelized. Its main blocks of computation are identified, separately parallelized, and finally connected together. The training time of MLPs is reduced by a factor greater than 70 executing the new learning algorithm on 7 Inmos transputers.

Contents

Chapter 1

Introduction

1.1 General overview

In recent years there has been an increasing interest in the development and application of artificial neural networks technology. Neural network research has spread through almost every field of science, several journals [1][2][3] and major conferences [4][5] now being entirely devoted to this subject.

Control systems form one area where artificial neural networks have found ready application. In every major conference in the field of control systems, sessions are now devoted to artificial neural networks applications. Special issues [6][7][8] of important publications on control systems have been dedicated to this new subject.

The objectives of the work described in this thesis are to investigate the potential applications of neural networks to control systems, to identify suitable applications where neural network solutions are superior to conventional techniques, and to further develop the relevant approaches, with a view to practical implementation.

When this PhD. programme started, in October 1988, applications of neural networks in control systems were in a embryonic state. From a literature survey conducted at that time [9], two important properties of artificial neural networks were identified as being of great potential interest to the area of control systems: their capacity to approximate nonlinear mappings, and their ability to learn and adapt. These properties could then be exploited for nonlinear systems identification, adaptive control of nonlinear systems and PID autotuning, among other areas. Clearly, this preliminary analysis was found to be correct, as confirmed by the number of publications, appeared during these last three years, where applications of neural networks on nonlinear systems identification and adaptive control of nonlinear systems have been proposed.

Because of the practical importance of the problem, the ability of artificial neural networks to arbitrarily approximate nonlinear mappings is applied, in this thesis, to PID autotuning. Despite all the advances made in control theory, the majority of the regulators in industry are of the PID type. Large industrial plants may contain hundreds of these simple regulators. Manual tuning is an operation that, to be accurately performed, takes a considerable amount of time. Whether because of plant changes or because of component

ageing, retuning needs to be performed regularly. Methods that can provide automatic tuning of PID controllers are, therefore, of a great practical importance.

Several different methods have been proposed for tuning PID regulators, starting from the work of Ziegler-Nichols [10]. For instance, Åstrom and Hägglund [11] proposed the use of relay feedback, in conjunction with different design methods [11][12], to automate the tuning process of PID regulators. This is a robust technique; however some reservations on its use have been expressed by commissioning engineers, mainly because the method is so different from the standard procedures used for manual tuning. In practice, the commissioning engineer applies an input step to either the plant or the closed-loop system, and, based on the system response and on his experience on the plant, iteratively tunes the PID controller. In this respect, there are similarities to the method proposed in this thesis. The main difference, however, lies in the replacement of the human operator by one type of artificial neural network, multilayer perceptrons [13]. These gain their experience in tuning by being trained off-line. The skill level of the engineer corresponds to the criterion used to derive the PID parameters used in the training phase. By using criteria that produce good responses, the neural PID tuner mimics an experienced plant operator, with the advantage that several iterations are not needed for tuning.

One problem often reported in applications of multilayer perceptrons is the enormous number of iterations (tens or hundreds of thousands) needed for training, particularly when the well known error back-propagation algorithm [13] is used. This means, in practice, that the training of multilayer perceptrons is a time consuming operation. As the technique proposed for PID autotuning employs this type of artificial neural network, trained off-line, the time-consuming training phase incurred by the use of the error back-propagation algorithm would constitute one drawback of the new method. For these reasons, off-line training of MLPs was investigated, in the context of nonlinear optimization.

The error back-propagation algorithm is an efficient implementation of a steepest-descent method [14], which intelligently exploits the topology of the MLPs in order to minimize the computational costs per iteration. However, as efficient (per iteration) as it can be, it inherits the disadvantages of the steepest descent method, namely, poor reliability and poor rate of convergence. Three major possibilities arise for the reduction of the training time: using higher-order methods of optimization, fully exploiting the topology of the MLPs, and using parallel processing techniques. All these possibilities have been fully exploited in this thesis, and order of magnitude reductions in training time have been achieved as a result of their joint application.

Second-order optimization methods, in contrast with the error back-propagation algorithm, are guaranteed to converge to a minimum (although, of course this may be local),

and in certain ideal conditions achieve a quadratic rate of convergence. Three second-order methods [14] (quasi-Newton, Gauss-Newton, and Levenberg-Marquardt) were applied to the training of MLPs and their performance compared using common examples. As a result of this research, one method was clearly identified as producing the best convergence rate.

The error back-propagation algorithm exploits the topology of the multilayer perceptrons to minimize the cost of computing the gradient vector. In the proposed PID autotuning method, and indeed in the majority of the control systems applications, multilayer perceptrons are employed for nonlinear function approximation purposes. For this reason, the neurons in the output layer employ a linear activation function, instead of a nonlinear (typically a sigmoid) function. This linearity, as explained later, can be exploited and a new training criterion developed. The main advantage of using this approach, as will be shown, is a great reduction in the number of iterations needed for convergence.

As a result of applying a reformulated criterion, together with the second-order optimization method previously chosen, a new learning algorithm is obtained. Separate blocks of computation can be identified in this new algorithm. Each one of these computational tasks is separately parallelized, the main effort being put into the most computationally intensive block. It is shown that by performing a careful partitioning of the operations involved in each block and by exploiting the characteristics of the target processor (the Inmos transputer [15]), and by choosing a topology for the processors which matches well with the partitioning, it is possible to obtain a good parallel efficiency for each one of the blocks. This results in an overall parallel algorithm with parallel efficiency very close to the theoretical optimum.

Having investigated the learning problem associated with the proposed neural PID autotuner, the practical implementation of this technique is addressed next. The new technique is implemented in real-time. With a view to easing the experimental task, and to allow easy upgradabilty to future phases of this work, a modular real-time system is built in Occam [16] and implemented on a network of Inmos transputers [15].

1.2 Outline of thesis and major achievements

Chapter 1 presents a general overview of the work developed during the course of this research, stating, at the same time, the main contributions of this work. An outline of the thesis, by chapters, is also given.

Although work on artificial neural networks began some forty years ago, widespread interest in this field of research has only taken place in the past six or seven years. Consequently, for the large majority of researchers, artificial neural networks are a new field of science. Chapter 2 gives the background information needed for the next Chapters, and gives the reader a concise introduction to the field of artificial neural networks. Special

attention is given to applications of artificial neural networks in control systems.

A chronological perspective of neural network research is presented, and different proposals for the neuron model, the pattern of interconnectivity, and, especially, the learning mechanisms, are pointed out. Comparisons between aspects found in artificial neural networks and the archetypal neural network, the human brain, are also made.

Among several different ANN models, three of them are found to be most relevant for control systems applications. The Hopfield model, the Cerebellar Model Articulation Controller (CMAC), and the multilayer perceptrons (MLPs) are described in Chapter 2.

A detailed, up-to-date, review of the applications of neural networks in control systems is also conducted in Chapter 2. The main advantages of using this new technology, in comparison with conventional techniques, are highlighted, and the current major open questions pointed out.

The detailed overview of current applications of neural networks to control systems is considered to be an important contribution of Chapter 2.

The PID controller, both in its continuous and discrete versions, is introduced in Chapter 3. An overview of existing methods for automatically tuning PID controllers is also presented.

This Chapter describes an original approach to PID autotuning. In this new approach, multilayer perceptrons, exploiting their mapping capabilities, are employed to supply PID values to a standard PID controller.

Details of the method are given and its performance assessed using off-line simulations. An example is given where the neural PID autotuner is used to control a plant, which has time-varying parameters as well as a varying transfer function structure.

This new approach to the problem of PID autotuning is one of the major contributions of this thesis.

The technique described in Chapter 3 employs, for on-line control, multilayer perceptrons previously trained off-line. The enormous time taken to perform the training, using the standard error back-propagation learning algorithm [13], was soon identified as one of the drawbacks of the proposed PID autotuning technique. For this reason, Chapter 4 is devoted to a study of training algorithms for multilayer perceptrons, with the aim of reducing the training time.

Starting with a derivation of the error back-propagation algorithm, the reasons for its poor performance are pointed out using a simple example. Three second order optimization methods [14] (quasi-Newton, Gauss-Newton and Levenberg-Marquardt) are introduced and

their performance compared using common examples.

Next, it is shown that, for the class of multilayer perceptrons most employed for control systems applications, the standard learning criterion can be reformulated into a new, yet equivalent, learning criterion. It is shown that the use of this reformulated criterion achieves a considerable reduction in the number of iterations needed for convergence. When considering the use of this new formulation in Gauss-Newton and Levenberg-Marquardt methods, a Jacobian matrix must be computed. For this purpose, three Jacobian matrices, one of them proposed by the author, are introduced, and compared in terms of computational complexity and rates of convergence achieved by the Levenberg-Marquardt method.

The proposal of the reformulated criterion, for the learning problem of multilayer perceptrons, is one major contribution of Chapter 4. The proposal of a new form for the Jacobian matrix is also an important innovation. Together, they reduce the training time of MLPs by at least one order of magnitude, often more.

As a result of the research described in Chapter 4, a new learning algorithm was introduced. Chapter 5 describes how further reductions in the training time can be achieved by means of a parallel version of this algorithm. The parallel learning algorithm is coded in the Occam language [16], and executed on an array of Inmos transputers [15].

The main blocks of computation in the learning algorithm are identified and separately parallelized. The major effort, however, is directed towards the most computationally intensive task of the algorithm:- the least squares solution of overdetermined systems of equations. This problem is solved employing a QR decomposition [14] obtained using Householder reflections [14].

By modifying the sequence of operations that are performed in a known parallel solution for this type of problem, it is shown, in Chapter 5, that a boost in efficiency can be obtained. After successfully parallelizing this phase, the other blocks of the learning algorithm are parallelized and afterwards interconnected, to form a complete parallel learning algorithm.

Because of its widespread use, the set of modifications introduced in the parallel QR algorithm, is considered to be an important contribution in itself. The parallel learning algorithm enables the training of multilayer perceptrons to be performed orders of magnitude faster. It is then considered to be a major contribution of this thesis.

In Chapter 6 the practical implementation of the new PID autotuning method is addressed. A real-time system is built, in Occam, using an array of transputers, to assess the performance of the neural PID autotuner in real-time.

For experimental convenience the plant is simulated digitally at this stage of the work but the system has been constructed such that the transition to the control of real plants can be

carried out in a straightforward manner. During the course of this work some implementation problems were identified and these are reported in this Chapter. Real-time results of the connectionist approach to PID autotuning are presented.

Because of its importance for the practical implementation of the new PID autotuning technique, the implementation of the real-time system is also considered to be a significant contribution of this thesis.

Finally, Chapter 7 completes the thesis with conclusions and suggestions for further work.

Chapter 2

Artificial Neural Networks

2.1 Introduction

In the past few years a phenomenal growth in the development and application of artificial neural networks technology has been observed. Neural networks research has spread through almost every field of science, covering areas as different as character recognition [17][18][19], medicine [20][21][22], speech recognition and synthesis [23][24], image processing [25][26], robotics [27][28] and control systems [29][30].

Although work on artificial neural networks began some forty years ago, widespread interest in this area has only taken place in the past six or seven years. This means that, for the large majority of researchers, artificial neural networks are a new and unfamiliar field of science. For this reason a brief introduction to this reborn field is one of the objectives of this Chapter.

Writing an introduction to artificial neural networks, in a short number of pages, is not an easy task. Although active research in this area is recent, or precisely because of that, a large number of neural networks have been proposed. Almost each one of them is introduced with a specific application in mind, and uses its own learning rule. This task becomes more complicated if we consider that contributions to this field come from a broad range of areas of science, like neurophysiology, psychology, physics, electronics, control systems and mathematics, and applications are targeted for an even wider range of disciplines. For these reasons, this introduction is confined to a broad view of the field of artificial neural networks, without entering into the detail of any particular subject. Rather, references to work which was found important for a clear view of the area will be given.

Some ANN models are described in this introduction. As the focus of this thesis is in control systems applications, the models that are described are the ones that were found more relevant for this specific area.

Another objective of this Chapter is to give ideas of where and how neural networks can be applied in control systems, as well as to summarize the main advantages that this new technology might offer over conventional techniques. There are at present no definitive answers to these questions. However, since this PhD. programme started, a large number of applications of ANNs to control systems have been proposed, enabling trends to be identified.

These will be pointed out in this Chapter and some of the most relevant applications will be discussed.

The final objective of this Chapter is to introduce conventions which will be used throughout the thesis. This will be done throughout the text.

The outline of this Chapter is as follows:

An introduction to artificial neural networks is given in section 2.2. Primarily, a chronological perspective of ANN research is presented. As artificial neural networks are, to a great extent, inspired by the current understanding of how the brain works, a brief introduction to the basic brain mechanisms is given in section 2.2.1. Section 2.2.2 shows that the basic concepts behind the brain mechanism are present in artificial neural networks. However, different approximations to the neuron transfer function, the pattern of connectivity and the learning mechanisms exist, giving rise to different neural network models. Three of those models will be discussed. Hopfield networks are described in section 2.2.2.1. CMAC networks are discussed in section 2.2.2.2. Finally multilayer perceptrons are introduced in section 2.2.2.3.

Section 2.3 presents an overview of the applications of artificial neural networks to control systems. The attractive features offered by neural networks to this area are described and this is followed by a brief summary of relevant applications for robotics and fault detection systems. Section 2.3.1 will discuss the role of ANN for nonlinear systems identification. It will be shown that neural networks have been extensively used to approximate forward and inverse models of nonlinear dynamical systems. Building up on the modelling concepts introduced, neural control schemes are discussed in section 2.3.2. During the last three years, artificial neural networks have been integrated into almost every available design approach. The largest number of applications, however, seems to be centred around model reference adaptive control and predictive control schemes. Therefore these different designs will be highlighted in sections 2.3.2.1 and 2.3.2.2 respectively.

Finally, some conclusions are given in section 2.4.

2.2 Artificial neural networks: a brief tutorial

The field of artificial neural networks (they also go by the names of connectionist models, parallel distributed processing systems and neuromorphic systems) is very active nowadays. Although for many people artificial neural networks can be considered a new area of research, which developed in the late eighties, work in this field can be traced back to more than forty years ago [31][32].

McCulloch and Pitts [33], back in 1943, pointed out the possibility of applying

Boolean algebra to nerve net behaviour. They essentially originated neurophysiological automata theory and developed a theory of nets.

In 1949 Donald Hebb, in his book *The organization of behaviour* [34] postulated a plausible qualitative mechanism for learning at the cellular level in brains. An extension of his proposals is widely known nowadays as the Hebbian learning rule.

In 1957 Rosenblatt developed the first neurocomputer, the perceptron. He proposed a learning rule for this first artificial neural network and proved that, given linearly separable classes, a perceptron would, in a finite number of training trials, develop a weight vector that would separate the classes (the famous *perceptron convergence theorem*). His results were summarised in a very interesting book, *Principles of Neurodynamics* [35].

About the same time Bernard Widrow modelled learning from the point of view of minimizing the mean-square error between the output of a different type of ANN processing element, the ADALINE [36], and the desired output vector over the set of patterns [37]. This work has led to modern adaptive filters. Adalines and the Widrow-Hoff learning rule were applied to a large number of problems, probably the best known being the control of an inverted pendulum [38].

Although people like Bernard Widrow approached this field from an analytical point of view, most of the research on this field was done from an experimental point of view. In the sixties many sensational promises were made which were not fulfilled. This discredited the research on artificial neural networks. About the same time Minsky and Papert began promoting the field of artificial intelligence at the expense of neural networks research. The death sentence to ANN was given in a book written by these researchers, *Perceptrons* [39], where it was mathematically proved that these neural networks were not able to compute certain essential computer predicates like the EXCLUSIVE OR boolean function.

Until the 80's, research on neural networks was almost nil. Notable exceptions from this period are the works of Amari [40], Anderson [41], Fukushima [42], Grossberg [43] and Kohonen [44].

Then in the middle 80's interest in artificial neural networks started to rise substantially, making ANN one of the most active current areas of research. The work and charisma of John Hopfield [45][46] has made a large contribution to the credibility of ANN. With the publication of the PDP books [13] the field exploded. Although this persuasive and popular work made a very important contribution to the success of ANNs, other reasons can be identified for this recent renewal of interest:

- One is the desire is to build a new breed of powerful computers, that can solve problems that are proving to be extremely difficult for current digital computers

(algorithmically or rule-based inspired) and yet are easily done by humans in everyday life [47]. Cognitive tasks like understanding spoken and written language, image processing, retrieving contextually appropriate information from memory, are all examples of such tasks.

- Another is the benefit that neuroscience can obtain from ANN research. New artificial neural network architectures are constantly being developed, and new concepts and theories being proposed to explain the operation of these architectures. Many of these developments can be used by neuroscientists as new paradigms for building functional concepts and models of elements of the brain.

- Also the advances of VLSI technology in recent years, turned the possibility of implementing ANN in hardware into a reality. Analog, digital and hybrid electronic implementations [48][49][50][51] are available today, with commercial optical or electro-optical implementations being expected in the future.

- The dramatic improvement in processing power observed in the last few years makes it possible to perform computationally intensive training tasks which would, with older technology, require an unaffordable time.

Research into ANNs has lead to several different architectures being proposed over the years. All of them try, to a greater or lesser extent, to exploit the available knowledge of the mechanisms of the human brain. Before describing the structure of artificial neural networks we should give a brief description of the structure of the archetypal neural network.

2.2.1 Biological neural networks

One point that is common to the various ANN models is their biological inspiration. Let us, then, very briefly consider the structure of the human brain.

It is usually recognized that the human brain can be viewed at three different levels [52]:

a) the neurons - the individual components in the brain ciruitry;

b) groups of a few hundreds neurons which form modules that may have highly specific functional processing capabilities;

c) at a more macroscopic level, regions of the brain containing millions of neurons or groups of neurons, each region having assigned some discrete overall function.

The basic component of brain circuitry is a specialized cell called the neuron. As Fig. 2.1 shows, a neuron consists of a *cell body* with finger-like projections called *dendrites* and a long cable-like extension called the *axon*. The axon may branch, each branch terminating in a structure called the *nerve terminal*.

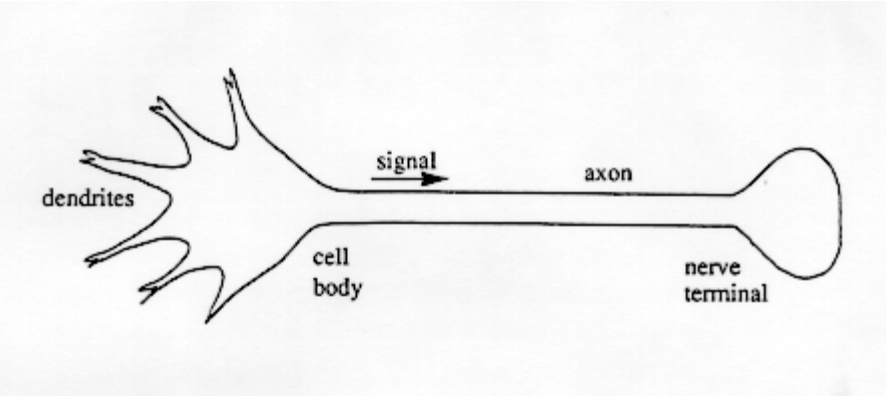


Fig. 2.1 - A biological neuron

Neurons are electrically excitable and the cell body can generate electrical signals, called *action potentials*, which are propagated down the axis towards the nerve terminal. The electrical signal propagates only in this direction and it is an all-or-none event. Information is coded on the frequency of the signal.

The nerve terminal is close to the dendrites or body cells of other neurons, forming special junctions called *synapses*. Circuits can therefore be formed by a number of neurons. Branching of the axon allows a neuron to form synapses on to several other neurons. On the other end, more than one nerve terminal can form synapses with a single neuron.

When the action potential reaches the nerve terminal it does not cross the gap, rather a chemical neurotransmitter is released from the nerve terminal. This chemical crosses the synapse and interacts on the postsynaptic side with specific sites called *receptors*. The combination of the neurotransmitter with the receptor changes the electrical activity on the receiving neuron.

Although there are different types of neurotransmitters, any particular neuron always releases the same type from its nerve terminals. These neurotransmitters can have an excitatory or inhibitory effect on the postsynaptic neuron, but not both. The amount of neurotransmitter released and its postsynaptic effects are, to a first approximation, graded events on the frequency of the action potential on the presynaptic neuron. Inputs to one neuron can occur in the dendrites or in the cell body. These spatially different inputs can have different consequences which alter the effective strengths of inputs to a neuron.

Any particular neuron has many inputs (some receive nerve terminals from hundreds

or thousands of other neurons), each one with different strengths, as described above. The neuron integrates the strengths and fires action potentials accordingly.

The input strengths are not fixed, but vary with use. The mechanisms behind this modification are now beginning to be understood. These changes in the input strengths are thought to be particularly relevant to learning and memory.

In certain parts of the brain groups of hundreds of neurons have been identified and are denoted as *modules*. These have been associated with the processing of specific functions. It may well be that other parts of the brain are composed of millions of these modules, all working in parallel, and linked together in some functional manner.

Finally, at a higher level, it is possible to identify different regions of the human brain. Of particular interest is the cerebral cortex, where areas responsible for the primary and secondary processing of sensory information have been identified. Therefore our sensory information is processed, in parallel, through at least two cortical regions, and converges into areas where some association occurs. In these areas, some representation of the world around us is coded in electrical form.

2.2.2 Different types of artificial neural networks

From the brief discussion on the structure of the brain, some broad aspects can be highlighted: the brain is composed of a large number of small processing elements, the neurons, acting in parallel. These neurons are densely interconnected, one neuron receiving inputs from many neurons and sending its output to many neurons. The brain is capable of learning, which is assumed to be achieved by modifying the strengths of the existing connections.

These broad aspects are also present in all ANN models. However a detailed description of the structure and the mechanisms of the human brain is currently not known. This leads to a profusion of proposals for the model of the neuron, the pattern of interconnectivity, and especially, for the learning mechanisms. Artificial neural networks can be loosely characterized by these three fundamental aspects [53].

Before developing a model for the neuron, it is convenient to introduce some conventions that will be used through this thesis.

Primarily, scalars, vectors and matrices will be employed. To enable an easy identification of these quantities with different dimensions, the following convention will be

used:

- a scalar is denoted by normal weight characters;

- a vector will be denoted by bold, lower case characters;

- a matrix will be denoted by bold, upper case characters.

In the case of vectors and matrices, it is sometimes necessary to isolate one element or one partition. An element will be identified by specifying, in subscript, the corresponding indices of the element. A range in one of the indices may be indicated by two different manners: explicitly, by specifying the initial and final elements, separated by two dots '..', or implicitly, by denoting, in italic, the set of the elements in the range; a single dot '.' denotes all the elements in the corresponding dimension. For instance:

$\mathbf{A}_{i,j}$ is the element which corresponds to the i^{th} row and the j^{th} column of matrix **A**;

$\mathbf{A}_{i..k,j}$ is the column vector that corresponds to the elements in the rows from i until k of column j of matrix **A**;

$\mathbf{A}_{l,j}$ is the column vector that corresponds to the elements of the j^{th} column matrix **A** whose row indices are the elements of the set *l*;

$\mathbf{A}_{.,j}$ is the column vector corresponding to the j^{th} column of matrix **A**.

Also, whenever variables are defined in an iterative way, the iteration number will be explicitly shown inside square brackets. For instance, x[k] denotes the value of x at iteration k.

Taking into account the brief description of the biological neuron given in the last section, a model of a neuron can be obtained [54] as follows. In the neuron model the frequency of oscillation of the i^{th} neuron will be denoted by $\mathbf{o}_i$ (we shall consider that **o** is a row vector). If the effect of every synapse is assumed independent of the other synapses, and also independent of the activity of the neuron, the principle of spatial summation of the effects of the synapses can be used. As it is often thought that the neuron is some kind of leaky integrator of the presynaptic signals, the following differential equation can be considered for $\mathbf{o}_i$:

$$\frac{d\mathbf{o}_i}{dt} = \sum_{i \in I} \mathbf{X}_{i,j}(\mathbf{o}_j) - g(\mathbf{o}_i) \qquad (2.1)$$

where $\mathbf{X}_{i,j}(.)$ is a function which denotes the strength of the j^{th} synapse out of the set *i* and g(.) is a loss term, which must be nonlinear in $\mathbf{o}_i$ to take the saturation effects into account.

Some ANN models are concerned with the transient behaviour of (2.1). The majority, however, consider only the stationary input-output relationship, which can be obtained by equating (2.1) to 0. Assuming that the inverse of g(.) exists, $\mathbf{o}_i$ can then be given as:

$$\mathbf{o}_i = g^{-1}\left(\sum_{i \in I} \mathbf{X}_{i,j}(\mathbf{o}_j) + \mathbf{b}_i \right) \tag{2.2}$$

where $\mathbf{b}_i$ is the bias or threshold for the i^{th} neuron. To derive function $\mathbf{X}_{i,j}(.)$ the spatial location of the synapses should be taken into account, as well as the influence of the excitatory and inhibitory inputs. If the simple approximation (2.3) is considered:

$$\mathbf{X}_{i,j}(\mathbf{o}_j) = \mathbf{o}_j \mathbf{W}_{j,i} \tag{2.3}$$

where $\mathbf{W}$ denotes the weight matrix, then the most usual model of a single neuron model is obtained:

$$\mathbf{o}_i = g^{-1}\left(\sum_{i \in I} \mathbf{o}_j \mathbf{W}_{j,i} + \mathbf{b}_i \right) \tag{2.4}$$

The terms inside brackets are usually denoted as the net input:

$$\mathbf{net}_i = \sum_{i \in I} \mathbf{o}_j \mathbf{W}_{j,i} + \mathbf{b}_i = \mathbf{o}_i \mathbf{W}_{i,i} + \mathbf{b}_i \tag{2.5}$$

Denoting $g^{-1}(.)$ by f, usually called the *output or activation function*, we then have:

$$\mathbf{o}_i = f(\mathbf{net}_i) \tag{2.6}$$

Sometimes it is useful to incorporate the bias in the weight matrix. This can be easily done by defining:

$$\begin{aligned} \mathbf{o}'_i &= \begin{bmatrix} \mathbf{o}_i & 1 \end{bmatrix} \\ \mathbf{W}' &= \begin{bmatrix} \mathbf{W} \\ \mathbf{b}^T \end{bmatrix} \end{aligned} \tag{2.7}$$

so that the net input is simply given as:

$$\mathbf{net}_i = \mathbf{o}'_i \mathbf{W}'_{i,i} \tag{2.8}$$

Typically f(.) is a sigmoid function, which has low and high saturation limits and a proportional range between. Other deterministic functions are, however, used. In a special type of ANN, the Boltzmann machine [13], this function is stochastic.

Other networks, like one type of Hopfield network and Grossberg's family of ANNs, take the transient behaviour into account. The latter also distinguishes between excitatory and

inhibitory inputs.

Another difference between the neuron models lies in the admissible value for their outputs: some assume that the output is unbounded continuous, others admit limited ranges $[-1, 1]$ or $[0, 1]$, and some employ discrete values $\{-1, 1\}$ or $\{0, 1\}$.

Considering now the pattern of connectivity, in certain ANN models which distinguish between excitatory and inhibitory inputs, two weight matrices are used. Some networks admit feedback connections, while others are feedforward structures. In some ANN models, the neurons may be grouped into layers, connections being allowed between neurons in consecutive layers.

With respect to the learning mechanism, it can be divided broadly into three major classes:

a) supervised learning - this learning scheme assumes that the network is used as an input-output system. Associated with some input matrix, **I**, there is a matrix of desired outputs, or teacher signals, **T**. The dimensions of these two matrices are $m*n_i$ and $m*n_o$, respectively, where m is the number of patterns in the training set, n_i is the number of inputs in the network and n_o is the number of outputs. The aim of the learning phase is to find values of the weights and biases in the network such that, using **I** as input data, the corresponding output values, **O,** are as close as possible to **T**. The most commonly used minimization criterion is:

$$\Omega = \mathrm{tr}(\mathbf{E}^T\mathbf{E}) \tag{2.9}$$

where tr(.) denotes the trace operator and **E** is the error matrix defined as:

$$\mathbf{E} = \mathbf{T} - \mathbf{O} \tag{2.10}$$

Examples of supervised learning rules are the Widrow-Hoff rule, or LMS (least-mean-square) rule [37] and the error back-propagation algorithm [13].

b) reinforcement learning - this kind of learning also involves minimisation of some cost function. In contrast with supervised learning, however, this cost function is only given to the network from time to time. In other words, the network does not receive a teaching signal at every training pattern but only a score that tells it how it performed over a training sequence. These cost functions are, for this reason, highly dependent on the application. A well known example of reinforcement learning can be found in [55].

c) competitive learning - this type of learning does not employ any teaching signal. Instead an input pattern is presented to the network and the neurons compete among themselves. The processing elements that emerge as winners of the competition are allowed to modify their weights (or modify their weights in a different way from those of the non-winning neurons). Examples of this type of learning are Kohonen learning [54] and the Adaptive Resonance Theory (ART) [56].

As already mentioned, several different ANN models have been proposed throughout these last 40 years of research. A description of all the different models, or even a significant subset of them, is outside the scope of this introduction. A detailed description of several different models can be found in, for instance, [13][51][54] [57][58]and [59]. We shall only describe the ones found more relevant in control systems applications, namely the Hopfield network (detailed also because of its historical importance), CMAC (Cerebellar Model Articulation Controller) networks and multilayer perceptrons.

2.2.2.1 Hopfield networks

The Hopfield networks, as Fig. 2.2 illustrates, are dense networks where each neuron is connected to one input and to the outputs of all the other neurons, via weights.

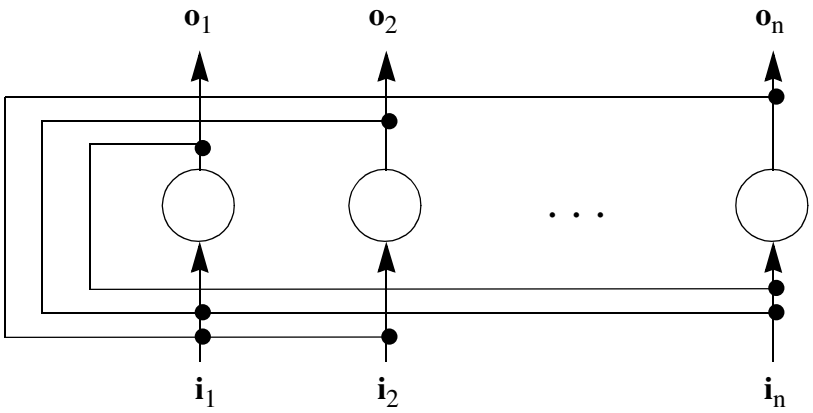


Fig. 2.2 - Hopfield network

Two models of neurons have been proposed by John Hopfield. The earliest model [45] employed two-states threshold neurons. In this case the net input of the neurons is given by:

$$\mathbf{net} = \mathbf{oW} + \mathbf{i} \tag{2.11}$$

The output of the neurons has two states, which can be represented as -1 and +1. Each neuron samples its net input at random times. It changes the value of its output or leaves it fixed according to the rule:

$$\begin{aligned} \mathbf{o}_i &\to 1 \qquad \text{if } \mathbf{net}_i > \mathbf{b}_i \\ \mathbf{o}_i &\to 0 \qquad \text{if } \mathbf{net}_i < \mathbf{b}_i \end{aligned} \tag{2.12}$$

where **b** is the threshold row vector.

One of the important contributions of Hopfield was to formulate the operation of the network in terms of an 'energy' function. Considering the function:

$$E = -\frac{1}{2}\mathbf{o}\mathbf{W}\mathbf{o}^T - \mathbf{i}\mathbf{o}^T + \mathbf{b}\mathbf{o}^T, \tag{2.13}$$

where **W** is symmetric and has a null diagonal, the change (ΔE) in energy due to the change of the output of the neuron i by $\Delta \mathbf{o}_i$ is:

$$\Delta E = -(\mathbf{net}_i - \mathbf{b}_i)\Delta \mathbf{o}_i \tag{2.14}$$

Independent of the sign of $\Delta \mathbf{o}_i$, the product in (2.14) is always positive, because of the threshold rule (2.12). This means that the energy can only decrease. As the energy is bounded, independently of the initial state of the network, the network will always converge to a stable state in a finite number of steps.

Later, Hopfield [46] released the restrictions of the stochastic update rule and the two states outputs.

The neuron i is now described by the following system of differential equations:

$$\frac{d\mathbf{u}_i}{dt} = \mathbf{o}\mathbf{W}_{.,i} + \mathbf{i}_i - \mathbf{b}_i\mathbf{u}_i \tag{2.15}$$

where $\mathbf{u}_i$ denotes the state of neuron i, and

$$\mathbf{o}_i = f(\mathbf{u}_i), \tag{2.16}$$

f(.) being a monotonically increasing function, typically a sigmoid function.

The energy function for the continuous case is [46]:

$$E = -\frac{1}{2}\mathbf{o}\mathbf{W}\mathbf{o}^T - \mathbf{i}\mathbf{o}^T + \sum_{i=1}^{n} \mathbf{b}_i \int_0^{\mathbf{o}_i} f^{-1}(\eta)d\eta \tag{2.17}$$

It can be proved [46] that assuming that **W** is symmetric (note that self-connections are now allowed) and that f(.) is a monotonically increasing function, $\frac{d}{dt}E \le 0$. This derivative is 0 only when $\frac{d\mathbf{u}_i}{dt} = 0$, for $i = 1, \ldots, n$. This occurs at the equilibrium points.

These equilibrium points, or stable states, are used to store important data that the user wants to recall later. To be more precise, when a network is started from an initial point that is close, say, in terms of Hamming distance, to a certain fixed point, the system state rapidly converges to that fixed point. The initial point is said to be in the *basis of attraction* of that stable state and the network is then said to recall the stored pattern, or complete the input pattern. Since the stored pattern is recalled by content and not, as in conventional computer architectures, by address, the network is acting like a content-addressable memory (CAM). This is one of the applications of Hopfield networks.

Another use of these type of networks is in optimization. Examples can be found in [60][61][62][63]:

Several different learning rules have been proposed for Hopfield networks. A good summary can be found in [64] and [65].

Current research on Hopfield networks is centred around implementation, increasing the network's capacity (number of stable states), effective means of removing spurious (not desired) stable states, and shaping basins of attraction [66].

2.2.2.2 CMAC networks

CMAC (Cerebellar Model Articulation Controller) was conceived by Albus [67] as a neuronal network model of the human cerebellar cortex. Different implementations of this concept exist [68][69][70].

CMAC is descended from Rosenblatt's perceptron and can be represented by the overall nonlinear mapping:

$$H:(\mathbf{s} \rightarrow \mathbf{p}) \tag{2.18}$$

by which, according to neurophysiological notation, the sensory input $\mathbf{s} = \left[\mathbf{s}_1 \ \ldots \ \mathbf{s}_n\right]$ of n variables $\mathbf{s}_i$, the dynamic range of each one divided into $\mathbf{r}_i$ classes, is mapped onto the m-dimensional output row vector $\mathbf{p} = \left[\mathbf{p}_1 \ \ldots \ \mathbf{p}_m\right]$ representing the axons of Purkinje cells leaving the cerebral cortex [71].

The overall mapping defined by (2.18) can be divided into two further mappings: a fixed stimulus/association cell mapping

$$h_1:(\mathbf{s} \rightarrow \mathbf{a}) \tag{2.19}$$

and an adaptive association cell/response mapping

$$h_2:(\mathbf{a} \rightarrow \mathbf{p}) \tag{2.20}$$

The first of these mappings is constructed in such a way that k (a constant integer) association cells are activated for each stimulus. A special encoding procedure is employed which guarantees that, for two different stimulus vectors, the number of common active association cells is proportional to Euclidean distance between the two vectors.

The association cells are pointers for weights. These weights are added up together to obtain the corresponding output. Denoting by $\mathbf{a}_i^*$ the i^{th} active cell associated with a particular stimulus vector, the j^{th} element of the corresponding output ($\mathbf{p}_j$) is obtained as:

$$\mathbf{p}_j = \frac{\sum_{i=1} \mathbf{W}_{j,i}(\mathbf{a}_i^*)}{k} \tag{2.21}$$

where $\mathbf{W}_{j,i}(\mathbf{a}_i^*)$ denotes the weight corresponding to the j^{th} output element pointed by the i^{th} association cell.

This last equation, together with the special encoding procedure used in the first mapping, is responsible for the important feature of *generalization*, i.e. similar inputs produce similar outputs [69].

Learning is achieved by adjusting the weights in the second mapping, usually with the Widrow-Hoff rule:

$$\mathbf{W}_{j,i}(\mathbf{a}_i^*) = \mathbf{W}_{j,i}(\mathbf{a}_i^*) + \eta(\mathbf{t}_j - \mathbf{p}_j) \tag{2.22}$$

where $\mathbf{t}_j$ denotes the j^{th} element of the desired output.

As will be shown in the next section, multilayer perceptrons (MLPs) can also be used for nonlinear function approximation. In comparison with CMAC, which generalizes only locally (each input only activates a small part of the existing weights), the MLPs employ a global generalization (every input employs all weights).

The use of CMAC neural networks is not so widespread in the neural network community as, for instance, MLPs and Hopfield networks, although they deserve more attention. They offer potential advantages over MLPs for nonlinear function approximation, namely [72][73]: faster training time, no local minima and are temporally stable (what is learnt is not forgotten).

The main disadvantage is the memory requirement, which grows exponentially with the number of inputs. For large-dimensional systems, random memory hashing is usually used [69]. Guidelines to determine the resolution to employ in the inputs and the number k of active association cells are also currently not available.

2.2.2.3 Multilayer perceptrons

Multilayer perceptrons are, perhaps, the best known and widely used artificial neural network. They are certainly the most commonly employed ANN in control systems applications.

As its name suggests, in this type of ANN, neurons are grouped into layers. Layers are classified as input, output or hidden depending on whether they receive data from, send data to, or do not have direct communication with the environment, respectively. Neurons in adjacent layers are fully connected, in a feedforward manner, connections between nonadjacent layers usually not being allowed. The topology of MLPS is usually indicated by specifying the number of neurons in each layer, starting with the input layer, between round brackets. For instance, Fig. 2.3 illustrates the topology of a multilayer perceptron with a single hidden layer, which could be denoted as (n,x,k).

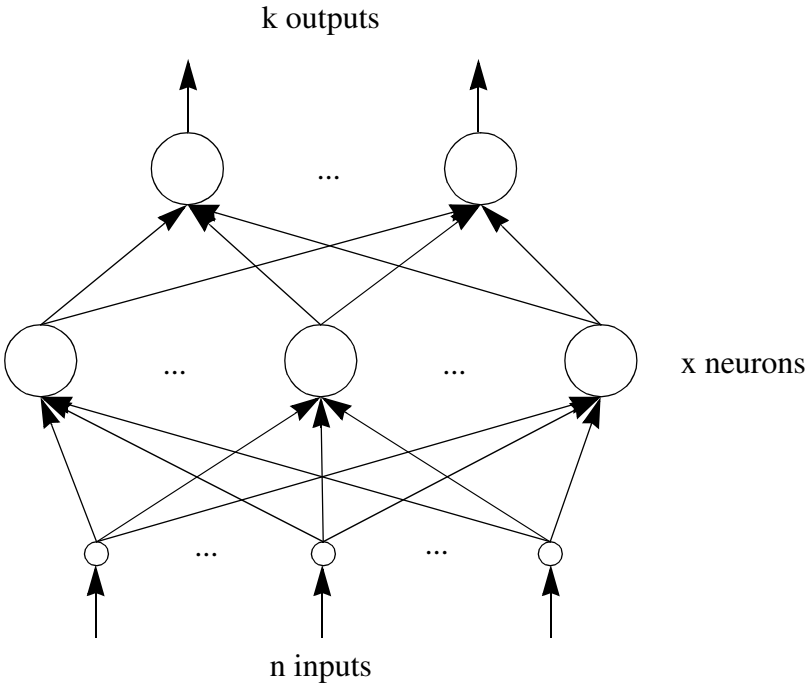


Fig. 2.3 - A multilayer perceptron

Variants of this standard topology exist. In some cases connections between nonadjacent layers are allowed. There is also a great deal of interest in structures of this type where feedback is allowed [74][75].

The operation of these networks is very simple. Data is presented at the input layer, where the neurons act as input buffers. For each neuron in the layer immediately above, its net input is computed using (2.5) and is then passed through an output function, as indicated by

(2.6). In this way the output row vector for the first hidden layer is obtained. This process is repeated through all the other layers, until the output layer is reached. This operation is usually called the *recall operation*.

Multilayer perceptrons are a natural extension of Rosenblatt's perceptrons [35]. A single layer perceptron (see page 9), has severe limitations. When considering classification applications, a single layer perceptron can only separate the input data between classes if this data is linearly separable, i.e., forms half-plane decision regions. However, as pointed out by Lippmann [31], the presence of a hidden layer of perceptrons may lead to convex regions (possibly unbounded) being produced by the overall network of perceptrons. Networks with two hidden layers are capable of creating arbitrary decision regions. This feature is responsible for the success of MLPs in a large number of pattern recognition applications.

The importance of hidden layers was recognized early on. However, as Minsky and Papert pointed out [39], in 1969 there was no known learning rule for networks with hidden units. This situation changed with the introduction of the error back-propagation algorithm (or generalized delta rule) by Rumelhart and the PDP group [13]. This rule involves a change in the output function of the constituent perceptrons. Rosenblatt's perceptrons employ a sign function as output function:

$$\mathbf{o}_i = \mathrm{sgn}(\mathbf{net}_i) \tag{2.23}$$

where **net** is defined in (2.5). Since the error back-propagation rule is a gradient descent method, it involves the computation of derivatives and so the output function must be a differentiable function. The most commonly used is the sigmoid function

$$\mathbf{o}_i = \frac{1}{1 + e^{-\mathbf{net}_i}} \tag{2.24}$$

although other functions, such as the hyperbolic tangent and the Gaussian function, are also used. Because of its importance, the error back-propagation algorithm is detailed in Chapter 4.

We have already mentioned the important application of multilayer perceptrons as pattern classifiers. Another application of these artificial neural networks, perhaps more useful in control systems, is in nonlinear function approximation. When MLPs are used for this purpose, the output function of neurons in the output layer is, instead of a sigmoid, a linear function. This fact is exploited, in Chapter 4, to reduce the training time. Funahashi [76] and Hornik et al. [77] theoretically proved that any continuous function defined in an n-dimensional space can be uniformly approximated by MLPs of this class, with a single hidden layer.

In contrast with CMAC networks, which obtain the property of generalization by

spreading each pattern of training data through a constant number of association cells, multilayer perceptrons achieve this important feature by producing an overall mapping from an n-dimensional space (assuming n inputs) to a k-dimensional output space (k outputs), which is a close approximation to the function underlying the training data, within the training range. They achieve thus a global generalization within the boundaries of the training data.

Multilayer perceptrons, due to their mapping capabilities and also to their wide use, are the artificial neural networks employed in the work described in this thesis. For this reason, it is advisable to introduce some nomenclature which will be used when dealing with this type of ANNs. As Chapter 4 is dedicated to off-line training of MLPs, it is beneficial to consider the operation of MLPs in terms of sets of presentations, instead of isolated presentations of patterns. For this reason, quantities like net inputs, outputs and errors are considered to be matrices, where the i^{th} row denotes the i^{th} presentation, out of m different presentations. For these quantities to be clearly identified they must also reflect the layered structure of MLPs. Accordingly, an index, in superscript and inside brackets, is used to identify the particular layer to which the quantity corresponds. With these considerations in mind, the following nomenclature will be used:

q - the number of layers of the MLP (the input layer is the first layer, and the output layer is the q^{th});

k -a vector with q elements, the z^{th} element denoting the number of neurons in layer z;

$\mathbf{Net}^{(z)}$ - the $m*\mathbf{k}_z$ matrix of the net inputs of the neurons in layer z; this matrix is not defined when z=1;

$\mathbf{O}^{(z)}$ - the $m*\mathbf{k}_z$ matrix of the outputs of the neurons in the z^{th} layer; the network input matrix is therefore denoted as $\mathbf{O}^{(1)}$;

T - the $m*\mathbf{k}_q$ matrix of desired outputs or targets;

E - the $m*\mathbf{k}_q$ matrix of the errors ($\mathbf{E} = \mathbf{T} - \mathbf{O}^{(q)}$);

$\mathbf{F}^{(z)}(.)$ - the matrix output function for the neurons in the z^{th} layer; it is applied on an element by element basis, i.e.:

$$\mathbf{O}^{(z)}_{i,j} = \mathbf{F}^{(z)}_{i,j}(\mathbf{Net}^{(z)}) = \mathbf{F}^{(z)}(\mathbf{Net}^{(z)}_{i,j}) \quad ; \tag{2.25}$$

$\mathbf{W}^{(z)}$ - the $(\mathbf{k}_z+1)*\mathbf{k}_{z+1}$ weight matrix between the z^{th} and the $(z+1)^{th}$ layers. The threshold associated with each neuron of the $(z+1)^{th}$ layer can be envisaged as a normal weight that connects an additional neuron (the $(\mathbf{k}_z+1)^{th}$ neuron) in the z^{th} layer,

which has a fixed output of 1. $\mathbf{W}^{(z)}_{i,j}$ is therefore the weight connecting neuron i in the z^{th} layer to neuron j in the $(z+1)^{th}$ layer.

In some instances it is convenient to represent these weight matrices as a vector. In this case:

$\mathbf{w}^{(z)}$ - the vector representation of $\mathbf{W}^{(z)}$, defined by (2.26):

$$\mathbf{w}^{(z)}. = \begin{bmatrix} \mathbf{W}^{(z)}_{., 1} \\ \cdots \\ \mathbf{W}^{(z)}_{., \mathbf{k}_{z+1}} \end{bmatrix} ; \tag{2.26}$$

$\mathbf{w}$ - the entire weight vector of the network:

$$\mathbf{w} = \begin{bmatrix} \mathbf{w}^{(q-1)} \\ \cdots \\ \mathbf{w}^{(1)} \end{bmatrix} \tag{2.27}$$

Multilayer perceptrons have been the focus of very active research. However, several problems remain to be solved:

- One open question is which topology to use. Theoretical proofs [76][77] exist on the capabilities of a single hidden layer MLP for approximating a nonlinear continuous mapping. However, these proofs do not give any guidelines on the number of hidden neurons to employ for a particular problem. Furthermore, for some practical problems, networks with two or more hidden layers may be more efficient in terms of the total hidden neurons required. Methodologies which can specify the number of hidden layers and/or the number of neurons per hidden layer are a very active area of research[78][79][80].

- Other open questions, obviously related to the previous one, are the training data size and how to obtain the training data (even/uneven sampling). Although research on this area is also very active [81][82], no clear answers are yet known.

- Another problem related to multilayer perceptrons is the large amount of time taken for learning, when employing the standard algorithm of error back-propagation. This is due to the large amount of data involved (large number of training patterns and large dimension of the model) but also to the slow convergence rate of the learning algorithm. This is discussed in detail in Chapter 4, where alternatives to the error back-propagation algorithm are given.

- Finally, one point that does not seem to have attracted the attention of the scientific community, is the choice of initial values for the training procedure. In standard non-linear regression applications, it is typical to begin with a reasonable estimate of the solution. This has the advantage of, afterwards, reducing the time taken by the iterative optimization procedure. In the training of MLPs the usual procedure is to choose random numbers, in the range of -1 to +1. It may be possible to produce better initial estimates, by analysing the range of input and target output data.

We finish this brief description on multilayer perceptrons by noting that a recently proposed layered neural network, the radial basis function (RBF) network [83], has been receiving increasing attention, due to its savings in training time.

2.3 Applications of neural networks to control systems: an overview

Today, there is a constant need to provide better control of more complex (and probably nonlinear) systems, over a wide range of uncertainty. Artificial neural networks offer a large number of attractive features for the area of control systems [84][53]:

- The ability to perform arbitrary nonlinear mappings makes them a cost efficient tool to synthesize accurate forward and inverse models of nonlinear dynamical systems, allowing traditional control schemes to be extended to the control of nonlinear plants. This can be done without the need for detailed knowledge of the plant.

- The ability to create arbitrary decision regions means that they have the potential to be applied to fault detection problems. Exploiting this property, a possible use of ANNs is as control managers, deciding which control algorithm to employ based on current operational conditions.

- As their training can be effected on-line or off-line, the applications considered in the last two points can be designed off-line and afterwards used in an adaptive scheme, if so desired.

- Neural networks are massive parallel computation structures. This allows calculations to be performed at a high speed, making real-time implementations feasible. Development of fast architectures further reduce computation time.

- Neural networks can also provide, as demonstrated in [85], significant fault tolerance, since damage to a few weights need not significantly impair the overall performance.

During the last years, a large number of applications of ANN to robotics, failure detection systems, nonlinear systems identification and control of nonlinear dynamical

systems have been proposed. As the subject of this thesis is the application of ANNs to control systems, only brief mention of the first two areas will be made in this Chapter, the main effort being devoted to a description of applications for the last two areas.

Robots are nonlinear, complicated structures. It is therefore not surprising that robotics was one of the first fields where ANNs were applied. It is still one of the most active areas for applications of artificial neural networks. A very brief list of important work on this area can be given. Elsley [85] applied MLPs to the kinematic control of a robot arm. Kawato et al. [86] performed feedforward control in such a way that the inverse system would be built up by neural networks in trajectory control. Guo and Cherkassky [87] used Hopfield networks to solve the inverse kinematics problem. Kuperstein and Rubinstein [27] have implemented a neural controller which learns hand-eye coordination from its own experience. Fukuda et al. [28] employed MLPs with time delay elements in the hidden layer for impact control of robotic manipulators.

Artificial neural networks have also been applied to sensor failure detection and diagnosis. In these applications, the ability of neural networks for creating arbitrary decision regions is exploited. As examples, Naidu et al. [88] have employed MLPs for the sensor failure detection in process control systems, obtaining promising results, when compared to traditional methods. Leonard and Kramer [89] compared the performance of MLPs against RBF networks, concluding that under certain conditions the latter has advantages over the former. Narenda [90] proposes the use of neural networks, in the three different capacities of identifiers, pattern recognizers and controllers, to detect, classify and recover from faults in control systems.

2.3.1 Nonlinear identification

The provision of an accurate model of the plant can be of great benefit for control. Predictive control, self-tuning and model reference adaptive control employ some form of plant model. In process control, due to the lack of reliable process data, models are extensively used to infer primary values, which may be difficult to obtain, from more accessible secondary variables.

Most of the models currently employed are linear in the parameters. However, in the real world, most of the processes are inherently nonlinear. Artificial neural networks, making use of their ability to approximate large classes of nonlinear functions accurately, are emerging as a recognized tool for nonlinear systems identification.

CMAC networks, RBF networks and specially MLPs, have been used to approximate forward as well as inverse models of nonlinear dynamical plants. These applications are discussed in the following sections.

2.3.1.1 Forward models

Consider the SISO, discrete-time, nonlinear plant, described by the following equation:

$$y_p[k+1] = f(y_p[k], \ldots, y_p[k-n_y+1], u[k], \ldots, u[k-n_u+1]) \quad , \qquad (2.28)$$

where n_y and n_u are the corresponding lags in the output and input and f(.) is a nonlinear continuous function. Equation (2.28) can be considered as a nonlinear mapping between an n_u+n_y dimensional space to a one-dimensional space. This mapping can then be approximated by a multilayer perceptron or a CMAC network (a radial basis function network can also be applied) with n_u+n_y inputs and one output.

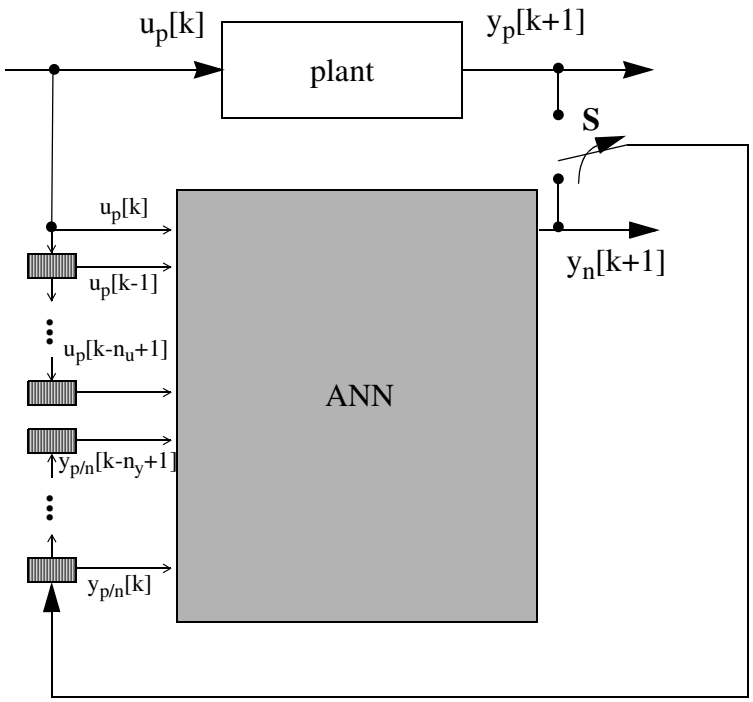


Fig. 2.4 - Forward plant modelling with ANNs

Fig. 2.4 illustrates the modelling the plant described by (2.28). In this Figure, the symbol ▭ denotes a delay. When the neural network is being trained to identify the plant, switch S is connected to the output of the plant, to ensure convergence of the neural model. In this approach, denoted as *series-parallel model*, the neural network model does not incorporate feedback. Training is achieved, as usual, by minimizing the sum of the squares of the errors between the output of the plant and the output of the neural network. After the neural network has completed the training, switch S can be connected to the output of the neural network, which acts as a model for the plant, within the training range. This last

configuration is called the *parallel model*.

The neural model can be trained off-line by gathering vectors of plant data and using them for training by means of one of the methods described in Chapter 4. In this case, it should be ensured that the input data used should adequately span the entire input space that will be used in the recall operation, as ANNs can be successfully employed for nonlinear function interpolation, but not for function extrapolation purposes.

The neural network can also be trained on-line. In the case of CMAC networks, the Widrow-Hoff rule is employed. For MLPs, the error back-propagation algorithm, in pattern mode, can be used. This will be explained in Chapter 4. Alternatively, the recursive-prediction-error (RPE) algorithm, essentially a recursive version of the batch Gauss-Newton method, due to Chen et al. [91] can be employed, achieving superior convergence rate. This algorithm was also introduced by Ruano [92] based on a recursive algorithm proposed by Ljung [93].

One fact was observed by several researchers [29][94], when performing on-line training of MLPs. In certain cases, the output of the model rapidly follows the output of the plant, but as soon as the learning mechanism is turned off, the model and the plant diverge. This may be explained by the fact that, at the beginning of the training process, the network is continuously converging to a small range of training data represented by a small number of the latest training samples. As learning proceeds (using the error back-propagation algorithm, usually for a very large number of samples) the approximation becomes better within the complete input training range.

A large number of researchers have been applying MLPs to forward nonlinear identification. As examples, Lapedes and Farber [95] have obtained good results in the approximation of chaotic time series. Narenda and Parthasarathy [29] have proposed different types of neural network models of SISO systems. These exploit the cases where eq. (2.28) can be recast as:

$$y_p[k+1] = h(y_p[k], \ldots, y_p[k-n_y+1]) + g(u[k], \ldots, u[k-n_u+1]) \quad , \qquad (2.29)$$

h(.) or u(.) possibly being linear functions. Bhat et al. [96] have applied MLPs to model chemical processes, with good results. Lant, Willis, Morris et al. [97][98] have been applying MLPs to model ill-defined plants commonly found in industrial applications, with promising results. In [98] they have proposed, for forward plant modelling purposes, to incorporate dynamics in an MLP, by passing the outputs of the neuron through a filter with transfer function:

$$G(s) = e^{-T_d s}\frac{N(s)}{D(s)} \qquad (2.30)$$

as an alternative to the scheme shown in Fig. 2.4. Recently, Billings et al. [99] have reminded the neural networks community that concepts from estimation theory can be employed to measure, interpret and improve network performance.

CMAC networks have also been applied to forward modelling of nonlinear plants. Ersü and others, at the Technical University of Darmstadt, have been working since 1982 in the application of CMAC networks for the control of nonlinear plants. Examples of their application can be found, for instance, in [100][101].

2.3.1.2 Inverse models

The use of neural networks for producing inverse models of nonlinear dynamical systems seems to be finding favour in control systems applications.

In this application the neural network is placed in series with the plant, as shown in Fig. 2.5. The aim here is, with the use of the ANN, to obtain $y_p[k]=r[k]$.

Assuming that the plant is described by (2.28), the ANN would then implement the mapping:

$$u[k] = f^{-1}(y_p[k+1], y_p[k], \ldots, y_p[k-n_y+1], u[k-1], \ldots, u[k-n_u+1]) \qquad (2.31)$$

As (2.31) is not realisable, since it depends on $y_p[k+1]$, this value is replaced by the control value r[k+1]. The ANN will then approximate the mapping:

$$u[k] = f^{-1}(r[k+1], y_p[k], \ldots, y_p[k-n_y+1], u[k-1], \ldots, u[k-n_u+1]) \qquad (2.32)$$

as shown in Fig. 2.5.

Training of the ANN involves minimizing the cost function:

$$J = \frac{1}{2}\sum_{j}(r[j] - y_p[j])^2 \qquad (2.33)$$

over a set of patterns *j*.

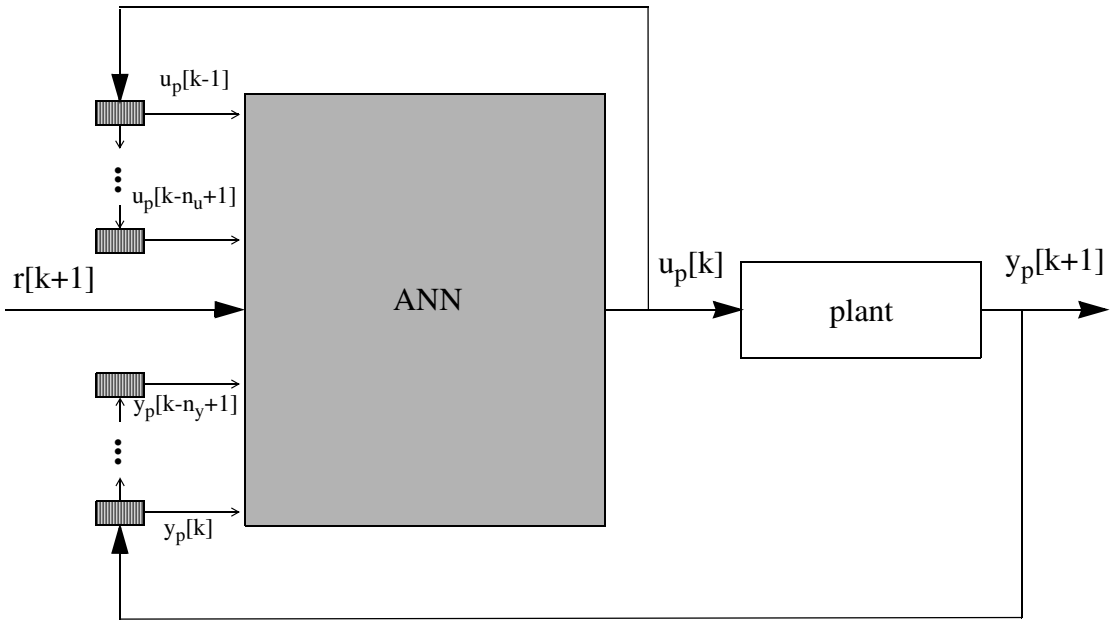


Fig. 2.5 - Inverse modelling with ANNs

There are several possibilities for training the network, depending on the architecture used for learning. Psaltis et al. [102] proposed two different architectures. The first one, denoted as *generalised learning architecture*, is shown in Fig. 2.6. The arrow denotes that the error 'e' is used to adjust the neural network parameters.

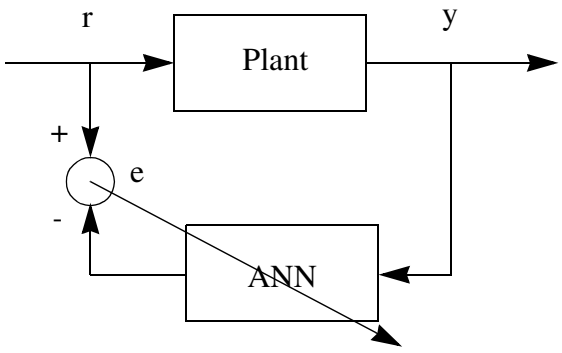


Fig. 2.6 - Generalised learning architecture

Using this scheme, training of the neural network involves supplying different inputs to the plant and teaching the network to map the corresponding outputs back to the plant inputs. This procedure works well but has the drawback of not training the network over the range where it will operate, after having been trained. Consequently, the network may have to

be trained over an extended operational range. Another drawback is that it is impossible to train the network on-line.

To overcome these disadvantages, the same authors proposed another architecture, the *specialized learning architecture*, shown in Fig. 2.7.

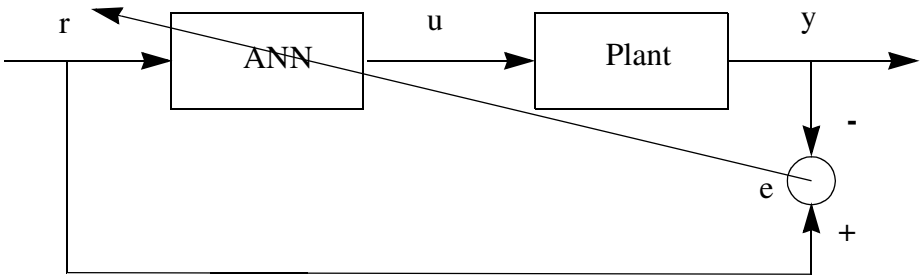


Fig. 2.7 - Specialized learning architecture

In this case the network learns how to find the inputs, u, that drive the system outputs, y, towards the reference signal, r. This method of training addresses the two criticisms made of the previous architecture because the ANN is now trained in the region where it will operate, and can also be trained on-line. The problem with this architecture lies in the fact that, although the error r-y is available, there is no explicit target for the control input u, to be applied in the training of the ANN. Psaltis proposed that the plant be considered as an additional, unmodifiable, layer of the neural network to backpropagate the error. This depends upon having *a priori* knowledge of the Jacobian of the plant, or determining it by finite difference gradients.

To overcome these difficulties, Saerens and Soquet [103] proposed either replacing the Jacobian elements by their sign, or performing a linear identification of the plant by an adaptive least-squares algorithm.

An alternative, which seems to be widely used nowadays, was introduced by Nguyen and Widrow in their famous example of the ‘truck backer-upper’ [104]. Here they employ another MLP, previously trained as a forward plant model, where the unknown Jacobian of the plant is approximated by the Jacobian of this additional MLP.

Examples of MLPs used for storing the inverse modelling of nonlinear dynamical plant can be found, for instance, in [94][105]. Harris and Brown [106] showed examples of CMAC networks for the same application.

Applications of ANN in nonlinear identification seems very promising. Lapedes and Farber [95] point out that the predictive performance of MLPs exceeds the known

conventional methods of prediction. Willis et al. [98] achieve good results when employing MLPs as nonlinear estimators of bioprocesses.

Artificial neural networks have been emerging, during the past three or four years, as another available tool for nonlinear systems identification. It is expected that in the next few years this role will be consolidated, and open questions like determining the best topology of the network to use will be answered.

2.3.2 Control

Neural networks are nonlinear, adaptive elements. They have found, therefore, immediate scope for application in nonlinear, adaptive control. During the past three years ANNs have been integrated into a large number of control schemes currently in use. Following the overall philosophy behind this Chapter, selected examples of applications of neural networks for control will be pointed out, references being given to additional important work.

2.3.2.1 Model reference adaptive control

The largest area of application seems to be centred on model reference adaptive control (MRAC). MRAC systems are designed so that the output of the system being controlled follows the output of a pre-specified system with desirable characteristics. To adapt the controller gains, traditional MRAC schemes initially used gradient descent techniques, such as the MIT rule [107]. With the need to develop stable adaptation schemes, MRAC systems based on Lyapunov or Popov stability theories were later proposed [107].

There are two different approaches to MRAC [29]: the *direct approach*, in which the controller parameters are updated to reduce some norm of the output error, without determining the plant parameters, and the *indirect approach*, where these parameters are first estimated, and then, assuming that these estimates represent the true plant values, the control is calculated.

These schemes have been studied for over 20 years and have been successful for the control of linear, time-invariant plants with unknown parameters. Exploiting the nonlinear mapping capabilities of ANNs, several researchers have proposed extensions of MRAC to the control of nonlinear systems, using both the direct and the indirect approaches.

Narenda and Parthasarathy [29] exploited the capability of MLPs to accurately derive forward and inverse plant models in order to develop different indirect MRAC structures. Assume, for instance, that the plant is given by (2.28), where h(.) and u(.) are unknown nonlinear functions, and that the reference model can be described as:

$$y_r[k+1] = \sum_{i=0}^{n-1} a_i y_r[k-i] + r[k] \tag{2.34}$$

where r[k] is some bounded reference input.

The aim of MRAC is to choose the control such that

$$\lim_{k \to \infty} e[k] = 0 \tag{2.35}$$

where the error e[k] is defined as:

$$e[k] = y_p[k] - y_r[k] \tag{2.36}$$

and $y_r[k]$ denotes the output of the reference model at time k.

Defining

$$\begin{aligned} g[k] &= g(u[k], \ldots, u[k-n_u+1]) \\ h[k] &= h(y_p[k], \ldots, y_p[k-n_y+1]) \end{aligned} \tag{2.37}$$

and assuming that (2.35) holds, we then have

$$\sum_{i=0}^{n-1} a_i y_r[k-i] + r[k] = g[k] + h[k] \tag{2.38}$$

Rearranging this last equation, (2.39) is obtained:

$$g[k] = \sum_{i=0}^{n-1} a_i y_r[k-i] + r[k] - h[k] \tag{2.39}$$

and u[k] is then given as:

$$u[k] = g^{-1}\left(\sum_{i=0}^{n-1} a_i y_r[k-i] + r[k] - h[k] \right) \tag{2.40}$$

If good estimates of h[k] and $g^{-1}[k]$ are available, (2.40) can be approximated as:

$$u[k] = \hat{g}^{-1}\left(\sum_{i=0}^{n-1} a_i y_r[k-i] + r[k] - \hat{h}[k] \right) \tag{2.41}$$

The functions h[k] and g[k] are approximated by two ANNs, interconnected as Fig. 2.8 suggests.

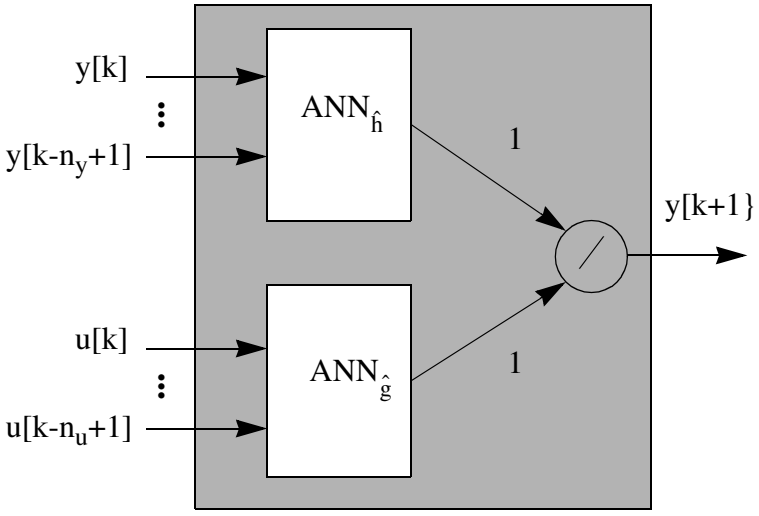


Fig. 2.8 - Forward model

Notice that this structure may be viewed as a forward model, with the difference that the weights at the outputs of the neural networks $ANN_{\hat{h}}$ and $ANN_{\hat{g}}$ are fixed and unitary. This does not involve any problem for training and any of the methods discussed in 2.3.1.1 can be used.

After $ANN_{\hat{h}}$ and $ANN_{\hat{g}}$ have been trained, then an additional network, approximating $g^{-1}[k]$ can be trained, in the specialized learning architecture, employing $ANN_{\hat{g}}$ as discussed in 2.3.1.2. The control signal can be computed using (2.41).

Examples of this approach can be seen in [29][105][94].

For direct neural MRAC, Lightbody and Irwin [94] propose the structure shown in Fig. 2.9.

The controller is a hybrid controller, composed of a fixed gain, linear controller in parallel with a MLP controller. A suitable reference model, with state vector $\mathbf{x}_m[k]$, is run in parallel with the controlled plant. Its output is compared with the output of the plant in order to derive an error, which is used to adapt the weights of the ANN. The plant is situated between this error and the output of the MLP. Consequently, in order to feed back this error to the network, they propose either to use Saerens and Soquet's technique [103] of approximating the elements of the Jacobian of the plant by their sign, or to use a previously trained network, which approximates the forward model of the plant.

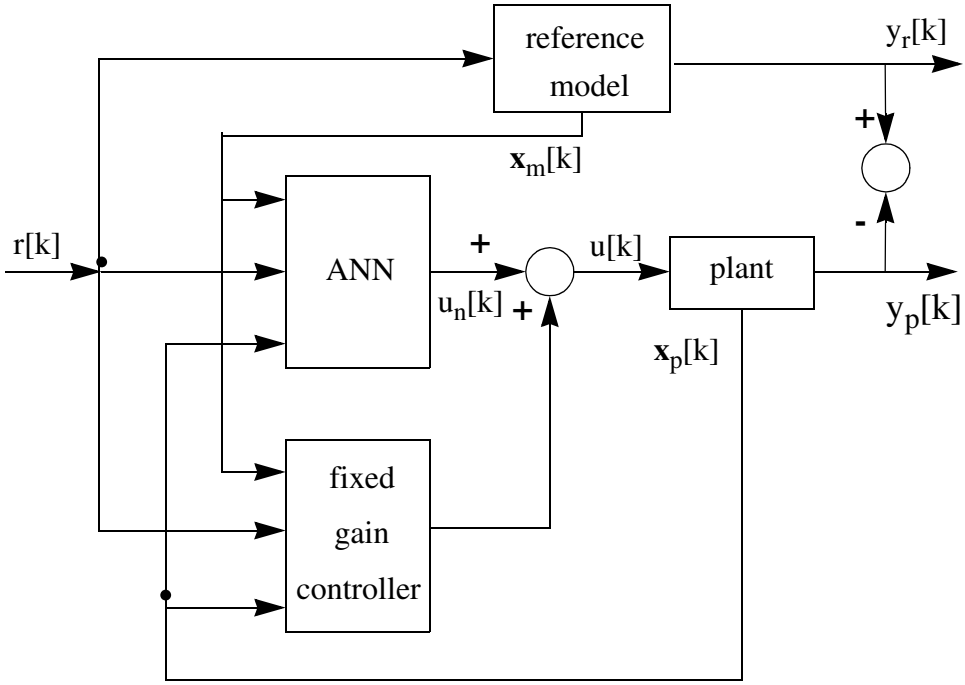


Fig. 2.9 - Direct MRAC controller

A similar scheme was proposed by Kraft and Campagna [108] using CMAC networks.

2.3.2.2 Predictive control

Artificial neural networks have also been proposed for predictive control. Ersü and others [69][109][101] have developed and applied, since 1982, a learning control structure, coined LERNAS, which is essentially a predictive control approach using CMAC networks.

Two CMAC networks (referred by Ersü as AMS) are used in their approach. The first one stores a predictive model of the process:

$$(\mathbf{x}_m[k], u[k], v[k]) \rightarrow y_m[k+1] \tag{2.42}$$

where $\mathbf{x}_m[k]$ and v[k] denote the state of the process and the disturbance at time k, respectively. The second ANN stores the control strategy:

$$(\mathbf{x}_c[k], w[k], v[k]) \rightarrow u[k] \tag{2.43}$$

where $\mathbf{x}_c[k]$ and w[k] denote the state of the controller and the value of the setpoint at time k, and u[k] is the control input to the plant.

At each time interval, the following sequence of operations is performed:

i) the predictive model is updated by the prediction error $e[k] = y_p[k] - y_m[k]$;

ii) an optimization scheme is activated, if necessary, to calculate the optimal control value $\hat{u}[k]$ that minimizes a *l*-step ahead subgoal of the type:

$$J[k] = \sum_{i=1}^{l} L_s(y_m[k+i], w[k+1], u[k-l+1]) \tag{2.44}$$

constrained in the trained region G_T of the input space of the predictive model of the plant. To speed up the optimization process, an approximation of $\hat{u}[k]$, denoted as $u[k]$, can be obtained from past decisions:

$$(\mathbf{x}_c[k], w[k], v[k]) \rightarrow u[k] \quad ; \tag{2.45}$$

iii) the control decision $\hat{u}[k]$ is stored in the controller ANN to be used, whether as a future initial guess for an optimization, or directly as the control input upon user decision;

iv) finally, before applying a control to the plant, a sub-optimal control input $\tilde{u}[k]$ is sought, such that:

$$\|\tilde{u}[k] - \hat{u}[k]\| < \varepsilon \qquad \tilde{u}[k] \notin G_T \tag{2.46}$$

for some specified ε. This process enlarges the trained region G_T and speeds up the learning.

More recently, Montague et al. [110] developed a nonlinear extension to Generalized Predictive Control (GPC) [111]. The following cost function is used in their approach:

$$J[k] = \sum_{i=N_1}^{N_2} (w[k+i] - y_m[k+i])^2 + \sum_{i=0}^{N_u} \lambda u[k+j])^2 \tag{2.47}$$

where $y_m[k+i], i = N_1 \ldots N_2$ are obtained from a previously trained MLP, which emulates the forward model of the plant. At each time step, the output of the plant is sampled, and the difference between the output of the plant and the predicted value is used to compute a correction factor. The MLP is again employed to predict the outputs over the horizon $n = N_1 \ldots N_2$. These values are later corrected and employed for the minimization of (2.47) with respect to the sequence $\{u[i], (i = k, \ldots, k + N_u)\}$. The first of these values, $\hat{u}[k]$, is then applied to the plant and the sequence is repeated.

The same cost function (2.47) is used by Sbarbaro et al. [112] who employs radial basis functions to implement an approach similar to the LERNAS concept. Hernandez and Arkun [113] have employed MLPs for nonlinear dynamic matrix control.

Artificial neural networks have been proposed for other adaptive control schemes. For instance, Chen [114] has introduced a neural self-tuner. Iiguni et al. [115] have employed MLPs to add nonlinear effects to a linear optimal regulator.

Conclusive results on the merit of neural controllers over traditional schemes are not currently available, due to the infancy of this new field. However, preliminary performance comparisons made by some researchers seems to indicate that ANN-based control delivers better results than traditional schemes when the plant is highly nonlinear [94][108][110] and also in the presence of noise [108]. For linear, noise free, plants no improvement over conventional techniques was obtained [94][108].

On the whole, ANN-based control offers definite promises. Several open questions exist, the biggest of all perhaps being stability issues. At present, there is no established methodology for determining the stability of ANN control schemes. However, this is not a unique situation in the area of control systems. The first self-tuners were used in industry before stability proofs were developed, and self-tuning is at present a consolidated field. Whether the same thing will happen to neural control schemes, only time and a large research effort will tell.

2.4 Conclusions

In this Chapter artificial neural networks have been introduced. A chronological perspective of the research on ANN has been given and some of the reasons for the renewal of interest in this technology pointed out. From a brief discussion of the structure of the human brain, it can be concluded that the broad aspects of this complicated structure are also found in artificial neural networks models. However, since detailed knowledge of the brain is not currently available, a large number of ANN models has been proposed. A brief discussion on neuron models, patterns of connectivity and learning mechanism has been conducted. Finally the details of three neural models, Hopfield networks, CMAC networks and multilayer perceptrons have been given.

An overview of current applications of artificial neural networks to nonlinear identification and control of nonlinear dynamical systems has been conducted. It has been shown that ANNs are emerging as a cost effective tool for nonlinear systems identification. Neural networks, during the past three years, have been integrated in several different control

schemes. Promising results have been obtained that, when compared with conventional techniques, seem to indicate that neural networks have an important role to play in nonlinear adaptive control.

From this overview, it is clear that MLPs are the most commonly employed ANN in control systems. For this field, the most exploited feature of MLPs is the ability to perform arbitrary nonlinear mappings. In the next Chapter this capacity of MLPs will again be exploited in a problem of great practical importance: the autotuning of PID controllers.

Chapter 3

A Connectionist Approach to PID Autotuning

3.1 Introduction

In the last Chapter, an overview of the applications of artificial neural networks in the field of control systems was presented. From that overview it was clear that multilayer perceptrons were the most commonly used neural networks in this area. This was even more the case when this research started, since, for instance, radial basis function networks were not known at that time. Another point which stands out from the preceding overview is that the most exploited feature of MLPs in control systems applications is their ability to perform accurate nonlinear mappings.

In this Chapter this feature of MLPs is also exploited in the context of PID autotuning. This is a very important subject because the majority of the controllers used in industry are of the PID type. These controllers are tuned by instrument engineers using simple empirical rules such as the Ziegler-Nichols tuning rule [10]. The controllers are often poorly tuned due to neglect or lack of time. Derivative action is seldom used.

Since there are significant benefits in having well-tuned control loops, several attempts have been made to tune regulators automatically. In this Chapter a novel method, employing multilayer perceptrons, is proposed.

The outline of this chapter is as follows:

In section 3.2 the PID controller is introduced, both in its continuous and discrete versions.

The next section presents an overview of existing approaches to PID tuning, focusing on automatic methods.

Section 3.4 gives the details of the proposed approach, in terms of multilayer perceptrons. Section 3.4.1 focuses on the nature of their inputs. The nature of their outputs will be pointed out in section 3.4.2. As the new technique involves off-line training of the MLPs section 3.4.3. discusses special cares to be taken in obtaining the training set and the criterion to be used when deriving the target output data for training. After having trained the MLPs off-line they can then be used for on-line control. The on-line procedure is discussed in section 3.4.4.

Section 3.5 assesses the performance of the new PID autotuning method using off-line simulations. An example is given, where the new technique is used to control a plant which has time-varying parameters as well as a varying transfer function. The performance, within the training set, of two MLPs with different topologies is analysed and compared with the use of second and third order polynomial approximations. Examples of the responses obtained with the neural PID autotuner are presented.

Finally, in section 3.6 the main advantages of this novel approach will be highlighted. The problems encountered will also be pointed out.

3.2 The PID controller

A text-book version of the continuous PID controller, in Laplace transform notation, is

$$U(s) = k_c\left(1 + T_i s + \frac{T_d s}{1 + T_f s}\right)E(s) \tag{3.48}$$

where U(s) is the Laplace transform of the control variable, u(t), and E(s) is the Laplace transform of the error, e(t), defined by

$$e(t) = r(t) - y(t) \tag{3.49}$$

In equation (3.49) r(t) is the reference input, and y(t) is the output. The terms k_c, T_i and T_d in (3.48) are the proportional gain, the integral or reset time, and the derivative time respectively. To reduce the gain at high frequencies, a filter, with time constant T_f, is incorporated in the controller.

In practice some modifications are introduced to this controller:

i) in order that the controller does not produce a large control signal for step changes in the reference signal (a phenomenon known as *derivative kick*) it is standard practice to let the derivative action operate only on the process output;

ii) the integral action is also modified so that it does not continue to integrate when the control variable saturates (a phenomenon known as *integrator windup*);

iii) it is necessary to ensure that the state of the PID controller is correct when changing between manual and automatic mode. When the system is in manual mode, the controller produces a control signal that may be different from the manually generated control signal. It is necessary to make sure that the value of the integrator is correct at the time of switching. This is called *bumpless transfer.*

A text book-version of the digital PID controller can be written as:

$$u[k] = k_D\left(1 + \frac{T_s}{T_{DI}}\frac{1}{q-1} + \frac{T_{DD}}{T_s}\frac{q-1}{q+\gamma}\right)e[k] \tag{3.50}$$

where u[k] and e[k] are the discrete versions of u(t) and e(t), respectively, q is the forward-shift operator and T_s is the sampling period. The terms k_D, T_{DI} and T_{DD} are dependent on how the continuous-time controller has been approximated, and γ is given by:

$$\gamma = -e^{-\frac{T_s}{T_f}} \tag{3.51}$$

Equation (3.50) is called the *position form* for the PID controller since the total output of the controller is calculated. If only the change in the control signal ($\Delta u[k]$) is calculated instead, the velocity form is obtained:

$$\Delta u[k] = u[k] - u[k-1] = k_D\left(1 - q^{-1} + \frac{q^{-1}T_s}{T_{DI}} + \frac{T_{DD}(1 - 2q^{-1} + q^{-2})}{T_s(1 + \gamma q^{-1})}\right)e[k] \tag{3.52}$$

In the last equation q^{-1} denotes the backward-shift operator. The use of this last form is a way of solving the problems referred in (ii) and (iii) [116]. The integrator windup is automatically avoided because the integration stops automatically when the output is limited. Bumpless transfer is always achieved because, when switching from manual to automatic mode, the actuator will not change until an error occurs.

3.3 An overview of existing methods of PID autotuning

The PID controller, introduced in the last section, is a standard building block for industrial automation. The popularity of this regulator comes from its robust performance in a wide range of operating conditions, and also from its functional simplicity, which makes it suitable for manual tuning.

To account for process changes and ageing, regular retuning is normally required. Accurate tuning is an operation which, to be done properly, takes considerable time. Since large plants can have hundreds of PID regulators, methods which automate the tuning of the PID compensators are of great practical importance.

A large number of methods for PID autotuning have been proposed. In this section some of them will be described. To ease the description these methods will be loosely classified into five classes: frequency response based, step response based, on-line parameter estimation based, expert and fuzzy systems based, and neural networks based.

3.3.1 Methods based on the frequency response

These types of methods are based on the knowledge of a few features of the Nyquist curve of the open loop transfer function, typically its behaviour close to the critical point. The *critical point* is defined as the first point where the Nyquist curve intersects the negative real axis of the s-plane.

The most widely known tuning method of this type is undoubtedly the closed loop Ziegler-Nichols method [10]. In this method proportional control is applied in a unit feedback configuration, the gain being increased until the system oscillates. The gain of the compensator is usually denoted as the *ultimate gain* (k_u) and the period of the oscillation the *ultimate period* (T_u).

Since this is an operation which in practice can seldom be performed, Åstrom and Hägglund [11] have proposed the use of relay feedback for the determination of the critical point, or other points in the Nyquist curve.

The basic idea behind this method is that many processes (the ones whose Nyquist curve intersects the negative real axis) will exhibit limit cycle oscillations under relay feedback.

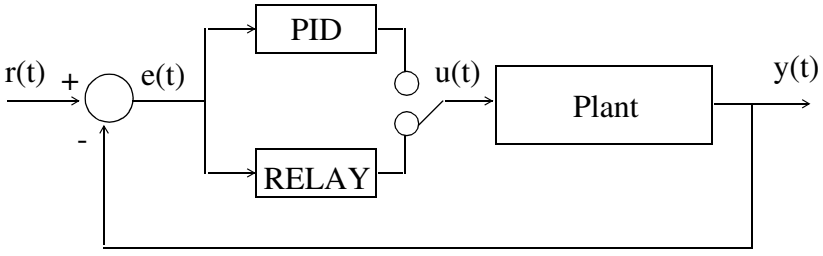


Fig. 3.10 - Block diagram of a relay auto-tuner

Considering Fig. 3.10, let D be the amplitude of the relay output and A be the amplitude of the first harmonic of the error signal. Also let r(t)=0. A Fourier series expansion of the relay output shows that the first harmonic has amplitude $4D / \pi$. If it is assumed that the dynamics of the system are of a low-pass type, and that the contribution from the first harmonic dominates the output, the amplitude of the error signal will be given by:

$$A = \frac{4D}{\pi}\left|G(jw_u)\right| \tag{3.53}$$

where w_u is the *ultimate frequency*:

$$w_u = \frac{2\pi}{T_u} \tag{3.54}$$

The ultimate gain can then be determined from (3.53):

$$k_u = \frac{1}{|G(jw_u)|} = \frac{4D}{\pi A} \tag{3.55}$$

So, by measuring the peak-to-peak amplitude of the output, and by counting the time between zero-crossings, k_u and T_u can be determined. Other points in the Nyquist curve can be determined by introducing known dynamics and hysteresis in the relay. The key difficulty with this approach is the determination of the initial amplitude of the relay [117].

Based on the knowledge of k_u and T_u several design approaches are possible. Ziegler and Nichols [10] introduced simple formulas to tune PID controllers:

k_c	T_i	T_d
$0.6\ k_u$	$T_u/2$	$T_u/8$

Table 3.1 - PID parameters based on Ziegler-Nichols closed loop method

These formulas have been redefined by several authors. Recently, Hang et al. [118] have reviewed this rule and expressed it in terms of the normalized process gain (κ):

$$\kappa = k_p k_u \tag{3.56}$$

where k_p is the steady-state gain of the plant.

Other methods have been proposed based on the frequency response. Åstrom and Hägglund [11] proposed using the criteria of phase and amplitude margins. Assuming that the response of the closed loop system can be characterized by a pair of dominant poles, a different design, based on the knowledge of two points near the critical point, is proposed by the same authors [12].

Examples of commercial devices based on relay oscillations, which employ a modified Ziegler-Nichols rule, are the PID regulators from SattControl and Fisher [107].

3.3.2 Methods based on the step response

The earliest method of PID tuning based on the analysis of the transients of the step response is also due to Ziegler and Nichols [10]. They assumed that the system could be well approximated by the transfer function:

$$G(s) = k_p \frac{e^{-Ls}}{(1 + sT)} \tag{3.57}$$

By applying a unit step to the plant, the time delay (L) and the time constant (T) can be determined from a graphical construction such as the one shown in Fig. 3.11. Such a method, however, is difficult to automate. For this reason, Nishikawa et al. [119] proposed the computation of characteristic areas of the step response.

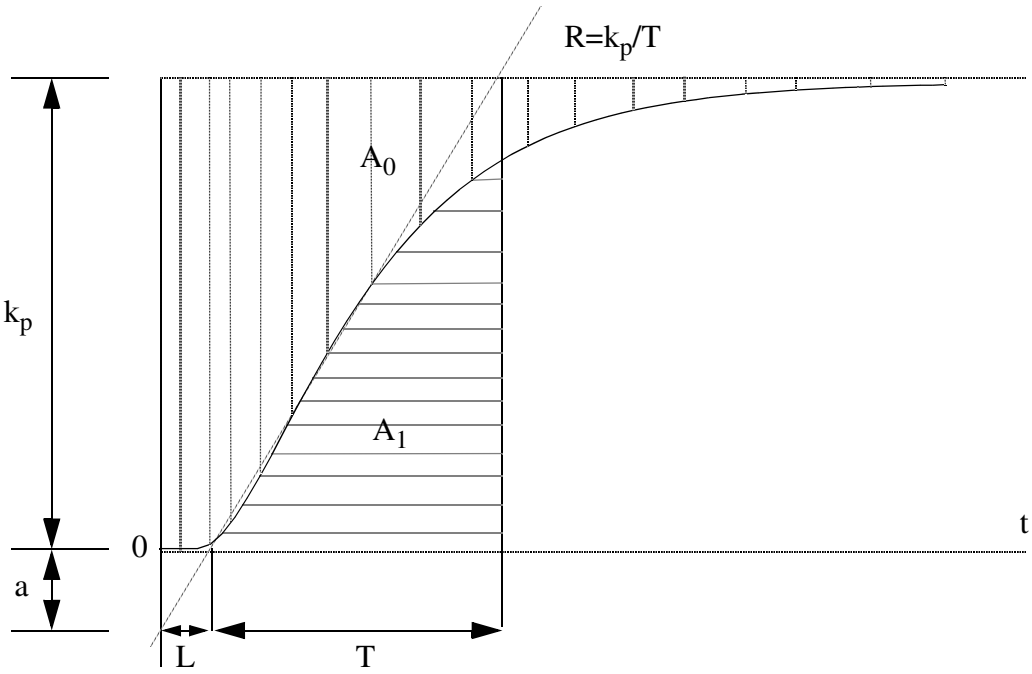


Fig. 3.11 - Open loop unit step response

Considering plants with transfer function (3.57), using Laplace transform properties it is easy to show that the area A_0 in Fig. 3.11 can be given as:

$$A_0 = \int_0^{\infty} (y(\infty) - y(\tau))d\tau = T_T k_p \tag{3.58}$$

where

$$T_T = T + L \tag{3.59}$$

and the area A_1 as:

$$A_1 = \int_0^{T_T} y(\tau)d\tau = \frac{k_p T}{e} \tag{3.60}$$

where the symbol ‘e’ in the last equation denotes the Napier number.

Using (3.58) to (3.60) it is then possible to obtain to obtain the values of L and T. Notice that the steady state gain (k_p) is just the steady-state value of the output, if a unit step input is considered.

Based on this knowledge different designs are possible. Ziegler and Nichols [10] proposed the following formulas for the PID values:

k_c	T_i	T_d
1.2 RL	2L	L/2

Table 3.2 - PID parameters based on Ziegler-Nichols open loop method

where R is the steepest slope of the open loop step response. Hang et al. [118] refined these relations in terms of the *normalised delay time* (Θ) :

$$\Theta = \frac{L}{T} = \frac{a}{k_p} \tag{3.61}$$

Nishikawa et al. [119] proposed the addition of one pole to the plant model (3.57). In order to estimate this additional time constant, an additional characteristic area of the open loop step response must be computed. They proposed to compute

$$A_2 = \int_0^{kT_T} y(\tau)d\tau \tag{3.62}$$

where k is a constant between 0 and 1. Notice that in this case T_T is given as:

$$T_T = L + T_1 + T_2 \tag{3.63}$$

To obtain better damped responses than the ones obtained with Ziegler-Nichols tuning rule, they suggested obtaining the PID parameters by minimizing the exponential time weighted integral of the squared error (ETWISE):

$$\text{ETWISE} = \int_0^{\infty} e^{\beta t} e^2(t)dt \qquad \beta > 0 \tag{3.64}$$

As this numerical procedure is too time-consuming to be performed in real-time, they

proposed to approximate the relationships between scaled values of the characteristic areas A_1 and A_2, denoted as σ and σ':

$$\sigma = \frac{A_1}{k_p T_T} \tag{3.65}$$

$$\sigma' = \frac{A_2}{k_p T_T} \tag{3.66}$$

and the corresponding normalized optimal PID values (ETWISE), by third-order polynomials.

Another approach is followed in the EXACT regulator, a commercial adaptive PID regulator from Foxboro [120]. This regulator is based on the analysis of the closed loop system to setpoint changes or load disturbances. A typical response of the error to a step disturbance is shown in Fig. 3.12. Logic is used to identify that a proper disturbance has occurred. The peaks e_1, e_2 and e_3 are detected, as well as the period T.

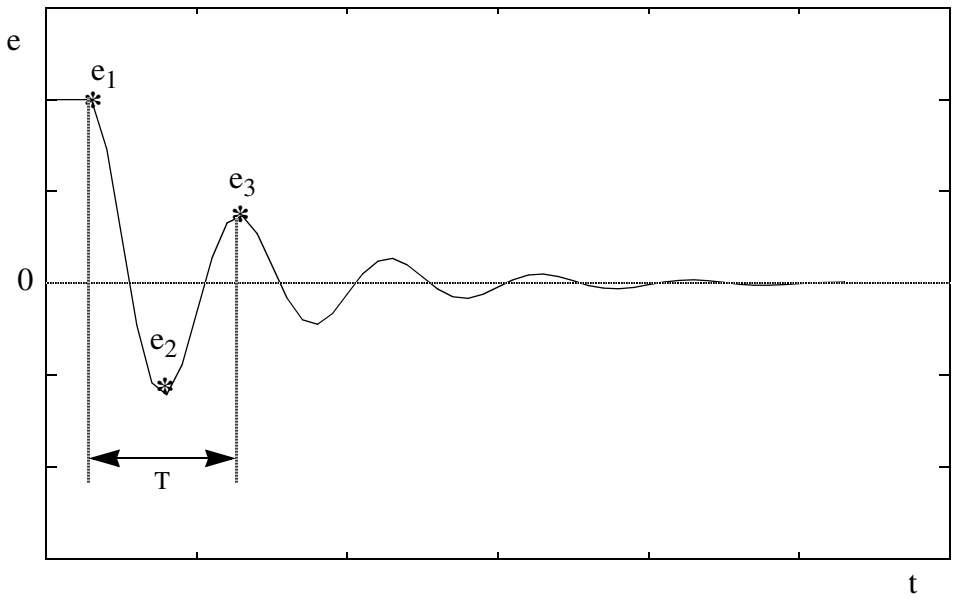


Fig. 3.12 - Typical response of the error to a step input

With the peak values, estimates of the damping (d) and the overshoot (o) are obtained:

$$d = \frac{e_3 - e_2}{e_1 - e_2} \tag{3.67}$$

$$o = -\frac{e_2}{e_1} \tag{3.68}$$

Empirical rules are used to calculate the regulator parameters using the measured values of T, d and o. These rules are based on the Ziegler-Nichols rules, augmented by experience of regulator tuning.

In this way, the controller parameters are adapted each time a disturbance (set point change or load disturbance) is sensed and observed. The tuning procedure requires prior information on the PID parameters, the time-scale and the process noise. There is a pretune feature which can be used if this information is not available. This consists of the application of a step to the plant and the use of rules similar to the Ziegler-Nichols open loop formulas to derive the initial PID parameters.

3.3.3 Methods based upon on-line parameter estimation

These methods assume a model for the process. The parameters of the model are estimated on line, usually by a recursive least-squares algorithm [107].

Based on this estimated model, different design methods, like pole placement, minimum variance and LQG are employed in self-tuning regulators. To obtain PID control algorithms the process model is limited to second order. So, as long as the process can be well approximated by such a model, self-tuning PID controllers work well. As with any self-tuning technique, self-tuning PID controllers require a previous knowledge of the time scale of the process, to determine the sampling time.

Some of the authors who have proposed self-tuning PID controllers are Wittenmark and Åstrom [121] and Gawthrop [122]. An example of a commercial PID self-tuner is the Electromax V regulator from Leeds and Northrup [107].

Radke and Isermann [123] employ on-line parameter estimation in conjunction with a numerical parameter optimization procedure. To determine the PID values they minimize the quadratic performance criterion:

$$J = \sum_{k=0}^{M} (e^2[k] + rk_p^2 \Delta u^2[k]) \tag{3.69}$$

where

$$\Delta u[k] = u[k] - u[k-1] \tag{3.70}$$

and r is the weighting factor for the control effort.

The performance criterion (3.69), following a step change in the reference variable, can be evaluated either in the time domain or in the z-domain. The minimum of (3.69) is found using the hill-climbing technique of Hooke and Jeeves. Since this process is time-consuming, the PID program is divided in two tasks: a high-priority task, which executes the

PID algorithm in real-time, and a low priority task, which performs the minimization.

3.3.4 Methods based on expert and fuzzy logic systems

Expert systems and systems fuzzy logic systems have also been proposed for PID autotuning.

An example of the first class can be found in Anderson et al. [124]. Here they propose an iterative rule-based method which analyses the response of the closed-loop system to a step change in the reference. Based on previous experience and knowledge, a new set of PID values is then chosen.

For each set-point disturbance, features of the output like overshoot, settling time, rise time, etc. are computed and compared with desired values. For each criterion not met, a rule is fired. Each rule computes the percentage of the change for each PID parameter, using a measure of the degree to which the criterion has not been satisfied, and a weight associated with that criterion. These weights, one for each parameter and for each criterion, are obtained from an expert's experience.

These percentages of change are accumulated over the criteria not met, to compute the total adjustment for each PID parameter. The PID parameters are then modified accordingly.

Lemke and De-zhao [125] introduced a fuzzy PID supervisor to adjust the settings of a PID controller. The error and its derivative, scaled by the value of the reference, are the input variables of the fuzzy supervisor. Different fuzzy regions are specified, in which the inputs are distinguished. For each fuzzy region, fuzzy rules and conditional statements are formulated, according to expert experience. After implementation of these rules the resulting fuzzy outputs are transformed into deterministic values, using a defuzzification rule. These values are the changes to be applied to the PID values.

3.3.5 Methods based on artificial neural networks

To the best of our knowledge, apart from our own approach to PID autotuning [126], to be introduced in the following section, only two other PID autotuning techniques involving artificial neural networks have been proposed.

The first, in a chronological order, was introduced by Swiniarski [127]. In this approach the open loop step response of a plant is discretized, its samples being used as inputs to a multilayer perceptron. The role of the neural network is, based on these inputs, to determine the corresponding PID parameters. The MLP has therefore three outputs, corresponding to the three PID parameters.

The open loop Ziegler-Nichols tuning rule is employed to obtain the PID parameters to

be used as targets for training. The training set is derived by varying the parameters L, T and k_p, shown in Fig. 3.11, within a specified range around nominal values. These ranges are discretized to obtain suitable examples for training. In the example proposed, the nominal values were k_p=1, L=0.2 and T=1, the range of change for the three parameters being $|0.2|$. Unfortunately, no results of this technique are presented in the paper.

Some observations can be made concerning this method of PID autotuning:

i) It is an open loop technique; it can not be applied in closed loop.

ii) Because samples of the open loop step response are used as inputs to the MLPs, obtaining good results might require a high number of samples. This leads to a large number of parameters in the MLP, resulting in large training times.

iii) This technique is likely to be heavily dependent on the sampling time chosen. Specifically, if we consider an example where the sampling time T is used, it is questionable whether the MLP will produce the same PID values if sampling times of T/2 or 2T are used.

iv) It is known that, in most cases, the responses obtained with Ziegler-Nichols tuning rule are not well damped. It seems sensible to obtain the target PID values using a better criterion.

v) If the Ziegler-Nichols technique is used to derive the target PID values then there is no need to consider three outputs for the MLP, since the integral and derivative time constants are linearly interdependent.

The second approach is due to Lightbody and Irwin [94]. Their example actually considered a PD controller, but their technique can be extended to PID regulators. In this approach the closed loop step response is discretized, the samples being used as inputs to one MLP. They consider a nominal plant, under PD control. The gains of the compensator are varied over some acceptable range, and a family of step responses obtained. The MLP is then trained to map these responses to the actual PD values which originated them.

Training having been completed, the closed loop step response of a plant (different from the nominal plant) with a PD gain vector $\mathbf{k}$ is then found. Applying this step response to the trained MLP would produce, in principle, the gain vector $\mathbf{k}_{nom}$ which would control the nominal plant in a similar way. Supposing that it was desirable to make the response of the actual system similar to some response of the nominal plant with a PD gain vector $\mathbf{k}_{des}$, the PD gain vector to produce this, $\mathbf{k}_{act}$, is given by:

$$\mathbf{k}_{act} = \mathbf{k}_{des} + \Delta\mathbf{k} \qquad \Delta\mathbf{k} = \mathbf{k} - \mathbf{k}_{nom} \tag{3.71}$$

Remarks (ii) (high number of samples required) and (iii) (sampling time dependency), made for the previous approach, are also valid for this second approach. Additionally, as this is a closed loop technique, it requires an *a priori* knowledge of a compensator **k** that is able to stabilize the actual plant.

3.4 A neural PID autotuner

The last section reviewed briefly the existing methods for PID autotuning. This section will detail our proposed connectionist approach to PID autotuning.

This novel approach exploits the mapping properties of the multilayer perceptrons. Three MLPs are used to supply the PID parameters, based on suitable plant identification measures, to a standard PID controller.

Before the MLPs can be employed in on-line control, they are trained off-line. This type of training was found preferable to on-line training, since, as pointed out in Chapter 2, training usually requires a large number of iterations. Envisaging the possibility of industrial applications of this technique, an on-line time-consuming training phase would not be desirable. Worse still, if the MLPs were started with random parameters, unstable systems would probably appear during training. This would not contribute to the practical success of this technique.

In the following description of the autotuning method, the nature of the inputs of the neural networks will be discussed first. Then we shall concentrate on the nature of their outputs. Considerations about training will be discussed next. Finally, the on-line operation will be detailed.

3.4.1 Inputs of the neural networks

The proposed technique is closely related to the method proposed by Nishikawa et al. [119]. As already mentioned, these authors proposed the use of σ (3.65) and σ' (3.66) as open-loop identification measures. For the sole purpose of closed loop identification, they also mention other identification measures, which apparently are not used afterwards.

These latter measures, however, have an important advantage over the former: they can be computed in open loop and in closed loop. This fact accounts for one of the advantages of the new technique when compared with existing methods, which is the flexibility of being applicable in open or in closed loop.

Let us then consider plants with the transfer function:

$$G(s) = \frac{Y(s)}{U(s)} = k_p e^{-Ls} \frac{\prod_{i=1}^{\bar{n}_p} (1 + T_{z_i} s)}{\prod_{i=1}^{n_p} (1 + T_{p_i} s)} , \tag{3.72}$$

which are typical of a wide range of industrial plants. The plant will be assumed to be BIBO stable.

The measures

$$F(\alpha) = \left. \frac{G(s)}{k_p} \right|_{s=\alpha} = e^{-L\alpha} \frac{\prod_{i=1}^{\bar{n}_p} (1 + T_{z_i} \alpha)}{\prod_{i=1}^{n_p} (1 + T_{p_i} \alpha)} , \tag{3.73}$$

for a set of values of α ($\alpha \in \Re$), will be used to characterize the system. An algebraic sum of the time constants and the time delay of the plant

$$T_T = L + \sum_{i=1}^{n_p} T_{p_i} - \sum_{i=1}^{n_z} T_{z_i} \tag{3.74}$$

is used as a scaling factor for α:

$$\alpha = \frac{\sigma}{T_T} \tag{3.75}$$

where σ takes values in the region $[0.1, 10]$. Other values of σ would result in values of $F(\alpha)$ too close to 1 or 0 to be useful in systems identification. As α is always scaled by T_T, it is convenient to express the identification measures not in terms of α but in terms of σ, as in the following equation

$$F(\sigma) = \left. \frac{G(s)}{k_p} \right|_{s=\frac{\sigma}{T_T}} = e^{-\frac{L\sigma}{T_T}} \frac{\prod_{i=1}^{\bar{n}_p} \left(1 + \frac{T_{z_i}\sigma}{T_T}\right)}{\prod_{i=1}^{n_p} \left(1 + \frac{T_{p_i}\sigma}{T_T}\right)} \tag{3.76}$$

The inputs of the MLPs will be the $F(\sigma)$ measures. To show how they can be obtained, we shall consider first the open-loop situation. Fig. 3.13 illustrates the output of the plant following an input step of amplitude B.

The DC gain, k_p, can be obtained using the steady state value of the output:

$$k_p = \frac{y(\infty)}{B} \tag{3.77}$$

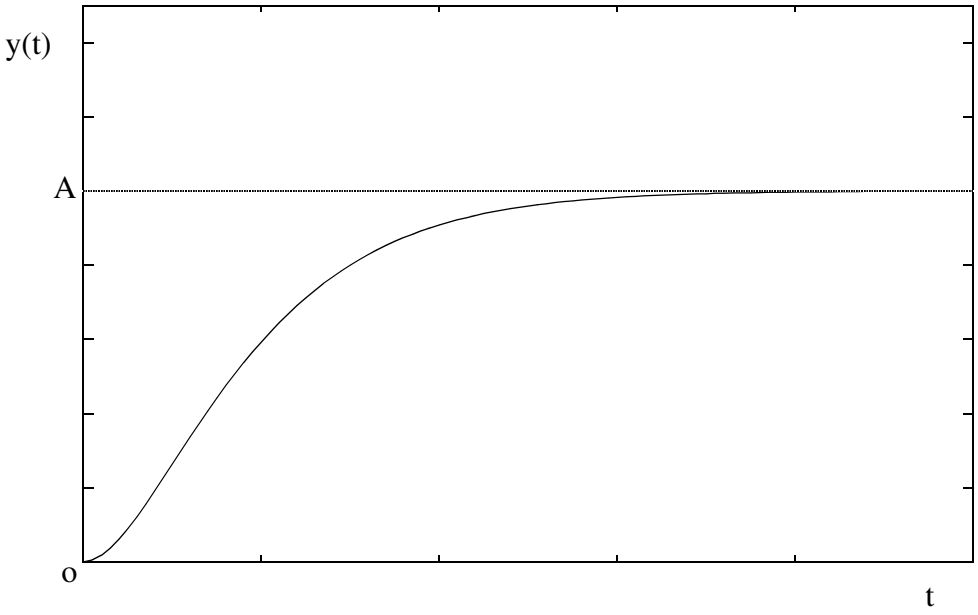


Fig. 3.13 - Open loop step response

By introducing the integral measures $S(\sigma)$ defined by:

$$S(\sigma) = \int_0^\infty e^{-\frac{\tau\sigma}{T_T}}[y(\infty) - y(\tau)]d\tau \quad , \tag{3.78}$$

and using the properties of Laplace transforms, the scaling factor, T_T, can be computed as:

$$T_T = \frac{S(0)}{k_p B} = \frac{S(0)}{y(\infty)} \tag{3.79}$$

The proof of this last equation and subsequent ones are given in Appendix A. The identification measures, $F(\sigma)$, can be obtained as:

$$F(\sigma) = 1 - \frac{\sigma S(\sigma)}{T_T y(\infty)} \tag{3.80}$$

These measures can also be obtained in a closed loop configuration. The PID compensation depicted in Fig. 3.14 will be used throughout the thesis. In order to avoid the

derivative 'kick' the derivative action operates only on the process output. As standard practice, a filter is used to reduce the noise. Without loss of generality, the time constant of the filter, T_f, is given as:

$$T_f = \frac{T_d}{\eta} \tag{3.81}$$

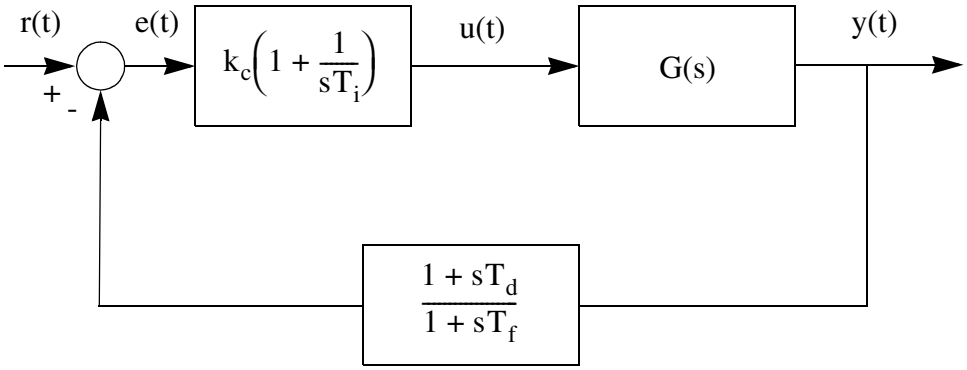


Fig. 3.14 - Closed loop system

Fig. 3.15 illustrates a typical control signal, after a step change with amplitude B is applied in the reference input.

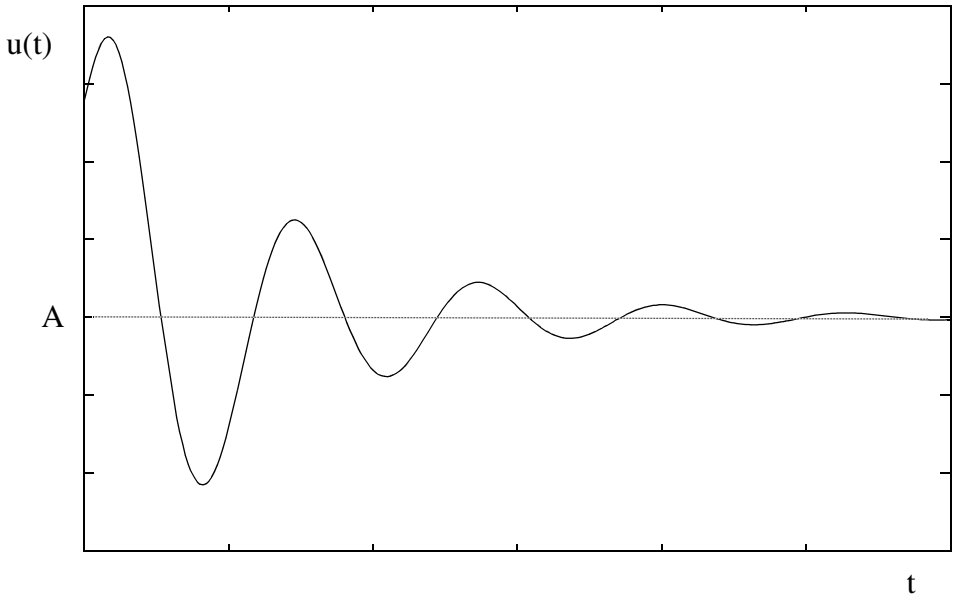


Fig. 3.15 - Control signal

The DC gain can be obtained from the steady-state value of the control signal:

$$k_p = \frac{B}{u(\infty)} \tag{3.82}$$

By computing integral measures of the control signal,

$$S_u(\sigma) = \int_0^\infty e^{-\frac{\tau\sigma}{T_T}}[u(\infty) - u(\tau)]d\tau \quad , \tag{3.83}$$

the scaling factor, T_T can be obtained as:

$$T_T = \frac{T_i}{k_p k_c} + T_d(1-\eta) - \frac{k_p S_u(0)}{B} = \frac{T_i}{k_p k_c} + T_d(1-\eta) - \frac{S_u(0)}{u(\infty)} \tag{3.84}$$

and the identification measures $F(\sigma)$ as:

$$F(\sigma) = \frac{1 + \frac{\sigma T_d}{\eta T_T}}{1 + \frac{\sigma T_d}{T_T}}\left(\frac{B}{B - \frac{\sigma k_p S_u(\sigma)}{T_T}} - \frac{\sigma T_i}{k_c k_p (T_T + \sigma T_i)}\right) \tag{3.85}$$

The measures $F(\sigma)$ can also be obtained using integral measures of the output signal,

$$S_y(\sigma) = \int_0^\infty e^{-\frac{\tau\sigma}{T_T}}[y(\infty) - y(\tau)]d\tau \quad , \tag{3.86}$$

as shown in (3.87):

$$F(\sigma) = \frac{T_i\left(1 + \frac{\sigma T_d}{\eta T_T}\right)\left(B - \frac{\sigma S_y(\sigma)}{T_T}\right)}{k_c k_p\left(1 + \frac{\sigma T_i}{T_T}\right)\left(S_y(\sigma)\left(1 + \frac{\sigma T_d}{T_T}\right) - B T_d\left(1 - \frac{1}{\eta}\right)\right)} \tag{3.87}$$

This is usually not used, since T_T and k_p must always be obtained from the control signal.

3.4.2 Outputs of the neural networks

Three multilayer perceptrons are used in this technique. All use the same inputs, the $F(\sigma)$ measures introduced in the last section. The output of each one of the MLPs should then be the corresponding PID parameter. However, a general approach, not depending on the absolute value of the time constants and delay of the plant, but only on their relative values, is desired. For this reason, the outputs of the MLPs responsible for the integral and derivative time mappings are not T_i and T_d, but these values scaled by T_T, defined in (3.74). Using the same argument, the output of the MLP responsible for the proportional gain of the controller should be (inversely) scaled by the plant steady-state gain, k_c. As $k_p k_c$ is, for the majority of the plants, greater than one, in order to keep the range of the output of the MLP responsible

for the proportional gain small, the inverse of that scaled value is actually used.

This way, the three MLPs should produce the mappings shown in (3.41).

$$\text{I}: \ F(\sigma) \rightarrow \overline{k_c} = \frac{1}{k_c k_p} \tag{3.88a}$$

$$\text{II}: \ F(\sigma) \rightarrow \overline{T_i} = \frac{T_i}{T_T} \tag{3.41b}$$

$$\text{III}: \ F(\sigma) \rightarrow \overline{T_d} = \frac{T_d}{T_T} \tag{3.41c}$$

3.4.3 Training considerations

As already stated, the MLPs, before being used in on-line control, will be trained off-line. As with all off-line training, the examples used for the training must adequately span the entire range of operation. In this case it is difficult, or even impossible, to know the operational range of the $F(\sigma)$ identification measures. One indirect way to solve this problem is to have previous knowledge about the type or types of the transfer function of the plant and, for each type of transfer function, the range of its time constants and delay time. For each time constant or delay of each transfer function, its range should then be discretized in order to have suitable examples for training. This discretization should be performed with care, to avoid ending up with duplicate examples in the training set. For instance, the plant with transfer function

$$G_1(s) = k_p \frac{e^{-0.5s}}{(s+1)(s+4)} \tag{3.89}$$

produces the same $F(\sigma)$ values as

$$G_2(s) = k_p \frac{e^{-4s}}{(s+8)(s+32)} \tag{3.90}$$

since the time constants and delay time of the second plant are the corresponding parameters of the first plant multiplied by 8.

Several criteria can be used to train the MLPs. For instance, the Ziegler-Nichols tuning rule could be used to determine the PID values used for the tuning. As an example, let us consider that the plant to be controlled is well represented by the transfer function:

$$G(s) = k_p \frac{e^{-Ls}}{1 + sT_p} \tag{3.91}$$

where $k_p>0$. Let us further assume that the range of variation of L and T_p is such that the normalized delay time varies between 0.2 and 2. If the PID parameters were chosen according to the closed-loop Ziegler-Nichols tuning rule, the relation between the identification measure F(1) and the normalized PID values is the one shown in Fig. 3.16.

We could then train the MLPs to approximate these mappings, within the range of F(1). In this case only two MLPs would be needed, since T_i and T_d are related by a constant. Assuming that a good approximation was obtained by the MLPs they could then be used to supply, either in open loop or in closed loop, the Ziegler-Nichols derived PID values. Note that, by using the normalized PID values as outputs of the networks, the tuning is independent of the steady-state gain of the plant and independent of the absolute value of L and T_p.

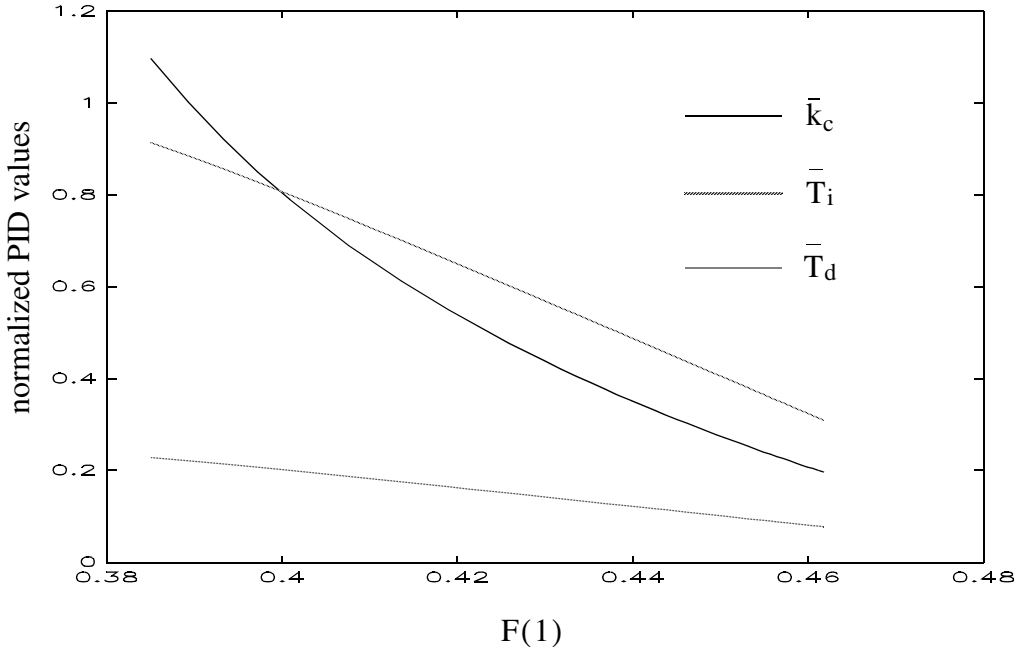


Fig. 3.16 - Normalized PID values versus F(1) for plant (3.91)

One of the advantages of the proposed technique is, however, not to be restricted to a specific design for determining the PID parameters of the training set. This feature should be exploited.

The simple fact that several methods have been proposed as alternatives to Ziegler-Nichols tuning rules is practical evidence that it does not produce satisfactory responses. It is usually recognized that well damped responses can be obtained using integral performance criteria [128][129]. Several criteria of this type exist. The most commonly employed are:

i) the integral of the squared error (ISE):

$$ISE = \int_0^\infty e^2(t)dt \tag{3.92}$$

ii) the time weighted ISE (TWISE):

$$TWISE = \int_0^\infty te^2(t)dt \tag{3.93}$$

iii) the exponential time weighted ISE (ETWISE):

$$ETWISE = \int_0^\infty e^{\beta t}e^2(t)dt \qquad \beta > 0, \text{ and} \tag{3.94}$$

iv) the integral of time multiplied by the absolute error (ITAE):

$$ITAE = \int_0^\infty t|e(t)|dt \tag{3.95}$$

When the plant has no time-delay, the first three criteria can be expressed analytically using Åstrom's integral formula [130]. When the plant has time-delay, in order to have an analytical expression for the integral, usually a second order Padé approximation is employed. To the best of our knowledge, an analytical expression for the ITAE criterion is not known. This implies that it must be computed by a quadrature or trapezoidal formula using samples of e(t), obtained from a digital simulation of the closed loop system.

The ISE criterion often produces relatively oscillatory closed loop step responses [119][129] due to the great contribution of the large errors that occur in the initial part of the response. The other three criteria usually produce better damped responses since more emphasis is put on the errors that occur later in the response. The ITAE is usually recognized as being more selective than the TWISE [131][132]. The ETWISE has the disadvantage of needing *a priori* definition of the parameter β. Another practical disadvantage of this criterion is that the closed loop system often becomes unstable during the course of the optimization. The ITAE criterion has the disadvantage of not possessing an analytical expression. Overall, the last three criteria produce well damped responses, with perhaps the ITAE being slightly better than the other two.

Nowadays there are commercially available packages that can minimize these performance criteria within a small amount of time using affordable processors (workstations, 386 PCs, etc.). However, the computational burden introduced by the numerical minimization procedure dictates that, in most cases, integral performance criteria cannot be applied in real-time applications. This situation is aggravated when the performance criteria used cannot be

expressed in an analytical form, as is the case with the ITAE. Further, as these methods rely on a transfer function for the process, its use is prohibited in situations where the plant is characterized by different transfer functions during its life time.

However, as in the proposed approach to PID autotuning, training is performed off-line, integral performance criteria can and should be used to compute optimal PID values for the plants in the training set. If an accurate approximation is obtained by the MLPs then the well damped responses typically obtained by the use of these criteria can also be obtained in real-time, with a minimum of delay.

To obtain the data needed to train the MLPs, the optimal PID parameters are computed according to either the TWISE, ETWISE, or ITAE criteria, for every model in the training set. These optimal values are then normalized using (3.41) and constitute the target output data. Also for every model in the training set, the $F(\sigma)$ identification measures are computed using (3.76). The number of measures to be computed and the actual values of σ to be employed are, however, not well known. To clarify this problem some experiments have been performed and will be discussed in section 3.5. The $F(\sigma)$ measures are the training input data. Once the training data is complete, three multilayer perceptrons are trained to approximate the mappings.

3.4.4 On-line operation

As we have seen, the off-line operations needed for this technique are certainly time-consuming. On the other hand, the on-line operations are simple and can be executed quickly.

If open-loop tuning is being performed, an input step is applied to the plant. The steady-state gain is obtained by observing the steady-state value of the output and the amplitude of the input step. The output is sampled, and S(0), defined by (3.78) is computed using a quadrature or trapezoidal formula. With S(0), T_T is computed using (3.79). Using the same procedure, a number of integral measures $S(\sigma)$, equal to the number used in the training phase are computed, for the same values of σ. Then the $F(\sigma)$ identification measures are obtained using (3.80) and applied to the trained MLPs. The normalized PID values are then output by the MLPs and the absolute values of the PID parameters obtained using

$$\begin{aligned} k_c &= \frac{1}{\overline{k}_c k_p} \\ T_i &= \overline{T}_i T_T \\ T_d &= \overline{T}_d T_T \end{aligned} \tag{3.96}$$

Closed-loop tuning follows a similar procedure. The only difference lies in using the control signal instead of the output signal. The DC gain is obtained from the steady-state value of the control and the amplitude of the reference step, using (3.82). To determine T_T, $S_u(0)$ is numerically computed from the samples of u(t) and then eq. (3.84) is used. The identification measures $F(\sigma)$ are obtained by first numerically obtaining the integrals $S_u(\sigma)$ and then using eq. (3.85).

3.5 Simulation results

To obtain a preliminary assessment of the results of the new technique, we considered a hypothetical situation of a plant described by four types of transfer function: time-delay systems with one and two poles, three-pole systems and four-pole systems. For each transfer function a large range of time constants and delay time was considered. The range of time constants and time delay was discretized, in the manner shown below.

i) $G(s) = k_p \frac{e^{-Ls}}{(s+p)}$ $\quad L \in \{0.1{:}0.1{:}3\},\ p = -1$ 30 examples

ii) $G(s) = k_p \frac{e^{-Ls}}{(s+p_1)(s+p_2)}$

$p_1 = -1,\ p_2 \in \{-1{:}{-1}{:}{-6}\},\ L \in \{0.1{:}0.3{:}2.8\}$ 60 examples

iii) $G(s) = \frac{k_p}{(s+p_1)(s+p_2)(s+p_3)}$

$p_1 \in \{-0.1, -0.2, -0.4, -0.8\}$

$p_2 \in \{-1, -2, -4, -8\},\ p_3 \in \{-1.5, -3, -6, -12\}$ 37 examples

iv) $G(s) = \frac{k_p}{(s+p_1)(s+p_2)(s+p_3)(s+p_4)}$

$p_1 = -1,\ p_2 \in \{-1, -2, -4, -8\}$

$p_3 \in \{-1, -1.5, -3, -4.5\},\ p_4 \in \{-1, -1.25, -2.5, -3.75\}$ 64 examples

In cases (i) and (ii) the middle value inside the curly brackets denotes the step-size used in the discretization. For case (iii) the way that the discretization was performed led to duplicate values of $F(\sigma)$. After these duplicates have been removed, the $64(=4^3)$ possible examples had been reduced to 37. In total, the training set was composed of 191 different models.

For each one of the 191 examples the optimal values of the PID parameters, according to the ITAE criterion, was computed. For all the examples, a unitary DC gain was assumed. To obtain the optimal PID values, the 'fminu' function of the MATLAB [133] optimization toolbox [134] was employed. This function implements a quasi-Newton method of unconstrained optimization, with BFGS (Broyden, Fletcher, Goldfarb and Shanno) update, and a mixed quadratic and cubic polynomial line search. In each iteration of the optimization algorithm, the closed loop step response was obtained by digitally simulating the system in MATLAB. The ITAE value was computed using a trapezoidal formula. Gradient information needed for the optimization routine was derived using a numerical differentiation method.

At first, the starting PID values for the optimization of each example were the ones obtained using the Ziegler-Nichols tuning rule. However, it was soon realized that better starting values might be obtained using the optimal values of previous optimizations. This way, the starting values for each example were the ones, among the previously computed optimal values, which corresponded to the plant whose $F(\sigma)$ values were the closest (Hamming distance) to the $F(\sigma)$ of the plant to optimize.

After the optimal PID values were obtained for all the 191 examples, they were normalized according to (3.41). For each of the 191 examples, five measures of $F(\sigma)$ were computed, using values of σ belonging to the set $\{0.5, 1, 2, 3, 5\}$.

After this time-consuming process, the training data set is complete. Two topologies of MLPs were tested for storing the mappings. Both had five input neurons as five identification measures were used, and one output neuron with a linear activation function. The first MLP had one hidden layer with 10 neurons. The second MLP had two hidden layers, each one with 7 neurons. All the neurons in the hidden layers had sigmoid activation functions. Because of computational constraints in the early stages of this work, small MLPs had to be employed. As stated in Chapter 2, there are at present no definite guidelines for determining the number of hidden layers and/or the number of neurons per hidden layer to employ. The actual number of neurons employed, for the two MLPs, was therefore chosen based on past experience in different problems.

To train the MLPs several methods were tried, starting with the error back-propagation algorithm. This method was found to have a very slow rate of convergence. Several thousand of iterations were performed without the convergence criteria being met. Additionally, a good value for the learning parameter employed in this algorithm was very difficult to find. This led to research on more powerful methods of training, which are described in full in Chapter 4.

After the MLPs have been trained, their performance within the training set is compared. As in the approach of Nishikawa et al. [119] polynomials are used to approximate

mappings between open-loop identification measures and optimal PID values (ETWISE sense), second and third order polynomials were also used to approximate the mappings.

Denoting by **X** the (191*5) matrix of the identification measures, a second order expansion matrix of **X**, $\mathbf{A}^{(2)}$, was obtained. This matrix may be interpreted as a partitioned matrix:

$$\mathbf{A}^{(2)} = \left[1 \ \mathbf{X} \ \mathbf{X}^{(2)}\right] \tag{3.97}$$

where element i,k of matrix $\mathbf{X}^{(2)}$ is obtained as:

$$\mathbf{X}^{(2)}_{i,k} = \mathbf{X}_{i,j}\mathbf{X}_{i,m} \qquad \begin{cases} j = 1, \ldots 5 \qquad m = j, \ldots, 5 \\ k = m - j + 1 + \sum_{n=0}^{j-2} (5 - n) \end{cases} \tag{3.98}$$

Three linear models, with a number of parameters equal to the number of columns of $\mathbf{A}^{(2)}$, were afterwards fitted to the training data by solving equations of the type:

$$\mathbf{a}^{(2)} = (\mathbf{A}^{(2)})^{+}\mathbf{t} \tag{3.99}$$

In the last equation **t** denotes the vector of the optimal PID values (one for each PID parameter), $\mathbf{a}^{(2)}$ is the parameter vector of the second order polynomial, and the symbol '+' denotes a pseudo-inverse operation [135].

To obtain the third order polynomial, a third order expansion of the matrix **X** was obtained. Denoting this matrix by $\mathbf{A}^{(3)}$:

$$\mathbf{A}^{(3)} = \left[\mathbf{A}^{(2)} \ \mathbf{X}^{(3)}\right] \tag{3.100}$$

where the element i,k of matrix $\mathbf{X}^{(3)}$ is given as:

$$\mathbf{X}^{(3)}_{i,k} = \mathbf{X}_{i,j}\mathbf{X}_{i,m}\mathbf{X}_{i,n} \qquad \begin{cases} j = 1, \ldots, 5 \qquad m = j, \ldots, 5 \qquad n = m, \ldots, 5 \\ k = n - m + 1 + \sum_{l=2}^{j} \sum_{i=1}^{5-l+2} i + \sum_{i=i}^{m-1} (6 - i) \end{cases} \tag{3.101}$$

Again three linear models were fitted to the training data using:

$$\mathbf{a}^{(3)} = (\mathbf{A}^{(3)})^{+}\mathbf{t} \tag{3.102}$$

To compare the quality of the four approximations, the sum of the square of the errors (s.s.e.) and the major relative absolute error (m.r.a.e.) were obtained for the three mappings and are summarized in Table 3.3. The m.r.a.e is defined as:

$$\text{m.r.a.e.} = \max\left(\frac{|\hat{\mathbf{e}}_k|}{\mathbf{t}_k}\right) \qquad k = 1, \ldots, 191 \tag{3.103}$$

where $\mathbf{t}$ is the target vector and $\hat{\mathbf{e}}$ is the optimal error vector.

	k_c		T_i		T_d	
	s.s.e.	m.r.a.e.	s.s.e.	m.r.a.e.	s.s.e.	m.r.a.e.
MLP. (1 h l.)	.28	1.09	.89	1.04	.14	1.77
MLP. (2 h l.)	.27	.95	.62	.69	.13	1.5
2nd O. Pol.	.57	2.12	1.36	3.81	.20	4.06
3rd O. Pol.	.43	.89	1.04	1.66	.19	1.89

Table 3.3 - Error measures obtained after learning with the training data

It can be observed that the two MLPs obtain better performance than the polynomial approximations, according to both criteria. Only in the case of the mapping of the proportional gain does the third order polynomial achieve the best result, according to the m.r.a.e. criterion, but its performance in the overall training set is much worse than any of the MLPs.

Comparing the performance of the two MLPs, it is clear that the one with two hidden layers achieves the best results for all the mappings according to both criteria.

Polynomial expansions of the input data were also used to address another problem. In the example that is being followed, five different measures of $F(\sigma)$ are used to identify the plant. However, as the maximum number of parameters of the plants in the training set is four, and as the identification measures do not depend on the absolute value of the parameters but only on their relative values then, in principle, three measures of $F(\sigma)$ would be sufficient to identify the plants. However it is admitted that this is a minimum number and that better results could be obtained with a larger number of identification measures.

Clearly this assumption should be verified. For this purpose, we then conducted a further experiment. For each one of the three mappings, we were interested in comparing the accuracy of the approximations obtained using five measures of $F(\sigma)$, with the accuracy obtained when only three or four of these values were used. Assuming that σ takes values within the set $\{0.5, 1, 2, 3, 5\}$, ten different combinations are possible with three identification measures. Five combinations are obtained if four are employed. This meant that,

to test the validity of the assumption made above, 72 additional training sessions should be performed.

At the time that this problem was being handled, training was performed using the 'leastsq' function of the Optimization Toolbox of MATLAB, running on Sun 3-60 workstations. Each training session, for the MLP with 2 hidden layers, took, on average, two to three days. Clearly, to perform this large number of training sessions was simply not feasible. This was an additionally strong motivation for the study of training methods which could reduce this enormous training time.

In view of this practical impossibility, another solution was sought. Third order polynomial expansions were used instead of MLPs. An additional assumption was made that the qualitative results obtained with polynomial approximations would also be obtained with MLPs. This is clearly a conjecture but it was, at that time, the only practical way of throwing some light on the number of identification measures to use.

Fig. 3.17 shows the norm of the error vector for the integral mapping obtained for the different combinations of σ. The numbers inside each bar denote the values of σ employed. As can be observed, on average, the use of four identification measures achieves better accuracy than three identification measures. The highest accuracy is obtained using five identification measures.

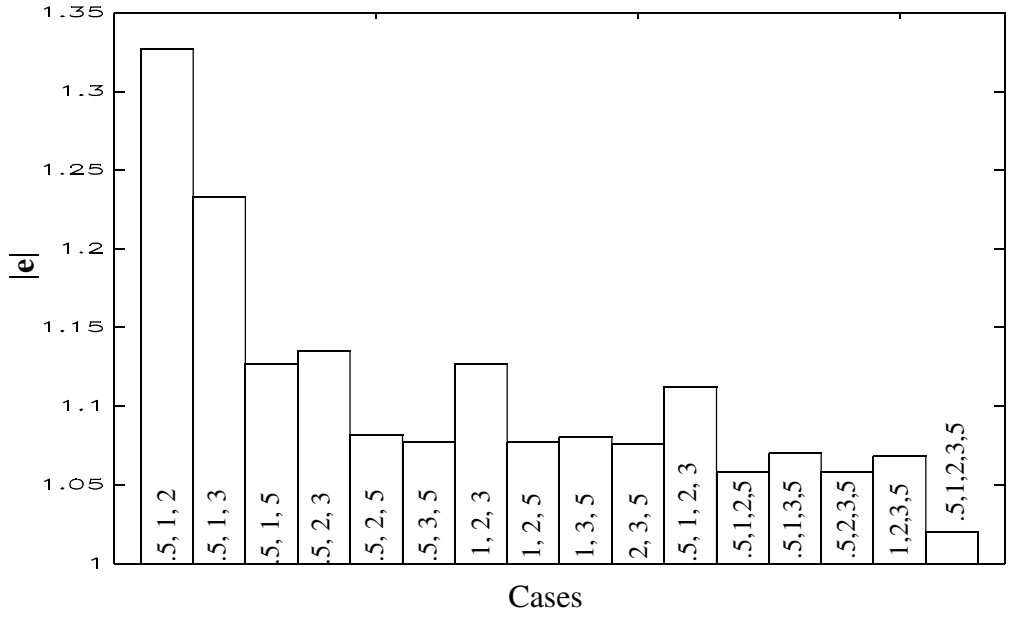


Fig. 3.17 - Norm of the error vector versus different combinations of σ for the integral mapping

Similar results were obtained for the two other mappings, indicating that a larger number of identification measures leads to better approximations.

Finally, the quality of the step responses obtained with the neural PID auto-tuner, employing the two hidden layers MLPs, was assessed. For plants within the training range no unstable systems were obtained. The majority of the responses were very close to the ones obtained by using the optimal (ITAE sense) PID values, and much better than using the Ziegler-Nichols tuning rule. In some cases, slightly overdamped responses were obtained.

Figures 3.9 and 3.10 show closed loop step responses obtained with the PID parameters derived from the open loop step response. For comparison, the responses obtained using the closed loop Ziegler-Nichols tuning rule are also shown. In Fig. 3.18, plant A has transfer function $G_A(s) = \frac{e^{-0.5s}}{(1+0.2s)(1+0.8s)}$. The response obtained was very close to the optimal one.

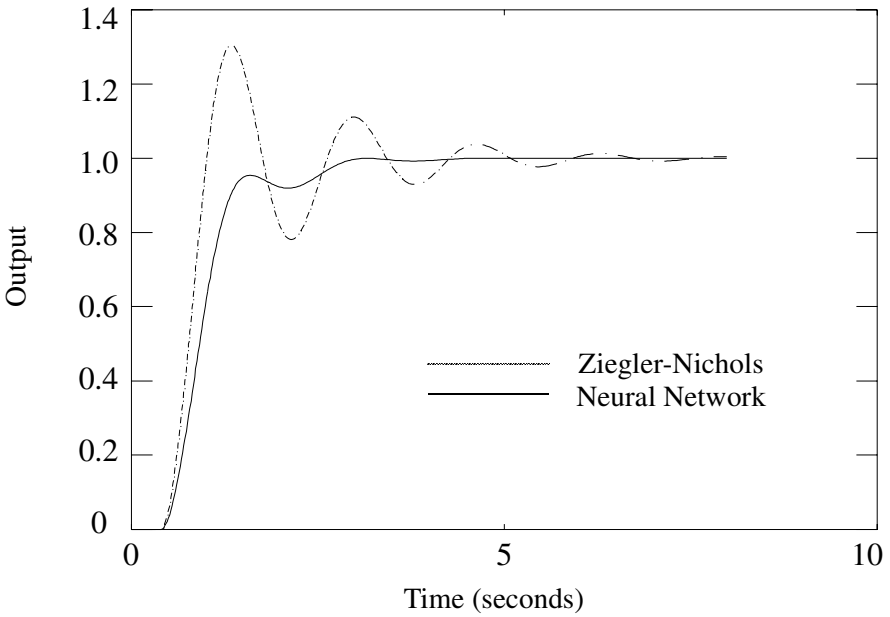


Fig. 3.18 - Open loop tuning for plant A

One example, where less good results were obtained, is shown in Fig. 3.19. Plant B has transfer function $G_B(s) = \frac{10}{(1+6s)(1+9s)(1+10s)}$. It should be noted that while these two plant descriptions fall within the bounds of the training set, neither of them is actually a member of the original training set.

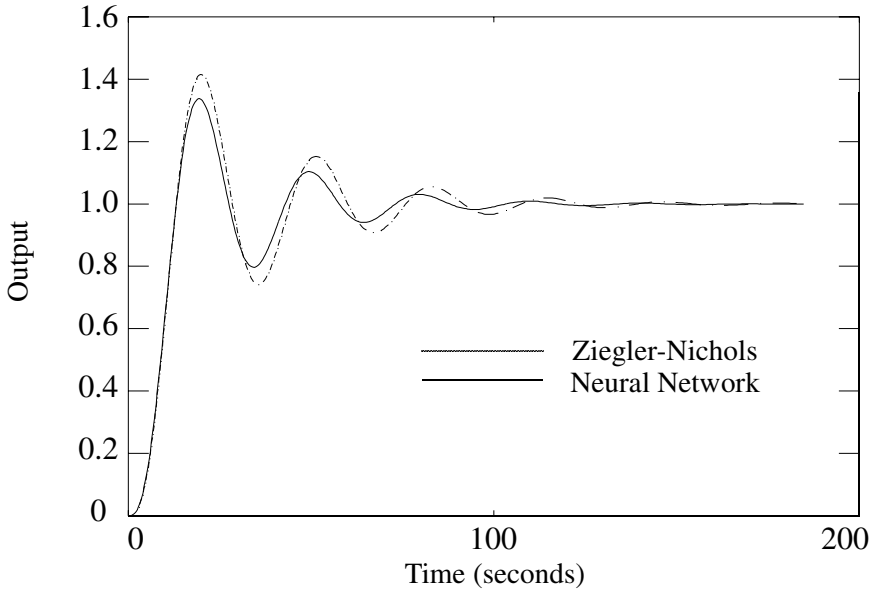


Fig. 3.19 - Open loop tuning for plant B

To illustrate closed-loop retuning, the closed-loop response to an input waveform, composed of a series of steps was simulated and is shown in Fig. 3.20. The plant was allowed to change between steps, once the transients related to the previous step had vanished. The initial value of the PID parameters was computed from the open loop step response. Further PID values were computed from the closed-loop step response.

The following sequence of changes to the plant was investigated:-

$$\frac{e^{-s}}{1+s} \rightarrow \frac{e^{-s}}{(1+s)^2} \rightarrow \frac{1}{(1+s)^2(1+0.5s)} \rightarrow \frac{1}{(1+s)^2(1+0.5s)^2}$$

At time 0 the loop was closed, and step I was applied to the reference input. The transfer function of the plant was $\frac{e^{-s}}{1+s}$. When step II was applied, the plant had been changed to $\frac{e^{-s}}{(1+s)^2}$. Since the PID parameters were tuned for the first plant, a considerable overshoot is present. The plant remains unchanged when step III is applied and better results are obtained, since the PID has retuned as a consequence of the observation of the results of the last step. In step IV a plant with transfer function of $\frac{1}{(1+s)^2(1+0.5s)}$ is assumed. The response is now typically overdamped. The controller retunes and a faster response is now

obtained in step V. In step VI the plant had changed to $\frac{1}{(1+s)^2(1+0.5)^2}$, and an oscillatory response is obtained. The controller is again retuned for step VII, and again a better response was achieved.

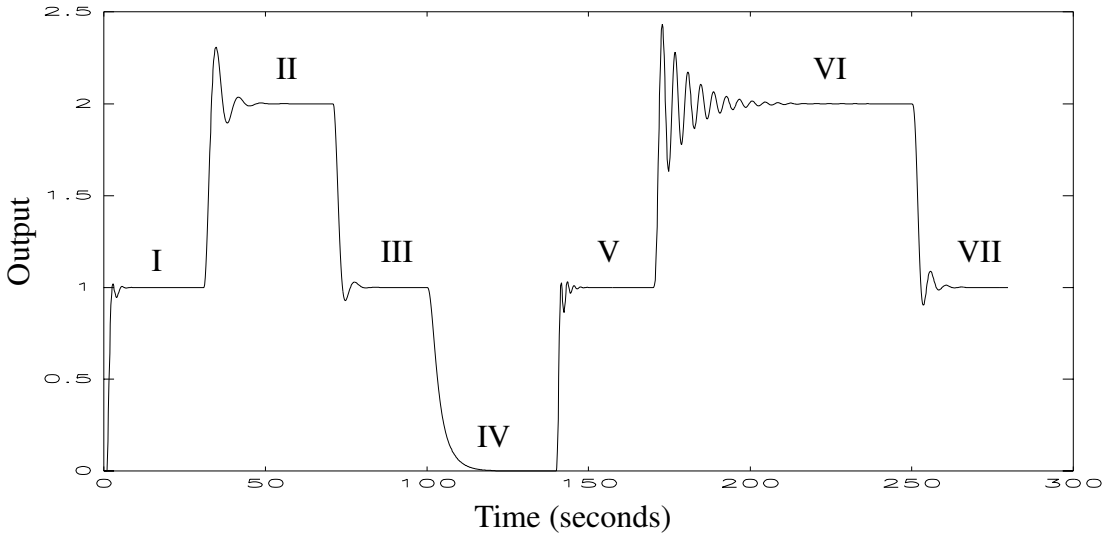


Fig. 3.20 - Step response for changing plant transfer function

3.6 Conclusions

In this Chapter a neural network approach to PID autotuning was proposed. This method has a number of attractive features, such as:

i) It can be applied both in open-loop and in closed-loop, identical tunings being obtained for either cases.

ii) In contrast with relay auto-tuners, no special hardware is needed. The tuning procedure is similar to the actual process employed by the instrument engineer, when performing manual tuning. This is an advantage if industrial application of this technique is to succeed.

iii) It does not rely on any special model for the plant. In fact simulation results suggest that changing model structures can be accommodated.

iv) Well damped responses, typical of time-consuming optimization procedures, can be obtained with a minimum amount of delay. The total time of tuning is essentially the time the plant takes to reach steady-state.

The approach, however, has some disadvantages. These are:

i) It requires a time-consuming off-line phase, prior to application in on-line control.

Most of this time is spent in the training of the MLPs.

ii) A certain amount of knowledge about the plant is required, to produce suitable examples for training. This knowledge is expressed in terms of the plant model (or models) and the relative range of variation of its time constants. When precise information is not available, an extended range of values should be assumed so that the whole operational range of the controller is covered by the training set.

iii) Theoretical proofs concerning the robustness or even the stability of this approach are not known and may even be impossible to find. As we have seen in the last Chapter, this is, at present, a characteristic of neural networks applications.

Overall, the results obtained are promising enough to deserve further investigation. These results, however, were obtained with off-line simulations. The performance of this technique, in real-time, should therefore be investigated. This is done in Chapter 6.

One problem has been identified during the course of the work presented in this Chapter: the enormous time taken to train the MLPs. This also has consequences on further experiments which should be carried out in order to answer open questions which always arise when a new method is proposed. For instance, polynomial approximations had to be used in place of MLPs to clarify the problem of how many identification measures to use. Also, we have considered just two topologies of MLPs. It is expected that, by employing larger multilayer perceptrons, a better accuracy could be achieved for the mappings between the identification measures and the normalized optimal PID values.

These considerations led to detailed research into the training of multilayer perceptrons, which is the subject of the next Chapter.

Chapter 4

Training Multilayer Perceptrons

4.1 Introduction

In the last Chapter a method for autotuning PID controllers, employing multilayer perceptrons, was introduced. Until now the existence of algorithms to perform the training process has been assumed; it has been noted that this is usually a time-consuming operation. The aim of this Chapter is the introduction and development of training algorithms, their comparison in terms of execution time, and the choice of the one that achieves the fastest training phase.

The training algorithms which will be introduced and developed in this Chapter are targeted to the type of MLP, and its method of use, when applied to the auto-tuned compensator. As explained previously, the multilayer perceptrons, before being used in on-line control, must be trained off-line. Training is therefore assumed to be an off-line procedure and all the training data must be available before learning takes place.

The algorithms to be introduced are based on the further assumption that the MLPs to be trained are small size networks. By a small size MLP, it is meant that the total number of weights and biases of the network is in the range of hundreds, a condition which is satisfied in the great majority of the current MLP applications reported in the literature. Training algorithms for large size networks, with thousands of neurons and millions of connections, are therefore outside the scope of this thesis.

Finally, in the proposed connectionist approach to PID autotuning, three different MLPs with just one output - each one being responsible for one PID parameter - were employed This approach was found preferable to using one MLP with three outputs since a better accuracy for the mappings is achieved by the first approach. The training algorithms developed in this Chapter will therefore be presented assuming multilayer perceptrons with just one output. Those algorithms, however, can be extended to a multi-output case.

With these considerations in mind, the outline of this Chapter is as follows:

In section 4.2 the error back-propagation algorithm is introduced and developed from its specifications. It is shown that each iteration of this algorithm is efficient, computationally speaking, since it fully exploits the topology of the multilayer perceptron. However, when the total learning process is considered, its performance is far from attractive. A very simple

example demonstrates this fact, and highlights the reasons for the poor performance of the error back-propagation algorithm.

Based on these results, alternatives to the error back-propagation algorithm are given in section 4.3. Three second-order optimization methods (quasi-Newton, Gauss-Newton and Levenberg-Marquardt) are introduced and their performance on some common examples compared with that of the error back-propagation algorithm.

The great majority of the MLPs used in control systems applications use linear activation functions in the neurons in the output layer. In section 4.4 this particular feature is exploited, and a different, yet equivalent, criterion for learning is developed. It is shown that the use of this reformulated criterion (proposed by us for the purpose of MLP training [136]) achieves an important reduction in the number of iterations needed for convergence. When considering the use of this new formulation in Gauss-Newton and Levenberg-Marquardt methods a Jacobian matrix must be computed. For this purpose, three different matrices (one of them proposed by us [137]) are introduced, and compared in terms of computational complexity and rates of convergence achieved by the Levenberg-Marquardt method.

In section 4.5 the results previously obtained in the preceding sections are summarised. Based on that, the best method for training the type of MLPs that are used for the PID compensator is chosen.

4.2 Error back-propagation

The aim of training the MLPs is to find the values of the weights and biases of the network that minimize the sum of the square of the errors between the target and the actual output (4.1).

$$\Omega = \frac{1}{2}(\mathbf{e}^{\mathrm{T}}\mathbf{e}) \tag{4.1}$$

The error back-propagation (BP) algorithm is the best known learning algorithm for performing this operation. In fact, MLPs and the BP algorithm are so intimately related that it is usual to find in the literature that this type of artificial neural network is referred to as a ‘back-propagation neural network’.

This algorithm has been independently developed by several people from wide-ranging disciplines [51]. Apparently it was introduced by Paul Werbos [138] in his doctoral thesis, rediscovered by Parker [139] and widely broadcast throughout the scientific community by Rumelhart and the PDP group [13].

The original error back-propagation algorithm implements a steepest descent method. In each iteration, the weights of the multilayer perceptron are updated by a fixed percentage in

the negative gradient direction:

$$\mathbf{w}[k+1] = \mathbf{w}[k] - \eta \mathbf{g}[k] \tag{4.2}$$

The parameter η is usually defined as the learning rate and $\mathbf{g}$ denotes the gradient of Ω:

$$\mathbf{g} = \begin{vmatrix} \frac{\partial \Omega}{\partial \mathbf{w}_1} \\ \cdots \\ \frac{\partial \Omega}{\partial \mathbf{w}_n} \end{vmatrix} \tag{4.3}$$

Several modifications of this algorithm have been proposed. One of them is to perform the update of the weights each time a pattern is presented. This mode of operation is usually denoted as *pattern* mode, in contrast with performing the update only when all the patterns in the training set have been presented, usually called *epoch* mode. The reasoning behind pattern mode update is that, if η is small, the departure from true gradient descent will be small and the algorithm will carry out a very close approximation to gradient descent in sum-squared error [13].

Another modification, also introduced by Rumelhart and the PDP group [13], is the inclusion of a portion of the last weight change, called the *momentum* term, in the weights update equation:

$$\mathbf{w}[k+1] = \mathbf{w}[k] - \eta \mathbf{g}[k] + \alpha(\mathbf{w}[k] - \mathbf{w}[k-1]) \tag{4.4}$$

The use of this term would, in principle, allow the use of a faster learning rate, without leading to oscillations, which would be filtered out by the momentum term. It is however questionable whether in practice this modification increases the rate of convergence. Tesauro et al. [140] suggest that the rate of convergence is essentially unmodified by the existence of the momentum term, a fact that was also observed in several numerical simulations performed by us.

Several other modifications to the BP algorithm were proposed. To name but a few, Jacobs [141] proposed an adaptive method for obtaining the learning rate parameter, Watrous [142] has employed a line search algorithm, Sandom and Uhr [143] have developed a set of heuristics for escaping local error minima.

The error back-propagation algorithm, in its original version, is developed below. Subsequently, the limitations of this algorithm are pointed out.

4.2.1 The error back-propagation algorithm

The error back-propagation algorithm, when used in epoch mode, and without a momentum term, implements a steepest descent method. In each iteration, the weights of the MLP are updated using (4.2).

One advantage of the error back-propagation algorithm is that it enables the computation of the gradient vector in an efficient way, computationally speaking. This efficiency derives from the fact that this algorithm fully exploits the layered topology of the MLPs.

To prove this, the steps involved in the computation of the gradient vector are given below. Starting from (4.3), the gradient of Ω can be given as:

$$\mathbf{g}^T = -\mathbf{e}^T\mathbf{J} \tag{4.5}$$

where $\mathbf{J}$, usually called the Jacobian matrix, is the matrix of the derivatives of the output vector of the MLP with respect to (w.r.t.) its weights:

$$\mathbf{J} = \frac{\partial}{\partial \mathbf{w}}\mathbf{o}^{(q)} \ddagger \tag{4.6}$$

One possibility for computing $\mathbf{g}$ is to use (4.5) directly, by obtaining the error vector and the Jacobian, and then computing the product. Following this approach, it is advisable to compute $\mathbf{J}$ in a partitioned fashion, reflecting the topology of the MLP:

$$\mathbf{J} = \left[\mathbf{J}^{(q-1)} \ \cdots \ \mathbf{J}^{(1)}\right] \tag{4.7}$$

In equation (4.7), $\mathbf{J}^{(z)}$ denotes the derivatives of the output vector of the MLP w.r.t. the weight vector $\mathbf{w}^{(z)}$. The chain rule can be employed to compute $\mathbf{J}^{(z)}$:

$$\begin{aligned}\mathbf{J}^{(z)} &= \frac{\partial}{\partial \mathbf{w}^{(z)}}\mathbf{o}^{(q)} = \frac{\partial}{\partial \mathbf{net}^{(q)}}\mathbf{o}^{(q)} \\ &\left\{\frac{\partial}{\partial \mathbf{O}^{(q-1)}}\mathbf{net}^{(q)} \cdots \frac{\partial}{\partial \mathbf{Net}^{(z+1)}}\mathbf{O}^{(z+1)}\right\}\frac{\partial}{\partial \mathbf{w}^{(z)}}\mathbf{Net}^{(z+1)}\end{aligned} \tag{4.8}$$

In this last equation, the terms inside the curly brackets occur in pairs, since the

‡. Lower case letters are used for $\mathbf{o}^{(q)}$ and $\mathbf{net}^{(q)}$ since we are considering just one neuron in the output layer

derivatives $\frac{\partial}{\partial \mathbf{Net}^{(j-1)}}\mathbf{Net}^{(j)}$ can be obtained as:

$$\frac{\partial}{\partial \mathbf{Net}^{(j-1)}}\mathbf{Net}^{(j)} = \frac{\partial}{\partial \mathbf{O}^{(j-1)}}\mathbf{Net}^{(j)}\frac{\partial}{\partial \mathbf{Net}^{(j-1)}}\mathbf{O}^{(j-1)} \tag{4.9}$$

The number of these pairs of terms is q-z-1, which means that they do not exist for z=q-1.

If (4.8) is followed, three-dimensional quantities must be employed. Their use does not clarify the text, and if introduced, additional conventions should have to be defined. To avoid this additional complexity, and since there is no dependence from presentation to presentation in the operation of the MLPs, the computation of $\mathbf{J}^{(z)}$ will be developed in a pattern basis, rather than in epoch basis. So, the Jacobian of layer z, for presentation i, is just:

$$\begin{aligned}\mathbf{J}^{(z)}_{i,.} &= \frac{\partial \mathbf{o}^{(q)}_i}{\partial \mathbf{w}^{(z)}} = \frac{\partial \mathbf{o}^{(q)}_i}{\partial \mathbf{net}^{(q)}_i} \\ &\left\{\frac{\partial \mathbf{net}^{(q)}_i}{\partial \mathbf{O}^{(q-1)}_{i,.}}\cdots\frac{\partial \mathbf{O}^{(z+1)}_{i,.}}{\partial \mathbf{Net}^{(z+1)}_{i,.}}\right\}\frac{\partial \mathbf{Net}^{(z+1)}_{i,.}}{\partial \mathbf{w}^{(z)}}\end{aligned} \tag{4.10}$$

Detailing now each constituent element of the right hand-side of (4.10), it is easy to see that:

a) $\frac{\partial \mathbf{O}^{(j)}_{i,.}}{\partial \mathbf{Net}^{(j)}_{i,.}}$, with $q<j<1$, is a diagonal matrix whose k^{th} diagonal element is $\mathbf{f}'(\mathbf{Net}^{(j)}_{i,k})$; for the case of the output layer (j=q), it is just a scalar equal to $f'(\mathbf{net}^{(q)}_i)$;

b) $\frac{\partial \mathbf{Net}^{(j)}_{i,.}}{\partial \mathbf{O}^{(j-1)}_{i,.}} = (\mathbf{W}^{(j-1)}{}_{1..\mathbf{k}_j,.})^T$, i.e., it is the transpose of the weight matrix between layers (j-1) and j, without considering the threshold vector; for the special case of the output layer, this derivative is a row vector;

c) $\frac{\partial \mathbf{Net}^{(z+1)}_{i,.}}{\partial \mathbf{w}^{(z)}}$ can best be described as a partitioned matrix, with $\mathbf{k}_{z+1}$ column partitions; its j^{th} partition ($\mathbf{A}_j$) has the form:

$$\mathbf{A}_j = \begin{bmatrix} 0 \\ \left[\mathbf{O}^{(z)}_{i,.}\ 1\right] \\ 0 \end{bmatrix} \tag{4.11}$$

where the top and the bottom null matrices on the right hand-side have dimensions of

$(j-1)*(\mathbf{k}_z+1)$ and $(\mathbf{k}_{z+1}-j)*(\mathbf{k}_z+1)$, respectively. $\mathbf{1}$ denotes a column vector of ones of suitable dimension.

Using relationships (a) to (c), equation (4.10) can be further manipulated to yield a more compact form for the computation of $\mathbf{J}_{i,.}$.

The matrices described in a) and c) are sparse matrices, with a format that reflects the organization of one layer into different neurons. This way, another simplification can be achieved if we further partition $\mathbf{J}^{(z)}_{i,.}$ among the different neurons within the layer z+1. Using the same type of conventions used so far, the sub-partition of $\mathbf{J}^{(z)}$ corresponding to the weights that connect the neurons in the z^{th} layer with the p^{th} neuron in layer z+1, should be denoted as $\mathbf{J}^{(z)}_{i,p}$, where:

$$p = ((p-1)(\mathbf{k}_z+1)+1)..p(\mathbf{k}_z+1) \tag{4.12}$$

To keep the notation as simple as possible, this sub-partition will be denoted as:

$$\mathbf{J}^{(z)}_{i,p} = \frac{\partial \Omega}{\partial \mathbf{W}^{(z)}_{.,p}} \tag{4.13}$$

Since the p^{th} partition of $\frac{\partial \mathbf{Net}^{(z+1)}_{i,.}}{\partial \mathbf{w}^{(z)}}$ has just the p^{th} row non-null (see (4.11)), $\mathbf{J}^{(z)}_{i,p}$ can be given as:

$$\mathbf{J}^{(z)}_{i,p} = \frac{\partial \mathbf{o}^{(q)}_i}{\partial \mathbf{Net}^{(z+1)}_{i,.}} \frac{\partial \mathbf{Net}^{(z+1)}_{i,.}}{\partial \mathbf{W}^{(z)}_{.,p}} = \left[\frac{\partial \mathbf{o}^{(q)}_i}{\partial \mathbf{Net}^{(z+1)}_{i,.}}\right]_p \left[\mathbf{O}^{(z)}_{i,.} \; 1\right] \tag{4.14}$$

The diagonallity of matrix $\frac{\partial \mathbf{O}^{(j)}_{i,.}}{\partial \mathbf{Net}^{(j)}_{i,.}}$ can also be exploited, to simplify further the calculations. Using this property, it is easy to see that the product of each pair of matrices inside the curly brackets of (4.10) can be given as:

$$\frac{\partial \mathbf{Net}^{(j)}_{i,.}}{\partial \mathbf{Net}^{(j-1)}_{i,.}} = (\mathbf{W}^{(j-1)}{}_{1..\mathbf{k}_j,.})^T \times \mathbf{f}'(\mathbf{Net}^{(j)}_{i,.}) \tag{4.15}$$

where the symbol $\times$ has the meaning that each column of the premultiplying matrix is multiplied by the corresponding element of the postmultiplying row vector.

The final consideration before presenting an efficient algorithm to compute $\mathbf{J}_{i,.}$ concerns the order of the computations of its different partitions. Following (4.10), we note

that, for every layer z, $\frac{\partial \mathbf{o}^{(q)}_i}{\partial \mathbf{Net}^{(z+1)}_{i,.}}$ must be obtained. Further, because of the layered structure of the MLPs, these derivatives are obtained in a recursive fashion, i.e., to compute $\frac{\partial \mathbf{o}^{(q)}_i}{\partial \mathbf{Net}^{(z+1)}_{i,.}}$, the matrices $\frac{\partial \mathbf{o}^{(q)}_i}{\partial \mathbf{Net}^{(j)}_{i,.}}$, with $q \geq j > z+1$, must be calculated beforehand. This way, it is clear that no duplicate calculations will be made if $\mathbf{J}_{i,.}$ is computed starting from the output layer towards the input layer.

Keeping the preceding points in mind, Algorithm 1 illustrates an efficient way of computing the Jacobian matrix of an MLP.

Algorithm 1 - Computation of $\mathbf{J}_{i,.}$

```
z := q
d := [f'(net^(q)_i)]
while z>1
    for j=1 to k_z
        J^(z-1)_{i,j} := d_j [O^(z-1)_{i,.}  1]
    end
    if z>2
        d := d(W^(z-1))^T
        d := d × f'(Net^(z-1)_{i,.})
    end
    z := z-1
end
```

Denoting by k the number of neurons in the MLP (without considering the input layer) and by n the total number of weights in the network, an analysis of the operations involved in Algorithm 1 shows that, roughly, 2mn multiplications, mn additions and km derivatives are needed to obtain the Jacobian matrix.

Using this algorithm, in conjunction with (4.5), to compute the gradient vector, requires that two other points be addressed:

a) $\mathbf{Net}^{(j)}_{i,.}$ and $\mathbf{f}'(\mathbf{Net}^{(j)}_{i,.})$, for $q \geq j > 1$, must be available; also, $\mathbf{O}^{(j)}_{i,.}$, for $q > j \geq 1$, must be known beforehand;

b) the error vector, and consequently the output vector, must be available.

Point b) implies that a recall operation must be executed. If this operation is performed

before the computation of the Jacobian matrix takes place, then $\mathbf{Net}^{(j)}_{i,.}$ and $\mathbf{O}^{(j)}_{i,.}$ can be obtained. Also for output functions like the sigmoid or the hyperbolic tangent functions, the computation of the derivative can be obtained from the output, which simplifies the calculations.

The recall operation can be implemented using Algorithm 2, which, for consistency with Algorithm 1, is also derived on a presentation mode.

Algorithm 2 - Computation of $\mathbf{o}^{(q)}_i$

```
z := 1
while z<q
    Net^(z+1)_{i,.} := [O^(z)_{i,.}  1] W^(z)
    O^(z+1)_{i,.} := f^(z+1)(Net^(z+1))
    z := z+1
end
```

Using this algorithm, mk output function evaluations and mn additions and multiplications are needed to compute $\mathbf{o}^{(q)}$.

Employing algorithms 1 and 2, the gradient vector can be obtained using Algorithm 3:

Algorithm 3 - Computation of $\mathbf{g}$ using (4.5)

```
g^T := 0
i := 1
while i ≤ m
    ... compute o^(q)_i                    -- (Algorithm 2)
    e_i := t_i – o^(q)_i
    ... compute J_{i,.}                    -- (Algorithm 1)
    g^T := g^T – e_i J_{i,.}
end
```

This last algorithm computes the gradient vector, with the same computational complexity as the error back-propagation algorithm, as presented by Rumelhart et al. [13]. However, it does not share one of its advantages because not all the computations are performed locally in the neurons.

This property can be provided by slightly modifying Algorithm 3. By doing that, we obtain the gradient vector as computed by the BP algorithm:

Algorithm 4 - Computation of **g** using BP

$\mathbf{g}^T := 0$
i := 1
while i ≤ m
    ... compute $\mathbf{o}^{(q)}_i$ -- (Algorithm 2)
    $\mathbf{e}_i := \mathbf{t}_i - \mathbf{o}^{(q)}_i$
    z := q
    $\mathbf{d} := [f'(\mathbf{net}^{(q)}_i)\mathbf{e}_i]$
    while z>1
        for j=1 to $\mathbf{k}_z$
            $\mathbf{g}^{(z-1)^T}_j := \mathbf{g}^{(z-1)^T}_j - \mathbf{d}_j \left[\mathbf{O}^{(z-1)}_{i,.} \; 1\right]$
        end
        if z>2
            $\mathbf{d} := \mathbf{d}(\mathbf{W}^{(z-1)})^T$
            $\mathbf{d} := \mathbf{d} \times \mathbf{f}'(\mathbf{Net}^{(z-1)}_{i,.})$
        end
        z := z-1
    end
end

4.2.2 Limitations of the error back-propagation algorithm

It was shown above that each iteration of the error back-propagation algorithm is computationally efficient. However, its performance with regard to the total process of training is very poor. It is common to find in the specialized literature that hundreds of thousands of iterations are needed to train MLPs. This fact is related to the underlying minimization technique that this algorithm implements, the steepest descent method.

To show the problems associated with this method, an example will be employed.

Example 1 - x^2 mapping

The aim of this example is to train one MLP to approximate the mapping $y = x^2$ in a specified range of x. The error back-propagation algorithm, with different learning rates, will be employed. The simplest possible MLP will be used, namely a perceptron with a linear output function, as shown in Fig. 4.21. The same starting point ($\mathbf{w} = \left[0.9 \; 0.9\right]^T$) will be used for all the examples.

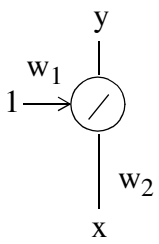


Fig. 4.21 - Simplest possible MLP

The performance of the algorithm will be evaluated using four different graphs:

a) illustrates the evolution of **w**. Contour lines are superimposed on the graph, to illustrate the behaviour of the training criterion;

b) illustrates the evolution of the criterion, which is the usual sum of the square of the errors ($\|\mathbf{e}\|^2$);

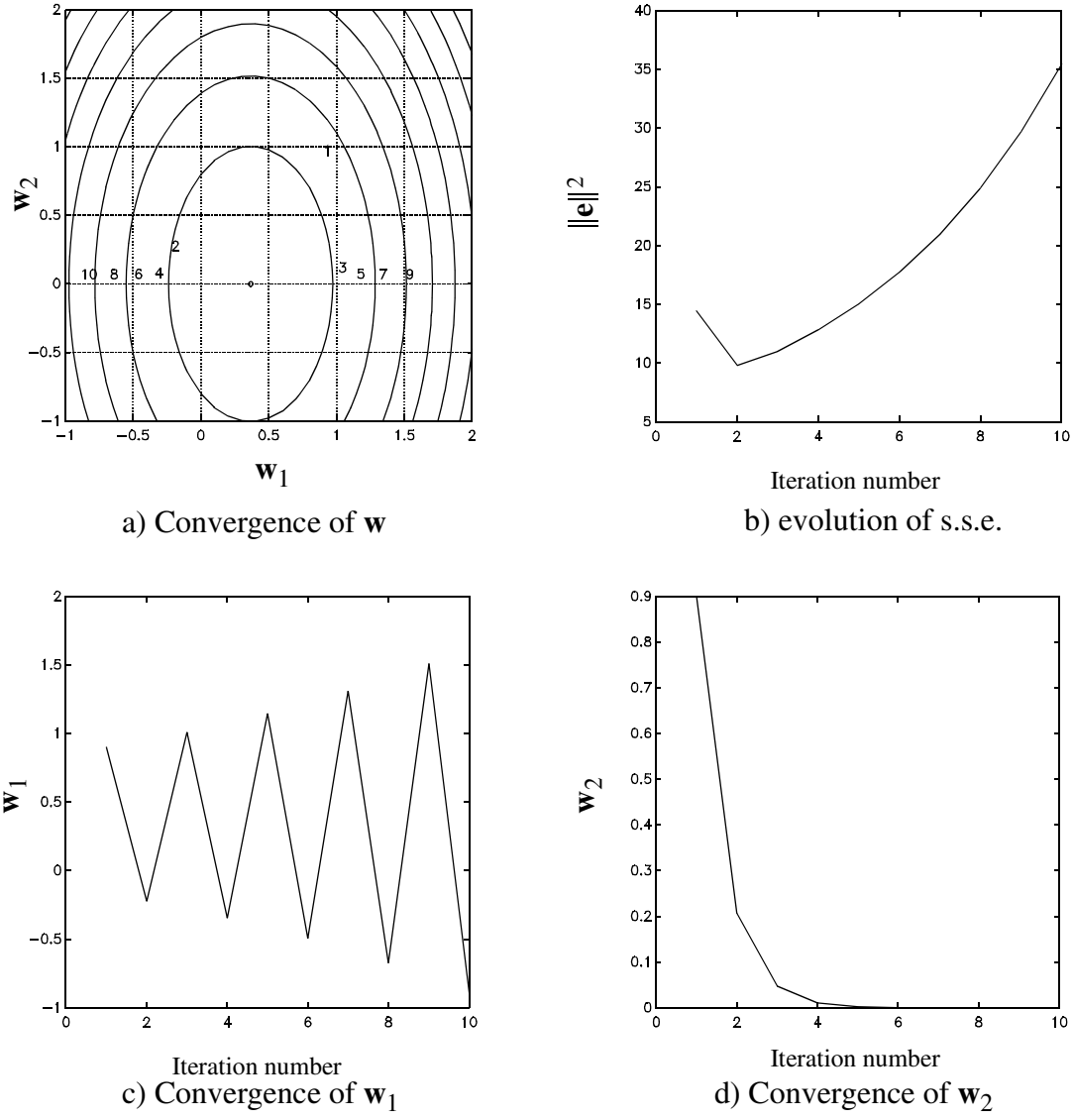
c) shows the convergence of the threshold ($\mathbf{w}_1$);

d) shows the convergence of the weight ($\mathbf{w}_2$).

Figures 4.2 to 4.4 illustrate the performance achieved by the algorithm for $x \in \left[-1\ \ 1\right]$‡, for the first ten iterations. The training range was discretized into 21 evenly spaced points, starting from x=-1, and separated by $\Delta\mathbf{x} = 0.1$.

As shown in Fig. 4.22, the training procedure diverges when a learning rate of 0.1 is used. Although $\mathbf{w}_2$ converges to its optimum value (0), the evolution of the bias is clearly an unstable process.

‡. It can be shown that, for the continuous mapping, $\hat{\mathbf{w}} = \left[\frac{1}{3}\ \ 0\right]^T$



a) Convergence of **w** b) evolution of s.s.e.

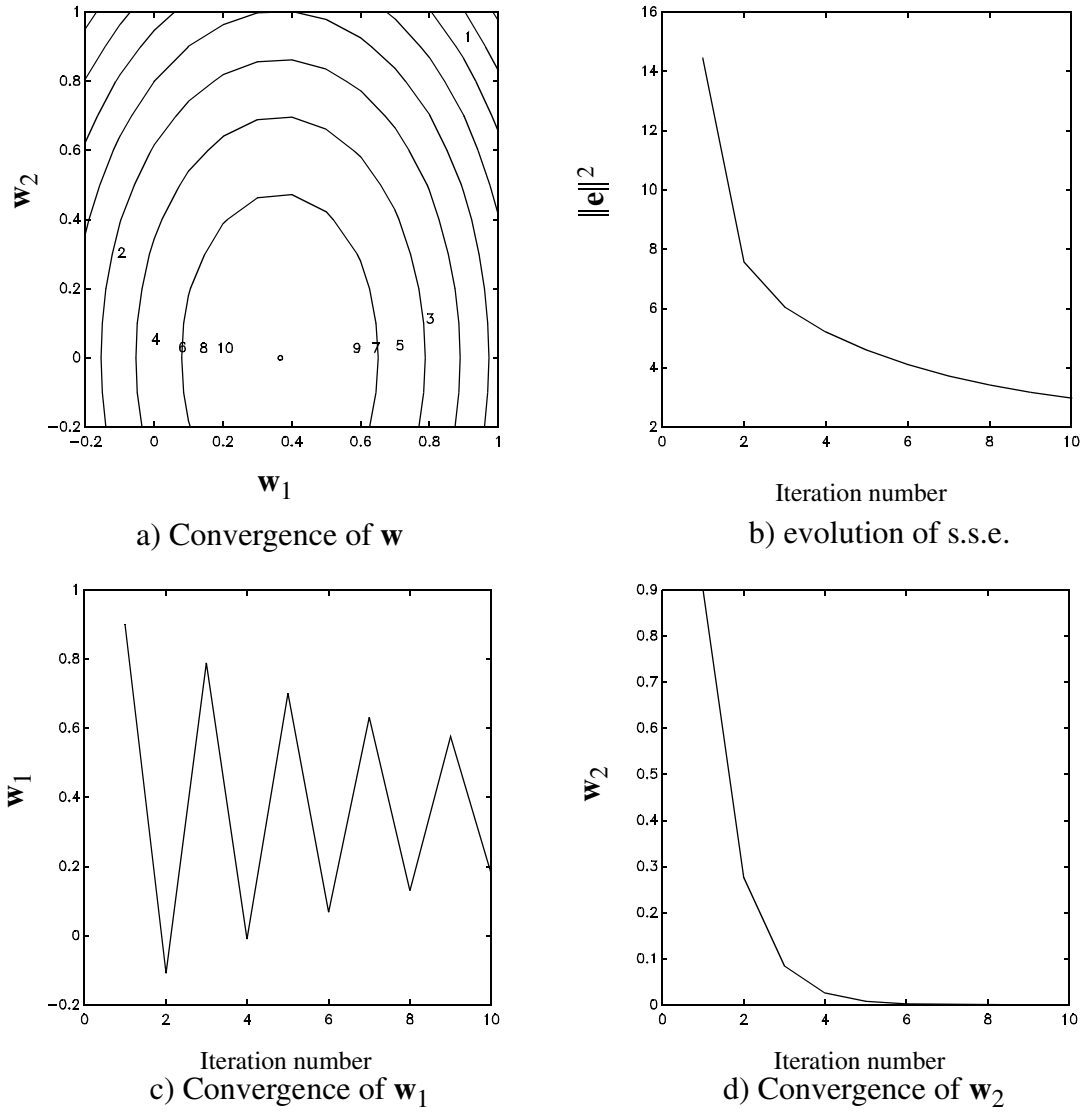
c) Convergence of $\mathbf{w}_1$ d) Convergence of $\mathbf{w}_2$

Fig. 4.22 - x^2 problem; $x \in [-1\ 1]$, $\Delta\mathbf{x} = 0.1$, $\eta = 0.1$

Fig. 4.23 illustrates the performance of the training procedure for the same problem, this time using a learning rate of 0.09.

With this learning rate the training procedure is now stable. The evolution of the bias is an oscillatory process, with slower convergence rate than the evolution of the weight. The convergence of this last parameter ($\mathbf{w}_2$) is now slightly slower than was obtained with $\eta = 0.1$.

.



a) Convergence of **w**

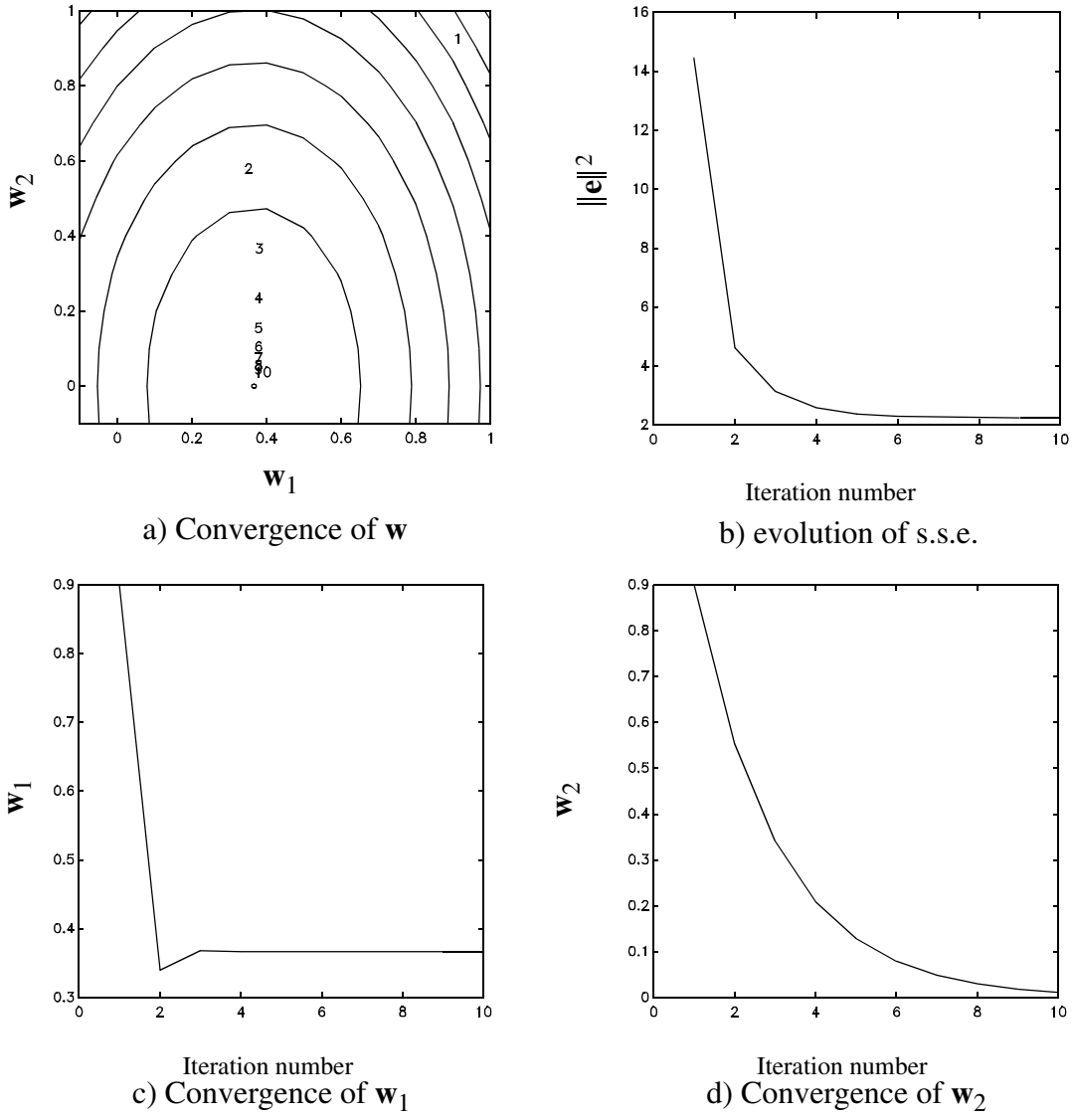
b) evolution of s.s.e.

c) Convergence of $\mathbf{w}_1$

d) Convergence of $\mathbf{w}_2$

Fig. 4.23 - x^2 problem; $x \in [-1 \ 1]$, $\Delta \mathbf{x} = 0.1$, $\eta = 0.09$

Fig 4.4 shows what happens when the learning rate is further reduced to 0.05. The training process is perfectly stable. Although the learning rate used is smaller than the one employed in the last example, the evolution of the training criteria is faster. The convergence rate for $\mathbf{w}_1$ is now better than for $\mathbf{w}_2$. This latter parameter converges slower than with $\eta = 0.09$ and clearly dominates the convergence rate of the overall training process.



a) Convergence of **w**

b) evolution of s.s.e.

c) Convergence of $\mathbf{w}_1$

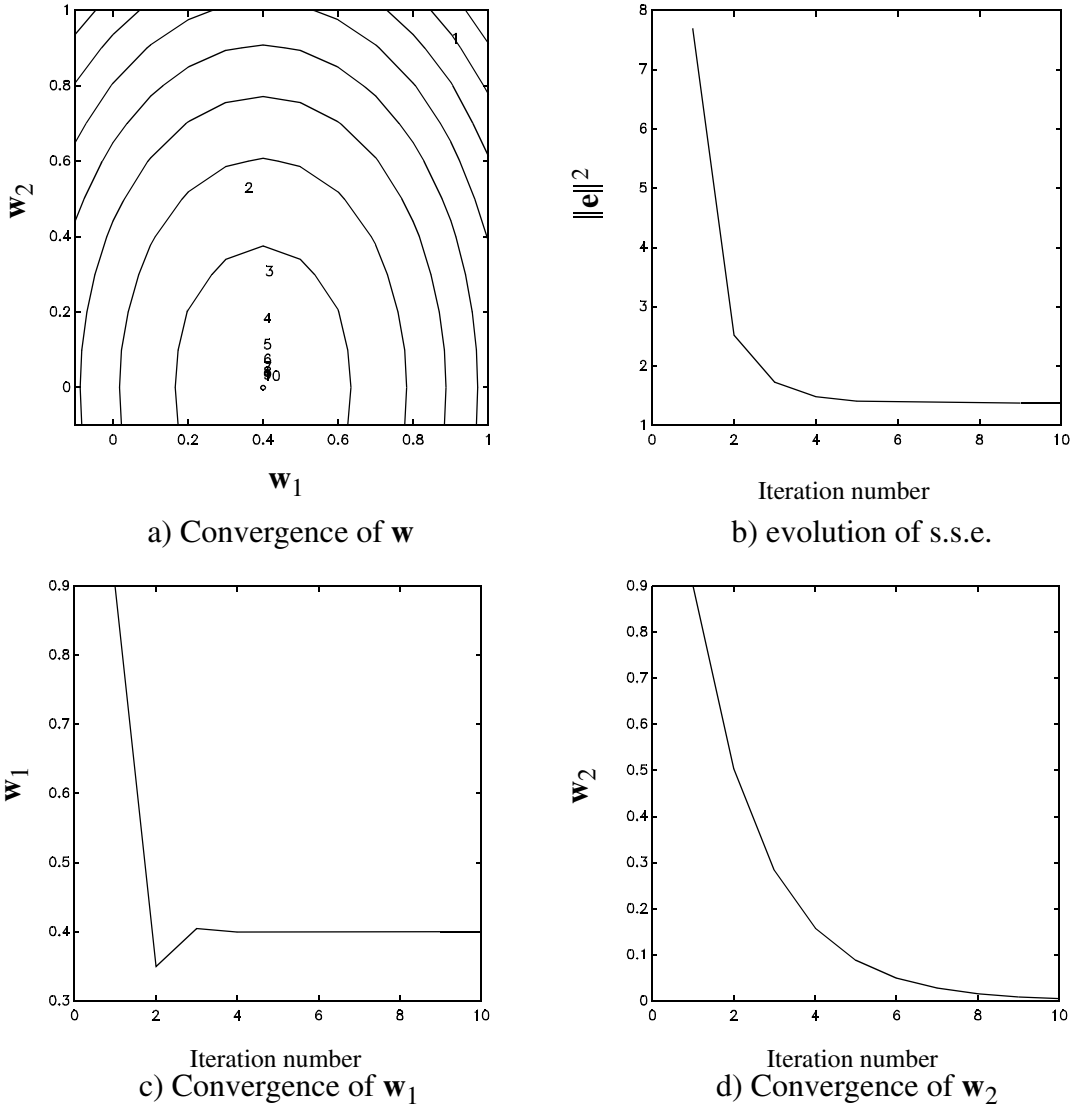
d) Convergence of $\mathbf{w}_2$

Fig. 4.24 - x^2 problem; $x \in [-1 \ 1]$, $\Delta\mathbf{x} = 0.1$, $\eta = 0.05$

As a final example, the same range of x will be used, this time with a separation between samples of $\Delta\mathbf{x} = 0.2$. Fig 4.5 illustrates the results obtained when $\eta = 0.1$ is used.

Although the learning rate employed is the same as in the example depicted in Fig. 4.22, the results presented in Fig 4.5 resemble more closely the ones shown in Fig 4.4.

.



a) Convergence of **w** b) evolution of s.s.e.

c) Convergence of $\mathbf{w}_1$ d) Convergence of $\mathbf{w}_2$

Fig. 4.25 - x^2 problem; $x \in [-1\ 1]$, $\Delta \mathbf{x} = 0.2$, $\eta = 0.1$

From the four graphs shown, two conclusions can be drawn:

a) - the error back-propagation algorithm is not a reliable algorithm; the training procedure can diverge;

b) the convergence rate obtained depends on the learning rate used and on the characteristics of the training data.

This example was deliberately chosen as it is amenable for an analytical solution. Since the output layer has a linear function, the output of the network is:

$$\mathbf{o}^{(q)} = [\mathbf{x}\ 1]\,\mathbf{w}, \tag{4.16}$$

or in the more general case:

$$\mathbf{o}^{(q)} = \mathbf{A}\mathbf{w} \tag{4.17}$$

where **A** denotes the input matrix, augmented by a column vector of ones to account for the threshold.

In this special (linear) case, the training criteria, given by (4.1), becomes:

$$\Omega^{l} = \frac{\|\mathbf{t} - \mathbf{A}\mathbf{w}\|^2}{2} \tag{4.18}$$

where the delimiters $\| \ \|$ denote the 2-norm.

Deriving (4.18) w.r.t. **w**, the gradient can then be expressed as:

$$\mathbf{g}^{l} = -\mathbf{A}^{T}\mathbf{t} + (\mathbf{A}^{T}\mathbf{A})\mathbf{w} \tag{4.19}$$

The minimal solution, or, as it is usually called, the least-squares solution of (4.18) is then obtained by first equating (4.19) to **0**, giving then rise to the *normal equations* [144]:

$$(\mathbf{A}^{T}\mathbf{A})\mathbf{w} = \mathbf{A}^{T}\mathbf{t} \tag{4.20}$$

which, when **A** is not rank-deficient, has a unique solution, given by (4.21):

$$\hat{\mathbf{w}} = (\mathbf{A}^{T}\mathbf{A})^{-1}\mathbf{A}^{T}\mathbf{t} \tag{4.21}$$

where it is assumed that **A** has dimensions m*n, $m \geq n$‡.

In the case where **A** is rank-deficient, there is an infinite number of solutions. The one with minimum 2-norm can be given by:

$$\hat{\mathbf{w}} = \mathbf{A}^{+}\mathbf{t} \tag{4.22}$$

where $\mathbf{A}^{+}$ denotes the pseudo-inverse of **A** [135].

If (4.19) is substituted into the back-propagation update equation (4.2), we obtain:

$$\begin{aligned} \mathbf{w}[k+1] &= \mathbf{w}[k] + \eta(\mathbf{A}^{T}\mathbf{t} - (\mathbf{A}^{T}\mathbf{A})\mathbf{w}[k]) = \\ &(\mathbf{I} - \eta(\mathbf{A}^{T}\mathbf{A}))\mathbf{w}[k] + \eta\mathbf{A}^{T}\mathbf{t} \end{aligned} \tag{4.23}$$

This last equation explicitly describes the training procedure as a discrete MIMO system, which can be represented as [145]:

‡. Throughout this thesis, it will always be assumed that the number of presentations is always greater than or equal to the number of parameters.

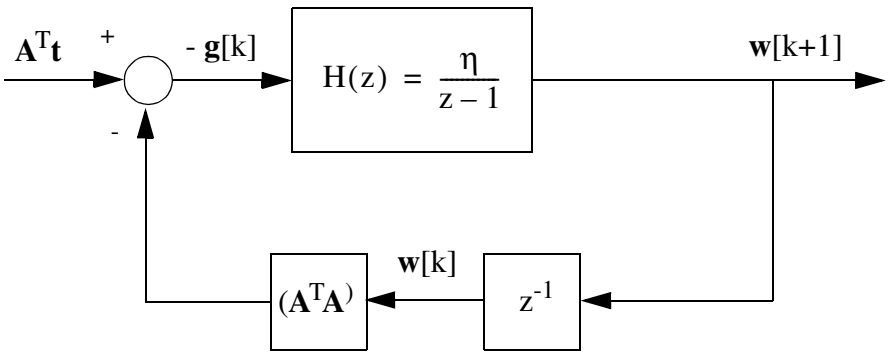


Fig. 4.26 - Closed-loop representation of (4.23)

The rate of convergence and the stability of the closed loop system are governed by the choice of the learning rate parameter. To examine them, it is then convenient to decouple the $\mathbf{k}_1+1$ simultaneous linear difference equations of (4.23) by performing a linear transformation of **w**.

Since $\mathbf{A}^T\mathbf{A}$ is an symmetric, it can be decomposed [146] as:

$$\mathbf{A}^T\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{U}^T \tag{4.24}$$

where **U** is an orthonormal square matrix and **S** is a diagonal matrix where the diagonal elements are the eigenvalues (λ) of $\mathbf{A}^T\mathbf{A}$.

Replacing (4.24) in (4.23), its j^{th} equation becomes:

$$\mathbf{v}_j[k+1] = (1-\eta\lambda_j)\mathbf{v}_j[k] + \eta\mathbf{z}_j \qquad j = 1, \ldots, \mathbf{k}_1 + 1 \tag{4.25}$$

where

$$\mathbf{v} = \mathbf{U}^T\mathbf{w} \tag{4.26}$$

and

$$\mathbf{z} = \mathbf{U}^T\mathbf{A}^T\mathbf{t} \tag{4.27}$$

It is clear that (4.25) converges exponentially to its steady-state value if the following condition is met:

$$\left|1-\eta\lambda_j\right| < 1 \tag{4.28}$$

which means that the maximum admissible learning rate is governed by:

$$\eta < \frac{2}{\lambda_{max}} \tag{4.29}$$

where λ_{max} denotes the maximum eigenvalue of $\mathbf{A}^T\mathbf{A}$.

From (4.25) it can also be concluded that the convergence rate for the j^{th} weight is related to $|1 - \eta\lambda_j|$, faster convergence rates being obtained as the modulus becomes smaller. The desirable condition of having fast convergence rates for all the variables cannot be obtained when there is a large difference between the largest and the smallest of the eigenvalues of $\mathbf{A}^T\mathbf{A}$. To see this, suppose that η is chosen close to half of the maximum possible learning rate. In this condition, the variable associated with the maximum eigenvalue achieves its optimum value in the second iteration. If m denotes the equation related with the smallest eigenvalue of $\mathbf{A}^T\mathbf{A}$, this equation may be represented as:

$$\mathbf{v}_m[k+1] = \left(1 - \frac{\lambda_{min}}{\lambda_{max}}\right)\mathbf{v}_m[k] + \frac{\mathbf{z}_m}{\lambda_{max}} \tag{4.30}$$

which has a very slow convergence if $\lambda_{min} \ll \lambda_{max}$

It is then clear that the convergence rate ultimately depends on the condition number of $\mathbf{A}^T\mathbf{A}$, defined as:

$$\kappa(\mathbf{A}^T\mathbf{A}) = \frac{\lambda_{max}}{\lambda_{min}} \tag{4.31}$$

Referring back to Example 1, $\mathbf{A}^T\mathbf{A}$ is a diagonal matrix with eigenvalues {21,7.7}, for the cases where $\Delta\mathbf{x} = 0.1$ and {11,4.4}, when $\Delta\mathbf{x} = 0.2$. Table 4.1 illustrates the values of $(1 - \eta\lambda_i)$, for the cases shown in Figs. 4.2 to 4.5.

	Fig 4.2	Fig 4.3	Fig 4.4	Fig 4.5
$\mathbf{w}_1$	-1.1	-0.89	-0.05	-0.1
$\mathbf{w}_2$	0.23	0.31	0.62	0.56

Table 4.1 - Values of $(1 - \eta\lambda_i)$ for Figs. 4.2 to 4.5

The evolution of **w** completely agrees with the values shown in the Table.

The above considerations, for the case of a linear perceptron, can be extrapolated for the case of interest, the nonlinear multilayer perceptron. This will be done in the following section, where alternatives to back-propagation, which do not suffer from its main limitations, namely unreliability and slow convergence, will be introduced.

4.3 Alternatives to the error back-propagation algorithm

In last section it has been shown that error back-propagation is a computationally efficient algorithm, but, since it implements a steepest descent method, it is unreliable and can have a very slow rate of convergence. Also it has been shown that it is difficult to select appropriate values of the learning parameter.

In this section alternatives to error back-propagation are given, in the framework of unconstrained deterministic optimization. Other approaches could be taken. For instance, the emerging field of genetic algorithms [147] [148] offers potential for the training of neural networks [149]. This approach is, however, beyond the scope of this thesis.

The first limitation of the error back-propagation algorithm, the unreliability of the method, can be eliminated by incorporating a line search algorithm. For the purpose of the interpretation of the error back-propagation algorithm in the framework of unconstrained optimization, it is advisable to divide the update equation (4.2) into three different steps:

Algorithm 5 - Sub-division of (4.2)

... compute a search direction: $\mathbf{p}[k] = -\mathbf{g}[k]$
... compute a step length: $\alpha[k] = \eta$;
... update the weights of the MLP: $\mathbf{w}[k+1] = \mathbf{w}[k] + \alpha[k]\mathbf{p}[k]$

This algorithm has now the usual structure of a step-length unconstrained minimization procedure. The second step could then be modified, to compute, in each iteration, a step length $\alpha[k]$ such that (4.32) would be verified in every iteration.

$$\Omega(\mathbf{w}[k] + \alpha[k]\mathbf{p}[k]) < \Omega(\mathbf{w}[k]) \tag{4.32}$$

Hence, step 2 would implement what is usually called a line search algorithm. Several strategies could be used, namely search procedures (Fibonnaci, golden section) or polynomial methods (quadratic, cubic) involving interpolation or extrapolation [150]. The line search is denoted exact if it finds $\alpha[k]$, such that it minimizes (4.33), partial or inexact if it does not.

$$\Omega(\mathbf{w}[k] + \alpha[k]\mathbf{p}[k]) \tag{4.33}$$

The inclusion of a line search procedure in the back-propagation algorithm guarantees global convergence to a stationary point [151]. This solves the first limitation of the BP

algorithm.

However, even with this modification, the error back-propagation algorithm is still far from attractive. It can be shown [152] that, for the linear perceptron case, discussed in the last section, employing an exact line search algorithm to determine the step length, the rate of convergence is linear, and given by:

$$\frac{\Omega(\mathbf{w}[k+1]) - \Omega(\hat{\mathbf{w}})}{\Omega(\mathbf{w}[k]) - \Omega(\hat{\mathbf{w}})} \approx \frac{(\kappa(\mathbf{A}^T\mathbf{A}) - 1)^2}{(\kappa(\mathbf{A}^T\mathbf{A}) + 1)^2} \tag{4.34}$$

Even with a mild (in terms of MLPs) condition number of 1000, simple calculations show that the asymptotic error constant is 0.996, which means that the gain of accuracy, in each iteration, is very small.

For the case of nonlinear multilayer perceptrons, as will be shown afterwards, the convergence rate can be approximated by (4.34), with the matrix $\mathbf{A}$ replaced by $\mathbf{J}(\hat{\mathbf{w}})$, the Jacobian matrix at the solution point. In practice, large condition numbers appear in the process of training of MLPs (which seems to be a perfect example of an ill-conditioned problem), and are responsible for the very large numbers of iterations typically needed to converge, as reported in the literature.

To speed up the training phase of MLPs, it is clear that some other search directions $\mathbf{p}[k]$, other than the steepest descent, must be computed in step 1 of Algorithm 5. Some guidelines for this task can be taken from section 4.2.2. Equation (4.21) gives us a way to determine, in just one iteration, the optimum value of the weights of a linear perceptron. This contrasts with the use of the normal error back-propagation update (4.2), where several iterations (depending on the η used and $\kappa(\mathbf{A}^T\mathbf{A})$) are needed to find the minimum.

One way to express how the two different updates appear is to consider that two different approximations are used for Ω, and that (4.2) and (4.21) minimize these approximations. For the case of (4.2), a first-order approximation of Ω is assumed:

$$\Omega(\mathbf{w}[k] + \mathbf{p}[k]) \approx \Omega(\mathbf{w}[k]) + \mathbf{g}^T[k]\mathbf{p}[k] \tag{4.35}$$

It can be shown [14] that the normalized (Euclidean norm) $\mathbf{p}[k]$ that minimizes (4.35) is:

$$\mathbf{p}[k] = -\mathbf{g}[k] \ , \tag{4.36}$$

clearly the steepest-descent direction.

Equation (4.21) appears when a second order approximation is assumed for Ω:

$$\Omega(\mathbf{w}[k] + \mathbf{p}[k]) \approx \Omega(\mathbf{w}[k]) + \mathbf{g}^T[k]\mathbf{p}[k] + \frac{1}{2}\mathbf{p}^T[k]\mathbf{G}[k]\mathbf{p}[k] \quad (4.37)$$

The matrix $\mathbf{G}[k]$ denotes the matrix of the second order derivatives of Ω at the k^{th} iteration. This matrix is usually called the Hessian matrix of Ω.

Formulating the quadratic function in terms of $\mathbf{p}[k]$:

$$\Phi(\mathbf{p}[k]) = \mathbf{g}^T[k]\mathbf{p}[k] + \frac{1}{2}\mathbf{p}^T[k]\mathbf{G}[k]\mathbf{p}[k] \quad (4.38)$$

if $\mathbf{G}[k]$ is positive definite (all its eigenvalues are greater than 0) then the unique minimum of (4.38) is given by (4.39).

$$\hat{\mathbf{p}}[k] = -\mathbf{G}^{-1}[k]\mathbf{g}[k] \quad (4.39)$$

The Hessian matrix has a special structure when the function to minimize is, as in the case in consideration, a sum of the square of the errors. For the nonlinear case, the Hessian can be expressed [151] as:

$$\mathbf{G}[k] = \mathbf{J}^T[k]\mathbf{J}[k] - \mathbf{Q}[k] \quad (4.40)$$

where $\mathbf{Q}[k]$ is:

$$\mathbf{Q}[k] = \sum_{i=1}^{m} \mathbf{e}_i[k]\mathbf{G}_i[k] \quad (4.41)$$

$\mathbf{G}_i[k]$ denoting the second matrix of the second-order derivatives of $\mathbf{o}^{(q)}_i$ at iteration k.

Thus, in nonlinear least-square problems, the Hessian matrix is composed of first and second order derivatives. For the linear case, $\mathbf{G}_i[k]$, $i = 1, \ldots, m$ is $\mathbf{0}$, so that (4.39) becomes (4.21).

As in the linear case, a better rate of convergence is usually obtained using second order information. Second order methods usually rely in eq. (4.39) (or similar) to determine the search direction.

Depending on whether the actual Hessian matrix is used, or an approximation to it, and on the way that this approximation is performed, second-order methods are categorized into different classes. A Newton method implements (4.39) directly. It can be proved [151] that, provided $\mathbf{w}[k]$ is sufficiently close to $\hat{\mathbf{w}}$ and $\hat{\mathbf{G}}$ is positive definite, then the Newton method converges at a second-order rate. However, at points far from the solution, the quadratic model may be a poor approximation of Ω and $\mathbf{G}[k]$ may not be positive definite, which means that (4.39) may not possess a minimum, nor even a stationary point.

This serious limitation, together with the high computational costs of the method (the second order derivatives are needed, and **G** must be inverted) stimulated the development of alternatives to Newton's method, the Quasi-Newton methods. These will be introduced in section 4.3.1. These methods are general unconstrained optimization methods, and therefore do not make use of the special structure of nonlinear least square problems. Two other methods that exploit this structure are the Gauss-Newton and the Levenberg-Marquardt methods, which will be presented in sections 4.3.2 and 4.3.3 respectively.

4.3.1 Quasi-Newton method

This class of methods employs the observed behaviour of Ω and $\mathbf{g}$ to build up curvature information, in order to make an approximation of $\mathbf{G}$ (or of $\mathbf{H}=\mathbf{G}^{-1}$) using an appropriate updating technique. The update formulae used possess the property of hereditary positive definiteness, i.e., if $\mathbf{G}[k]$ is positive definite, so is $\mathbf{G}[k+1]$.

A large number of Hessian updating techniques have been proposed. The first important method was introduced by Davidon [153] in 1959 and later presented by Fletcher and Powell [154], being then known as the DFP method. Its update equation [151] is:

$$\mathbf{H}_{\mathrm{DFP}}[k+1] = \mathbf{H}[k] + \frac{\mathbf{s}[k]\mathbf{s}^{\mathrm{T}}[k]}{\mathbf{s}^{\mathrm{T}}[k]\mathbf{q}[k]} - \frac{\mathbf{H}[k]\mathbf{q}[k]\mathbf{q}^{\mathrm{T}}[k]\mathbf{H}[k]}{\mathbf{q}^{\mathrm{T}}[k]\mathbf{H}[k]\mathbf{q}[k]} \tag{4.42}$$

where:

$$\begin{aligned} \mathbf{s}[k] &= \mathbf{w}[k+1] - \mathbf{w}[k] \\ \mathbf{q}[k] &= \mathbf{g}[k+1] - \mathbf{g}[k] \end{aligned} \tag{4.43}$$

It is now usually recognised [150] that the formula of Broyden [155], Fletcher [156], Goldfarb [157] and Shanno [158] is the most effective for a general unconstrained method. The BFGS update [151] is given by:

$$\begin{aligned} \mathbf{H}_{\mathrm{BFGS}}[k+1] = \mathbf{H}[k] + \left(1 + \frac{\mathbf{q}^{\mathrm{T}}[k]\mathbf{H}[k]\mathbf{q}[k]}{\mathbf{s}^{\mathrm{T}}[k]\mathbf{q}[k]}\right)\frac{\mathbf{s}[k]\mathbf{s}^{\mathrm{T}}[k]}{\mathbf{s}^{\mathrm{T}}[k]\mathbf{q}[k]} - \\ \left(\frac{\mathbf{s}[k]\mathbf{q}^{\mathrm{T}}[k]\mathbf{H}[k] + \mathbf{H}[k]\mathbf{q}[k]\mathbf{s}^{\mathrm{T}}[k]}{\mathbf{s}^{\mathrm{T}}[k]\mathbf{q}[k]}\right) \end{aligned} \tag{4.44}$$

Traditionally these methods update the inverse of the Hessian to avoid the costly inverse operation. Nowadays this approach is questionable and, for instance, Gill and Murray [14] propose a Cholesky factorization of **G** and the iterative update of the factors.

In practice, the application of the BFGS version of a Quasi-Newton (QN) method always delivers better results than the BP algorithm in the training of MLPs. Two examples of

this are given in Fig. 4.28 and in Fig. 4.30, where the performance of these two algorithms, as well as the Gauss-Newton and the Levenberg-Marquardt methods is compared on common examples. Further comparisons with BP can be found in [142].

4.3.2 Gauss-Newton method

While the quasi-Newton methods are usually considered 'state of the art' in general unconstrained minimization, they do not exploit the special nature of the problem at hand, the nonlinear least-squares structure of the problem.

As noted previously, in this type of problem the gradient vector can be expressed, as in (4.5), by a product of the Jacobian matrix and the error vector, and the Hessian as a combination of a product of first derivatives (Jacobian) and second order derivatives. So, if first order information is available, in terms of the Jacobian matrix, one part of the Hessian matrix is exactly known.

The basis of the Gauss-Newton (GN) method lies in dropping the use of second order derivatives in the Hessian, so that (4.40) is approximated as:

$$\mathbf{G}[k] \approx \mathbf{J}^T[k]\mathbf{J}[k] \tag{4.45}$$

The reasoning behind this approximation lies in the belief that, at the optimum, the error will be small, so that $\hat{\mathbf{Q}}$, given by (4.41), is very small compared with $\hat{\mathbf{J}}^T\hat{\mathbf{J}}$. This assumption may not be (and usually is not) true at points far from the local minimum.

The validity of this approximation is more questionable in the training of MLPs, since, almost in every case, it is not known whether the particular MLP topology being used is the best for the function underlying the training data, or even whether the chosen topology is a suitable model for the underlying function in the range employed.

The search direction obtained with the Gauss-Newton method is then the solution of:

$$\mathbf{J}^T[k]\mathbf{J}[k]\mathbf{p}_{GN}[k] = -\mathbf{J}^T[k]\mathbf{e}[k] \tag{4.46}$$

As already mentioned, this system of equations always has a solution, which is unique if $\mathbf{J}$ is full column rank; an infinite number of solutions exist if $\mathbf{J}$ is rank-deficient. The problem of applying the Gauss-Newton method to the training of MLPs lies in the typical high degree of ill-conditioning of $\mathbf{J}$. This problem is exacerbated [151] if (4.46) is solved by first computing the product $\mathbf{J}^T\mathbf{J}$, and then inverting the resultant matrix, since

$$\kappa(\mathbf{J}^T\mathbf{J}) = (\kappa(\mathbf{J}))^2 \tag{4.47}$$

where $\kappa(\mathbf{J})$, as $\mathbf{J}$ is a rectangular matrix, is the ratio between the largest and the smallest

singular values of **J**. As will be mentioned in Chapter 5, QR or SVD factorizations of **J** should be used to compute (4.46).

Even using a QR factorization and taking some precautions to start the training with a small condition number of **J**, in most of the situations where this method is applied, the conditioning becomes worse as learning proceeds. A point is then reached where the search direction is almost orthogonal to the gradient direction, thus preventing any further progress by the line search routine.

The Gauss-Newton method is therefore not recommended for the training of MLPs, but fortunately, a better alternative exists, the Levenberg-Marquardt method.

4.3.3 Levenberg-Marquardt method

A method which has global convergence property [151], even when **J** is rank-deficient, and overcomes the problems arising when **Q**[k] is significant, is the Levenberg-Marquardt (LM) method.

This method uses a search direction which is the solution of the system (4.48):

$$(\mathbf{J}^{\mathrm{T}}[k]\mathbf{J}[k] + \upsilon[k]\mathbf{I})\mathbf{p}_{\mathrm{LM}}[k] = -\mathbf{J}^{\mathrm{T}}[k]\mathbf{e}[k] \tag{4.48}$$

where the scalar $\upsilon[k]$ controls both the magnitude and the direction of **p**[k]. When $\upsilon[k]$ is zero, **p**[k] is identical to the Gauss-Newton direction. As $\upsilon[k]$ tends to infinity, **p**[k] tends to a vector of zeros, and a steepest descent direction.

In contrast with the Gauss-Newton and quasi-Newton methods, which belong to a step-length class of methods (Algorithm 5), the LM method is of the 'trust-region' or 'restricted step' type. Basically this type of method attempts to define a neighbourhood where the quadratic function model agrees with the actual function in some sense. If there is good agreement, then the test point is accepted and becomes a new point in the optimization; otherwise it may be rejected and the neighbourhood is constricted. The radius of this neighbourhood is controlled by the parameter υ, usually denoted the *regularization factor.*

Levenberg [159] and Marquardt [160] were the first to suggest this type of method in the context of nonlinear least-squares optimization. Many different algorithms of this type have subsequently been suggested.

To introduce the algorithm actually employed [151], one way of envisaging the approximation of the Hessian employed in GN and LM methods is that these methods, at every k^{th} iteration, consider a linear model for generating the data:

$$\mathbf{o}^{(nl)}[k] = \mathbf{J}[k]\mathbf{w}[k] \tag{4.49}$$

Using (4.49), the predicted error vector, after taking a step $\mathbf{p}[k]$ is:

$$\mathbf{e}^p[k] = \mathbf{e}[k] - \mathbf{J}[k]\mathbf{p}[k] \quad , \tag{4.50}$$

so that the predicted reduction of Ω is:

$$\Delta\Omega^p[k] = \Omega(\mathbf{w}[k]) - \frac{(\mathbf{e}^p[k])^T(\mathbf{e}^p[k])}{2} \tag{4.51}$$

As the actual reduction is given by:

$$\Delta\Omega[k] = \Omega(\mathbf{w}[k]) - \Omega(\mathbf{w}[k] + \mathbf{p}[k]) \quad , \tag{4.52}$$

then the ratio r[k]

$$r[k] = \frac{\Delta\Omega[k]}{\Delta\Omega^p[k]} \tag{4.53}$$

measures the accuracy to which the quadratic function approximates the actual function, in the sense that the closer that r[k] is to unity, the better the agreement.

Then the LM algorithm can be stated, for iteration k:

Algorithm 6 - Levenberg-Marquardt algorithm

... obtain $\mathbf{J}[k]$ (Algorithm 1) and $\mathbf{e}[k]$ (using Algorithm 2)
... compute $\mathbf{p}[k]$ using (4.48) (if $(\mathbf{J}^T[k]\mathbf{J}[k] + \upsilon[k]\mathbf{I})$ is not positive definite then $(\upsilon[k] := 4\upsilon[k])$ and repeat)
... evaluate $\Omega(\mathbf{w}[k] + \mathbf{p}[k])$ and hence r[k]
if r[k]<0.25
$\upsilon[k+1] := 4\upsilon[k]$
elseif r[k]>0.75
$\upsilon[k+1] := \frac{\upsilon[k]}{2}$
else
$\upsilon[k+1] := \upsilon[k]$
end
if $r[k] \leq 0$
$\mathbf{w}[k+1] := \mathbf{w}[k]$
else
$\mathbf{w}[k+1] := \mathbf{w}[k] + \mathbf{p}[k]$
end

The algorithm is usually initiated with $\upsilon[1] = 1$ and is not sensitive to the change of the parameters 0.25, 0.75, etc. [151].

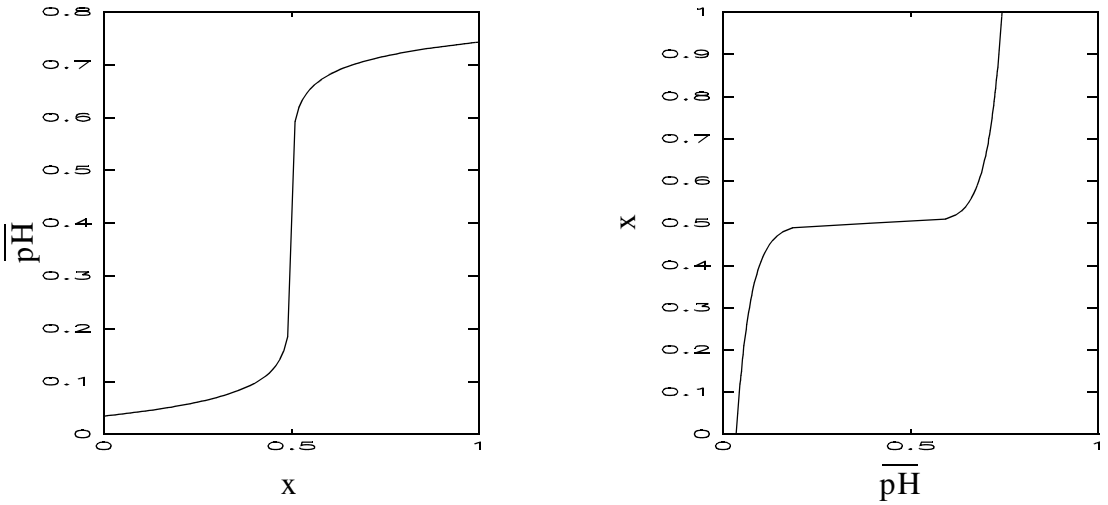
It is usually agreed that the Levenberg-Marquardt method is the best method for nonlinear least-squares problems [151]. This was confirmed in this study, in a large number of training examples performed.

To illustrate the performance of the different methods, two examples will now be presented.

Example 2 - Inverse of a titration curve [107]

The aim of this example is to approximate the inverse of a titration-like curve. This type of nonlinearity relates the pH (a measure of the activity of the hydrogen ions in a solution) with the concentration (x) of chemical substances. pH control is a difficult control problem, because the presence of this static nonlinearity causes large variations in the process dynamics [107]. An example of a normalized titration curve is shown in Fig. 4.27a. The equation used to generate the data for this graph was:

$$\overline{\text{pH}} = -\frac{\log\left(\sqrt{\frac{y^2}{4} + 10^{-14}} - \frac{y}{2}\right) + 6}{26}, \qquad y = 2*10^{-3}x - 10^{-3} \tag{4.54}$$



a) forward nonlinearity b) inverse nonlinearity

Fig. 4.27 - Titration-like curves

One strategy used in some pH control methods, to overcome the problems caused by the titration nonlinearity, is to use the concentration as output, rather than the pH, by linearizing the control [107]. Fig. 4.27b illustrates the inverse of the nonlinearity (4.54).

In this example a MLP with 1 input neuron, 2 hidden layers with 4 nonlinear neurons in each, and one linear neuron as output layer was trained to approximate the inverse of (4.54). 101 values ranging from -1 to 1, and with an even spacing of 0.01, were used as x and the

corresponding $\overline{\mathrm{pH}}$ values computed using (4.54). Then the x values were used as target output data and the $\overline{\mathrm{pH}}$ values as input training data.

Four different algorithms were used to train the MLP. To compare their performance, the error norm for the first 100 iterations of three of those algorithms is shown in Fig. 4.28.

The solid line denotes the performance of the error back-propagation, with $\eta = 0.005$. Two first trials of this method were performed with learning rate values of 0.05 and 0.01, but the algorithm diverged in both situations.

The dashed line denotes the performance of the 'fminu' function of the MATLAB [133] Optimization Toolbox [134]. This function implements a quasi-Newton method with BFGS update, with a mixed quadratic and cubic polynomial line search. Another function of this Toolbox, 'leastsq', was used for this example, employing a Gauss-Newton update. At the 5^{th} iteration, because of ill-conditioning of the Jacobian, this function changed automatically to a Levenberg-Marquardt update. For this reason, no results of the Gauss-Newton method can be presented.

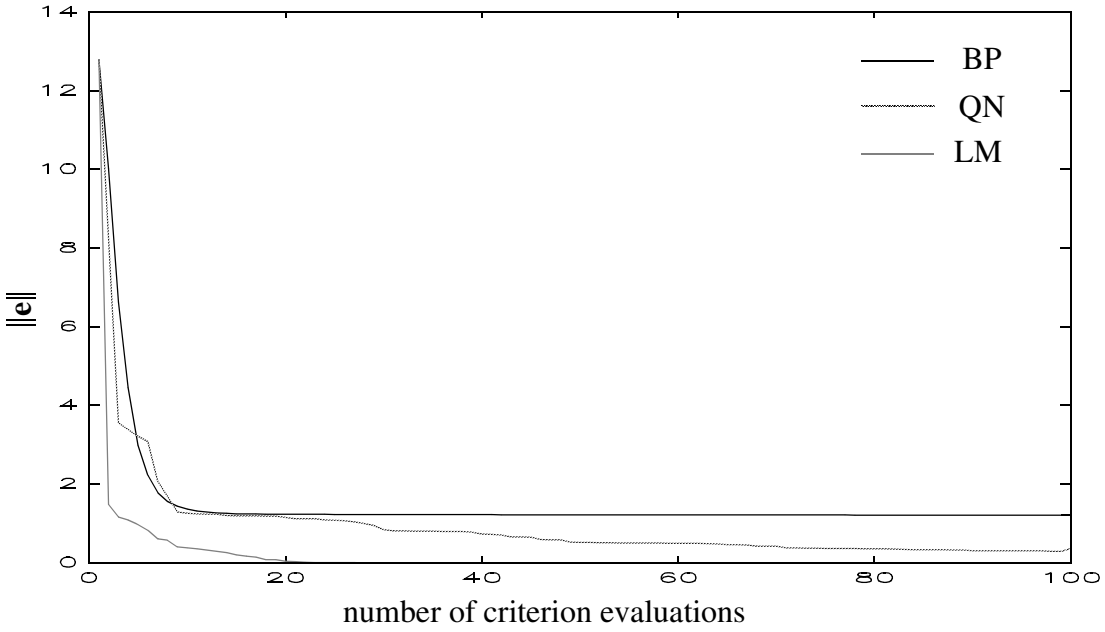


Fig. 4.28 - Convergence of training algorithms for Example 2

The dotted line illustrates the performance of the Levenberg-Marquardt algorithm (Algorithm 6).

An analysis of this example confirms the preferred 'ranking' of the methods for the training of MLPs, the best being the LM method, and the worst the BP algorithm.

It should be noted that in Fig. 4.28, and in later Figures illustrating convergence of QN and LM methods, 'spikes' in the data resulting from the line search algorithm, or from a poor choice of the regularization factor were removed. This was done to keep the range of the criterion values as small as possible, to illustrate the convergence properties of the methods. When spikes occur, a linear interpolation of the past and future values of the criterion is performed and shown in the graph.

Example 3 - Inverse coordinate transformation

This example illustrates an inverse kinematic transformation between Cartesian coordinates and one of the angles of a two link manipulator. Referring to Fig. 4.29, the angle of the second joint (θ_2) is related to the Cartesian coordinates of the end effector (x,y) by the expressions (4.55)

$$\begin{aligned} \theta_2 &= \pm \text{atan}\left(\frac{s}{c}\right) \\ c &= \frac{(x^2 + y^2 - l_1^2 - l_2^2)}{2 l_1 l_2} = \cos\theta_2 \\ s &= \sqrt{1 - c^2} = \sin\theta_2 \end{aligned} \tag{4.55}$$

where l_1 and l_2 represent the lengths of the arm segments..

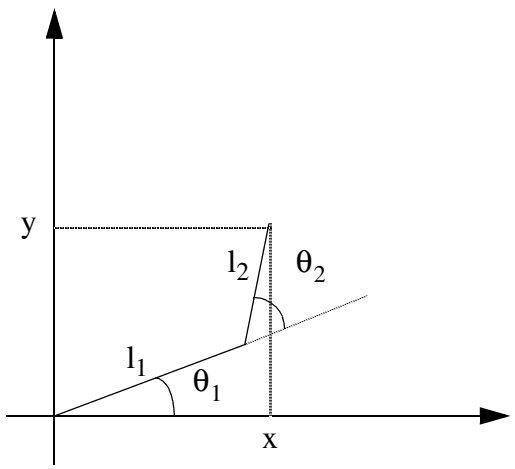


Fig. 4.29 - Two-links robot manipulator

The aim of this example is to approximate the mapping: $(x, y) \rightarrow +\theta_2$, in the first quadrant. The length of each segment of the arm is assumed to be 0.5m. To obtain the training data, 110 pairs of x and y variables were generated as the intersection of 10 arcs of circle centred in the origin (with radius ranging from 0.1m until 1.0m, with an even separation of

0.1m) with 11 radial lines (angles ranging from 0 to 90^0, with an even separation of 9^o). For each of the 110 examples, the corresponding θ_2 value was computed as the positive solution of (4.55) and used as the target output.

An MLP with 2 input neurons, one hidden layer with 5 nonlinear neurons and one linear neuron as output layer was employed to approximate this mapping.

Fig. 4.30 illustrates the performance of the error back-propagation algorithm, the BFGS version of a quasi-Newton method, and the Levenberg-Marquardt method applied to this problem.

The solid line denotes the performance of the error back-propagation algorithm with $\eta = 0.005$. As in Example 2, the use of learning parameter values of 0.05 and 0.01 caused the training procedure to diverge.

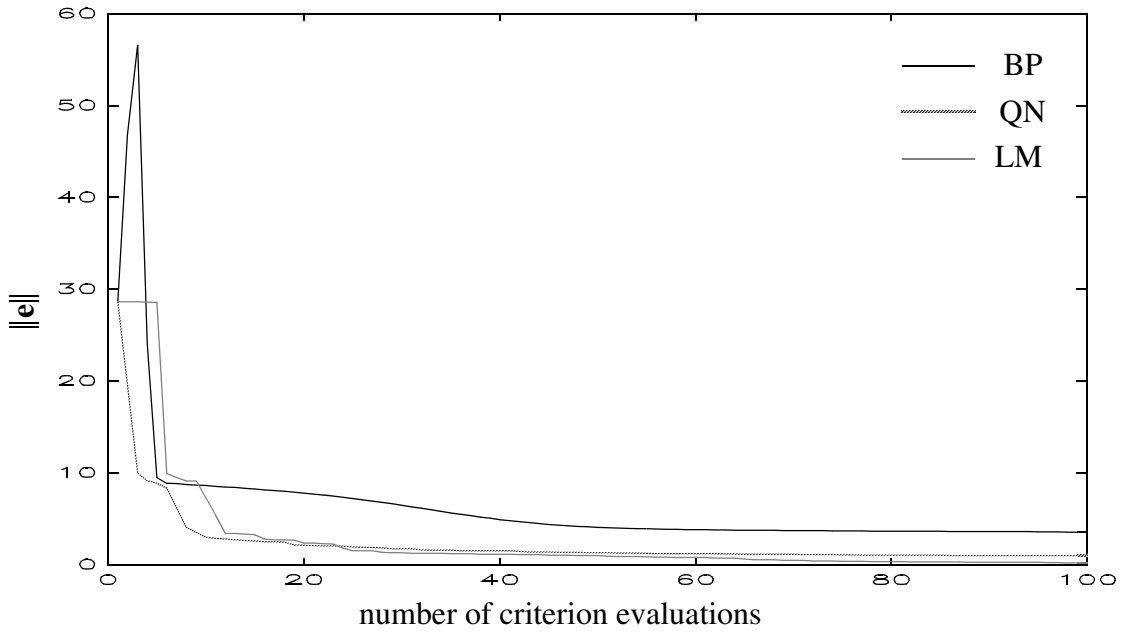


Fig. 4.30 - Convergence of training algorithms for Example 3

The dashed line illustrates the use of the 'fminu' function of the MATLAB optimization toolbox. It converges much more quickly than the error back-propagation algorithm, but again the Levenberg-Marquardt (dotted line) method achieves the best results.

The Gauss-Newton version of the 'leastsq' function of the MATLAB optimization toolbox was also employed in this problem, but it failed to achieve any progress, staying at an approximately constant value of 28.2 from iteration 10 till 100. This illustrates the fact that the approximation of the Hessian matrix by (4.45) is not reasonable in regions far from the optimum.

4.4 A new learning criterion

In the preceding section alternatives to the error back-propagation algorithm have been presented. It was shown that quasi-Newton algorithms have a more robust performance and achieve a faster rate of convergence than the error back-propagation algorithm. The LM method, which exploits the least-squares nature of the criterion emerges as the best method.

The convergence of the learning algorithms can be further improved if the topology of the MLPs is further exploited. In all the examples shown so far, and indeed, in most of the applications of MLPs to control systems, the networks act as function approximators. Consequently, the most important topology of this type of artificial neural network, as far as control systems applications are concerned, seems to be the one depicted in Fig. 4.31.

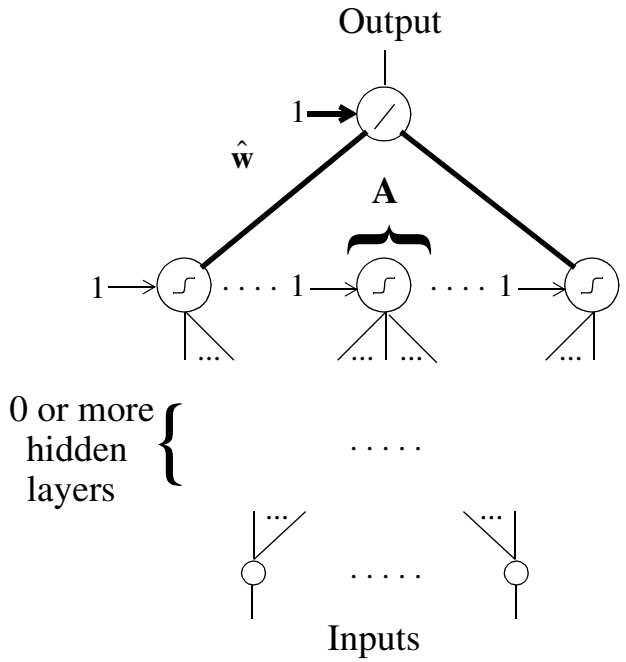


Fig. 4.31 - Most important topology of MLPs for control systems applications

The only difference between this topology and the standard one lies in the activation function of the output neuron, which is linear. This simple fact, as will be shown in the following treatment, can be exploited to decrease the number of iterations needed for convergence [136].

For the purposes of explanation, let us partition the weight vector as:

$$\mathbf{w} = \begin{bmatrix} \mathbf{w}^{(q-1)} \\ \begin{bmatrix} \mathbf{w}^{(q-2)} \\ \cdots \\ \mathbf{w}^{(1)} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} \tag{4.56}$$

where $\mathbf{u}$ denote the weights that connect to the output neuron, and $\mathbf{v}$ denotes all the other weights. Using this convention, the output vector can be given as:

$$\mathbf{o}^{(q)} = \left[\mathbf{O}^{(q-1)} \; 1\right]\mathbf{u} \tag{4.57}$$

When (4.57) is substituted in (4.1), the resulting criterion is:

$$\Omega = \frac{\left\|\mathbf{t} - \left[\mathbf{O}^{(q-1)} \; 1\right]\mathbf{u}\right\|_2^2}{2} \tag{4.58}$$

The dependence of this criterion on $\mathbf{v}$ is nonlinear and appears only through $\mathbf{O}^{(q-1)}$; on the other hand, Ω is linear in the variables $\mathbf{u}$, i.e. the weights in criterion (4.58) can be separated into two classes: nonlinear ($\mathbf{v}$) and linear ($\mathbf{u}$) weights. For any value of $\mathbf{v}$, the minimum of (4.58) w.r.t. $\mathbf{u}$ can be found using (4.22). The optimum value of the linear parameters is therefore conditional upon the value taken by the nonlinear variables.

$$\hat{\mathbf{u}}(\mathbf{v}) = \left(\left[\mathbf{O}^{(q-1)}(\mathbf{v}) \; 1\right]\right)^{+}\mathbf{t} \tag{4.59}$$

In (4.59), a pseudo-inverse is used for the sake of simplicity; however, it should be noticed that $\left[\mathbf{O}^{(q-1)} \; 1\right]$ is assumed to be full-column rank. Denoting this matrix by $\mathbf{A}$, for simplicity, the last equation can then be replaced into (4.58), creating therefore a new criterion:

$$\psi = \frac{\left\|\mathbf{t} - \mathbf{A}\mathbf{A}^{+}\mathbf{t}\right\|_2^2}{2} = \frac{\left\|\mathbf{P}_{\mathbf{A}_\perp}\mathbf{t}\right\|_2^2}{2}, \tag{4.60}$$

where the dependence of $\mathbf{A}$ on $\mathbf{v}$ has been omitted. In (4.60) $\mathbf{P}_{\mathbf{A}_\perp}$ is the orthogonal projection matrix to the complementary space spanned by the columns of $\mathbf{A}$.

This new criterion (4.60) depends only on the nonlinear weights, and although different from the standard criterion (4.58), their minima are the same. The proof of this statement can be found in [161]. We can therefore, instead of determining the optimum of

(4.58), first minimize (4.60), and then, using $\hat{\mathbf{v}}$ in eq. (4.59), obtain the complete optimal weight vector $\hat{\mathbf{w}}$.

Besides reducing the dimensionality of the problem, the main advantage of using this new criterion is a faster convergence of the training algorithm, a property which seems to be independent of the actual training method employed. This reduction in the number of iterations needed to find a local minimum may be explained by two factors:

- a better convergence rate is normally observed with this criterion, compared with the standard one;

- the initial value of the criterion (4.60) is usually much smaller than the initial value of (4.1), for the same initial value of the nonlinear parameters. This is because at each iteration, including the first, the value of the linear parameters is the optimal, conditioned by the value taken by the nonlinear parameters. As the criterion is very sensitive to the value of the linear parameters, a large reduction is obtained, in comparison with the situation where random numbers are used as starting values for these weights.

One problem that may occur using this formulation is a poor conditioning of the **A** matrix during the training process, which results in large values for the linear parameters. This happened in a very few out of a large quantity of examples where this new formulation was employed, and was resolved by taking certain precautions when obtaining the initial values of the nonlinear parameters.

In order to perform the training, the derivatives of (4.60) must be obtained. For this purpose the derivative of **A** w.r.t. **v** must be introduced. This is a three-dimensional quantity and will be denoted as:

$$(\mathbf{A})_{\mathbf{v}} = \frac{\partial \mathbf{A}}{\partial \mathbf{v}} \tag{4.61}$$

Considering first the use of the error back-propagation algorithm or the quasi-Newton with this new criterion, the gradient of ψ must be obtained.

It is proved in Appendix B that the gradient vector of ψ can be given as:

$$\begin{bmatrix} 0 \\ \mathbf{g}_{\psi} \end{bmatrix} = \mathbf{g}_{\Omega}\Big|_{\mathbf{u}=\hat{\mathbf{u}}} = -\begin{bmatrix} (\mathbf{o}^{(q-1)})^{\mathrm{T}} \\ 1 \\ \boldsymbol{J}^{\mathrm{T}} \end{bmatrix} \boldsymbol{e} \tag{4.62}$$

where $\mathbf{g}_{\Omega}\Big|_{\mathbf{u}=\hat{\mathbf{u}}}$, $\boldsymbol{J}$ and $\boldsymbol{e}$ denote respectively the gradient, the partition of the Jacobian matrix associated with the weights **v** and the error vector of Ω obtained when the values of linear

weights are their conditional optimal values, i.e.:

$$\boldsymbol{J} = (\mathbf{A})_{\mathbf{v}}\mathbf{A}^{+}\mathbf{t} \tag{4.63}$$

$$\boldsymbol{e} = \mathbf{P}_{\mathbf{A}_{\perp}}\mathbf{t} \tag{4.64}$$

The simplest way to compute $\mathbf{g}_{\psi}$ is expressed in Algorithm 7:

Algorithm 7 - Computation of $\mathbf{g}_{\psi}$

... compute $\mathbf{O}^{(q-1)}$ -- using Algorithm 2
... obtain $\hat{\mathbf{u}}$
... replace $\hat{\mathbf{u}}$ in the linear parameters, and complete Algorithm 2
... compute $\mathbf{g}_{\psi}$ -- using Algorithm 4.

The computation of $\mathbf{g}_{\psi}$ has the additional burden of determining $\hat{\mathbf{u}}$. However, since the dimensionality of the problem has been reduced, there is also some gain at each iteration that must be taken into account. Several algorithms can be used to compute these optimal values, but QR or SVD factorization are advisable, because of possible ill-conditioning of **A**.

Although there is an additional cost in complexity per iteration to pay using this approach, a large reduction in the number of iterations needed for convergence is obtained if (4.60) is used as the training criterion. To illustrate this, for the error back-propagation algorithm and the BFGS version of the quasi-Newton method, Example 2 will be used. To perform a fair comparison, the initial values of the nonlinear parameters were the ones used in Example 2. Fig. 4.32 illustrates the performance of the error back-propagation when criterion (4.60) is used as the learning criterion. The learning rate is set at 0.05. which is 10 times higher than the maximum allowed using (4.1). It should be noted that the initial norm of the error vector (0.53), in the reformulated case, is less than the value (1.2) obtained after 100 iterations of the standard criterion.

It was impossible to compare the convergence rate obtained using the two criteria with the BP algorithm. To force the same starting point for the standard criterion, the linear weights were initialized with their optimal value, in accordance with the initial value of the nonlinear parameters. With this starting point, even with very small values for η (as small as $5*10^{-4}$), the method diverged. This characteristic was found in several examples, and so it can be concluded that this technique cannot be employed to find a good starting point for the training, if criterion (4.1) is employed in conjunction with the error back-propagation algorithm.

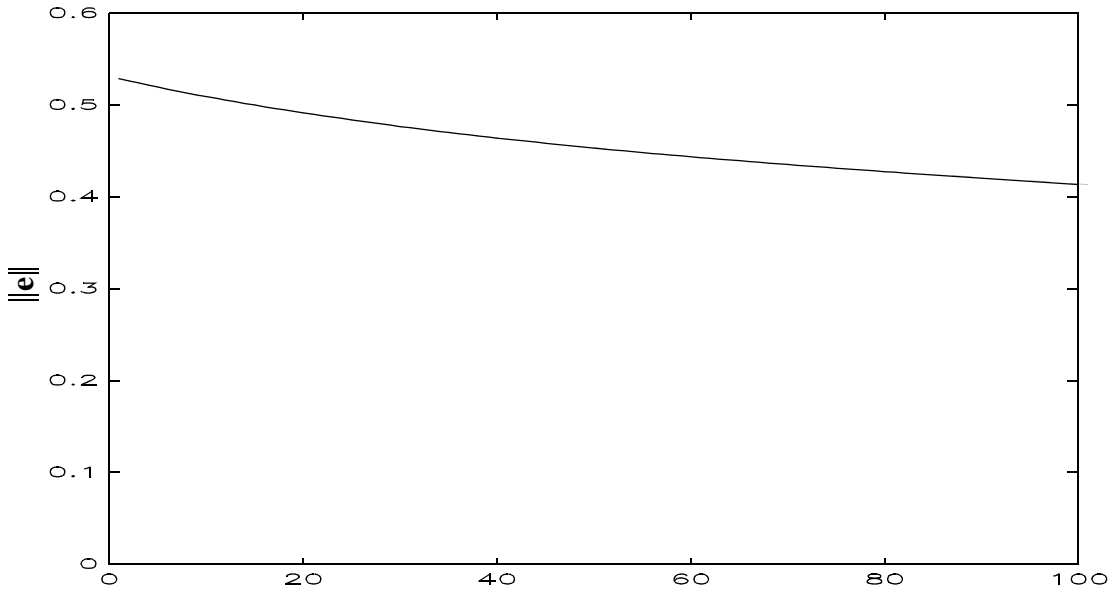


Fig. 4.32 - Convergence of BP (reformulated criterion) for Example 2

Fig. 4.33 illustrates the performance of the 'fminu' function of the MATLAB optimization toolbox. The solid line shows the convergence employing the standard criterion, where the linear weights were initialized with their conditional optimal values. The dashed line illustrates the convergence of the method with the new criterion. It can be observed that the convergence obtained with the new criterion is better than using the standard one.

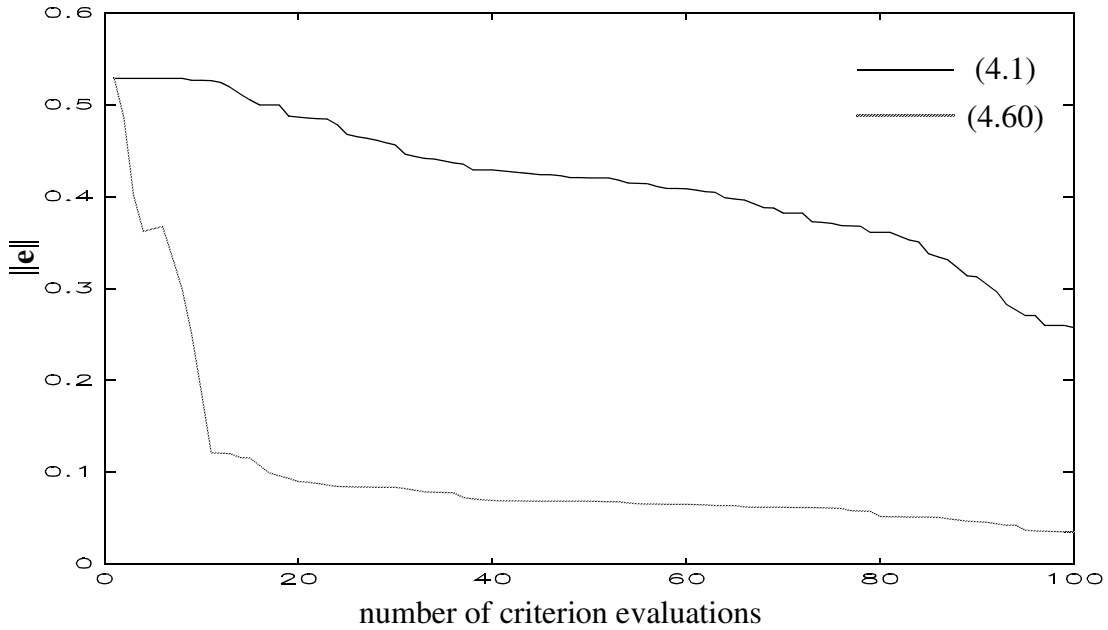


Fig. 4.33 - Convergence of QN methods for Example 2

Another example of the application of the new criterion in conjunction with the BP and the quasi-Newton methods can be found in [136] which deals with the training of one of the MLPs used in the PID compensator, discussed in Chapter 3.

If the Gauss-Newton or Levenberg-Marquardt methods are employed with this new formulation, then the Jacobian of (4.60) must be obtained. Three different Jacobian matrices have been proposed: the first was introduced by Golub and Pereyra [161], who were also the first to introduce the reformulated criterion. To reduce the computational complexity of Golub and Pereyra's approach, Kaufman [162] proposed a simpler Jacobian matrix. Recently, Ruano et al. [137] proposed another Jacobian matrix, which further reduces the computational complexity of each training iteration.

As the matrix $\boldsymbol{J}$ (4.63) will be employed in the computation of the three proposed Jacobian matrices, it is advisable to introduce now a procedure to obtain it. Following the same line of reasoning that led to Algorithm 7, the cheapest way to obtain $\boldsymbol{J}$ is using Algorithm 8:

Algorithm 8 - Computation of $\boldsymbol{J}$

... compute $\mathbf{O}^{(q-1)}$ -- using Algorithm 2
... obtain $\hat{\mathbf{u}}$
... replace $\hat{\mathbf{u}}$ in the linear parameters, and complete Algorithm 2
... compute $\boldsymbol{J}$ -- using Algorithm 1

As in the case of Algorithm 7, apart from the computation of $\hat{\mathbf{u}}$, the computation of $\boldsymbol{J}$ does not involve more costs than the computation of $\mathbf{J}$.

The three proposed Jacobian matrices will now be introduced, along with a discussion on the computational cost incurred in their computation. Theoretical considerations and mathematical proofs concerning the validity of those matrices will be deferred to Appendix B. The performance obtained by the Levenberg-Marquardt method, using the new formulation and employing the three Jacobian matrices, will then be compared between themselves and against the results obtained with the standard formulation.

4.4.1 Golub and Pereyra's approach

Golub and Pereyra [161] have proved that the Jacobian matrix of (4.60), which will be denoted here as $\mathbf{J}_{GP}$, can be given by:

$$\mathbf{J}_{GP} = \mathbf{P}_{\mathbf{A}_\perp}(\mathbf{A})_{\mathbf{v}}\mathbf{A}^{+}\mathbf{t} + \mathbf{A}^{+T}(\mathbf{A})_{\mathbf{v}}^{T}\mathbf{P}_{\mathbf{A}_\perp}\mathbf{t} \tag{4.65}$$

As will be shown, the computational complexity and the storage required to calculate (4.65) are both very large.

Considering first $(\mathbf{A})_{\mathbf{v}}$, if the number of nonlinear parameters is denoted by n_v, its dimensions are $(m*\mathbf{k}_{q-1}+1*n_v)$, which represents a requirement for a large amount of storage. This space, as well as the computational complexity to compute $(\mathbf{A})_{\mathbf{v}}$ and (4.65), can be reduced if the topology of the MLPs is taken into account. For that purpose, it can be observed that the last column of **A** (the one associated with the bias of the output neuron) is fixed and so does not depend on the nonlinear variables **v**. The second possible simplification is related to the fact that each matrix $(\mathbf{A})_{\mathbf{W}_{i,j}^{(q-2)}}$ has just one column different from **0**, since the weight $\mathbf{W}_{i,j}^{(q-2)}$ is connected to the j^{th} neuron of the $(q-1)^{th}$ layer, and so affects only this column of the **A** matrix.

The simplest way to incorporate these observations in the computation of $(\mathbf{A})_{\mathbf{v}}$ seems to be to consider primarily a second MLP, obtained from the original by just ignoring the output layer (and the weights connecting to it). This second MLP is then partitioned into $\mathbf{k}_{q-1}$ smaller MLPs, each one having just one output neuron corresponding to its partition number. Then Algorithm 1 can be employed to compute the Jacobian matrix for each partitioned MLP. The collection of these $\mathbf{k}_{q-1}$ Jacobian matrices constitute the non-zero elements of $(\mathbf{A})_{\mathbf{v}}$. In terms of computational complexity, the number of multiplications and additions reported for Algorithm 1 are increased by an order of $\mathbf{k}_{q-1}$.

Having obtained $(\mathbf{A})_{\mathbf{v}}$, in order to compute (4.65) in the cheapest possible way, it is beneficial, using simple algebraic manipulations, to express it as:

$$\mathbf{J}_{GP} = \mathbf{A}^{+T}((\mathbf{A})_{\mathbf{v}}^{T}\boldsymbol{e} - \mathbf{A}^{T}\boldsymbol{J}) + \boldsymbol{J} \tag{4.66}$$

Matrix $\boldsymbol{J}$ (and vector $\boldsymbol{e}$) can be obtained using Algorithm 8. To determine roughly the total computational cost of (4.66), the following terms must be added to the complexity of Algorithm 8 and the one involved in determining $(\mathbf{A})_{\mathbf{v}}$:

- $(\mathbf{A})_{\mathbf{v}}^{T}\boldsymbol{e}$: $n_v'*\mathbf{k}_{q-1}*m$ additions and multiplications, where n_v' is the number of parameters in each one of the $\mathbf{k}_{(q-1)}$ partitioned MLPs;

- $\mathbf{A}^{T}\boldsymbol{J}$: $n_v*\mathbf{k}_{q-1}*m$ additions and multiplications;

- $n_v*\mathbf{k}_{q-1}*m$ multiplications and additions for the rest of the operations.

The computational complexity of Golub and Pereyra's Jacobian matrix is therefore much higher than the one required to compute the Jacobian matrix of criterion (4.1); this constitutes a major disadvantage.

4.4.2 Kaufman's approach

To reduce the complexity incurred in the computation of Golub and Pereyra's Jacobian matrix, Kaufman [162] proposed the following Jacobian matrix ($\mathbf{J}_K$):

$$\mathbf{J}_K = \mathbf{P}_{\mathbf{A}_\perp} \boldsymbol{J} \tag{4.67}$$

The matrix $\boldsymbol{J}$ can be obtained using Algorithm 8. By expressing (4.67) as:

$$\mathbf{J}_K = \boldsymbol{J} - \mathbf{A}(\mathbf{A}^+ \boldsymbol{J}) \ , \tag{4.68}$$

if $\mathbf{A}^+$ is available, an additional cost of ($2*n_v*\mathbf{k}_{nl-1}*m$) additions and multiplications will be needed to compute (4.67) ‡.

4.4.3 A new approach

To reduce even further the complexity of computing a Jacobian matrix for (4.60), Ruano et al. [137] proposed the use of $\boldsymbol{J}$.

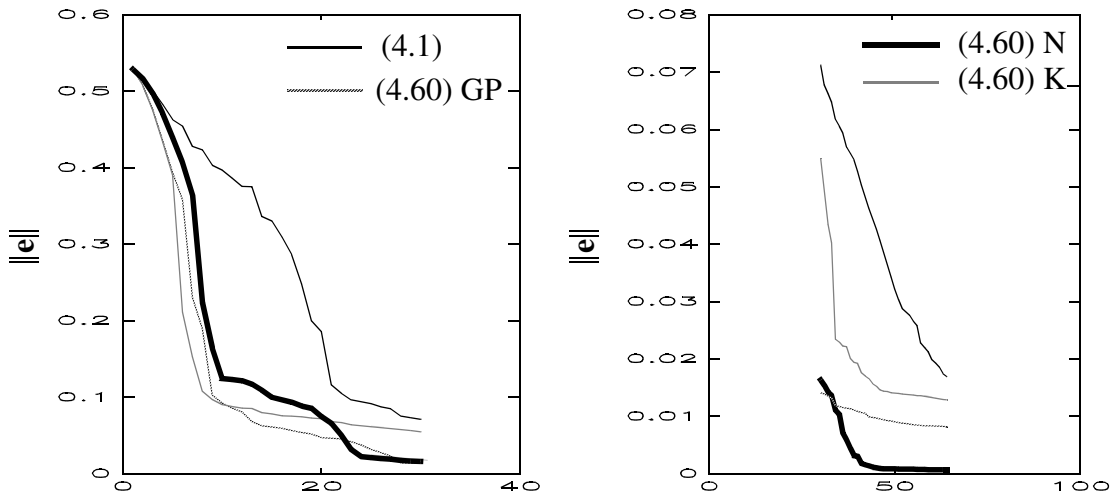
$$\mathbf{J}_N = \boldsymbol{J} \tag{4.69}$$

With this new approach, although the complexity involved in the computation of $\boldsymbol{J}$ exceeds that associated with $\mathbf{J}$ by the number of operations incurred in the calculation of $\hat{\mathbf{u}}$, the total complexity of the search direction procedure performed by a Gauss-Newton or a Levenberg-Marquardt method is actually reduced. The search direction in these methods is obtained by solving, in a least square sense, a linear system of equations. Although the complexity of this operation depends on the method applied, it is usually quadratic in the number of variables. By computing $\hat{\mathbf{u}}$, which has $\mathbf{k}_{q-1}+1$ variables, and afterwards obtaining the search direction $\mathbf{p}$ (n_v variables), we are actually solving two systems of equations whose total number of unknowns (n) equals the number of variables in the standard criterion. Because the complexity of the solution of the system of equations is quadratic in the number of unknowns, each iteration of the Gauss-Newton and Levenberg-Marquardt methods using the new formulation and the Jacobian matrix (4.69) is actually cheaper than an iteration of the same methods minimizing (4.1).

‡. Different implementations of the GN and LM methods using Golub-Pereyra and Kaufman's Jacobian matrices can be found in [163].

The three proposed Jacobian matrices have been introduced, and compared in terms of the complexity of their computation. The convergence rates obtained with the LM method (new criterion) employing Golub-Pereyra's and Kaufman's Jacobian matrices are usually similar, and normally faster than the one obtained with the standard criterion. The use of the new Jacobian matrix, in spite of having the least computational complexity, achieved, in the great majority of the examples tried, the fastest rate of convergence.

Fig. 4.34 compares the first 64 iterations of four LM algorithms (standard criterion, and reformulated criterion with the three Jacobians) using Example 2. To enable the use of two different scales for the criterion, the figure has been split into two graphs.



a) first 30 b) 31 to 64

Fig. 4.34 - Convergence of different LM methods for Example 2

The light solid line denotes the performance of the standard criterion. As in Fig. 4.33, the linear parameters were initialized with their optimal values. The dashed line reflects the performance of the new criterion with Golub and Pereyra's Jacobian matrix. The dotted line is related to the new criterion with Kaufman's Jacobian matrix. Finally, the dark solid line shows the behaviour of the new criterion with the new Jacobian matrix.

This last method converged after 64 iterations, which is the reason why this particular range was chosen for the x axis. The criterion for convergence, which was used for all the examples in this Chapter, is a logical AND of the conditions

$$\max\left|\mathbf{w}[k+1]-\mathbf{w}[k]\right| < 1*10^{-2} \tag{4.70}$$

$$\left|\mathbf{g}^{T}(\mathbf{w}[k+1]-\mathbf{w}[k])\right| < 1*10^{-2} \tag{4.71}$$

$$\max|\mathbf{g}| < 1*10^{-3} \tag{4.72}$$

It can be observed that the Golub-Pereyra and Kaufman versions of the LM method achieve similar rates of convergence, better than the standard LM method. The use of the new Jacobian, however, undoubtedly achieves the best results. Further comparisons between the four different LM algorithms can be found in [136] and [137].

4.5 Conclusions

In this Chapter the problem of training multilayer perceptrons was discussed. It was shown that the error back-propagation algorithm, the method most used for training this type of artificial neural networks, is an efficient algorithm, in terms of the computational cost per iteration. However, it has severe drawbacks when the overall training process is considered, namely:

- it is not reliable;

- a good value of the learning parameter is difficult to find;

- and has a very poor convergence.

One standard unconstrained minimization method, the quasi-Newton method, and two nonlinear least-squares methods, the Gauss-Newton and Levenberg-Marquardt methods, have been proposed as alternatives to the BP algorithm. It was found that the LM method achieved the best rate of convergence. Although these alternative training methods require more computations than the BP algorithm, for small size MLPs, the use of the quasi-Newton approach, and especially the LM method, reduces significantly the time required to find a local minimum.

Next, for the class of multilayer perceptrons most employed in control systems application, the linearity of the output layer was exploited. It was shown that a new training criterion can be formulated, which significantly reduces the time taken for the training. When considering the use of nonlinear least squares methods with this reformulated criterion, three different Jacobian matrices (Golub-Pereyra's, Kaufman's and a new formulation) have been described. It was then shown that the convergence rate obtained with LM methods, incorporating the first two Jacobian matrices is very similar, faster convergence being obtained with the newly proposed Jacobian. Since this matrix involves the least computation (in fact each iteration of this version of a LM method is cheaper than each iteration of the standard method) it can be concluded that the LM method, minimizing the new criterion, and using our proposed Jacobian matrix must be chosen as the best training algorithm, amongst all those considered.

Over the last two years, progress towards a more efficient training method, together

with an improvement of processing power, has led to a substantial practical reduction in the training time of MLPs. Taking, for example, the training of the MLPs used in the PID compensator which is considered in Chapter 3, a reduction factor of more than one hundred was obtained. Standard LM methods were the first to be applied to this particular example, this computation being implemented in MATLAB, running on a Sun 3/60 workstation. Under these conditions, approximately three days of background processing were employed in the training of each one of the MLPs. Nowadays, employing a LM reformulated method and the new Jacobian matrix, implemented in MATLAB on a Sun-SLC Sparc workstation, the training times are reduced to roughly half an hour. Although this considerable reduction in the training time is partly due to the improvement in the performance of both machines, even considering a factor of 10 for this technology factor, another factor of 10 (to be conservative) is really due to improvement of the algorithm in itself.

To enable further reductions in the training time, the chosen training algorithm was implemented in the Occam language, successfully parallelized and executed on an array of Inmos transputers. This is the subject of the next Chapter.

Chapter 5

Parallelization of the Learning Algorithm

5.1 Introduction

In the last Chapter several different training algorithms were compared in terms of reliability, convergence rate and execution time. A Levenberg-Marquardt method, minimizing a new learning criterion, together with a new Jacobian matrix, was the chosen algorithm. In spite of drastically reducing the execution time needed for the training of multilayer perceptrons, this problem still remains a highly computationally intensive task. It is therefore a suitable candidate for parallelization, which is the subject of this Chapter.

Several authors have applied parallel processing techniques to the same problem. Paugam-Moisy [164] has implemented a 'spy' device, using Inmos transputers, to select, amongst different candidate topologies and parameter algorithms, the best topology and set of parameters for a given application. Afterwards, she extended the 'spy' concept to accelerate the recall and learning phases of MLPs [165]. The learning algorithm used is a modified version of the error back-propagation algorithm. The parallelization of this algorithm, whether in its original version or slightly modified, has also been proposed by Singer [166], Zhang and co-workers [167], Petrowsky and co-workers [168], to mention just some of the authors in this specialized field.

Although some valid insight can be obtained from the work described above, the proposed learning algorithm has a structure which is different to the error back-propagation algorithm. One of the steps, the recall operation, is common to the error back-propagation algorithm. The computation of the Jacobian is also very similar, in the operations involved, to the calculation of the gradient vector as performed in the error back-propagation algorithm. However, the new learning algorithm involves the solution of two least-squares problems, one of them clearly being the most computationally intensive task. For this reason considerable effort was put into the parallelization of this type of problem.

This Chapter is organized as follows:

The learning algorithm is coded in Occam 2 [16] and executed on Inmos transputers [15]. Next section gives the minimal background needed, on these two subjects, for the understanding of this Chapter. Metrics employed to quantify parallel performance are also introduced.

In section 5.3 the main blocks of computation are identified, and the possibility of implementing some of them in parallel is addressed.

Section 5.4 concerns the parallelization of a least squares solution of overdetermined systems of equations. As this is the most computationally intensive task of the learning algorithm, it is also the largest section of the Chapter. Sequential methods for solving this kind of problem will be compared, and one of them chosen for parallelization. Then different schemes of parallel implementation will be discussed. Finally the chosen parallel solution will be adapted for the two different least squares problems present in the learning algorithm

A recall operation must be performed at each iteration of the learning algorithm. Its parallelization is discussed in section 5.5.

The last large computational block of the learning algorithm is the calculation of the Jacobian matrix. The parallelization of this operation is described in section 5.6.

Having introduced parallelization schemes for the main computational blocks of the learning algorithm, they must be interconnected. This will be the subject of section 5.7.

Finally, in the last section of this Chapter a summary of the most important results throughout the Chapter is produced. Some points that deserve further considerations are highlighted. Some suggestions for further reducing the training time are also given.

5.2 Background

This Chapter deals with the parallelization of the new learning algorithm, introduced in Chapter 4. The parallel algorithm will be coded in the Occam language and executed in Inmos transputers. This section, rather that being an introduction to Occam and the transputer, which can be found, for instance, in [16][169][170][171][15], aims to provide the minimal background needed for the understanding of this Chapter. Measures of parallel performance are also introduced.

The transputer is a new generation VLSI architecture which explicitly supports concurrency and synchronisation. Of special interest to us, the IMS T800 (a member of the transputer family) comprises a 32-bit microprocessor, a 64-bit floating point unit, four high-speed serial communication links, a microcode scheduler, 4 Kbytes of on-chip memory and external memory interface, all in a single chip.

In addition to executing processes in conventional sequential mode, it may execute processes concurrently. Two levels of process priority are supported by the process scheduler. At the low priority level, processes are time-sliced in round-robin fashion. The high priority level has no time-slicing, high priority processes being executed in preference to low priority processes.

Each transputer communication link can transfer data at over 1MByte per sec., with automatic hand-shaking synchronization in each direction, and provides a bi-directional, point-to-point connection between transputers. A single transputer links implements two Occam channels, one in each direction. As the transputer has four links, each transputer can be connected with up to four transputers, allowing networks of various sizes and topologies to be built up. One important feature of the transputer, is the possibility of communicating and processing simultaneously. This feature is employed in the parallel implementations presented later.

The algorithms presented in this Chapter are essentially parallel algorithms. For this reason, additional pseudo-instructions, which reflect the parallelism, must be incorporated into these algorithms. As the parallel learning algorithm is coded in Occam, the additional pseudo-instructions will follow the Occam syntax.

Essentially, three Occam constructs, and two Occam instructions are used.

In Occam, synchronization and communication of processes is achieved using *channels*. Each channel is unidirectional and can only be used by one calling process and one called process. Communication is synchronous, which means that the calling process is delayed until the called process is ready to accept the message. Associated with channels, two operations are employed in the algorithms: the input operation and the output operation.

The input process has the syntax:

c ? v

and inputs a value from the channel **c**, assigning the value received to the variable **v**.

The output process is defined as:

c ! v

and has the meaning that the value of the variable **v** is output along a channel **c**.

The Occam constructions employed are:

. SEQ - sequential construction,

. PAR - parallel construction, and

. ALT - alternation construction.

The sequential construction causes the indented processes to be executed one after the other, as in conventional computers. The format of this construction is:

SEQ

process 1

.

.

process n

The parallel construction causes the indented processes to be executed concurrently. It has a format similar to the sequential construction:

PAR

process 1

.

.

process n

This construction terminates only after all of the component processes have terminated.

The alternation construct allows a particular process, from a list of component processes, to be selected for execution. For the cases presented here, it is sufficient to consider that each component process is guarded by an input process. The process associated with the first input guard to be ready is the one chosen for execution. This construction has the format:

ALT

input 1

process 1

.

.

input n

process n

The performance of the parallel algorithms described in this Chapter is evaluated using three common metrics: execution time, speed-up and efficiency.

Execution time (t) is defined as the effective elapsed time taken to run a particular job in a given machine.

Speed-up (s) is defined as the ratio of the elapsed time when executing a program in a single processor (t_1) to the execution time when p processors are used (t_p):

$$s = \frac{t_1}{t_p} \tag{5.1}$$

Efficiency (e) of a parallel program is defined as the ratio between the speed-up and the number (p) of processors employed:

$$e = \frac{s}{p} \tag{5.2}$$

5.3 First steps towards parallelization

It is advisable, before starting to parallelize the chosen learning algorithm, to express each iteration in the form expressed by Algorithm 9. It is assumed that the necessary initializations have already taken place.

Algorithm 9 - One iteration of the learning algorithm

```
... compute O^(q-1)                          -- first part of Algorithm 2
... compute û (4.59); w^(q-1) = û
... compute o^(q)                            -- complete Algorithm 2
e := t – o^(q)
... compute J                                -- Algorithm 1

... compute r (4.53) and update υ            -- see Algorithm 6
if r>0
    v_old := v; J_old := J; e_old := e
end

... compute p   (4.48)
v_old := v_old+p
... compute e^p (4.50)
```

To parallelize Algorithm 9, the possibility of executing some of its steps in parallel should be explored first. It can easily be seen that the computation of $\boldsymbol{J}$ can be executed concurrently with the computations of $\boldsymbol{e}$, r, and the update of υ. Apart from this possibility, all the other steps must be executed sequentially. However, within each step, there is considerable scope for parallelization, as will be explained during this Chapter.

Five main blocks of computation can be identified in Algorithm 9:

a) $\boldsymbol{o}$ - Algorithm 2;

b) $\hat{\mathbf{u}}$ - (4.59);

c) $\boldsymbol{J}$ - Algorithm 1;

d) $\mathbf{p}$ - (4.48);

e) $\boldsymbol{e}^p$ - (4.50).

Among these blocks, the one with highest complexity is undoubtedly the computation of the LM increment, **p**. A good overall efficiency for the parallel training algorithm can only be obtained if this block is efficiently parallelized. And as block (b) is similar to block (d), in the sense that both can be formulated as least squares solutions of linear systems, a good algorithm (in terms of parallelization efficiency) for this type of problem should be sought.

As block (e) has the lowest complexity and also is easily parallelized, our efforts will be concentrated on the parallelization of the recall operation, the Jacobian, and in particular, the least squares solution of linear system equations. For each of these blocks, the possibilities for parallelization will be discussed and a parallel algorithm will then be chosen. Finally, these algorithms will be integrated and, together with the parallelized versions of the remaining steps of Algorithm 9, a parallel version of the complete algorithm is obtained.

Because of its importance for the overall efficiency of the learning algorithm, the parallelization of the least squares solution of overdetermined systems of linear equations will be discussed first.

5.4 Least squares solution of overdetermined systems

This is a very important problem, which appears in a broad range of disciplines (for instance, control systems, optimization, statistics, signal processing, etc.). The basic problem is to derive a vector **x**, such that:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{5.3}$$

where the dimensions of A are m*n, $m \geq n$. Usually this system has no solution. A commonly used alternative is the minimization of:

$$\rho = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_p \tag{5.4}$$

for a suitable norm p.

As the use of the 2-norm makes (5.4) a continuous differentiable function of **x**, and therefore makes the problem easier to solve than employing the 1-norm or the ∞-norm, the usual least-squares formulation is the minimization of (5.5).

$$\rho = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 \tag{5.5}$$

In Chapter 4 it has already been mentioned that the use of a pseudo-inverse allows us to express the solution of this last equation in a compact form:

$$\mathbf{x} = \mathbf{A}^{+}\mathbf{b} \tag{5.6}$$

and, when $\mathbf{A}$ is full-column rank, $\mathbf{x}$ can be obtained as:

$$\mathbf{x} = (\mathbf{A}^{\mathrm{T}}\mathbf{A})^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{b} \tag{5.7}$$

Eq. (4.59) is already in the form of (5.6). The computation of $\mathbf{p}$ can also be expressed in this form if $\mathbf{J}$ and $\mathbf{e}$ in (4.48) are replaced by $\mathbf{J}$' and $\mathbf{e}$', where

$$\mathbf{J}' = \begin{bmatrix} \boldsymbol{J} \\ \sqrt{\upsilon}\mathbf{I} \end{bmatrix} \tag{5.8}$$

$$\mathbf{e}' = \begin{bmatrix} \boldsymbol{e} \\ 0 \end{bmatrix} \tag{5.9}$$

Using these modified equations, the least-squares solution of

$$\mathbf{J}'\mathbf{p} = -\mathbf{e}' \tag{5.10}$$

is, using (5.6) to (5.9):

$$\begin{aligned} \mathbf{p} = -\mathbf{J}'^{+}\mathbf{e}' = -(\mathbf{J}'^{\mathrm{T}}\mathbf{J}')^{-1}\mathbf{J}'^{\mathrm{T}}\mathbf{e}' &= -(\boldsymbol{J}^{\mathrm{T}}\boldsymbol{J} + \upsilon\mathbf{I})^{-1}\begin{bmatrix} \boldsymbol{J}^{\mathrm{T}} & \sqrt{\upsilon}\mathbf{I} \end{bmatrix}\begin{bmatrix} \boldsymbol{e} \\ 0 \end{bmatrix} \\ &= -(\boldsymbol{J}^{\mathrm{T}}\boldsymbol{J} + \upsilon\mathbf{I})^{-1}\boldsymbol{J}^{\mathrm{T}}\boldsymbol{e} \end{aligned} \tag{5.11}$$

Eq. (5.5) may be solved by means of (5.7) but, in fact, it is not the recommended solution for the current problem. It was already mentioned that $\mathbf{O}^{(q-1)}$ and $\boldsymbol{J}$ are ill-conditioned matrices. If (5.7) is used, this ill-conditioned problem is aggravated, since $\kappa(\mathbf{A}^{\mathrm{T}}\mathbf{A}) = (\kappa(\mathbf{A}))^2$ [135], and unnecessary magnification of round-off errors, which can even lead to numerical singularity of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$, occurs.

Fortunately there are methods that can be used to solve (5.5) that do not exacerbate the ill-conditioning of the problem. Two factorizations of the matrix $\mathbf{A}$ can be used: QR and singular value decomposition (SVD). The latter has an advantage when $\mathbf{A}$ is rank deficient. In this case, there is an infinite number of solutions for (5.5) and SVD is able to compute the one with the smallest norm, in contrast to QR. On the other hand, QR factorization requires fewer computations than SVD.

As it is assumed that $\left[\mathbf{O}^{(q-1)}\ \mathbf{1}\right]$ is always full-column rank, and the computation of $\mathbf{p}$ is guarded against the loss of full rank, QR decomposition will be used for the solution of least-squares problems.

As its name indicates, this factorization decomposes a matrix $\mathbf{A}$ into an orthonormal matrix ($\mathbf{Q}$) and an upper triangular matrix ($\mathbf{R}_1$), such that:

$$\mathbf{A} = \mathbf{QR} = \left[\mathbf{Q}_1\ \mathbf{Q}_2\right]\begin{bmatrix}\mathbf{R}_1\\ 0\end{bmatrix} \tag{5.12}$$

where $\mathbf{Q}$ is m*m, $\mathbf{R}$ is m*n, $\mathbf{Q}_1$ is m*n, $\mathbf{Q}_2$ is m*(m-n), $\mathbf{R}_1$ is n*n and $\mathbf{0}$ is (m-n)*(m-n).

To solve the least-squares problem, (5.12) can be replaced in (5.5), and taking advantage of the property of orthonormal matrices:

$$\mathbf{Q}^T\mathbf{Q} = \mathbf{I} \tag{5.13}$$

eq. (5.5) can then be transformed into:

$$\rho = \left\|\mathbf{R}_1\mathbf{x} - \mathbf{Q}_1^T\mathbf{b}\right\|_2^2 + \left\|\mathbf{Q}_2^T\mathbf{b}\right\|_2^2 \tag{5.14}$$

This last equation is minimized when the first term in the right hand-side vanishes, so the vector $\mathbf{x}$ can be given as:

$$\mathbf{x} = \mathbf{R}_1^{-1}\mathbf{y} \tag{5.15}$$

where

$$\mathbf{y} = \mathbf{Q}_1^T\mathbf{b} \tag{5.16}$$

Note that $\mathbf{Q}_1$ does not actually have to be computed; only $\mathbf{y}$ must be obtained. Usually QR methods for solving least-squares systems divide their operation into two phases: a triangularization phase, where $\mathbf{R}_1$ and $\mathbf{y}$ are computed, and a solution phase, where the triangular system is solved, usually by back-substitution. For the first phase, there are, however, several algorithms to choose from. These are covered in the following section.

5.4.1 Sequential QR algorithms

There are several algorithms that can be used to compute the triangularization phase, but it is usually recognized [135] that the Householder method, the fast Givens method and the modified Graham-Schmidt method are the most important algorithms for performing this task.

As the three methods have similarly good numerical properties, it was decided to parallelize the one that involves the smallest sequential computational cost. If a good parallel efficiency can be obtained the problem is then solved; if not, another candidate method must be parallelized.

In terms of sequential computational cost, the modified Graham-Schmidt method is more expensive than the Householder algorithm [135]. Fast Givens approaches were developed as a rearrangement of the Givens method, so that they could be performed with 'Householder speed'. Although a straightforward fast Givens method has the same computational complexity as the Householder method, it is found in practice that monitoring for overflow of terms makes fast Givens methods slower, and more complicated to implement, than the Householder approach [135].

Based on these considerations, the Householder scheme was the algorithm chosen to parallelize.

This algorithm is based on the repeated use of elementary reflectors or Householder matrices:

$$\mathbf{H} = \mathbf{I} - \mathbf{h}\mathbf{h}^{\mathrm{T}} \tag{5.17}$$

The crucial point in favour of elementary reflectors is that they can be used to introduce zeros into a vector. Given a vector $\mathbf{x}$, an elementary reflector can be found such that:

$$\mathbf{H}\mathbf{x} = -\sigma\mathbf{e}_1 \tag{5.18}$$

where $\mathbf{e}_1$ denotes a vector of zeros, with the exception of the first element, which is unitary.

It can be shown [172] that the following algorithm computes both σ and the Householder vector ($\mathbf{h}$) that satisfies (5.18):

Algorithm 10 - Computation of σ and $\mathbf{h}$

$\zeta := \|\mathbf{x}\|_2$
$\sigma := \mathrm{sgn}(\mathbf{x}_1)\zeta$
$\mu := \zeta + |\mathbf{x}_1|$
$\mathbf{x}_1 := \mathrm{sgn}(\mathbf{x}_1)\mu$
$\pi := \zeta\mu$
$\mathbf{h} := \dfrac{\mathbf{x}}{\sqrt{\pi}}$

The above mentioned property of elementary reflectors can be used to triangularize a rectangular matrix, by successive applications of Householder matrices:

$$\mathbf{H}[n]\mathbf{H}[n-1]\ldots\mathbf{H}[2]\mathbf{H}[1]\mathbf{A} = \mathbf{A}[n+1] = \begin{bmatrix}\mathbf{R}_1\\ 0\end{bmatrix} \tag{5.19}$$

where **A** is assumed to have n columns. To see this, let us suppose that, at the end of iteration k, the matrix **A**[k] has the following form:

$$\mathbf{A}[k] = \begin{bmatrix}\mathbf{R} & \mathbf{r} & \mathbf{B}\\ 0 & \mathbf{x} & \mathbf{C}\end{bmatrix} \tag{5.20}$$

where **R** is an upper triangular matrix of size k*k. Denoting **H**[k+1] as:

$$\mathbf{H}[k+1] = \begin{bmatrix}\mathbf{I} & 0\\ 0 & \mathbf{H}\end{bmatrix} \tag{5.21}$$

where **H** satisfies (5.18), it is easy to see that:

$$\mathbf{A}[k+1] = \mathbf{H}[k+1]\mathbf{A}[k] = \begin{bmatrix}\mathbf{R} & \mathbf{r} & \mathbf{B}\\ 0 & -\sigma\mathbf{e}_1 & \mathbf{HC}\end{bmatrix} \tag{5.22}$$

so that, at each iteration, another column of **A** is put in a triangular form.

Comparing (5.12) with (5.19) it is clear that the orthogonal matrix **Q** can be obtained from the Householder matrices by:

$$\mathbf{Q} = (\mathbf{H}[n]\mathbf{H}[n-1]\ldots\mathbf{H}[2]\mathbf{H}[1])^T \tag{5.23}$$

When the Householder technique is employed for solving least-squares problems (5.5), there is no need to actually compute the matrices **H**[k]. The following algorithm can be used instead:

Algorithm 11 - Least squares solution using Householder orthogonalization

$\mathbf{R} := \mathbf{A}$; $\mathbf{y} := \mathbf{b}$

for i=1 to n

  $\mathbf{x} := \mathbf{R}_{i..m,\,i}$

  ... compute **h** and σ such that (5.18) is satisfied -- Algorithm 10

  $\mathbf{R}_{i..m,\,i} := -\sigma\mathbf{e}_1$

  $\mathbf{r} := \mathbf{h}^T\mathbf{R}_{i..m,\,i+1..n}$

  $\mathbf{R}_{i..m,\,i+1..n} := \mathbf{R}_{i..m,\,i+1..n} - \mathbf{hr}$

  $l := \mathbf{h}^T\mathbf{y}_{i..m}$

  $\mathbf{y}_{i..m} := \mathbf{y}_{i..m} - l\mathbf{y}_{i..m}$

end

... solve $\mathbf{R}_{1..n,.}\mathbf{x} := \mathbf{y}_{1..n}$ by back-substitution

Algorithm 11 is our candidate for parallelization.

5.4.2 Parallelization of the selected algorithm

Analysing Algorithm 11 it is evident that two different phases can be identified: the triangularization phase (the 'for' loop) and the solution phase (last pseudo-instruction). In terms of computational complexity, the former uses roughly $m*n^2$ floating point operations, while the latter only needs n^2 operations. This way, to achieve a good efficiency for the overall algorithm special care must be taken in the triangularization phase.

5.4.2.1 Triangularization phase

Analysing the complexity of each step of the triangularization phase, it is clear that the bulk of the computation is concentrated in the update of the columns which remain to be triangularized, computed by the following pair of equations:

$$\mathbf{r} = \mathbf{h}^T \mathbf{R}_{i..m,\, i+1..n} \tag{5.24}$$

$$\mathbf{R}_{i..m,\, i+1..n} = \mathbf{R}_{i..m,\, i+1..n} - \mathbf{h}\mathbf{r} \tag{5.25}$$

Different parallelization schemes can be applied to these steps. To introduce them, it will be assumed that p processors will be used, and that *j* denotes the set of columns or rows associated with the j^{th} processor. Taking as example the computation of **r** (the same reasoning can be applied to (5.25)), this vector can be computed as [173]:

a) $\mathbf{r} = \sum_{i=1}^{P} \mathbf{r}^{(j)} = \sum_{i=1}^{P} \mathbf{h}_j^T \mathbf{R}_{j,\, i+1..n}$

b) $\mathbf{r}_j = \mathbf{h}^T \mathbf{R}_{i..m,\, j}, \qquad j = 1, \ldots, p$

c) a mixture of the two first approaches.

It is clear that these different approaches make different demands concerning the data required for each process. This has consequences for the implementation of the other steps. We shall describe possible parallel algorithms for approaches (a) and (b), since a parallel algorithm for the last approach can be deduced from a combination of the algorithms for the two first schemes.

If approach (a) is followed, matrix **A** and the **b** vector must be partitioned by row, i.e., process j has its own private set *j* of row vectors of **A** and its *j* set of elements of **b**. Assuming this distribution of data, an algorithm can be proposed for the j^{th} process. In this algorithm, it is assumed that this process, named *worker*, communicates with an additional process by

means of two channels, named From.Worker and To.Worker. The algorithm for this additional process will not be present for the time being, since it is highly dependent on the actual topology of the network of transputers used, and is not necessary for the comprehension of the following algorithm.

Algorithm 12 - Triangularization phase, row based, for process j

$\mathbf{R}_{j,.} := \mathbf{A}_{j,.}$; $\mathbf{y}_j := \mathbf{b}_j$

for i=1 to n

$\mathbf{x} := \mathbf{R}_{\{i..m\} \cap j, i}$

$\zeta^{(j)} := \|\mathbf{x}\|_2^2$

From.Worker ! $\zeta^{(j)}$

To.Worker ? $\zeta := \sum_{i=1}^{P} \zeta^{(j)}$

if $i \in j$

... compute π and update $\mathbf{x}_1$ -- Steps 2 to 5 of Algorithm 10

From.Worker ! π

$\mathbf{R}_{\{i.., m\} \cap j, i} := -\sigma \mathbf{e}_1$

else

To.Worker ? π

end

$\mathbf{h}_{\{i..m\} \cap j} := \frac{\mathbf{x}_{\{i..m\} \cap j}}{\sqrt{\pi}}$

$\mathbf{r}^{(j)} := \mathbf{h}^T_{\{i..m\} \cap j} \mathbf{R}_{\{i..m\} \cap j, i+1..n}$

From.Worker ! $\mathbf{r}^{(j)}$

To.Worker ? $\mathbf{r} := \sum_{i=1}^{P} \mathbf{r}^{(j)}$

$\mathbf{R}_{\{i..m\} \cap j, i+1..n} := \mathbf{R}_{\{i..m\} \cap j, i+1..n} - \mathbf{h}_{\{i..m\} \cap j} \mathbf{r}$

$l^{(j)} := \mathbf{h}^T_{\{i..m\} \cap j} \mathbf{y}_{\{i..m\} \cap j}$

From.Worker ! $l^{(j)}$

To.Worker ? $l := \sum_{i=1}^{P} l^{(j)}$

$\mathbf{y}_{\{i..m\} \cap j} := \mathbf{y}_{\{i..m\} \cap j} - l \mathbf{y}_{\{i..m\} \cap j}$

end

In Algorithm 12 the symbol $\cap$ denotes set intersection. It is evident, by inspecting Algorithm 12, that a considerable number of communications must be performed within each iteration, which is a disadvantage of this algorithm.

In approach (b) the parallelization is done by column rather than by row. It is now assumed that process j has its own set j of columns of **A**. Using this approach, the operations on the right-hand side vector **b** can be performed either by a different process or by one of the processes responsible for the actual triangularization of matrix **A**. Anticipating that a master process will be needed for the parallel implementation of the complete training algorithm, we assume that the operations on **b** will be performed by this additional process, identified as process 0.

Algorithm 13 - Triangularization phase, column based, for process j

$\mathbf{R}_{.,j} := \mathbf{A}_{.,j}$

for i=1 to n

  if $i \in j$

    $\mathbf{x} := \mathbf{R}_{i..m,\,i}$

    ... compute **h** and σ -- Algorithm 10

    From.Worker ! **h**

    $\mathbf{R}_{i..m,\,i} := -\sigma\mathbf{e}_1$

    $j := j - i$

  else

    To.Worker ? **h**

  end

  $\mathbf{r} := \mathbf{h}^T\mathbf{R}_{i..m,\,j}$

  $\mathbf{R}_{i..m,\,j} := \mathbf{R}_{i..m,\,j} - \mathbf{h}\mathbf{r}$

end

Algorithm 14 - Triangularization phase, column based, for process 0

$\mathbf{y} := \mathbf{b}$

for i=1 to n

  To.Worker ? **h**

  $l := \mathbf{h}^T\mathbf{y}_{i..m}$

  $\mathbf{y}_{i..m} := \mathbf{y}_{i..m} - \mathbf{h}l$

end

Algorithms 13 and 14 require only one communication during each iteration, contrasting with 7 communications needed by Algorithm 12. This reflects the fact that the Householder algorithm is in fact a column-based algorithm. The amount of data to be communicated in approach (b) is greater than that in approach (a), but the fact that the processes are more loosely coupled in approach (b) makes it a stronger candidate for an efficient parallel algorithm.

Having decided to implement a column-based partition, two further points need to be addressed: the actual partitioning of columns between the processes and some mechanism to transfer the data between the processes.

In order to have an efficient parallel algorithm, good load balancing must be obtained across all the processes. Clearly, this is related to the order in which the Householder vectors are computed and how the columns of **A** are partitioned amongst the different processes. If the columns of **A** are partitioned in the natural order (i.e. the first n/p columns are allocated to process 1, the second n/p to process 2, etc.), and the Householder vectors are computed in the order described in Algorithm 13, a very poor load balancing is obtained. If this scheme was used, process j would be idle after the first nj/p iterations of the algorithm, and the efficiency obtained would certainly be poor.

Keeping the order of computation of the Householder vectors as described in Algorithm 13, the best scheme of column organization would be to assign to the i^{th} column of the j^{th} process the column k of matrix **A** satisfying (5.26):

$$k = (i - 1)p + j \tag{5.26}$$

With this partitioning scheme all the processes will have columns to update until iteration number n-p, each process becoming idle in each consecutive iteration.

The last point that needs to be addressed before implementing the algorithm is the way that communications will be performed. At each iteration, one process computes an Householder vector and must broadcast it to all the other p-1 processes, as well as to the process that is computing **y**. A general scheme, which is independent of either the actual number of processes used or the dimensions of **A**, is desired. Since each process will be allocated to one processor, and each transputer has only four bidirectional links, some form of intermediate communication must be implemented. To isolate the algorithm from the actual topology used, communications between different transputers will be performed by a priority process, whose only responsibility is the transfer of data. This additional process will be denoted as the *router*, and the process which implements the actual computation called the *worker*. This way there will be one router and one worker per processor.

Since intermediate communications will have to be implemented, at each iteration different processes will receive the computed Householder vector with different delays. In order not to degrade the performance of the system, it is vital that the process responsible for the computation of the next Householder vector receives the current vector with a minimum of delay.

Taking into account the discussion of the partitioning of columns between processes, it seems that the best topology for the network of transputers is a ring. At this stage of the work it was decided to concentrate on the actual triangularization of the matrix and to postpone the computation of **y** in processor 0 to some later time. The topology adopted is shown in Fig. 5.1.

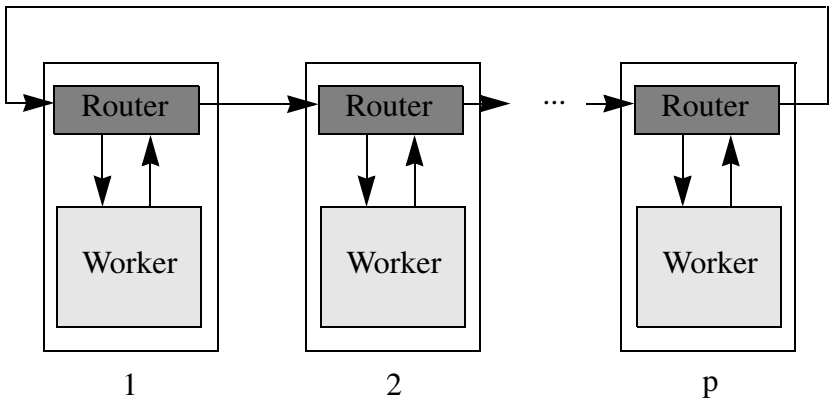


Fig. 5.1 - Ring Topology

In this Figure, the outer boxes denote processors, while the inner shaded boxes denote processes. Each one of the workers executes a slightly modified version of Algorithm 13. A simplified version of the algorithm that is implemented in the routers is given in Algorithm 15. In this algorithm two channels, denoted From.Left and To.Right, are used to implement the ring topology. The identifiers of the logical channels that connect the router and its corresponding worker process are the same as in Algorithm 13.

In this last algorithm a counter variable, identified as ‘rem’, is used to ensure that the Householder vector is broadcast to the remaining p-1 processes in each iteration.

This parallelization scheme was implemented in Occam and executed on a Transtech transputer platform [174], hosted by a Sun 4/110 workstation. This platform is populated with IMS T800-25 Inmos transputers, with the speed of the links set at 20 Mbits/second.

Algorithm 15 - Router process j

```
for i=1:n

    ALT
        From.Worker ? h
            To.Right ! [p]
                       [h]

        From.Left ? [rem]
                    [ h ]
            SEQ
                rem := rem-1
                if rem>0
                    PAR
                        To.Right ! [rem]
                                   [ h ]
                        To.Worker ! h
                else
                    To.Worker ! h
                end
end
```

In order to compute the efficiency of the parallel version, the triangularization phase of Algorithm 11 (excluding the computation of **y**) was implemented in Occam, and executed on a single transputer connected to the host transputer via a bidirectional link. The execution times shown in Table 1 were obtained using the high priority timer of the former transputer. They denote the time taken to execute the 'for' loop in Algorithm 11, for nine matrices with different dimensions (m and n denote their number of rows and columns, respectively).

n	Execution time (ms)
10	24
25	129
50	408

a) m=50

n	Execution time (ms)
25	279
50	999
100	3 181

a) m=100

n	Execution time (ms)
50	2 182
100	7 872
200	25 130

a) m=200

Table 1 - Sequential executions times for triangularization phase

Afterwards, Algorithm 13 was implemented on 5 transputers employing the topology depicted in Fig. 5.1. The data was partitioned in the way already described. The values of speed-up and efficiency obtained, for the 9 different matrices, are shown in Tables 2 and 3:

n	speed-up
10	1.98
25	3.05
50	3.82

a) m=50

n	speed-up
25	3.06
50	3.77
100	4.29

a) m=100

n	speed-up
50	3.76
100	4.24
200	4.55

a) m=200

Table 2 - Speed-up for triangularization phase

n	efficiency
10	.40
25	.61
50	.76

a) m=50

n	efficiency
25	.61
50	.75
100	.86

a) m=100

n	efficiency
50	.75
100	.85
200	.91

a) m=200

Table 3 - Efficiency for triangularization phase

Analysing the results obtained it is clear that the efficiency depends only weakly on the number of rows of the matrix but is heavily influenced by the number of columns allocated to each process, increasing as more columns are allocated to each process. To explain this result, we can approximate the time taken for each iteration (t_{iter}) by:

$$t_{iter} = t_h + t_c + t_u \tag{5.27}$$

where t_h denotes the time to compute the Householder vector, t_c is the time spent in communicating this vector to the process to compute the next **h**, and t_u is the time taken to update the columns of the next process that are not yet triangularized. When the number of columns to be updated is large, t_h and t_c are small compared with t_u and the efficiency obtained, in that iteration, is considerable. As the algorithm proceeds t_u decreases because the number of columns to be updated, in the p workers, is decreased by one at each p iterations of the algorithm. Consequently the ratio t_u/t_{iter} is reduced and so is the efficiency.

Employing this reasoning, as the ratio n/p increases, so do the initial values of efficiency, and accordingly the efficiency of the overall algorithm.

Although the results obtained with this scheme are generally good, especially when a high ratio n/p is considered, a more efficient solution is desirable. This can only be achieved by reducing t_c in (5.27). A network topology with higher connectivity, as shown in Fig. 5.2, was investigated.

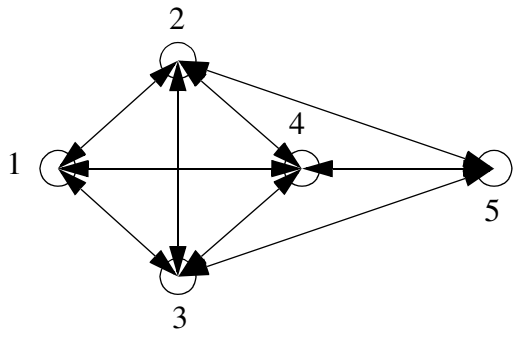


Fig. 5.2 - Connected topology

In this Figure each one of the nodes represents one transputer, with a router and a worker process. This topology allows the broadcasting of data with a minimum of intermediate nodes. For the case shown in Fig. 5.2, the Householder vector computed in one transputer can be transmitted to all the other processors involved in the triangularization with a maximum delay equal to $2t_c$, if communications are performed in parallel. For the same number of processors in the ring topology, this maximum delay is $4t_c$.

The worker processes are the same as in the ring topology. However, an implementation which is independent of the number of transputers used requires more complex routers. For this purpose, in the case of the ring topology, a counter was added to the Householder vector to be broadcast. By initializing this counter with the number of transputers in the ring, and decrementing and testing the counter at each transputer, a solution independent of the dimension of the ring is easily obtained. One possible solution for the connected topology also involves the addition of a header to the Householder vector to be broadcast. In this case the header is a set of identifiers of the processes to which the vector must be transmitted. If each router has available a map between the other worker processes and the identifiers of the external channels used for communication then it is possible to derive a procedure to implement the broadcasting required. Taking, for example, Fig. 5.2 and assuming that it was decided to implement communications between transputer 1 and 5 using transputer 2 as an intermediate node then, once worker 1 has computed its Householder vector, its router would perform (in parallel) the following output operations:

Chan1.to. 2 ! $\left[\left[2\ 2\ 5\right]\ \mathbf{h}^{T}\right]^{T}$

Chan1.to.3 ! $\left[\left[1\ 3\right]\ \mathbf{h}^{T}\right]^{T}$

Chan1.to.4 ! $\left[\left[1\ 4\right]\ \mathbf{h}^{T}\right]^{T}$

where Chani.to.j denotes the identifier of the channel connecting process i to process j. The first element in the header is the number of processes to which the Householder vector must be transmitted. When routers 3 and 4 receive these messages, they recognize that it should not be forwarded to any transputer. Router 2 analyses its header and identifies that it should be forwarded to router 5. Using its private map, it is able to conclude that the external channel Chan2.to.5 should be used to communicate the data, and will therefore execute the following instructions in parallel:

Chan2.to.5 ! $\left[\left[1\ 5\right]\ \mathbf{h}^{T}\right]^{T}$

To.Worker ! **h**

To implement this scheme, each router process must then receive two additional arguments:

- a list of the identifiers of the processes to which the Householder vector, computed by its own worker, must be transmitted;

- one vector that maps the processes identifiers to the physical channel identifiers of the router.

Table 4 shows the efficiency figures obtained using this approach.

n	efficiency
10	.35
25	.55
50	.70

a) m=50

n	efficiency
25	.57
50	.72
100	.83

a) m=100

n	efficiency
50	.73
100	.83
200	.90

a) m=200

Table 4 - Efficiency for triangularization phase, connected topology

The efficiency obtained with this scheme is clearly poorer than was achieved by the first approach, the ring topology (Table 3). To identify the reasons for the inefficiency of this second scheme, a third approach was implemented. This time, the topology shown in Fig. 5.2 was used, but router processes, specifically targeted for the particular topology employed, were implemented for each transputer. The output operations are the same as performed by each router in the general solution, except that the header is omitted. This reduces both the communication overhead and the processing in the routers. The efficiency results for this last approach are shown in Table 5:

n	efficiency
10	.40
25	.61
50	.75

a) m=50

n	efficiency
25	.61
50	.75
100	.85

a) m=100

n	efficiency
50	.76
100	.86
200	.91

a) m=200

Table 5 - Efficiency for triangularization phase, connected topology, targeted routers

The results obtained with this scheme are nearly identical to the ones achieved by the ring topology. Analysing the results shown in tables 3 to 5, it may be concluded that:

- using Algorithm 13 as the worker process, nothing is gained by using a more connected topology than the ring topology. This is due to the fact that t_c, in (5.27), already has its minimum for the ring topology. The efficiency of the algorithm does not depend on the maximum delay incurred in broadcasting the Householder vector;

- the poorer results shown in Table 4 are due to the increase in the amount of data to be communicated (a header is also transmitted) and particularly to the amount of processing that each router must perform;

- if a more efficient solution is desired than clearly the worker algorithm must be modified.

Analysing Algorithm 13, it can be observed that another sequence of operations can be employed. To explain this modification, let us assume that, at iteration k, process j must compute the Householder vector, **h**[k]. In the preceding iteration, process j received **h**[k-1], updated the columns yet to be triangularized (j[k-1]) and only then computed **h**[k], transmitted it onwards and finally updated the columns $j[k] = j[k-1] - k$ using **h**[k].

However, to compute **h**[k] it is not necessary to update all the *j*[k-1] columns with **h**[k-1], but simply the first one of these columns. After performing this operation, **h**[k] can then be sent following which the columns *j*[k] are updated, first with **h**[k-1] and then with **h**[k]. In principle, this sequence of operations has the advantage of removing the t_c term from (5.27), since the communication is now being performed in parallel with the computation. Note also that this modification to the algorithm of the worker process does not in any way affect the router processes, since, from their point of view, the sequence of operations remains the same.

This new worker was implemented in Occam, and used in conjunction with the router described by Algorithm 15. The topology depicted in Fig. 5.1 was employed, with p=5. The efficiency figures for the nine different matrices are shown in Table 6:

n	efficiency
10	.47
25	.76
50	.87

a) m=50

n	efficiency
25	.76
50	.87
100	.92

a) m=100

n	efficiency
50	.87
100	.92
200	.95

a) m=200

Table 6 - Efficiency for triangularization phase, new worker, ring topology

Comparing these values with Table 3, it is seen that improvements in efficiency ranging from 4% to 15% are obtained with the new worker.

The two versions of routers for the connected topology (Fig. 5.2) were also employed with the new worker. Tables 7 and 8 show the efficiency obtained when the general and the targeted routers are used, respectively.

n	efficiency
10	.40
25	.75
50	.84

a) m=50

n	efficiency
25	.73
50	.85
100	.91

a) m=100

n	efficiency
50	.85
100	.91
200	.94

a) m=200

Table 7 - Efficiency for triangularization phase, new worker, connected topology, general routers

n	efficiency
10	.47
25	.76
50	.87

a) m=50

n	efficiency
25	.76
50	.87
100	.92

a) m=100

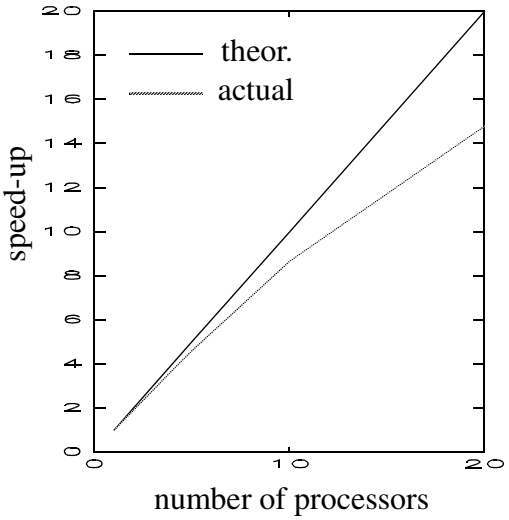
n	efficiency
50	.87
100	.92
200	.95

a) m=200

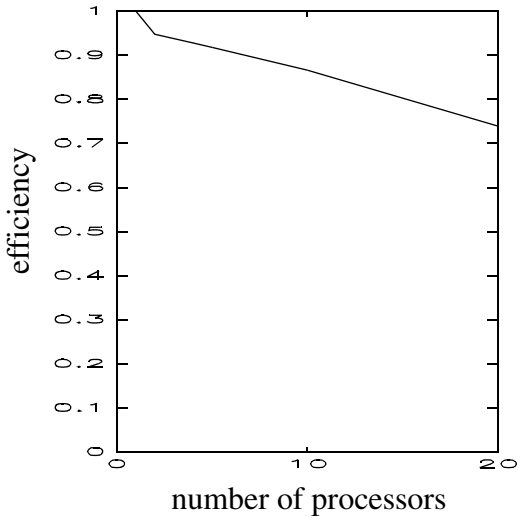
Table 8 - Efficiency for triangularization phase, new worker, connected topology, targeted routers

The targeted version of the connected topology and the ring topology achieve the best efficiency results among the different parallelized schemes experimented. As the ring topology is a simpler and general solution, which is independent of the number of transputers used, it is chosen as the parallel scheme for the triangularization phase.

Up to now the same number of transputers (5) was used in all the different schemes of parallelization tried. To investigate the performance of the chosen scheme with different numbers of transputers, it was decided to triangularize a matrix with 200 rows and 100 columns. These dimensions were chosen because they are similar to the dimensions of the ***J*** matrices appearing in the training of the MLPs used in the PID autotuning approach described in Chapter 3. Fig. 5.3 illustrates the speed-up and the efficiency obtained when 2, 4, 5, 10 and 20 transputers were used. In the case of the speed-up graph, the theoretical speed-up line is also shown.



a) speed-up



b) efficiency

Fig. 5.3 - Efficiency and speed-up for triangularization phase, new worker, ring topology

As expected, the deviation between the theoretical and actual speed-up curves increases as the number of transputers increases. Nevertheless, even with 20 transputers, speed-up is far from being exhausted, as can be seen from the curve in Fig. 5.3a).

When analysing the efficiency results shown in Table 3, where a fixed number (5) of transputers was used, it was concluded that this figure was almost independent of the number of rows of the matrix to be triangularized, and depended on the number of columns per processor. Analysing the results of Fig. 5.3 and Table 6, it can be stated that efficiency is also almost independent of the number of transputers used. In order to see this more clearly, a graph of the dependence of efficiency on the number of columns per processor, for the new worker, is shown in Fig. 5.4. In this figure the solid line is obtained from Table 6 where 5 transputers are used. The points denoted by the symbols '*' are taken from Fig. 5.3, where a variable number of transputers was employed. It can be seen that there is good agreement between fixed and variable number of transputers, the biggest discrepancy (2%) being obtained for 5 columns per processor, where the efficiencies are obtained using 5 and 20 transputers. Fig. 5.4 can then be used to obtain a good first approximation of the efficiency, and indirectly of the speed-up, obtained for the parallelization of a matrix of any size.

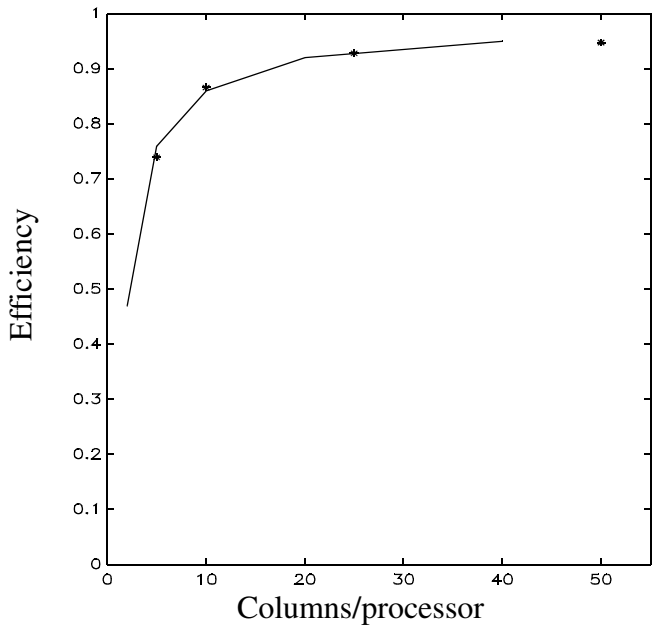


Fig. 5.4 - Efficiency versus columns per processor, triangularization phase, new worker, ring topology

5.4.2.2 Solution phase

Having successfully parallelized the computation of $\mathbf{R}_1$, it is now necessary to incorporate the computation of $\mathbf{y}$ and of the solution $\mathbf{R}_1^{-1}\mathbf{y}$ into the parallel algorithm.

As pointed out in the last section, the update of the right hand-side is to be performed by a master process, denoted by process 0. To allow the Householder vector to be received by this process, another channel was added to the p^{th} router, and its pseudo-instructions inside the 'if' block in Algorithm 15 are slightly modified.

It has also been noted that, sequentially, the solution phase is usually performed using a back-substitution algorithm.

The importance of the triangularization phase for the performance of the overall least-squares solution forced the **A** matrix to be partitioned on a column basis. This has disadvantages for this second phase because, in order to implement the standard parallelization of the back-substitution algorithm [173], it is rows, rather than columns, which must be stored in each processor.

To implement the standard parallel back-substitution algorithm the elements of **A** should then be distributed amongst the p processes in such a way that each process has a set of rows of **A**. However, this was not done for the following reasons:

a) good efficiency cannot be expected for this parallel algorithm using transputers, as it involves very small computations and a fairly large amount of communications;

b) the contribution of this phase to the overall performance of the least-squares solution is very small, especially when large matrices are considered. Table 9 shows the sequential execution times for the complete least-squares solution. Comparing these values with Table 1, it can be seen that when dimensions of 200*100 are considered, which is the most similar case to the MLPs employed in the PID autotuning approach, the solution phase only accounts for 2.7% of the total execution time.

n	Execution time (ms)
10	27
25	137
50	425

a) m=50

n	Execution time (ms)
25	297
50	1 036
100	3 267

a) m=100

n	Execution time (ms)
50	2 260
100	8 073
200	25 620

a) m=200

Table 9 - Sequential executions times for least-squares solution

Consequently it was decided to implement the solution phase using the sequential back-substitution algorithm, but split among the p processes.

The topology used for the complete least-squares solution is shown in Fig. 5.5.

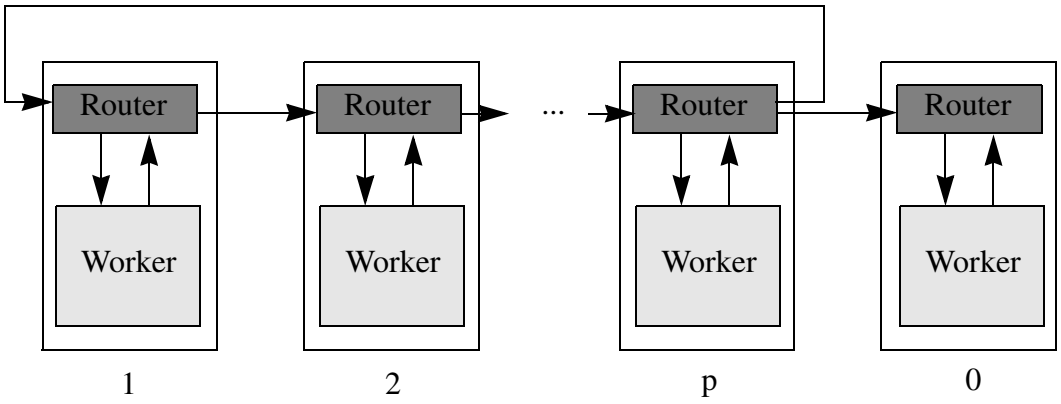


Fig. 5.5 - Ring topology for the complete least-squares solution

A bi-directional ring is now used. In the triangularization phase the Householder vectors circulate from left to right; in the solution phase the right-hand side vector circulates from right to left. A simplified algorithm for the solution phase of the j^{th} worker is:

Algorithm 16 - Solution phase for worker j

$i := \frac{n}{p}$

for s=1 to n/p
  k=n+j-s*p
  To.Worker ? $\mathbf{y}$
  $\mathbf{x}_i := \frac{\mathbf{y}_k}{\mathbf{R}_{k,i}}$
  $\mathbf{y}_{1..k-1} := \mathbf{y}_{1..k-1} - \mathbf{R}_{1..k-1,i}\mathbf{x}_i$
  From.Worker ! $\mathbf{y}_{1..k-1}$
  $i := i - 1$
end

In this algorithm it is assumed that n/p columns of the **A** matrix have been assigned to matrix **R** of process j using the assignment scheme described by (5.26).

A simplified algorithm of the j^{th} router for this phase is simply:

Algorithm 17 - Solution phase for router j

```
for k=1:n/p
    From.Right ? y
    To.Left ! y
end
```

Two channels, denoted as From.Right and To.Left are used in this algorithm to implement the ring topology in the direction needed for this phase.

The parallel version of the complete least-squares solutions was coded in Occam and afterwards executed on a network of transputers with the topology shown in Fig. 5.5. As in the case of the triangularization phase, the **A** matrix was partitioned across 5 transputers (p=5), and one additional transputer was used to compute **y**.

Table 10 shows the efficiency figures obtained. To obtain these values, the values of speed-up were divided by p=5, since it is assumed that the additional transputer will have the role of a master in the overall training algorithm.

n	efficiency
10	.47
25	.73
50	.81

a) m=50

n	efficiency
25	.76
50	.85
100	.89

a) m=100

n	efficiency
50	.87
100	.92
200	.94

a) m=200

Table 10 - Efficiency for least-squares solution

Comparing Table 10 and Table 6 it can be concluded that the use a sequential algorithm to solve the triangular system does not cause a large overhead in the efficiency of the overall problem. For matrices with n large, or $m \gg n$, the computation of **y** in the additional transputer completely compensates for the overhead incurred in the solution phase.

5.4.2.3 Specializing the algorithm

In the two last sections a parallel algorithm was proposed to implement the least-squares solution of overdetermined systems. As explained before, two of the more computationally costly steps in the learning algorithm can be formulated in this manner.

However, both have special structures that allow some savings in computation to be performed.

Starting with the computation of the optimal values of the linear parameters ($\hat{\mathbf{u}}$) (4.59) it can be observed that the column related to the bias of the output neuron is always fixed. If the columns of the right-hand side of (4.59) are reordered such that this column is the first one, there is no need to compute and transmit the first Householder vector, since it is also fixed. Then **h**[1] can instead be computed within the initialization code of all the different workers, and used immediately by them at each iteration of the learning algorithm.

The computational savings involved in the computation of $\hat{\mathbf{u}}$, although not negligible, are not very important when the complexity of the complete training algorithm is concerned. On the other hand, the special structure of the LM update can lead to important savings in the overall learning algorithm.

The LM update can be expressed in the least-squares format shown in (5.10), and so can be solved using the algorithms (sequential and parallel) described so far. But, assuming that the dimensions of $\boldsymbol{J}$ are m*n, a [(m+n)*n] least-squares problem would then have to be solved. In this problem, however, the last n rows of **J**' (5.8) are a diagonal matrix and the last n elements of **e**' are zero.

Taking this special structure of **J**' and **e**' into account, it can be easily seen that if the row range $i..m$ of the different vector and matrix quantities, in the i^{th} iteration of Algorithm 11, is replaced by $i..m+i-1$, no unnecessary computations will be made throughout the triangularization phase.

The sequential version of the least-squares solution was modified in this sense to compute the LM update. The execution times obtained with this approach are summarised in Table 11, where m and n refer to the dimensions of $\boldsymbol{J}$ in (5.8).

n	Execution time (ms)
10	31
25	173
50	664

a) m=50

n	Execution time (ms)
25	339
50	1 303
100	5 105

a) m=100

n	Execution time (ms)
50	2 627
100	10 280
200	40 670

a) m=200

Table 11 - Sequential execution times for LM update

The chosen parallel implementation of the least-squares solution was also modified in the same way. Using p=5, the efficiency results obtained are shown in the next Table.

n	efficiency
10	.52
25	.76
50	.85

a) m=50

n	efficiency
25	.79
50	.88
100	.92

a) m=100

n	efficiency
50	.91
100	.95
200	.97

a) m=200

Table 12 - Efficiency for LM update

High values of efficiency were obtained for the LM update. As this is the most computationally intensive step of the training algorithm, it is unnecessary to investigate further the parallelization of other sequential QR algorithms. The results also forecast a good efficiency for the overall learning algorithm

5.5 Recall operation

Having successfully parallelized equations (4.48) and (4.59), the parallelization of Algorithm 2 will now be considered. Although this algorithm is split into two parts in the training algorithm (the optimal value of the linear parameters must be computed before the output vector of the MLP is obtained) it will be considered in this section as a whole.

The computation of the output vector was presented in Chapter 4 on a pattern basis. When considering possible parallelization schemes, however, it is advisable to consider the operations on an epoch basis.

The most important operations, as far as parallelization is concerned, are:

$$\mathbf{Net}^{(z+1)} = \left[\mathbf{O}^{(z)}\ 1\right]\mathbf{W}^{(z)} \qquad z = 1, \ldots, q-1 \tag{5.28}$$

$$\mathbf{O}^{(z+1)} = \mathbf{F}^{(z+1)}(\mathbf{Net}^{(z+1)}) \qquad z = 1, \ldots, q-1 \tag{5.29}$$

Three possibilities of parallelization exist for these equations:

a) $\mathbf{O}^{(z)}$, $\mathbf{O}^{(z+1)}$and $\mathbf{Net}^{(z+1)}$ are partitioned in rows, the operations in the different partitions being performed in parallel;

b) $\mathbf{W}^{(z)}$, $\mathbf{Net}^{(z+1)}$ and $\mathbf{O}^{(z+1)}$ are partitioned in columns, the operations in each column partition being executed in parallel;

c) a mixture of the first two approaches.

Approach (a) reflects the fact that the MLP has no dynamics, i.e., present behaviour is independent of past behaviour. In this approach each process can perform the total recall operation for its own set of data, without needing any communication to other processes. If there is the need to reconstruct the complete output vector, then the different partitions of that vector must be communicated in some way.

This first approach is therefore very promising. However, the assignment of data using this approach is the opposite of the one employed in the least-squares solution problem. Using this approach for the recall operation, each process must have a complete copy of the entire weight vector, $\mathbf{w}$. In contrast, only partitions of $\hat{\mathbf{u}}$ and $\mathbf{p}$ are available in each process. To compute these vectors, the matrices $\mathbf{O}^{(q-1)}$ and $\boldsymbol{J}$ are partitioned, on a column basis, by several processes. As for the computation of $\boldsymbol{J}$, $\mathbf{O}^{(z)}$ and $\mathbf{Net}^{(z)}$, $q < z < 1$, must be known previously. The use of approach (a) would lead to a great number of communications to obtain the data in a format suitable for the parallel implementation of the two least-squares problems.

Approach (b) has the advantage of demanding a distribution of the data similar to the least-squares problems. It reflects the general knowledge that, within each layer, all the neurons can compute their output independently of the other neurons in the same layer. However, it has the disadvantage of requiring several communications to perform the recall operation, since to compute any column partition of $\mathbf{Net}^{(z+1)}$, the entire matrix $\mathbf{O}^{(z)}$ must be known.

In order to have a common data distribution among all the phases of the learning algorithm, it was decided to follow approach (b). If a poor efficiency was obtained using this scheme, approach (a) would then be considered. For the same reasons as discussed in the parallelization of the least-squares solution, each processor will have a router and a worker process, the former being responsible for the transfer of data between different processes.

Although the parallelization of equations (5.28) and (5.29) is a straightforward operation, as in the least-squares problem, the sequence in which the operations are performed has a strong influence on the overall efficiency of the parallel recall operation. In the least-squares solution, a more efficient algorithm was obtained by exploiting a strong point of the Inmos transputer: the possibility of communicating while performing useful computation. This facility can also be exploited in this phase, with a view to reducing the disadvantage already mentioned for this partitioning.

In order to describe how this facility can be useful during the recall operation, one convention must be established. In each layer, the constituent neurons will be partitioned

amongst the various worker processes. To identify those neurons, the symbol $\boldsymbol{j}^{(z)}$ will denote the set of indices of neurons on the z^{th} layer assigned to process j. As the input training data is constant throughout the learning algorithm, then it is assumed that it is common to all the processes:

$$\boldsymbol{j}^{(1)} = \{1, 2, \ldots, \mathbf{k}_1\} \ . \tag{5.30}$$

To compute $\mathbf{Net}^{(z)}_{:,\boldsymbol{j}^{(z)}}$, for $1 < z < q$, process j must have $\mathbf{O}^{(z-1)}$, but only a subset of this matrix, $\mathbf{O}^{(z-1)}_{:,\boldsymbol{j}^{(z-1)}}$, is computed by itself. All the other partitions are computed by different processes, and must be transmitted to process j. This problem is common to all processes, and clearly constitutes a disadvantage of this parallelization approach. However, as soon as $\mathbf{O}^{(z-1)}_{:,\boldsymbol{j}^{(z-1)}}$ is obtained it can be broadcast to all the other processes. The other processes, also, may perform this operation. While these communications take place, $\mathbf{Net}^{(z)}_{:,\boldsymbol{j}^{(z)}}$ can be computed in an iterative way, as described in (5.31):

$$\mathbf{Net}^{(z)}_{:,\boldsymbol{j}^{(z)}} = \left[\mathbf{O}^{(z-1)}_{:,\boldsymbol{j}^{(z-1)}} \ 1\right] \begin{bmatrix} \mathbf{W}^{(z-1)}_{\boldsymbol{j}^{(z-1)},\boldsymbol{j}^{(z)}} \\ \mathbf{W}^{(z-1)}_{\mathbf{k}_{z-1}+1,\boldsymbol{j}^{(z)}} \end{bmatrix} + \sum_{\substack{i=1 \\ i \notin \boldsymbol{j}^{(z-1)}}}^{p} \mathbf{O}^{(z-1)}_{:,\boldsymbol{i}^{(z-1)}} \mathbf{W}^{(z-1)}_{\boldsymbol{i}^{(z-1)},\boldsymbol{j}^{(z)}} \tag{5.31}$$

In this last equation the first product on the right-hand side is computed first since it does not require communication. Assuming that, before this computation is completed, another partition of $\mathbf{O}^{(z-1)}$ becomes available in the j^{th} router, and that happens for the other terms, no actual overhead is produced by the communications.

Using this scheme, an algorithm for worker j can be presented:

Algorithm 18 - Recall operation for worker j

for z=1 to q-2

$$\mathbf{Net}^{(z+1)}_{.,\boldsymbol{j}^{(z+1)}} := \left[\mathbf{O}^{(z)}_{.,\boldsymbol{j}^{(z)}} \;\; 1\right] \begin{bmatrix} \mathbf{W}^{(z)}_{\boldsymbol{j}^{(z)},\boldsymbol{j}^{(z+1)}} \\ \mathbf{W}^{(z)}_{\mathbf{k}_z+1,\boldsymbol{j}^{(z+1)}} \end{bmatrix}$$

$$\boldsymbol{s} := \boldsymbol{j}^{(z)}$$

while $\#(\boldsymbol{s}) \neq \mathbf{k}_z$

To.Worker ? $\mathbf{O}^{(z)}_{.,\boldsymbol{t}}$

$$\mathbf{Net}^{(z+1)}_{.,\boldsymbol{j}^{(z+1)}} := \mathbf{Net}^{(z+1)}_{.,\boldsymbol{j}^{(z+1)}} + \mathbf{O}^{(z)}_{.,\boldsymbol{t}}\mathbf{W}^{(z)}_{\boldsymbol{t},\boldsymbol{j}^{(z+1)}}$$

$$\boldsymbol{s} := \boldsymbol{s} + \boldsymbol{t}$$

end

$$\mathbf{O}^{(z+1)}_{.,\boldsymbol{j}^{(z+1)}} := \mathbf{F}^{(z+1)}(\mathbf{Net}^{(z+1)}_{.,\boldsymbol{j}^{(z+1)}})$$

if $z < (q-2)$

From.Worker ! $\mathbf{O}^{(z+1)}_{.,\boldsymbol{j}^{(z+1)}}$

end

end

$$\mathbf{o}^{(q)} := \mathbf{O}^{(q-1)}_{.,\boldsymbol{j}^{(q-1)}}\mathbf{W}^{(q-1)}_{\boldsymbol{j}^{(q-1)},1}$$

From.Worker ! $\mathbf{o}^{(q)}$

In Algorithm 18 the existence of a master process is assumed, which computes $\mathbf{o}^{(q)}$ as the sum of its p components, each one being obtained by one of the workers. In this algorithm the $\#(\boldsymbol{s})$ denotes the cardinal of set $\boldsymbol{s}$.

To implement the parallel recall operation, the topology of the network of transputers must be specified and an algorithm for the routers must be derived. Experiments were conducted on two topologies, a one-directional and a bi-directional ring, in conjunction with several different routers. The solution that yielded the best results was the simplest topology, together with the simplest router.

The topology employed is shown in Fig. 5.6. As usual, each node denotes a processor, with a worker and a router process. Processor 0 denotes the master process.

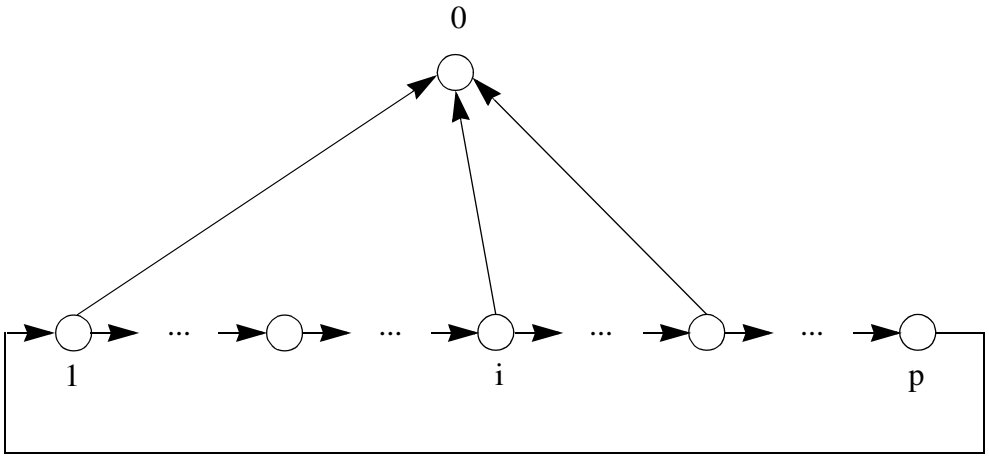


Fig. 5.6 - Ring topology for recall operation

Routers 1 to p must broadcast the data in such a manner that communication of one partition is performed simultaneously with the computation associated with the partition last received, or with the partition computed in the worker. To achieve this operation, the partition used in the current computation is sent in parallel with the receipt of a new partition, the latter being transmitted thereafter to the corresponding worker.

Finally, the output vector is computed in a distributed manner, as indicated by (5.32):

$$\mathbf{o}^{(q)} = W^{(q-1)}_{k_{q-1}+1,\,1}\,1 + \sum_{i=1}^{S}\left(\sum_{j=1}^{p_i} \mathbf{O}^{(q-1)}_{\cdot,j^{(q-1)}} W^{(q-1)}_{j^{(q-1)},\,1}\right) \tag{5.32}$$

The first term on the right-hand side and the outer summation are computed in the master process. The inner summation is computed in the routers. Each router can identify if it should compute the first element in these inner summations by testing one flag (‘first’). A proper assignment of channels, in the configuration file, ensures that the correct number of terms are added in each summation.

The algorithm for the j^{th} router is therefore:

Algorithm 19 - Recall operation for router j

```
for z=2 to q-1
    From.Worker ? O(z)_{.,1(z)}

    for k=1 to p-1
        PAR
            To.Right ! O(z)_{.,k(z)}
            From.Left ? O(z)_{.,(k+1)(z)}
        To.Worker ! O(z)_{.,(k+1)(z)}
    end
end

if not first
    PAR
        From.Left ? t
        From.Worker ? o
    o = o + t
else
    From.Worker ? o
end

To.Master ! o
```

A sequential version of the recall operation was implemented. Three different topologies of MLPs were tried, each one with three different numbers (m) of input patterns. The execution times obtained are shown in Table 13.

Execution time (ms) / m	(3,3,3,1)	(3,6,6,1)	(9,9,9,1)
100	24	54	108
200	47	107	214
300	71	160	320

Table 13 - Execution time for recall operation

Afterwards the parallel version of the recall operation was implemented and executed on three Inmos transputers (p=3). For the case of the MLP with the topology (3,3,3,1) one neuron in each hidden layer was assigned to each processor. This number changes to two and three when the topologies (3,6,6,1) and (9,9,9,1) are considered. The efficiency figures obtained are shown in the next table.

Efficiency / m	(3,3,3,1)	(3,6,6,1)	(9,9,9,1)
100	.84	.93	.96
200	.85	.93	.96
300	.85	.94	.97

Table 14 - Efficiency for recall operation

Good efficiency values are obtained with this parallelization scheme. As expected, efficiency increases as more neurons are assigned to each process.

Subsequently an MLP with the same topology as the ones used in the PID compensator, (5,7,7,1) was used as a test case. The sequential execution time, with 191 input patterns, was 133 ms. The same operation, distributed among 7 transputers, achieved an efficiency of 0.77, which is still a satisfactory value for the number of transputers used.

5.6 Jacobian computation

The last major step of the learning algorithm that remains to be parallelized is Algorithm 1. In the same way as in the recall operation, three possibilities arise for the parallelization of this phase:

a) the input training data is split, in row partitions, among the processes. Each process computes its partition of **J**;

b) the operations involved in the computation of each row of **J** are parallelized;

c) a mixture of both approaches.

The considerations of the last section about the different parallelization approaches are also valid for this phase. In order to have the same structure of data across the different phases of the learning algorithm, approach (b) will be followed.

The main operations to be parallelized in Algorithm 1 are:

1) the ‘for’ loop which computes the Jacobian for the current layer;

2) $\mathbf{d} = \mathbf{d}(\mathbf{W}^{(z-1)})^T$;

3) $\mathbf{d} = \mathbf{d} \times \mathbf{f}'(\mathbf{Net}^{(z-1)}_{i,.})$.

The first and third operations have a straightforward parallelization. In the former the for loop is simply partitioned among the different processors. In the last operation **d** and **Net** are partitioned into column partitions, the necessary operations being applied to this reduced data.

The bottleneck, as far as parallelization is concerned, lies in the second operation. Each process has its own column partition of the different weight matrices $\mathbf{W}^{(z)}$ and of **d**, and is only concerned with the computation of a column partition (possibly with different column indices) of the new **d**, denoted in this section as $\bar{\mathbf{d}}$, as illustrated in the next equation:

$$\bar{\mathbf{D}}_{j^{(z-1)}} = \mathbf{D}_{j^{(z)}}(\mathbf{W}^{(z)}_{j^{(z-1)},\, j^{(z)}})^T + \sum_{\substack{i=1 \\ i \neq j}}^{p} \mathbf{D}_{i^{(z)}}(\mathbf{W}^{(z)}_{j^{(z-1)},\, i^{(z)}})^T = \bar{\mathbf{D}}^{(j,j)} + \sum_{\substack{i=1 \\ i \neq j}}^{p} \bar{\mathbf{D}}^{(j,i)} \qquad (5.33)$$

In this equation **D** represents the matrix composed of the m $\mathbf{d}_i$ vectors.

The first term on the right-hand side of (5.33) can be computed by worker j, since all the data to perform the computation is available to it. However, every i^{th} term of the following summation must be computed by the corresponding i^{th} process, and afterwards the summation made available to process j.

Clearly, the parallel implementation of this operation involves a large amount of communications. To achieve a good efficiency for the parallel implementation of the Jacobian phase these communications must be performed while useful computation is being done. This is possible if another sequence of operations is used for Algorithm 1. It can be seen that, for each hidden layer, except the first, the terms $\bar{\mathbf{D}}^{(i,j)}$ for i=1,...,p and $i \neq j$ can be computed immediately. If this is done, in parallel with the transmission of the $\bar{\mathbf{D}}^{(i,j)}$ terms, performed by a router process, the worker process can compute its own partition of the Jacobian of that layer and the term $\bar{\mathbf{D}}^{(j,j)}$.

A formal algorithm for the worker (and the router) processes will not be presented, since it would require additional complex notation. But it can be easily deduced from Algorithm 1, taking the considerations mentioned above into account.

Having the experience of the recall phase in mind, the topology of the network and the router algorithms were kept as simple as possible. A ring topology was also used for this

phase. The router process must communicate the data and compute the summation on the right-hand side of (5.33). It performs this operation in p-1 iterations. At the first iteration, the matrices $\overline{\mathbf{D}}^{(i,j)}$, i=1,...,p and $i \neq j$ are computed by its worker and sent to the next router in the ring while, in parallel, the matrices $\overline{\mathbf{D}}^{(i,j-1)}$, i=1,...,p and $i \neq j-1$, computed by the left-hand neighbouring worker, are received. This completes the first iteration. In the next iteration the (p-2) matrices $\overline{\mathbf{D}}^{(i,j-1)}$, i=1,...,p and $i \neq j-1$, $i \neq j$ are sent to the right-hand side neighbouring router while receiving in parallel the (p-2) matrices $\overline{\mathbf{D}}^{(i,j-2)}$, and assigning, also in parallel, $\overline{\mathbf{D}}^{(j,j-1)}$ to a temporary variable so that the summation on the right hand-side of (5.33) can be computed iteratively. This concludes the second iteration. All subsequent iterations execute the same parallel input and output operations, while adding, the current matrix $\overline{\mathbf{D}}^{(i,j-k)}$ to the temporary variable. At the end of the (p-1) iterations it is communicated to the worker, which will then complete the computation of $\mathbf{D}_{j^{(z-1)}}$.

When Algorithm 1 was introduced in Chapter 4 it was mentioned that, prior to the computation of the Jacobian, a recall operation should be performed so that the output values of the various neurons were known. For this reason, the execution times and efficiency values shown for this phase include also a complete recall phase. The same topologies and number of input training patterns employed for the recall phase are also used in this phase. The following Table illustrates the execution times obtained.

Execution time (ms) / m	(3,3,3,1)	(3,6,6,1)	(9,9,9,1)
100	38.9	95	205
200	77	189	407
300	115	283	608

Table 15 - Execution time for the recall and Jacobian operations

The corresponding efficiency values, using 3 transputers and the same partitioning employed in last section, are shown in Table 16.

Efficiency / m	(3,3,3,1)	(3,6,6,1)	(9,9,9,1)
100	.85	.88	.91
200	.87	.89	.91
300	.87	.89	.92

Table 16 - Efficiency for recall and jacobian operations

It may be observed that good values of efficiency were obtained for the two combined phases.

5.7 Connecting the blocks

The most important phases of the training algorithm have been parallelized separately. Now it is necessary to interconnect them and to integrate the remaining steps of the learning algorithm.

One common point between the parallel solutions chosen is the topology of the network of transputers. A ring topology is used in every phase discussed, which enables the use of a fixed topology throughout the learning algorithm.

Among the four phases discussed, only one - the computation of the Jacobian matrix - does not employ an additional process in its operation. As noted in section 5.3, the Jacobian phase can be implemented in parallel with the computations of $\boldsymbol{e}$, r and υ. Therefore these calculations will also be performed in this additional process. Since it is also responsible for determining whether the last iteration was successful ($r>0$) or not ($r \leq 0$), this process acts as a master process for the overall network.

The target vector, $\mathbf{t}$, is only needed for the right-hand side of the two least-squares problems that are solved at each iteration of the learning algorithm. This data need only be available to the master process, since it performs the update of the right-hand side in the triangularization phase of the least-square problems. On the other hand, this process does not need to know the Jacobian matrix, $\boldsymbol{J}$, provided that the results obtained for the calculation of the predicted error vector, $\boldsymbol{e}^{\mathrm{p}}$, are made available to the master process. Recalling eq. (4.50):

$$\boldsymbol{e}^{\mathrm{p}} = \boldsymbol{e} - \boldsymbol{J}\mathbf{p}, \tag{5.34}$$

the last term can be computed in a distributed way using the same approach employed in the parallel computation of $\boldsymbol{o}^{(q)}$:

$$\boldsymbol{J}\mathbf{p} = \sum_{i=1}^{3} \left(\sum_{i=1}^{p_i} \boldsymbol{J}_{\cdot, \boldsymbol{v}_j} \mathbf{p}_{\boldsymbol{v}_j} \right) = \sum_{i=1}^{3} \boldsymbol{e}_i^p , \tag{5.35}$$

where $\boldsymbol{v}_j$ denotes the set of nonlinear weights assigned to process j.

In this equation the external summation is computed in the worker process of the master transputer, the p terms $\boldsymbol{J}_{\cdot, \boldsymbol{v}_j} \mathbf{p}_{\boldsymbol{v}_j}$ are computed by each worker process of the other transputers and the $\boldsymbol{e}_i^p$ terms are computed by the corresponding router processes.

The final question to be addressed is how to split the MLP between the p processes. When discussing the parallel implementation of the recall and the Jacobian operations the neurons within each hidden layer were split among these processes. Good efficiencies were achieved for the various test cases where a perfect load balancing between the different processes was obtained. A perfect load balancing can always be achieved if the number of neurons in each hidden layer is a multiple of the number of processes, p, used. This is not, however, always possible. Efficiency should, in principle, decrease as the ratios

$$\frac{\max(n_{\boldsymbol{j}^{(z)}}) - \min(n_{\boldsymbol{j}^{(z)}})}{\mathbf{k}_z / p}, \text{ for } 1 < z < q \tag{5.36}$$

increase. In the last equation $n_{\boldsymbol{j}^{(z)}}$ denotes the number of neurons on the z^{th} layer allocated to process j.

Assuming a perfect load balancing for the recall and the Jacobian phases, the partitioning of columns for the least-squares problems becomes straightforward. For the computation of $\hat{\mathbf{u}}$, each j^{th} processor has its $\boldsymbol{j}^{(q-1)}$ set of output vectors of the last nonlinear hidden layer to work with. In the case of the computation of the LM increment, the v_j set of columns of the Jacobian matrix $\boldsymbol{J}$ is used by processor j. This has the advantage of making any additional transfer of data between the processors unnecessary.

When a perfect load balancing can not be achieved by distributing the neurons between processors, a different solution can be used. As the most computationally intensive task in the training algorithm is the LM update, an even distribution of columns of $\boldsymbol{J}$ among the p processors is desirable. This can be obtained by prior distribution of the columns of $\boldsymbol{J}$ among the processors, before computing $\mathbf{p}$, and then redistributing the solution obtained.

To specify the complete parallel training algorithm, perfect load balancing is assumed. Algorithms 20 and 21 specify, in a simplified manner, the operations executed by the worker processes of the master and the j^{th} transputers of the rest of the network. In these algorithms the existence of another process responsible for the user interface is assumed. In each iteration the norm of the error vector and the current value of υ are sent from the master process to this interface process, which in turn outputs them to the screen. The interface process also monitors the keyboard, so that the user can interrupt the learning algorithm at any given time. The master, and the other p transputers, test this possibility by examining, at the end of each iteration, the value of a flag, 'go.on'.

Algorithm 20 - Worker of master transputer

```
... initializations
while go.on
    ... compute Q_{I_u}^T t                                  -- see section 5.4.2
    ... input o_i^(q), i = 1, ..., 3  and compute  o^(q) := sum_{i=1}^{3} o_i^(q)
    ... e := t - o^(q) ; compute ||e||
    ... compute r (4.53) and update υ                        -- see Algorithm 6
    s := FALSE
     if r>0
        e_old := e; s := TRUE
    end
    From.Worker ! υ , s
    ... compute Q_{I_v}^T e.old                              -- see section 5.4.2
    ... input e_i^p, i = 1, ..., 3 , compute e^p := e - sum_{i=1}^{3} e_i^p  and ||e^p||
    ... output results of the current iteration to its router
    To.Worker ? go.on
end
```

In both algorithms, all the calculations involved in the computation of the optimal values of the linear weights are performed in double precision, to avoid possible numerical problems. All communication related to this phase is, however, made in single precision, by converting the data before and after communication is performed.

It is evident that the worker of the master transputer has a very small computational load. For this reason the master transputer can even be the host transputer, which is why the master is not taken into account in efficiency calculations.

.

Algorithm 21 - Worker of transputer j

... initializations
while go.on
... compute $\mathbf{O}^{(q-1)}_{.,\boldsymbol{n}_j^{(q-1)}}$ -- see section 5.5
... compute $\hat{u}_{\boldsymbol{n}_j^{(q-1)}}$ -- see section 5.4.2.3
... compute $\boldsymbol{o}^{(q)} := \mathbf{O}^{(q-1)}_{.,\boldsymbol{n}_j^{(q-1)}} \hat{u}_{\boldsymbol{n}_j^{(q-1)}}$ and send it
... compute $\boldsymbol{J}_{\boldsymbol{v}_j}$ -- see section 5.6
To.Worker ? υ , s
if s
$\mathbf{v.old}_{\boldsymbol{v}_j} := \mathbf{v}_{\boldsymbol{v}_j}$; $\boldsymbol{J.old}_{\boldsymbol{v}_j} := \boldsymbol{J}_{\boldsymbol{v}_j}$
end
... compute $\mathbf{p}_{\boldsymbol{v}_j}$ -- see section 5.4.2.3
$\mathbf{v}_{\boldsymbol{v}_j} := \mathbf{v.old}_{\boldsymbol{v}_j} + \mathbf{p}_{\boldsymbol{v}_j}$
... compute $\boldsymbol{J.old}_{\boldsymbol{v}_j} \mathbf{p}_{\boldsymbol{v}_j}$ and send it
To.Worker ? go.on
end

The sequential training algorithm was implemented in Occam and executed for the same test cases as employed in the previous two sections. The learning algorithm is not applicable to the case of the MLP (9,9,9,1) with m=100, since the number of nonlinear variables is greater than the number of training patterns. Table 17 shows the execution times of each iteration of the training algorithm, obtained as an average of the first 20 iterations.

Execution time (ms) / m	(3,3,3,1)	(3,6,6,1)	(9,9,9,1)
100	413	2 558	*
200	818	5 055	34 600
300	1 220	7 550	51 680

Table 17 - Execution times for each iteration of the learning algorithm

Subsequently, the parallel learning algorithm was executed in three transputers (p=3), the MLPs being partitioned in the same way as in the two last Chapters. The efficiency values obtained are shown in Table 18.

Efficiency / m	(3,3,3,1)	(3,6,6,1)	(9,9,9,1)
100	.82	.92	*
200	.85	.93	.95
300	.86	.93	.95

Table 18 - Efficiency for learning algorithm

The efficiencies obtained are very good. As expected, efficiency marginally increases with the number of input training patterns and is strongly dependent on the number of nonlinear variables per processor.

For completion, one MLP with the topology employed in the PID compensation was also used as a test case. The sequential execution times, for different numbers of input patterns, is shown in the next Table.

m	100	191	200	300
Execution time (ms)	5 370	10 135	10 660	15 930

Table 19 - Execution times for each iteration of the learning algorithm (5,7,7,1)

Partitioning this MLP among seven processors (p=7), the efficiency values obtained are shown in Table 20.

m	100	191	200	300
Efficiency	.84	.85	.85	.86

Table 20 - Efficiency for learning algorithm (5,7,7,1)

As expected, efficiency drops when more processors are used. In section 5.4.2.1 it has been shown that, for the standard least squares solution, the efficiency of the proposed parallelization scheme was practically independent of the number of rows of the matrix to be triangularized, varying slightly with the number of processors used and dominated by the

dependence on the number of columns per processor. Specializing this parallel solution to the computation of LM updates, an inspection of Table 12 shows a stronger dependence of efficiency on the number of rows of **A**. Finally, comparing the results of tables 18 and 20 it can be concluded that the efficiency of the complete training algorithm also depends on the number of transputers used.

As can be seen, the parallelization of the MLP (5,7,7,1), using 7 transputers achieves similar results to those obtained for the MLP (3,3,3,1). However, the number of columns of $\boldsymbol{J}$ per transputer in the former case (14) is almost double that of the latter case (8).

It can be concluded then that the efficiency of the complete training algorithm depends on more factors than just the number of nonlinear parameters per processor. However, as a rough first approximation, this figure can be used, and efficient solutions can be expected if this number is high.

5.8 Conclusions

A parallel implementation of the training algorithm chosen in Chapter 4 was investigated in this Chapter. The parallel algorithm was coded in Occam and executed on T800 Inmos transputers.

The main blocks of computation in each iteration were identified and separately parallelized. The main effort was concentrated, however, on the most computationally intensive step of the learning algorithm, the computation of the Levenberg-Marquardt update.

For accuracy reasons, this step was formulated as a least-squares problem and solved by means of a QR factorization. By modifying the sequence of operations that are performed in a known parallel solution for this type of problem, a boost in efficiency was obtained. As the LM update is the most time consuming operation in the training algorithm, this new parallelization scheme is the major factor in obtaining good efficiencies for the overall training algorithm.

A few points still need to be addressed before this parallel implementation has all the facilities found in the sequential version. In the parallel least squares solution no test of singularity is performed. This testing is easy to be introduced, by comparing, in each iteration of the triangularization phase, σ, computed in Algorithm 10, with a suitable tolerance value, usually function of the dimensions of the matrix to be triangularized and the floating point accuracy. Also, this parallel algorithm currently works only if perfect load balancing is obtained. This is because the code of each worker and router processes, in the least squares solution phase, assumes implicitly the column partitioning described in page 119. Extending this solution to situations where the number of columns per processor is not the same

throughout is a straightforward operation. If each worker and the corresponding router receive, explicitly, the indices of the columns allocated to them, then their code can be slightly modified so that they can handle any column partitioning. Finally, the only convergence criterion of the parallel algorithm is the accuracy (norm of the error vector) achieved by the mapping. More objective criteria, like the ones employed in the sequential training algorithms, should be incorporated in the parallel algorithm.The only difficulty in implementing the criteria (4.70) to (4.72) lies in the fact that, in the parallel algorithm, the gradient vector is not computed. Two possibilities arise for its parallel computation:- either to introduce another block in the algorithm, specifically for its calculation, or, profiting of the fact that $\mathbf{g}_{\psi} = -\boldsymbol{J}^{\mathrm{T}}\boldsymbol{e}$ (see (4.62) in page 97), broadcast the vector $\boldsymbol{e}$, computed by the master processor, through all the workers, and calculate the gradient vector in a distributed fashion. This last possibility would, in principle, require less computation time.

The improvements in technology and the use of a better learning algorithm enabled the training time of the MLPs used in the PID compensation to be reduced from three days to approximately half an hour. By using 7 T800 Inmos transputers this time was further reduced to roughly five minutes.

This small training time can be decreased further. Employing more transputers is one solution. As has been shown, even with 20 transputers used in the computation of the least squares solution of a problem with similar dimensions, speed-up is still far from being exhausted. These results should also be reflected in the overall training algorithm, since the LM update is the dominant task of the algorithm.

Another possibility for reducing the training time involves yet another improvement in technology. The T800 transputer, whose performance is quoted [15] at 10 MIPS and 1.5 Mflops, could simply be replaced by the new T9000, expected to be released soon. This new generation of transputers [175], capable of peak performances of 200 MIPS and 25 Mflops should speedup the learning process by another order of magnitude, assuming that comparable values of efficiency are obtained. This assumption is not unrealistic, since communication bandwidth is also projected to increase by a factor of 10.

Fast training is important, in many ways, for the new approach to PID autotuning. Larger networks, potentially capable of obtaining better accuracy in the approximations, can be trained in a reasonable amount of time. Different topologies of MLPs can be tried out to determine which one delivers the best performance. The results obtained using different criteria in the determination of the PID optimal values can be compared without a lengthy training process being needed. Neural network PID controllers for plants characterized by

different types of transfer functions can be synthesized more quickly. On the whole, small training times make the connectionist PID autotuning approach more attractive.

However, the validity of this approach is still based on off-line simulations. In the next Chapter one step towards reality will be taken and a real-time implementation of this new PID autotuning approach will be discussed.

Chapter 6

Real-Time Implementation

6.1 Introduction

In Chapters 4 and 5 of this thesis one major problem of the connectionist approach to PID autotuning, the training of multilayer perceptrons, was investigated. The results of this research considerably reduce the burden of the off-line phase of this technique and make available the means to perform necessary experimentation.

The results shown in Chapter 3 were obtained from off-line simulations. In this Chapter a step towards reality will be taken and a real-time implementation of the new technique will be described. To enable the necessary tests for the proposed technique to be done, in a user-friendly way, a real-time system [176] was implemented in Occam, on an array of Inmos transputers. This Chapter describes the system architecture and reports on the results obtained.

The outline of this Chapter is as follows:

Section 6.2 introduces a preliminary architecture for the system and discusses the motivations behind its choice.

Section 6.3 gives an overview of the facilities of the system, describes its operational modes and specifies its input/output devices.

The real-time system is partitioned into different blocks, each one implemented as an Occam process. Section 6.4 describes each one of these processes, a special emphasis being put on the process responsible for the automatic PID tuning. Accordingly, section 6.4.1 introduces the user-interface process, the controller process is detailed in section 6.4.2. and section 6.4.3. describes the block responsible for the plant simulation. Section 6.4.4 details the operations needed in the adaptation block. Finally, section 6.4.5 presents a graphical overview of the system that has been implemented.

Results obtained with the real-time system are given in section 6.5.

Conclusions are drawn in section 6.6.

6.2 Preliminary remarks

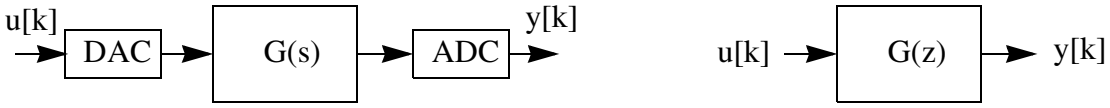
At this stage of the work, we wish to assess the real-time performance of the neural

PID autotuner for plants with varying time delays and number, and location, of poles. We wish to allow these changes to be effected easily. On the other hand, the final aim of this work is the control of real plants.

The use of an analog simulator to represent the different plants was considered. The analog computers available, however, could not simulate time-delays, which would have to be implemented digitally. Also, the practical implementation of plants with different transfer functions and varying time constants on an analog computer, although not intrinsically difficult, would be a time-consuming operation. For these reasons, it was decided to simulate the plants digitally, but to construct the system in such a way that an easy update path to the control of real plants was provided.

The use of the transputer and the Occam language allowed an easy implementation of such a system. By isolating the digital simulation of the plant in a separate process, which communicates with the controller process by means of channels, it is then possible to implement such a scheme.

Fig. 6.7 shows the simplified pseudo-Occam code for the controller process in two different situations. In case (a) a real plant is assumed. To compute the control signal, u[k], the controller must sample the output of the process. Typically this is done by sending, via an external link, the channel identifier to an ADC board and then inputting the sampled value via another physical link. It is assumed that the control value is afterwards computed, and finally applied to the plant. This is done by sending the control signal to a DAC board via a link, and then, if several output channels are available in the DAC, by specifying which output channel to use.




```
to.adc ! adc.channel
from.adc ? y
... compute control (u[k])
to.dac ! u
to.dac ! dac.channel
```

a) real process

```
from.dig.sim ? y
... compute control (u[k])
to.dig.sim ! u
```

b) digitally simulated process

Fig. 6.7 - Pseudo-Occam code for the controller operation

Case (b) illustrates the situation where the plant is digitally simulated in one process. The only differences in the code of the controller process are the omission of the output operations specifying the ADC and DAC channels. It is then very simple to update the real time system to the control of real plants.

The PID autotuner technique is intended to be applied using existing PID controllers. To keep the controller operations isolated from the tuning operations, it was decided to implement the controller and the autotuner as separate Occam processes. Additionally, one process responsible for the user interface is also needed.

Fig. 6.8 illustrates a preliminary architecture for the real-time system. The shaded boxes denote processes. In this figure, G(z), $G_{KI}(z)$ and $G_D(z)$ denote the Z-transforms of the plant, the forward and the feedback compensator, respectively.

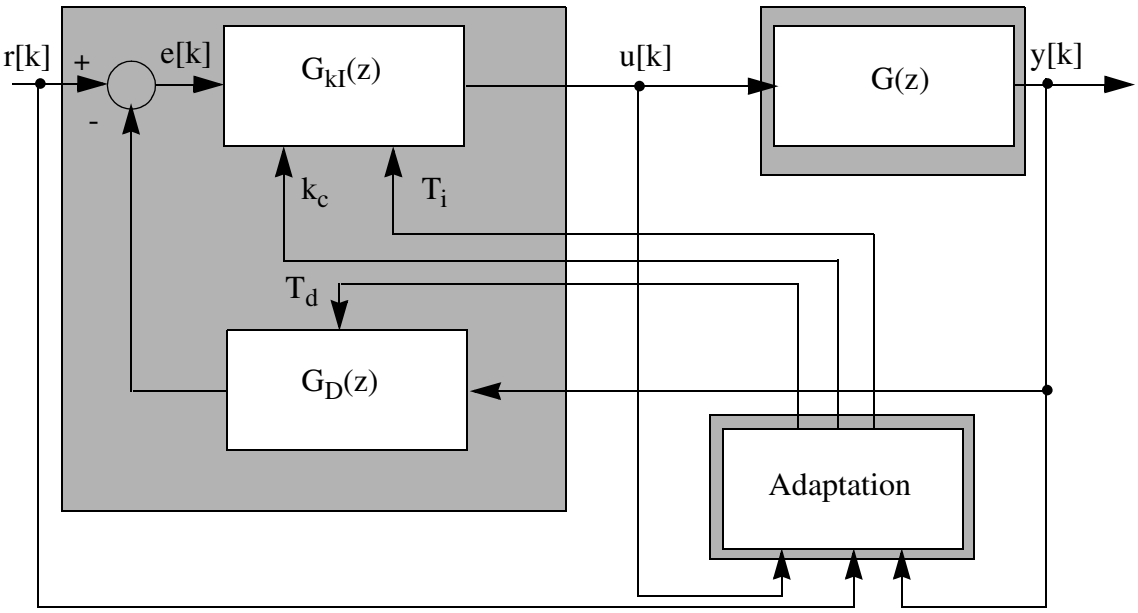


Fig. 6.8 - Preliminary architecture of the real-time system

Thus the intended system is composed of these four processes, which are denoted throughout this Chapter as: the user interface, the controller, the adaptation and the plant. Each process executes on a separate Inmos transputer. The use of multiple transputers was not specifically necessary, but provided flexibility for the testing procedure.

6.3 Operational modes and input/output devices

One of the advantages of the proposed technique of PID autotuning is the possibility of being applied in open and closed loop configurations. As explained in Chapter 3, different signals and different sets of equations are used to derive the PID values according to the loop

configuration. The status of the loop is specified by the user, at any time, and must be made available to the controller and the adaptation processes. The possibility of automatically closing the loop once a set of PID values has been computed, from the open loop step response, is also provided.

The system has two different operation modes: fixed and adaptive. These can be changed by the user, at any time. In fixed mode, which is only relevant when the loop is closed, the PID parameters remain fixed. This enables the user to manually tune the PID and to analyse the performance of a fixed PID when the plant changes. In adaptive mode, at each step, the open or closed loop response of the system is analysed by the adaptation block and new PID parameters are computed.

The implemented system was designed to include a convenient user interface for the specification of the plant. Normally, this is done by input from the keyboard of the host computer. However, there is also a facility whereby a list of different plants, each one with an associated time interval, can be read from the disk of the host computer during the initialization phase of the system. This way it is possible to specify, automatically, a sequence of different plants and the exact times when changes must occur. This facility is used to produce the results shown in section 6.5.

The input to the system is a square wave, taken from a signal generator. The number of outputs depends on the loop configuration. If in open loop, the output of the simulated plant is sent to one channel of a DAC board. In closed loop, both the output and the control signal are sent to two different channels of the DAC board. These signals are then available to be monitored on an oscilloscope.

6.4 System architecture

In this section a brief description of the operations performed by each constituent process of the real-time system will be made. The main emphasis will be given to the description of the adaptation process. We shall start by describing the user interface process.

6.4.1 The user interface

This process executes in the host transputer, an Inmos T414 [15] running at a clock frequency of 15 MHz. It is responsible, as its name suggests, for the interaction of the system with the user.

It allows the user to specify, at any time, using the computer keyboard, the following information:

i) The plant parameters. The user can change these values at any time, but the new parameters will only be made active when all the changes have been introduced.

ii) Open or closed loop configuration.

iii) Adaptive or fixed mode.

iv) The PID parameters, if in fixed mode. As in the case of the plant parameters, these will only be activated once all three PID parameters are introduced.

v) The sampling time.

This process also outputs to the monitor screen the following information:

i) The current plant parameters.

ii) In the case of adaptive mode, the current computed values of T_T (3.74) and the $F(\sigma)$ parameters (3.76). Their actual computation is described in section 6.4.4.

iii) The current PID parameters.

iv) The status of the system: normal or in error. Errors currently detected are:

a) In adaptive mode, the occurrence of a new input step before the system has settled within user-specified limits. This is necessary for the satisfactory performance of the system, since to compute the PID values it is necessary that the response of the stimulating step input has had sufficient time to die away.

b) Instability.

c) The user-specified sampling frequency can not be met.

If the facility of automatically specifying a list of plants has been selected, this process also outputs the parameters of the new plant to the plant simulator process, upon receipt of a signal from the controller process.

6.4.2 The controller

This process executes on an Inmos T800 transputer [15] with a clock frequency of 25 MHz. The clock used to sample the system is implemented in this process. This is easily done by using the timer instructions of the transputer, as specified in Algorithm 21:

Algorithm 21 - Implementing the system clock

```
WHILE TRUE
    clock ? now
    ... controller operations
    clock ? AFTER (now PLUS Ts)
```

In this algorithm, ‘clock’ is a high priority timer variable. The integer variable Ts is

the sampling time, in microseconds.

The controller process is responsible for:

i) Sampling the reference signal and the output signal at the user-specified sampling frequency. As the plant is digitally simulated, its output value at the current instant is obtained from the plant process by a channel input operation. The value of the reference signal is read from an ADC board.

ii) Detecting and dealing with instability. This situation is detected by testing for an output signal that is beyond the output range ([0, 10]) of the DAC employed. If this situation occurs, the loop is automatically opened and an input of 0 volts is supplied to the plant until it settles. The system remains in open loop until the user changes its configuration.

iii) Sending the plant input to the plant process. In the case of open loop, the reference signal is output to the plant process. In the closed loop case, the control value is computed, by solving the controller difference equations, and sent to the plant process.

iv) Computing the digital equivalents of the forward and feedback compensators, whenever there is a change in the PID values or a new sampling time is specified.

iv) Detecting whether the sampling frequency has been met. This is easily done by reading the clock channel, prior to execution of the last instruction in Algorithm 21. If the difference between the value read and the present value of the variable is greater than the sampling time, then an error message is sent to the user-interface process.

v) Signalling the user-interface process that a new plant must be activated, if the facility of automatically changing plants has been chosen.

The forward compensator is discretized using a Z-transform approximation. If the forward compensator is described by the Laplace transform:

$$G_{KI}(s) = k_c\left(1 + \frac{1}{sT_I}\right) \tag{6.37}$$

then its digital equivalent, using a Z-transform approximation is:

$$G_{KI}(z) = k_c\left(1 + \frac{T_s}{T_I(z-1)}\right) = \frac{k_c z + (k_c(T_s/T_i - 1))}{z-1} \tag{6.38}$$

To implement the forward compensator, the velocity form (see Section 3.2) is actually employed. $\Delta u[k]$ is computed as:

$$\begin{aligned}\Delta u[k] &= u[k] - u[k-1] \\ &= \Delta u[k-1] + k_c\left(e[k] + \left(\frac{T_s}{T_i} - 2\right)e[k-1] - \left(\frac{T_s}{T_i} - 1\right)e[k-2]\right)\end{aligned} \tag{6.39}$$

The feedback compensator is discretized using a Tustin approximation [177]. Using this discretization method, the feedback compensator, with Laplace transform:

$$G_D(s) = \frac{1 + sT_d}{1 + s\dfrac{T_d}{\eta}}, \tag{6.40}$$

is discretized as:

$$G_D(z) = G_D(s)\Big|_{s = \frac{2}{T_s}\frac{z-1}{z+1}} = \frac{z\dfrac{T_s + 2T_d}{T_s\eta + 2T_d}\eta + \dfrac{T_s - 2T_d}{T_s\eta + 2T_d}\eta}{z + \dfrac{T_s\eta - 2T_d}{T_s\eta + 2T_d}} \tag{6.41}$$

6.4.3 The plant

This process executes in an Inmos T800 transputer [15] with a clock frequency of 25 MHz. It performs the following tasks:

i) At each sampling instant, the current value of the output, computed during the previous iteration, is sent to the controller process.

ii) The current input of the plant is afterwards received from the controller processor. Both the current values of the input and output are sent to two channels of a DAC.

iii) The output value for the next sample is computed by solving the difference equation representing the plant.

iv) Whenever there is a change in either the sampling frequency or in the plant transfer function, the digital equivalent of the plant is recomputed.

The plant is discretized using a zero-pole mapping equivalent [177]. If the Laplace transform of the plant is:

$$G(s) = k_p e^{-Ls} \frac{\prod_{i=1}^{\bar{n}} (1 + T_{z_i} s)}{\prod_{i=1}^{n_p} (1 + T_{p_i} s)} \tag{6.42}$$

then its Z-transform, using a zero-pole mapping equivalent can be obtained as:

$$G(z) = k\frac{(z+1)^{n_p-n_z-1}\prod_{i=1}^{n_z}\left(z-e^{-\frac{T_s}{T_{z_i}}}\right)}{z^n\prod_{i=1}^{n_p}\left(z-e^{-\frac{T_s}{T_{p_i}}}\right)} \tag{6.43}$$

where k is given by:

$$k = k_p\frac{\prod_{i=1}^{n_p}\left(1+e^{-\frac{T_s}{T_{p_i}}}\right)}{2^{n_p-n_z-1}\prod_{i=1}^{n_z}\left(1+e^{-\frac{T_s}{T_{z_i}}}\right)} \tag{6.44}$$

and n is the nearest integer to L/T_s.

6.4.4 The adaptation

This block of operations, as will be explained shortly, is divided into two processes, each one executing in a T800 Inmos transputer [15], running at 25MHz. The purpose of this block is to determine the new set of PID values, based on the analysis of the response of the system to an input step. This requires several operations.

If the system is in adaptive mode, it receives from the controller process at each sampling instant, the values of the reference and the control or the output, according to the current loop configuration (closed or open loop).

The reference is used to detect the occurrence of a step and to compute its amplitude. As the reference is a square wave, the PID values are computed at the instant when a new input step is detected. Before doing so, the response associated with the previous step (used to identify the plant) must have settled within a user-specified tolerance. If this did not happen, the previous step cannot be employed to identify the plant, and neither can the current step because the system response is now a mixture of the transients associated with the last and the current steps; consequently an accurate computation of the identification indices $F(\sigma)$ cannot be guaranteed.

To ensure settling, the present and the last *n* values of the control or the output are checked, within each sampling interval. The arithmetic mean of these samples is computed,

recursively. It is then determined whether they all lie within a certain range (a user defined percentage) around the computed mean. If this condition is satisfied, it is accepted that settling has occurred; otherwise the same process is repeated at the next sampling interval. The number n of past samples is initialized to a user predefined constant. Whenever an underdamped response is detected, n is updated to the number of samples within the period of the signal.

Another task performed by this process is the computation of the scaling factor T_T, defined by (3.74). To calculate T_T, the integrals S(0), defined by (3.78), or $S_u(0)$, defined by (3.83), are computed recursively, using a trapezoidal formula.

S(0) for example (the computation of $S_u(0)$, for the closed loop case, is analogous) is obtained using (6.45):

$$S(0) \approx y(\infty)kT_s - T_s \sum_{i=1}^{k} \frac{y[i] + y[i-1]}{2} \qquad (6.45)$$

The summation in (6.45) is computed recursively. Currently, the computation of S(0) is continued until a new step is detected. Consequently, T_T, the identification measures $F(\sigma)$, and the PID values are only computed after the new step has been detected. This should be changed, so that the actual computation of S(0) takes place when settling has been detected.

After obtaining S(0), T_T can be computed. In the open-loop case eq. (3.79) is used; in the closed-loop case, (3.84) is employed.

The operations described so far can be performed within one sample period, without affecting the performance of the overall system. The same, however, can not be said about the rest of the operations that the adaptation block must perform, the computation of $F(\sigma)$ and subsequently, the PID parameters.

We note that, as $S(\sigma)$ or $S_u(\sigma)$ (which are needed to compute the identification measures) depend on T_T, they cannot be computed recursively. As a large amount of data is normally required to compute these integrals, this operation cannot be performed within one sampling period. As there were several transputers available, this problem was solved by splitting the adaptation block between two processes (running on different T800 transputers), the second of these being responsible for the actual computation of $F(\sigma)$, and subsequently, the PID parameters.

This second process receives from the first one the values of T_s, T_T, k_p and $y(\infty)$ or $u(\infty)$, according to the loop configuration. Additionally it also receives the samples of the output or the control signal. At present, the number of these samples is either the total number

of samples within the previous step or a user predefined maximum. In the same way as remarked when describing the computation of S(0), this should be modified so that $F(\sigma)$ would be computed using the samples available until settling had been detected.

To compute the identification measures, the integrals $S(\sigma)$, defined by (3.78), or $S_u(\sigma)$, defined by (3.83), are computed, using a trapezoidal formula.

Illustrating for the open loop case, $S(\sigma)$ is calculated as:

$$S(\sigma) \approx T_s \sum_{i=1}^{k} e^{-\left(\frac{T_s}{2} + (i-1)T_s\right)\frac{\sigma}{T_T}} \left(y[\infty] - \left(\frac{y[i] + y[i-1]}{2} \right) \right) \tag{6.46}$$

After the integrals have been obtained, the values $F(\sigma)$ are computed either using (3.80) or (3.85), according to the current loop configuration.

Using these last values, the scaled PID values are obtained by executing, concurrently, three processes which implement the recall operation of multilayer perceptrons. This code was already available from the work described in Chapter 5. The scaled PID values are output by the MLPs. They are then transformed back to their original form, using (3.96), and transmitted to the first adaptation process. This last process sends them to the controller process, where they are employed as the current PID parameters.

6.4.5 System overview

An overview of the complete real-time system is shown in the next figure. The white boxes denote processors. The dotted boxes denote input and output devices. The dashed boxes represent the digital-to-analog and the analog-to-digital boards.

6.5 Results

When testing the real-time system it was found that a high sampling rate was needed to reproduce the responses obtained in the off-line simulations described in Chapter 3. Using the well known rule of thumb for selecting the sampling frequency of 20 times the closed-loop bandwidth [177] produced, in the great majority of the cases, responses very different from the ones obtained in the off-line simulation.

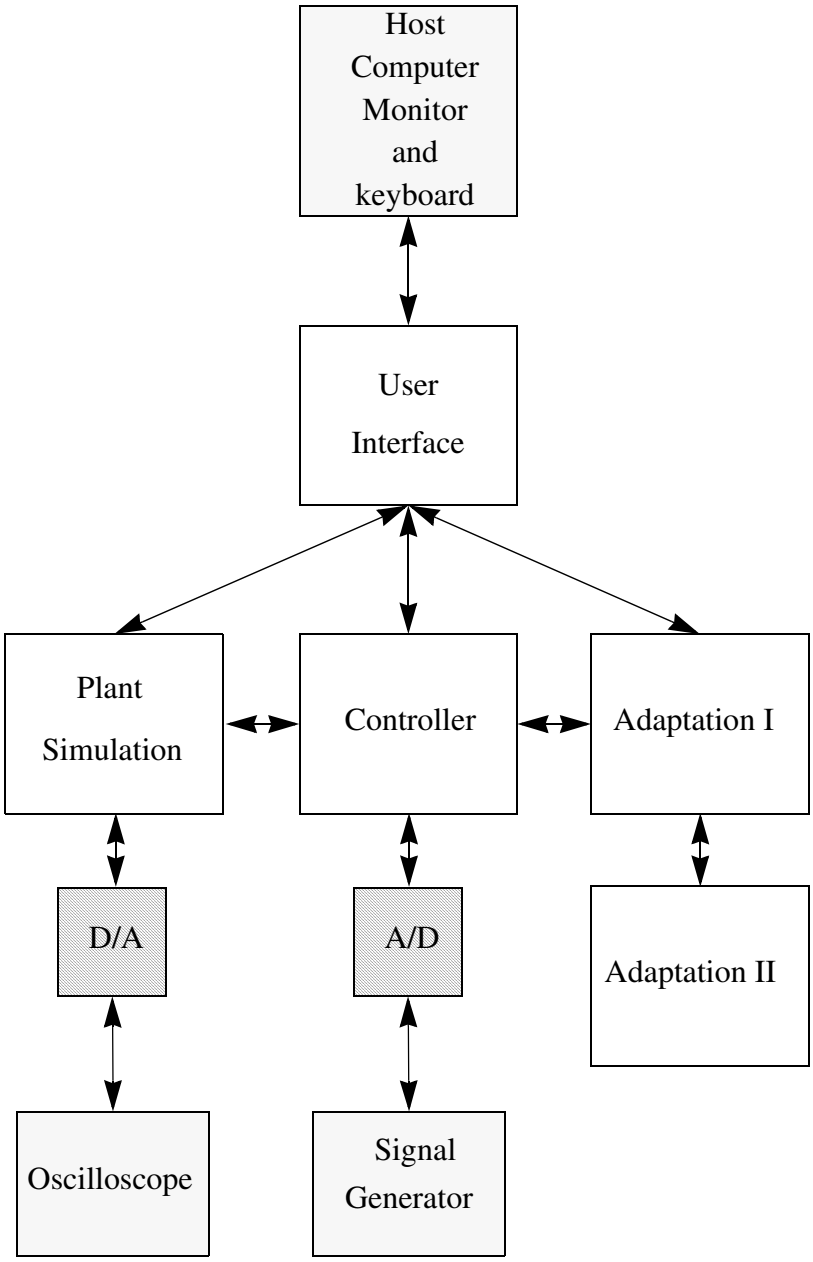


Fig. 6.9 - Overview of the real-time system

As an example we shall consider a three-pole plant with the transfer function:

$$G(s) = \frac{1}{(s + 0.8)(s + 8)(s + 12)} \tag{6.47}$$

Using PID parameters derived using the closed-loop Ziegler-Nichols tuning rule, the closed-loop bandwidth is 14 rad/s. Fig. 6.10 compares the continuous closed-loop output and the discrete closed-loop outputs, using different sampling rates.These results were obtained in off-line simulations using MATLAB.

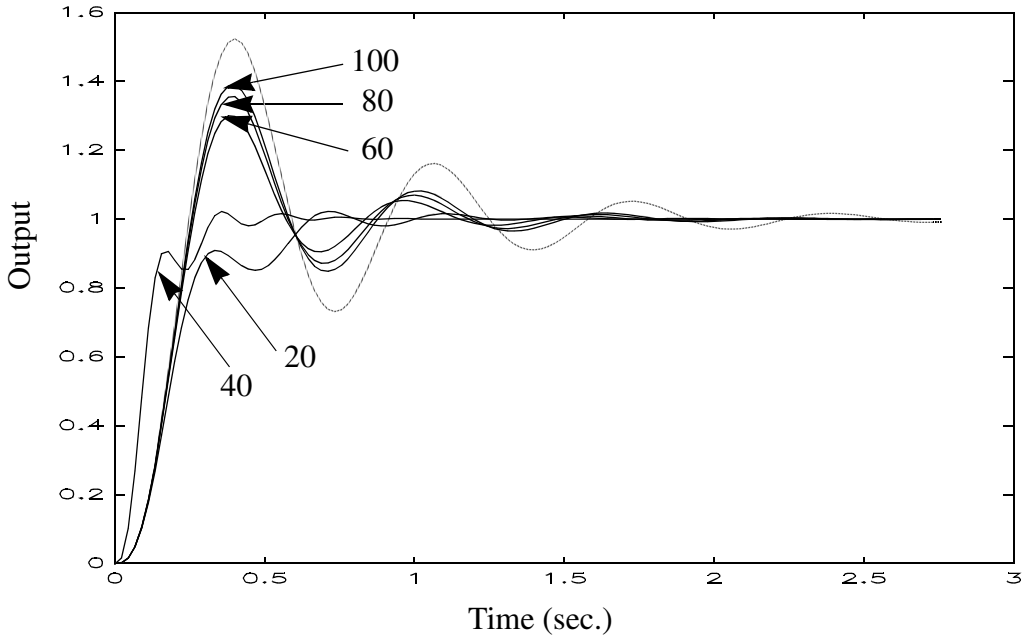


Fig. 6.10 - Responses obtained with different sampling frequencies

The dashed line denotes the output obtained by first calculating the continuous closed-loop transfer function and then applying the MATLAB ‘step’ function, where a sampling frequency 20 times greater than the closed-loop bandwidth was employed. The ‘step’ function first discretizes the continuous transfer function (assuming a zero-order hold in the input) using a exponential matrix and then solves the resulting difference equation. The results shown in Chapter 3, for the cases of plants without delay-time, were obtained using this procedure.

The solid lines in Fig. 6.10 denote the closed-loop outputs obtained by first discretizing each component of the closed-loop, using, for each component, the methods referred to in previous sections, then obtaining the Z-transform of the closed-loop and finally solving the resulting difference equation. The values indicated in Fig. 6.10 denote the multiple of the continuous closed-loop bandwidth used for obtaining the sampling frequency.

By inspection, it is clear that the responses obtained are remarkably different from the continuous case, if a factor less than 60 is used for the sampling frequency. This implies that a high sampling rate must be used in the real-time system, which has the drawback of enlarging the number of samples needed for the plant identification and therefore the computation time.

This problem should be further studied, by employing different discretization methods for each component of the loop and by analysing the closed-loop singularities, in both continuous and discrete cases.

To test the real-time performance of the new PID autotuning technique the parameters of the multilayer perceptrons were initialized with the same values employed in the off-line simulation described in Chapter 3.

Several tests were performed. Four of them are shown here. Fig. 6.11 illustrates the case of the control of a four-pole plant, with transfer function $G(s) = \frac{1}{(1+0.1s)^4}$. In this and subsequent figures the solid line denotes the reference signal and the dashed line the output signal. The system was started in open loop, with the option of automatically closing the loop once the PID parameters are obtained. A sampling period of 5ms was employed.

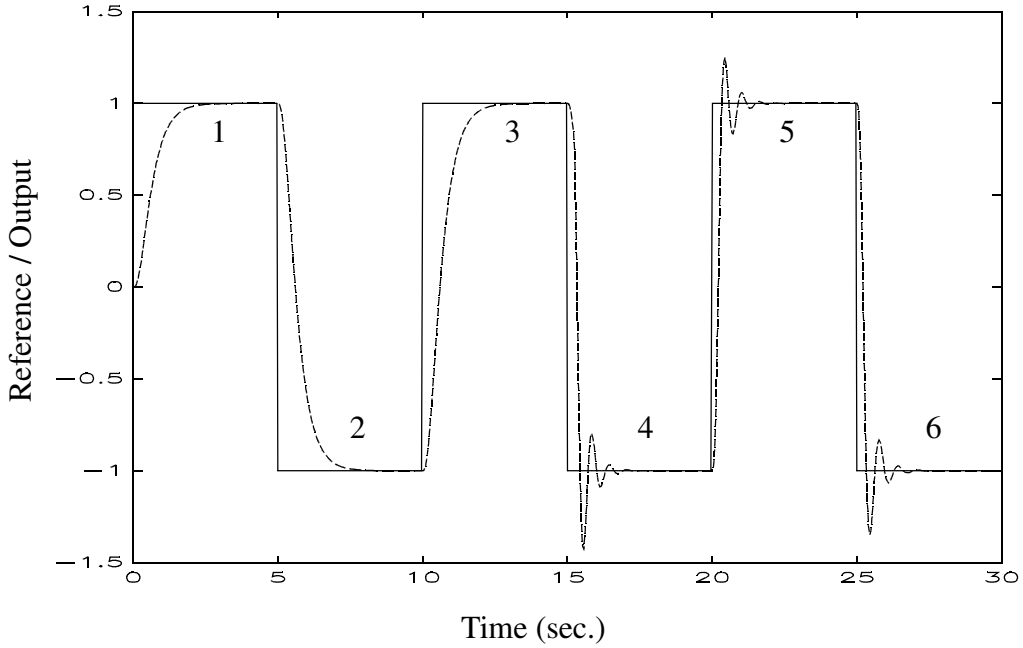


Fig. 6.11 - Example 1

During the second step, the mode was changed from fixed to adaptive. The response of the third step is therefore used to identify the plant. When the fourth step is detected, T_T is computed by the first adaptation process and the relevant information is passed to the second adaptation block. Shortly afterwards the PID values are obtained, transmitted to the controller process via the first adaptation process, and the loop automatically closed. The performance of the tuning can be assessed from the last two steps where it is clear that a well damped response is obtained.

The next Figure shows the control of a plant with transfer function $G(s) = \frac{e^{-0.05s}}{1+0.1s}$. The sampling period employed was 2ms. In this case the system was initialized in closed loop,

with the PID parameters manually chosen according to the closed-loop Ziegler-Nichols tuning rule for the specified plant. During the third step the operation mode changed from fixed to adaptive. During the fourth step the plant was identified. Shortly after the fifth step has been detected, the new PID values, have been made active.

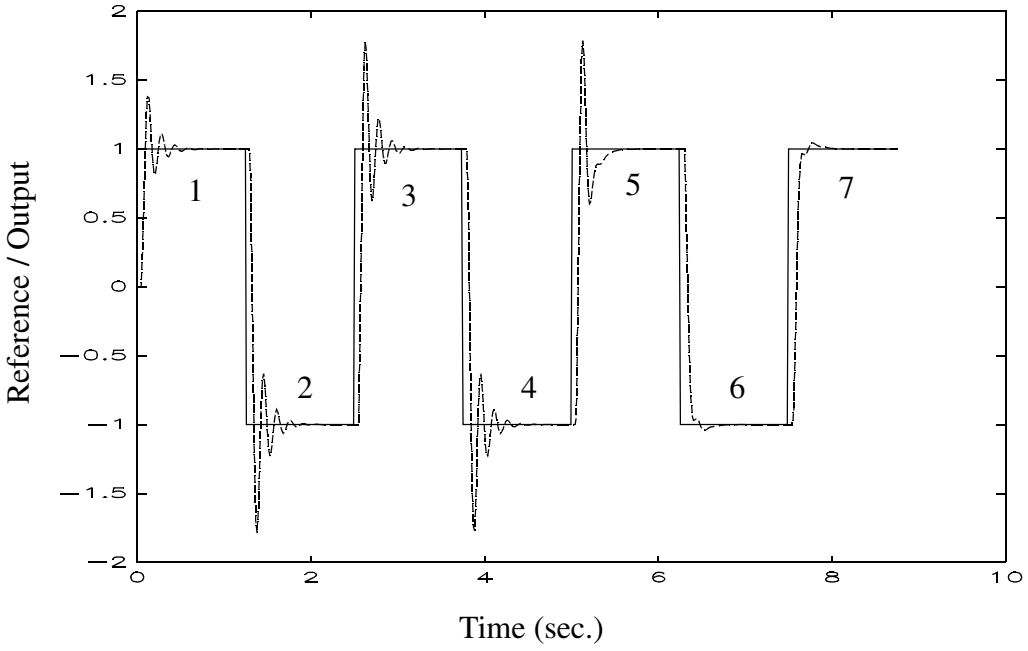


Fig. 6.12 - Example 2

Steps six and seven illustrate the performance obtained with the new tuning, where it can be seen that a better damped response has been obtained.

Fig. 6.13 illustrates the application of the new technique to a time-varying four-pole plant. The sampling period used was 5 ms. In this example the system was started in open loop, adaptive mode, with the option of closing the loop automatically once the PID values were obtained. The initial transfer function of the plant is:

$$G_1(s) = \frac{1}{(1+0.1s)^2(1+0.2s)^2}$$

In the second step the plant is identified. The PID values are computed in the third step and the loop is automatically closed. Steps four and five illustrate the tuning obtained from open loop identification.

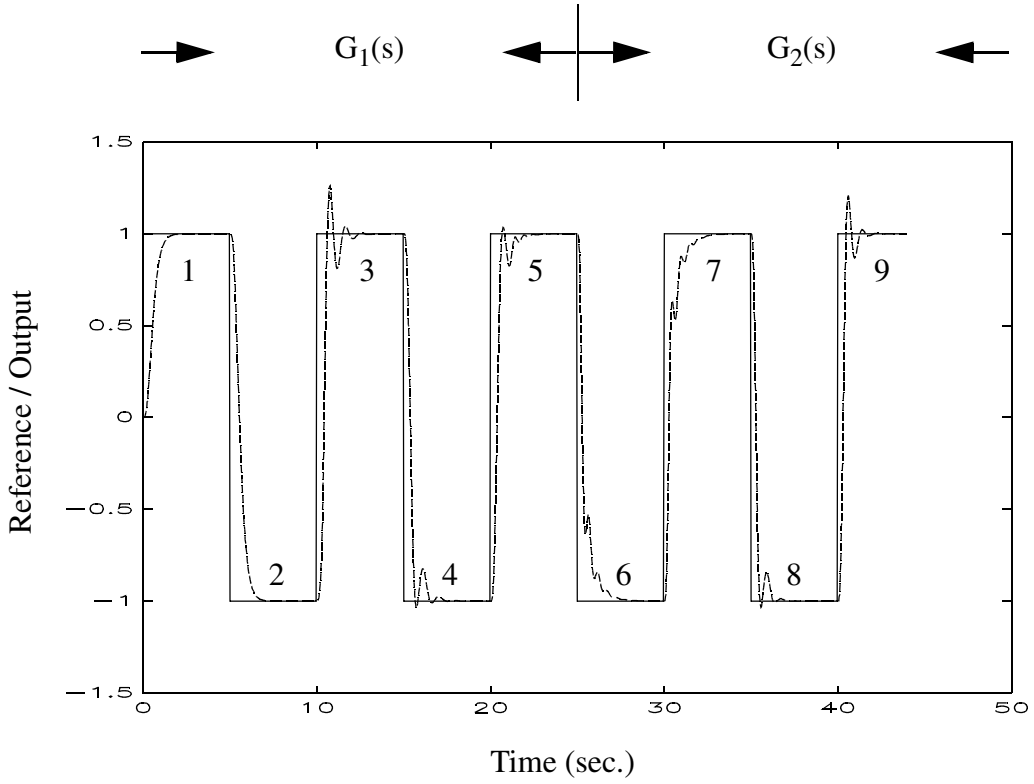


Fig. 6.13 - Example 3

For step six two plant time constants are changed and the transfer function becomes the one employed in the first example:

$$G_2(s) = \frac{1}{(1+0.1s)^4}$$

The response becomes overdamped. The new plant is identified during step seven. Step nine illustrates the response obtained after adaptation has taken place.

Finally Fig. 6.14 shows the control of a plant with varying transfer function. A sampling period of 5 ms was used in this example. As in the previous example, the system was started in open loop, adaptive mode. The transfer function of the plant is first set at:

$$G_3(s) = \frac{1}{(1+0.1s)(1+0.2s)(1+0.4s)}$$

During the second step the plant is identified, the PID values are obtained in the third step and the loop is automatically closed. Steps four and five denote the responses obtained after adaptation to the three-poles plant has occurred.

At step six the transfer function of the plant is changed. A time-delay plant with two poles, and transfer function

$$G_4(s) = \frac{e^{-0.1s}}{(1+0.4s)^2}$$

is now assumed.

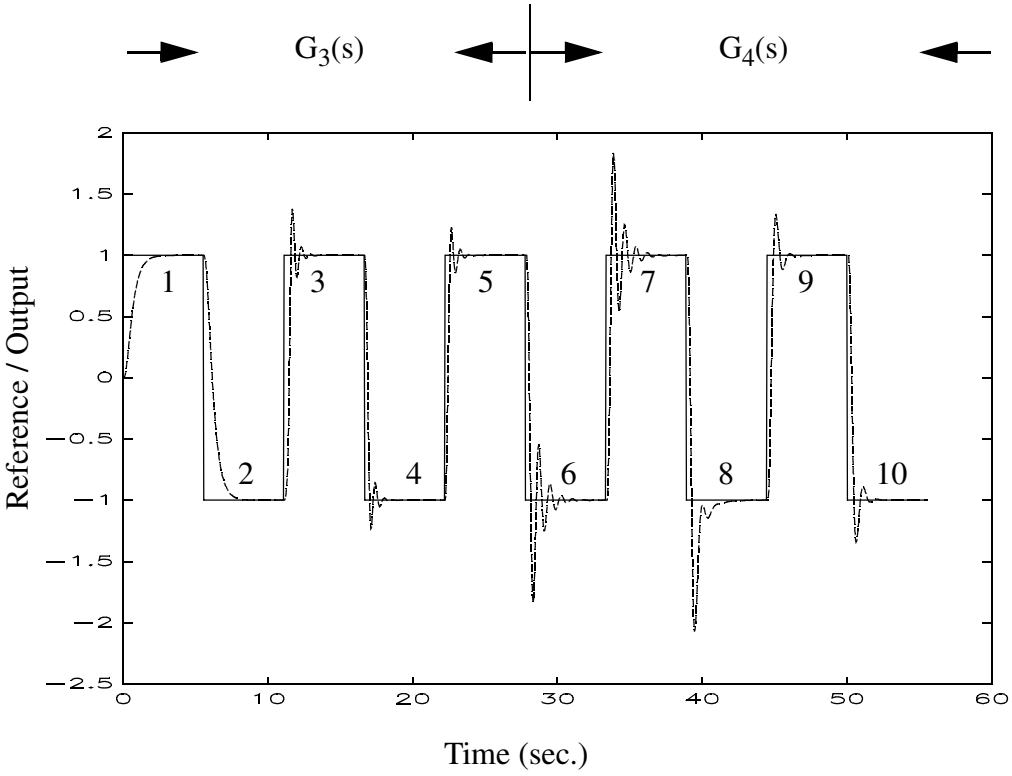


Fig. 6.14 - Example 4

An oscillatory response is obtained as result of this change. The new plant is identified during step seven, and the two final steps show the response obtained after adaptation has taken place, where it is clear that again a well damped response is obtained in steps 9 and 10.

6.6 Conclusions

In this Chapter the real-time implementation of the new neural PID autotuning technique is described. To enable different experiments to be performed, in a user-friendly way, a system was built in Occam, using an array of Inmos transputers. The different constituent blocks of the system are described. The system was also built with the purpose of easing the transition to the ultimate phase of this work, i.e., the control of real plants.

Results obtained with the real-time system show that, using the discretization methods currently employed, a high sampling rate needs to be employed to obtain similar results to the ones obtained in the continuous case. This has the drawback of increasing the number of samples required to identify the plant. This problem, however, as demonstrated in the last

section, is related to the discretization of the closed loop, and not to the proposed approach to PID autotuning.

This problem apart, the results show that well damped responses are obtained with this new PID autotuning technique, requiring a processing time less than 1 sec., once the response has settled.

As with any novel proposed technique, different stages must be followed before it becomes a practical proposition. The theoretical basis of this new method was introduced in Chapter 3, where off-line simulations showed promising results. The work described in this Chapter goes one step further towards reality and the technique was implemented in real-time. The next stage will involve the control of real plants. The method of building the real-time system allows this to be done easily.

Chapter 7

Conclusions

7.1 General conclusions

The application of artificial neural networks to control systems has been investigated. The ability of neural networks to represent nonlinear mappings has been found to be particularly important for the area of control systems. During the three years of this research, this property has been extensively applied for nonlinear systems identification and for the control of nonlinear plants.

In the work described in this thesis, this important property of neural networks has been applied to a commonly encountered problem, PID autotuning. A new method, involving multilayer perceptrons, was proposed. Since the tuning procedure has similarities to the actual process employed when performing manual tuning, no special hardware is required.

By using, as the input to the MLPs, identification indices that can be computed from the open or closed loop step response, this method is applicable to both situations. By defining the outputs of the MLPs to be the normalized PID values, this technique becomes independent of the absolute values of the time constants and time-delay of the plant, as well as of its DC gain. Training the multilayer perceptrons off-line, prior to their use in on-line control, avoids a long on-line training phase, where unstable close-loop systems might eventually appear. By using, for the purpose of deriving the target PID parameters, integral performance criteria, well damped responses are obtained in the on-line operation. On-line tuning is a fast operation, being determined essentially by the time it takes the plant to reach steady-state.

As with all applications where MLPs are previously trained off-line, it is essential that the examples used for training adequately span the entire range of operation. In this case, one indirect way of addressing this problem is to have previous knowledge about the type or types of the transfer function of the plant and, for each type of transfer function, the range of its time constants and delay time. For each time constant or delay of each transfer function, the range can then be discretized in order to produce suitable examples for training. Exactly how best to perform this discretization is currently not known.

In this novel PID autotuning method, the multilayer perceptrons are trained off-line, before being used for on-line control. The usual training method for MLPs, the error back-propagation algorithm, has been found to have severe drawbacks. It is an unreliable algorithm,

a good value for the learning parameter is difficult to find, and, in particular, it often has a very poor rate of convergence. Its use would have been reflected in a time-consuming off-line phase for the proposed PID technique, and the necessary experimentation, which is always needed when a new method is proposed, could not have been performed within a reasonable amount of time.

For these reasons, second order unconstrained optimization methods were investigated. Three methods (quasi-Newton, Gauss-Newton and Levenberg-Marquardt) were investigated, and their performance compared using common examples. It has been found that the Levenberg-Marquardt method achieved the best performance, since, in contrast with the quasi-Newton method, it explicitly exploits the least-squares nature of the problem; it is also less sensitive to ill-conditioning problems than the Gauss-Newton method.

To reduce the training time further, the topology of the MLPs employed was exploited. When MLPs are used for nonlinear function approximation purposes, their activation functions are linear, because the range of the outputs of the neurons in the output layer is typically unbounded. Exploiting this linearity, it has been shown that the usual training criterion can be reformulated into a new, yet equivalent, criterion. This, in contrast with the standard criterion, does not depend on the value of the weights and the biases associated with the neurons in the output layer. This new criterion was introduced by Golub and Pereyra [161], for general nonlinear regression problems. Its application for the specific purpose of training multilayer perceptrons was proposed by Ruano et al. [136]. By reformulating the learning problem, the dimensionality of the problem is reduced and, more importantly, there is a dramatic decrease in the number of iterations needed to find a local minimum. However, when the reformulated criterion is used with a Gauss-Newton or a Levenberg-Marquardt method, the calculation of its Jacobian matrix, as originally proposed by Golub and Pereyra [161], is a computationally-intensive task. In terms of training time, the benefits gained with the better convergence rate are lost with the higher computational costs per iteration. To reduce this computational complexity, Kaufman [162] introduced a different Jacobian matrix for the reformulated criterion. We have proposed another Jacobian matrix [137], which has been shown experimentally to produce better convergence rate than the existing approaches. Most importantly, the computational burden is reduced to such an extent that the computation at each iteration of the Levenberg-Marquardt method, using the reformulated criterion and the new Jacobian, is actually smaller than that using the standard criterion.

Employing the Levenberg-Marquardt algorithm, together with reformulated criterion and the new Jacobian matrix, a new sequential algorithm was proposed for the training of MLPs. In order to reduce the training times even further, parallel processing techniques were employed. Each main block of computation of the training algorithm was separately

parallelized. As the block with the highest computational complexity was, undoubtedly, the least-squares solution of an overdetermined system of linear equations, the main effort was put into the parallel implementation of this type of problem. For accuracy reasons, it was solved by means of a QR factorization. As a result of a modification introduced to the sequence of operations that are performed in a known parallel solution [173] for this type of problem, excellent values of parallel efficiency were obtained. For all the other blocks of computation, good parallel efficiency values were also obtained. Finally, the parallel solutions encountered were joined together to form a parallel learning algorithm.

The dramatic reduction in training time can be illustrated, taking as an example the training of an MLP with topology (5,7,7,1), with approximately 200 examples in the training set. By using the reformulated criterion, together with the new Jacobian matrix, reductions of one order of magnitude were obtained, when compared with the time taken by the Levenberg-Marquardt algorithm minimizing the standard criterion. By employing 7 Inmos T800 transputers, training sessions which used to take roughly three days on a Sun 3/60 workstation, using the Levenberg-Marquardt algorithm minimizing the standard criterion, were reduced to approximately 5 minutes. These dramatic savings in training time are not only important for the PID autotuning technique described in this thesis, but can be applied in all cases where MLPs are used for nonlinear mapping applications. This includes the majority of applications of MLPs in control systems. The parallelization scheme proposed for the least squares solution of a system of linear equations has probably a wider range of applications. Problems of this type appear in a broad range of disciplines such as statistics, optimization, control systems and signal processing. As mentioned during this thesis, however, the parallel solution proposed is suited to large problems. Experimental results enable us to conclude that, as long as the number of columns allocated to each processor is, say, greater than five, good parallel efficiency can be expected.

The practical implementation of the neural PID autotuner has also begun to be addressed. A modular real-time system was implemented in Occam. In order to allow easy experimentation with different solution methods, the main blocks of the system were implemented as different processes, each one executing on a separate transputer. For experimental convenience, the plant was also simulated digitally. The good performance obtained in off-line simulations was confirmed to be obtained in real-time as well. The time taken for the tuning procedure was found to be essentially the time that the plant takes to reach steady-state.

Extensive experimentation with the real-time system enables us to conclude that, as long as the examples used for their training phase adequately span the entire range of operation, multilayer perceptrons are a promising alternative to conventional PID autotuning

methods. However, before this technique becomes a practical proposition, further work, both theoretical and practical, must be performed.

7.2 Future Work

During this work, several problems were addressed. Answers were obtained for some of them, others remain open questions. There is, therefore, wide scope for future work.

- In Chapter 3, a set of integral measures $F(\sigma)$ was introduced to identify the plant. The values of σ employed in the experiences performed belonged to the set $\{0.5, 1, 2, 3, 5\}$. To obtain some insight into the "optimal" number of identification measures and the actual values of σ to use, third-order polynomial expansions of several combinations of the $F(\sigma)$ measures were employed to approximate the desired mappings. This was done, due to the practical impossibility, at that time, of performing the necessary training sessions of the multilayer perceptrons. As a result of the research described in Chapters 4 and 5 the means to perform this experimentation are now available, and so these tests should be conducted. A more desirable solution, however, would involve a theoretical study of this problem.

- Also in Chapter 3, two topologies of MLPs were considered for storing the mappings between the identification measures and the target normalized PID values. Further experimentation with other topologies is desirable, so that the best topology to use could be identified. As pointed out above, the means for doing this experimentation is now available. A more desirable solution would be to have guidelines available on the number of hidden layers and hidden neurons to use. As reported in Chapter 2, this is a subject of active research and so these guidelines might be expected to be available in the near future.

- Related to this last point, it was also speculated that larger networks could achieve a better accuracy for the desired mappings, so that closed loop step responses, much closer to the optimum, could be produced. The means to test this hypothesis, within a reasonable amount of time, is now available.

- It was evident from the literature reviewed that multilayer perceptrons were a suitable type of artificial neural network to be used in the proposed approach to PID autotuning, due to their nonlinear mapping capabilities. In Chapter 2, however, it was mentioned that CMAC and RBF networks are also being applied for nonlinear function approximation purposes. Comparison of the performance of MLPs, CMAC and RBF networks should be carried out.

- In Chapter 4 a detailed study of nonlinear optimization methods was conducted with a view to reducing the training time and to finding robust methods of training. In this context, two other points could be addressed. The first is to derive methods that could produce good initial values for the parameters of the MLPs. If a good starting point is available, then a reduction in the time taken by the iterative minimization procedure can be expected. Another point, related to the first, deals with scaling. It is well known that optimization methods work better if the decision variables (the weights and thresholds of the MLP) and the derivatives of the minimization criteria are well scaled. The training of MLPs, however, is a perfect example of an ill-conditioned problem. This ill-conditioning can, in principle, be alleviated, if some care is taken in the choice of the initial parameters. Empirical results suggest that the training process is improved, by first scaling the input and output data, and then choosing the initial weights and parameters associated with each nonlinear neuron such that the range of their net input vector lies within a region of, say, [-10, 10]. This should be the subject of further study.

- The parallel implementation of the learning algorithm, described in Chapter 5, does not have all the facilities of its sequential version. In the parallel least squares solution, a test for singularity should be incorporated. At present, this parallel block, and therefore the overall parallel algorithm, only works when the number of columns allocated to each processor is the same in every processor. This solution should be extended to the situation where such perfect load balancing cannot be achieved. Finally, more objective convergence criteria should be introduced into the parallel algorithm. All these modifications, as pointed out in the concluding remarks of Chapter 5, are straightforward to implement.

- The parallelization discussed in Chapter 5 allows the necessary experimentation to be performed and permits larger networks to be contemplated, with promising greater accuracy. However a large amount of time was spent on the parallel software development, by experimenting with different mapping strategies and testing the performance of different types of routers. Methods to aid the development of parallel software of this type, would clearly have been of great help in this work. The implementation of such techniques constitutes an important area of research.

- When implementing the new PID autotuning method in real-time, it was found that, to achieve the same kind of responses obtained in the continuous case, a high sampling rate had to be used. This problem should be further studied, by employing different discretization methods for each component of the loop and by analysing the closed-loop singularities, both in continuous and discrete cases.

- We have begun to address the practical implementation of the neural PID autotuner. However, before this technique can be considered a practical reality, further experimentation is still needed. The PID autotuner should be applied to real plants. Taking into account the way the real system described in Chapter 6 was built, the transition to this next step should be straightforward. During this work we have only considered ideal situations. In practical situations, noise is always present and several types of nonlinearities (controller saturation, process nonlinearities, etc.) must be taken into account. The performance of the new method should be assessed under these situations.

- The performance of the neural PID autotuner should be compared with commercially available devices, such as the PID regulators from SattControl and Fisher, Foxboro's Exact controller and the Electromax V regulator from Leeds and Northrup.

- The proposed PID autotuning method exploits one important feature of multilayer perceptrons, namely their capacity to approximate arbitrary nonlinear mappings. In the method described, MLPs are trained off-line, their weights remaining fixed afterwards. This has the disadvantage of requiring a considerable knowledge about the plant, namely in terms of its transfer function(s) and the range of its time constants and delay-time. This a priori knowledge is needed in order to generate training examples that adequately span the entire range of operation of the MLPs. In principle, this amount of a priori knowledge could be reduced if we exploit another important feature of MLPs, their ability to adapt. In this situation, the MLPs would also be previously trained off-line, but their parameters instead of remaining fixed afterwards, would change according to the performance of the system. We would then evolve from a broad tuning, achieved through the off-line training, to a finer tuning, specially adapted to the plant under consideration. This type of approach is advocated by several authors, in different contexts. For instance Psaltis [102] suggests using generalized learning (see section 2.3.1.2) in conjunction with specialized learning for the purpose of approximating inverse models of nonlinear dynamical plants.

- The autotuning method proposed in this thesis was targeted at PID controllers. In many situations, however, higher order controllers are able to produce better performance. The proposed technique can be readily extended to these situations. As the PID controller is characterized by three parameters (the DC gain and two time constants), three MLPs were used to approximate the mappings between identification measures and the three controller parameters. A larger number of time constants in the controller can be accommodated by simply employing more MLPs. This possibility is worthy of experimentation.

- Theoretical studies regarding the stability of the proposed technique should be conducted. Stability proofs are a key issue in control system design. Until now, and to the best of our knowledge, it is not possible to prove stability of any control system incorporating neural networks. The appearance of such types of proofs would make an important contribution to the credibility of this new technology in the eyes of the control systems community.

The ever-increasing technological demands of our society, together with the impressive development in control theory, open the door to new challenges. There is now the need to control increasingly complex, nonlinear, systems, over a wide range of uncertainty and under more stringent performance requirements. Neural networks, with their nonlinear mapping capabilities, massive parallelism and learning capabilities offer great promise in this area. Will this promise become a practical reality? - only time and a large research effort will tell.

References

[1] 'IEEE Transactions on neural networks', IEEE Press

[2] 'Neural networks', Pergamon Press

[3] 'Neural computation', MIT Press

[4] 'IEEE International Conference on Neural Networks'

[5] 'International Joint Conference on Neural Networks'

[6] 'Special section on neural networks for systems and control', IEEE Control Systems Magazine, Vol. 8, Nº 2, 1988

[7] 'Special Issue on neural networks for control systems', IEEE Control Systems Magazine, Vol. 9, Nº 3, 1989

[8] 'Special Issue on neural networks in control systems', IEEE Control Systems Magazine, Vol. 10, Nº 3, 1990

[9] Ruano, A., 'Application of neural networks to control systems: current trends', Internal Report, S.E.E.S., U.C.N.W., Dec. 1988

[10] Ziegler, J., Nichols, N., 'Optimum settings for automatic controllers', Transactions ASME, 65, 1942, pp. 759-768

[11] Åstrom, K., Hägglund, T., 'Automatic tuning of simple regulators with specifications on phase and amplitude margins', Automatica, Vol. 20, Nº 5, 1984, pp. 645-651

[12] Åstrom, K., Hägglund, T., 'A frequency domain method for automatic tuning of simple feedback loops', Proceedings of the 23rd Conference on Decision and Control, Las Vegas, U.S.A., 1984, pp. 299-304

[13] Rumelhart, D., McClelland, J. & the PDP Research Group, 'Parallel distributed processing', Vols. 1 and 2, MIT Press, 1986

[14] Gill, P., Murray, W. Wright, M., 'Practical optimization', Academic Press, 1981

[15] Inmos Ltd., 'The transputer databook', 2nd ed., Inmos Ltd., 1989

[16] Inmos Ltd., 'Occam 2 reference manual', C.A.R. Hoare (ed.), Prentice Hall, 1988

[17] Mehr, D., Richfield, S., 'Neural net application to optical character recognition', IEEE 1st International Conference on Neural Networks, Vol. 4, 1987, pp. 771-777

[18] Rajavelu, A., Musavi, M., Shirvaikar, V., 'A neural network approach to character recognition', Neural Networks, Vol.2, Nº 5, 1989, pp. 387-393

[19] Fukushima, K., Wake, N., 'Handwritten alphanumeric character recognition by the neocognitron', IEEE Transactions on Neural Networks, Vol. 2, Nº 3, 1991, pp. 355-365

[20] Bounds, D., Lloyd, P., Mathew, B., 'A comparison of neural networks and other

pattern recognition approaches to the diagnosis of low back disorders', Neural Networks, Vol. 3, Nº 5, 1990, pp. 583-591

[21] Kaufman, J., Chiabrera, A., Hatem, M., Hakim, N., Figueiredo, M., Nasser, P., Lattuga, S., Pilla, A., Siffert, R., 'A neural network approach for bone fracture healing assessment', IEEE Engineering in Medicine and Biology, Vol. 9, Nº 3, 1990, pp. 23-30

[22] Cios, K., Chen, K., Langenderfer, R., 'Use of neural networks in detecting cardiac diseases from echocardiographic images', IEEE Engineering in Medicine and Biology, Vol. 9, Nº 3, 1990, pp. 58-60

[23] Lippmann, R., 'Review of neural networks for speech recognition', Neural Computation, Vol. 1, Nº 1, 1989, pp. 1-38

[24] Sejnowsky, T., Rosenberg, C., 'Parallel networks that learn to pronounce English text', Complex Systems, 1, 1987, pp. 145-168

[25] Cottrel, G., Munro, P., Zipser, D., 'Image compression by back propagation: an example of an extensional programming', ICS Report 8702, Univ. of California at San Diego, 1987

[26] Shea, P., Liu, F., 'Operational experience with a neural network in the detection of explosives in checked airline luggage', IEEE INNS International Conference on Neural Networks, Vol.2, 1990, pp. 175-178

[27] Kuperstein, M., Rubinstein, J., 'Implementation of an adaptive neural controller for sensory-motor coordination', IEEE Control Systems Magazine, Vol. 9, Nº 3, 1989, pp. 25-30

[28] Fukuda, T., Shibata, T., kosuge, K., Arai, F., Tokita, M., Mitsuoka, T., 'Neuromorphic sensing and control - application to position, force and impact control for robotic manipulators', 30th IEEE Conference on Decision and Control, Brighton, England, 1991, pp. 162-167

[29] Narendra, K., Parthasarathy, K., 'Identification and control of dynamical systems using neural networks', IEEE Transactions on Neural Networks, Vol. 1, Nº 1, 1990, pp. 4-27

[30] Nguyen, D., Widrow, B., 'Neural networks for self-learning control systems', IEEE Control Systems Magazine, Vol. 10, Nº 3, 1990, pp. 18-23

[31] Lippmann, R., 'An introduction to computing with neural nets', IEEE ASSP Magazine, Vol. 4, Nº 2, 1987, pp. 4-22

[32] Anderson, J., Rosenfeld, E., 'Neurocomputing: foundations of research', MIT Press, 1988

[33] McCulloch, W., Pitts, W., 'A logical calculus of the ideas immanent in nervous activity', Bulletin of Mathematical Biophysics., Vol. 5, 1943, pp. 115-133

[34] Hebb, D., 'The organization of behaviour', Wiley, 1949

[35] Rosenblatt, F., 'Principles of neurodynamics', Spartan Books, 1962

[36] Widrow, B., 'An adaptive "Adaline", neuron using chemical "memistors" ', Stanford Electronics Labs., Stanford, USA, Tech. Report 1553-2, 1960

[37] Widrow, B., Hoff, M., 'Adaptive switching circuits', 1960 WESCON Convention Record, 1960, pp. 96-104

[38] Widrow, B., Smith, F., 'Pattern recognizing control systems', Computer and Information Sciences, Symposium Proceedings, Spartan Books, 1963

[39] Minsky, M., Papert, S., 'Perceptrons', MIT Press, 1969

[40] Amari, S., 'A theory of adaptive patterns classifiers', IEEE Transactions Electronic Computers, Vol. 16, Nº 3, 1967, pp. 299-307

[41] Anderson, J., 'A memory storage model utilizing spatial correlation functions', Kybernetik, Nº 5, 1968, pp. 113-119

[42] Fukushima, K., 'Cognitron: a self-organizing multilayered neural network', Biological Cybernetics, Vol. 4, Nº 20, 1975, pp.121-136

[43] Grossberg, S., 'Embedding fields: a theory of learning with physiological implications', J. Math. Psych., Nº 6, 1969, pp. 209-239

[44] Kohonen, T., 'Correlation matrix memories', IEEE Transactions on Computers, Vol. 21, Nº 4, 1972, pp. 353-359

[45] Hopfield, J., 'Neural networks and physical systems with emergent collective computational abilities', Proc. Nat. Acad. Sci., USA, 79, 1982, pp. 2554-2558

[46] Hopfield, J., 'Neurons with graded responses have collective computational properties like those of two-state neurons', Proc. Nat. Acad. Sci., USA, 81, 1984, pp. 3088-3092

[47] Vemuri, V., 'Artificial neural networks: an introduction', IEEE Technology Series - Artificial Neural Networks: Theoretical Concepts, 1988, pp. 1-11

[48] Hecht-Nielsen, R., 'Neurocomputing: picking the human brain', IEEE Spectrum, Vol. 25, Nº 3, 1988, pp. 36-41

[49] Mead, C., 'Analog VLSI and neural systems', Addison-Wesley, 1989

[50] Murray, A., 'Silicon implementations of neural networks', First IEE International Conference on Artificial Neural Networks', U.K., 1989, pp.27-32

[51] Hecht-Nielsen, R., 'Neurocomputing', Addison Wesley, 1990

[52] Strange, P., 'Basic brain mechanisms: a biologist's view of neural networks', IEE Colloquium on 'Current issues in neural network research', Digest Nº 1989/83, 1989, pp.1-5

[53] Antsaklis, P., 'Neural networks in control systems', IEEE Control Systems, Vol. 10, Nº 3, 1990, pp. 3-5

[54] Kohonen, T.,'Self-organization and associative memory', 2nd ed., Springer-Verlag, 1988

[55] Barto, A., Sutton, R., Anderson, C., 'Neuron-like adaptive systems that can solve difficult learning control problems', IEEE Transactions on Systems, Man &

Cybernetics, Vol. 13, Nº 5, 1983, pp. 834-846

[56] Carpenter, G., Grossberg, S., 'The ART of adaptive pattern recognition by a self-organizing neural network', IEEE Computer, Vol. 21, Nº 3, 1988, pp. 77-88

[57] Simpson, P., 'Artificial neural systems: applications, paradigms, applications, and implementation', Pergamon Press, 1990

[58] Aleksander, I. (ed.), 'Neural computing architectures: the design of brain-like machines', North Oxford Academic Press, 1989

[59] 'Neural works explorer: user's guide', NeuralWare, Inc, 1989

[60] Hopfield, J., Tank, D., 'Neural computation of decisions optimization problems', Biological Cybernetics, Vol. 52, 1985, pp. 141-152

[61] Hopfield, J., Tank, D., 'Simple "neural" optimization networks: an A/D converter, signal decision circuit, and a linear programming circuit', IEEE Transactions on Circuits and Systems, Vol. 33, Nº 5, 1986, pp. 533-541

[62] Kennedy, W., Chua, L., 'Neural networks for nonlinear programming', IEEE Transactions on Circuits and Systems, Vol. 35, Nº 5, 1988, pp. 554-562

[63] Ramanujam, J., Sadayappan, P., 'Optimization by neural networks', 2nd IEEE Conference on Neural Networks, Vol. 2, 1988, pp. 325-332

[64] Michel, A., Farrell, J., 'Associative memories via artificial neural networks', IEEE Control Systems, Vol.10, Nº 3, 1990, pp. 6-17

[65] Yen, G., Michel, A., 'A learning and forgetting algorithm in associative memories: the eigenstructure method', 30th IEEE Conference on Decision and Control, Brighton, England, 1991, pp. 847-852

[66] Venkatesh, S., Pancha, G., Psaltis, D., Sirat, G., 'Shaping attraction basins in neural networks', Neural Networks, Vol. 3, Nº 6, 1990, pp. 613-623

[67] Albus, J., 'The Cerebellar model articulation controller', Transactions ASME, series G, Vol. 97, Nº 3, 1975

[68] Miller W., Glanz, F., Kraft, L., 'Application of a general learning algorithm to the control of robotic manipulators', International Journal of Robotics Research, Vol. 6, Nº 2, 1987, pp. 84-98

[69] Ersü, E., Militzer, J., 'Software implementation of a neuron-like associative memory for control applications', Proceedings of the ISMM 8th Symposium MIMI, Switzerland, 1982

[70] Brown, M., 'Adaptive piloting systems for autonomous vehicles', Esprit II: Panorama Project rept. SOTONU TN/11/90 82p

[71] Ersü, E., Tolle, H., 'Learning control structures with neuron-like associative memory systems', in W. Seelen (ed.): 'Organization of neural networks. Structures and models', Weinheim, 1988, pp. 417-437

[72] Brown, M., Fraser, R., Harris, C., Moore, C., Intelligent self-organizing controllers for autonomous guided vehicles: comparative aspects of fuzzy logic and neural nets', IEE

International Conference Control 91, Edinburgh, U.K., 1991, pp. 134-139

[73] Mischo, W., Hormel, Tolle, H., 'Neurally inspired associative memories for learning control. A comparison', International Conference on Artificial Neural Networks, Espoo, Finland, 1991

[74] Almeida, L., 'Backpropagation in non-feedforward networks', in Igor Aleksander (ed.), 'Neural computing architectures: the design of brain-like machines', North Oxford Academic, 1989, pp. 74-91

[75] Pineda, J., 'Generalization of backpropagation to recurrent and higher-order neural networks', Proc. of the 1987 IEEE Conference on Neural Information Processing Systems - Natural and Synthetic, 1988, pp. 602-611

[76] Funahashi, K.,'On the approximate realization of continuous mappings by neural networks', Neural Networks, Vol. 2, Nº 3, 1989, pp. 183-192

[77] Hornik, K., Stinchcombe, M., White, H., 'Multilayer perceptrons are universal approximators', Neural Networks, Vol. 2, Nº 5, 1989, pp. 359-366

[78] Baum, E., Haussler, D., 'What size net gives valid generalization?', Neural Computation, Vol. 1, Nº 1, 1989, pp. 151-160

[79] Wang, Z., Tham, M., Morris, A., 'Multilayer feedforward neural networks: approximated canonical decomposition of nonlinearity', submitted to International Journal of Control

[80] Huang, S., Huang, Y., 'Bounds on the number of hidden neurons in multilayer perceptrons', IEEE Transactions on Neural Networks, Vol. 2, Nº 1, 1991, pp. 47-55

[81] Sartori, M., Antsalkis, P., 'A simple method to derive bounds on the size and to train multilayer perceptrons', IEEE Transactions on Neural Networks, Vol. 2, Nº 4, 1991, pp. 467-470

[82] Mehrota, K., Mohan, C., Ranka, S., 'Bounds on the number of samples needed for neural learning', IEEE Transactions on Neural Networks, Vol. 2, Nº 6, 1991, pp. 548-558

[83] Broomhead, D., Lowe, D., 'Multivariable functional interpolation and adaptive networks', Complex Systems, Nº 2, 1988, pp. 321-355

[84] Bavarian, B., 'Introduction to neural networks for intelligent control', IEEE Control Systems Magazine', Vol. 8, Nº 2, 1988, pp. 3-7

[85] Elsley, R., 'A learning architecture for control based on back-propagation neural networks', IEEE International Conference on Neural Networks, Vol. 2, 1988, pp. 587-594

[86] Kawato, M., Uno, Y., Isobe, M., Suzuki, R., 'Hierarchical neural network model for voluntary movement with application to robotics', IEEE Control Systems Magazine, Vol. 8, Nº 2, 1988, pp. 8-16

[87] Guo, J., Cherkassky, V., 'A solution to the inverse kinematic problem in robotics using neural network processing', IEEE International Conference on Neural Networks, Vol. 2, 1989, pp. 299-304

[88] Naidu, S., Zafiriou, E., McAvoy, T., 'Use of neural networks for sensor failure detection in a control system', IEEE Control Systems Magazine, Vol. 10, Nº 3, 1990, pp. 49-55

[89] Leonard, J., Kramer, M., 'Classifying process behaviour with neural networks: strategies for improved training and generalization', IEEE Automatic Control Conference, San Diego, Vol. 3, 1990, pp. 2478-2483

[90] Narenda, K., Mukhopadhyay, S., 'Multilevel control of dynamical systems using neural networks', 30th IEEE Conference on Decision and Control, Brighton, England, 1991, pp. 170-171

[91] Chen, S., Billings, S., Grant, P., 'Nonlinear systems identification using neural networks', International Journal of Control, Vol. 51, 1990, pp. 1191-1214

[92] Ruano, A., 'First year report', S.E.E.S., U.C.N.W., Sept. 1989

[93] Ljung, L., 'Analysis of a general recursive prediction error identification algorithm', Automatica, Vol. 17, Nº 1, 1981, pp. 89-99

[94] Lightbody, G., Irwin, G., 'Neural networks for nonlinear adaptive control', Proceedings of the First IFAC Workshop on 'Algorithms and Architectures for Real-Time Control', September 1991 (to be published)

[95] Lapedes, A., Farber, R., 'Nonlinear signal processing using neural networks: prediction and system modelling', Tech. Rept., Los Alamos National Lab., 1987

[96] Bhat, N., Minderman, P., McAvoy, T., Wang, N., 'Modelling chemical process systems via neural computation', IEEE Control Systems Magazine, Vol. 10, Nº 3, 1990, pp. 24-29

[97] Lant, P., Willis, M., Montague, G., Tham, M., Morris, A., 'A comparison of adaptive estimation with neural based techniques for bioprocess application', IEEE Automatic Control Conference, San Diego, Vol. 2I, 1990, pp. 2173-2178

[98] Willis, M., Massimo, C., Montague, G., Tham, M., Morris, A., 'Artificial neural networks in process engineering', IEE Proceedings Part D, Vol. 138, Nº 3, 1991, pp. 256-266

[99] Billings, S., Jamaluddin, H., Chen, S., 'Properties of neural networks to modelling non-linear dynamical systems', International Journal of Control, Vol. 55, Nº 1, 1992, pp. 193-224

[100] Ersü, E., Militzer, J., 'Real-time implementation of an associative memory-based learning control scheme for nonlinear multivariable processes', Proceedings of the Symposium 'Application of multivariable system technique', Plymouth, U.K., 1984, pp. 109-119

[101] Ersü, E., Wienand, S., 'An associative memory based learning control scheme with PI-controller for SISO-nonlinear processes', Proceedings of the IFAC Microcomputer Application in Process Control, Istanbul, Turkey, 1986, pp. 99-105

[102] Psaltis, D., Sideris, A., Yamamura, A., 'Neural Controllers', IEEE First Conference on Neural Networks, Vol.4, 1987, pp. 551-558

[103] Saerens, M., Soquet, A., 'Neural-controller based on back-propagation algorithm', IEE Proceedings, Part F, Vol. 138, Nº 1, 1991, pp. 55-62

[104] Nguyen, D., Widrow, B., 'The truck backer-upper: an example of self-learning in neural networks', Proceedings of the 1989 International Joint Conference on Neural Networks, Vol. 2, 1989, pp. 357-363

[105] Hunt, K., Sbarbaro, D., 'Neural networks for nonlinear internal control', IEE Proceedings, Part D, Vol. 138, Nº 5, 1991, pp. 431-438

[106] Harris, C., Brown, M., 'Autonomous vehicle identification, control and piloting through a new class of associative memory neural networks', IEE Colloquium on Neural Networks for Systems: Principles and Applications', January 1991, London, pp. 6/1-6/5, Digest Nº 1991/019

[107] Åstrom, K., Wittenmark, B., 'Adaptive Control', Addison-Wesley, 1989

[108] Kraft, L., Campagna, D., 'A comparison between CMAC neural networks and two traditional adaptive control systems', IEEE Control Systems Magazine, Vol. 10, Nº 3, 1990, pp. 36-43

[109] Ersü, E., Tolle, 'A new concept for learning control inspired by brain theory', Proceedings of the 9th IFAC World Congress, Budapest, Hungary, 1984, pp. 1039-1044

[110] Montague. G., Willis, M., Tham, M., Morris, A., 'Artificial neural network based control', Proceedings of the IEE International Conference Control 91, Edinburgh, U.K., Vol. 1, 1991, pp. 266-271

[111] Clarke, D., Mothadi, C., Tuffs, P., 'Generalized predictive control - part I: the basic algorithm', Automatica, Vol. 23, Nº 2, 1987, pp. 137-148

[112] Sbarbaro, D., Hunt, K., 'A nonlinear receding horizon controller based on connectionist models', 30th IEEE Conference on Decision and Control, Brighton, England, 1991, pp. 172-173

[113] Hernández, E., Arkun, Y., 'Neural network modelling and an extended DMC algorithm to control nonlinear systems', IEEE Automatic Control Conference, San Diego, Vol. 3, 1990, pp. 2454-2459

[114] Chen, F., 'Back-propagation neural networks for nonlinear self-tuning adaptive control', IEEE Control Systems, Vol. 10, Nº 3, pp. 44-48

[115] Iiguni, Y., Sakai, H., Tokumaru, H., 'A nonlinear regulator design in the presence of system uncertainties using multilayered neural networks', IEEE Transactions on Neural Networks', Vol. 2, Nº 4, 1991, pp. 410-417

[116] Åstrom, K., Wittenmark, B., 'Computer-controlled systems', Prentice-Hall, 1984

[117] Åstrom, K., 'Toward intelligent control', IEEE Control Systems Magazine, Vol. 9, Nº 3, pp. 60-64

[118] Hang, C., Åstrom, K., Ho, W., 'Refinements of the Ziegler-Nichols tuning formula', IEE Proceedings Part D, Vol. 138, Nº 2, 1991, pp. 111-118

[119] Nishikawa, Y. Sannomiya, N. Ohta, T. and Tanaka, H.,'A method for auto-tuning of PID control parameters', Automatica., Vol. 20, Nº 3, 1984, pp. 321-32

[120] Kraus, T., Myron, T., 'Self-tuning PID controller uses pattern recognition approach', Control Engineering, Vol. 31, Nº 6, 1984, pp. 106-111

[121] Wittenmark, B., Åstrom, K., 'Simple self-tuning regulators', in H. Unbehauen (ed.), 'Methods and applications in adaptive control', Springer, Berlin, 1980, pp. 21-30

[122] Gawthrop, P., 'Self-tuning PID controllers: algorithms and implementation', IEEE Transactions on Automatic Control, Vol. 31, Nº 3, 1986, pp. 201-209

[123] Radke, F. and Isermann, R., 'A parameter-adaptive PID-controller with stepwise parameter optimization', Automatica, Vol. 23, Nº 4, 1987, pp. 449-57

[124] Anderson, K., Blankenship, G., Lebow, L., 'A rule-based adaptive PID controller', 27th Conference on Decision and Control, Austin, U.S.A., Vol. 1, 1988, pp. 564-569

[125] Lemke, H., De-zhao, W., 'Fuzzy PID supervisor', 24th Conference on Decision and Control, Florida, U.S.A., Vol. I, 1985, pp. 602-608

[126] Ruano, A., P. Fleming, Jones, D., 'A connectionist approach to PID autotuning', IEE International Conference Control 91, Edinburgh, U.K., Vol. 2, 1991, pp. 762-767

[127] Swiniarski, R., 'Novel neural network based self-tuning PID controller which uses pattern recognition technique', IEEE Automatic Control Conference, Vol. 3, 1990, pp. 3023-3024

[128] Al-Assadi, S., , Al-Chalabi, L., 'Optimal gain for proportional-integral-derivative feedback', IEEE Control Systems Magazine, Vol. 7, Nº 6, 1987, pp. 16-19

[129] Zhuang, M., Atherton, D., 'Tuning PID controllers using integral performance criteria', IEE International Conference Control 91, Edinburgh, U.K., Vol. 1, 1991, pp. 481-486.

[130] Åstrom, K., 'Introduction to stochastic control theory', Academic Press, 1970

[131] Graham, D., Lathorp, R., 'The synthesis of "optimum" transient response: criteria and standard forms', AIEE Transactions, Nº 72, 1953, pp. 273-285

[132] Dorf, R., 'Modern control systems', 5th ed., Addison Wesley, 1989

[133] Moler, C., Little J., Bangert, S., 'PRO-MATLAB user's guide', The MathWorks, Inc., South Natick, MA 01760, USA, 1987

[134] Grace, A., 'Optimization toolbox user's guide', The MathWorks, Inc., South Natick, MA 01760, USA, 1990

[135] Golub, G., Van Loan, C., 'Matrix computations', North Oxford Academic, 1983

[136] Ruano, A., Fleming, P., and Jones, D., 'A connectionist approach to PID autotuning', accepted for publication in IEE Proceedings, Part D.

[137] Ruano, A., Jones, D. and Fleming P., 'A new formulation of the learning problem for a neural network controller' 30th IEEE Conference on Decision and Control, Brighton, England, Vol. 1, 1991, pp. 865-866

[138] Werbos, P., 'Beyond regression: New tools for prediction and analysis in the behavioral sciences', Doctoral Dissertation, Appl. Math, Harvard University, U.S.A., 1974

[139] Parker, D., 'Learning logic', Invention report, S81-64, File 1, Office of Technology Licensing, Stanford University

[140] Tesauro, G., He, Yu and Ahmad, S., 'Asymptotic convergence of backpropagation', Neural Computation, Vol. 1, Nº 3, 1989, pp.382-391

[141] Jacobs, R., 'Increased rates of convergence through learning rate adaptation', Neural Networks, Vol. 1, Nº 3, 1988, pp. 295-307

[142] Watrous, R., 'Learning algorithms for connectionist networks: applied gradient methods of nonlinear optimization',1st IEEE International Conference on Neural Networks, Vol. 2, 1987, pp. 619-627

[143] Sandom, P., Uhr, L., 'A local interaction heuristic for adaptive networks', 2nd IEEE International Conference on Neural Networks, Vol. I, 1988, pp. 317-324

[144] Strang, G., 'Linear algebra and its applications', Academic Press, 1980

[145] Proakis, J., Manolakis, D., 'Introduction to digital signal processing', Macmillan Publishing Company, 1988

[146] Marple, S., 'Digital spectral analysis with applications', Prentice-Hall International, 1987

[147] Holland, J.H., 'Adaptation in natural and artificial systems', Ann Arbor: The Univ. of Michigan Press, 1975

[148] Goldberg, D.,'Genetic algorithms in search, optimization and machine learning', Addison-Wesley, 1989

[149] de Garis, H., 'Genetic programming: modular neural evolution for Darwin machines', International Joint Conference on Neural Networks, Vol. 3, 1990, pp. 511

[150] Grace, A., 'Computer-aided control system design using optimization methods', PhD. Thesis, University College of North Wales, U.K., 1989

[151] Fletcher, R., 'Practical methods of optimization', Vol. I, John Wiley & Sons, 1980

[152] Luenberger, D., 'Introduction to linear and nonlinear programming', Addison-Wesley Publishing, 1973

[153] Davidon, W., 'Variable metric method for minimization', A.E.C. Research and Development Report, ANL - 5990, 1959

[154] Fletcher, R. Powell, M., 'A rapidly convergent descent method for minimization', Computer Journal, Vol. 6, 1963, pp. 163-168

[155] Broyden, C., 'The convergence of a class of double-rank minimization algorithms', Journal of the Institute of Mathematics and its Applications, Vol. 6, 1970, pp.76-90

[156] Fletcher, R., 'A new approach to variable metric algorithms', Computer Journal, Vol. 13, 1970, pp. 317-322

[157] Goldfarb, D., 'A family of variable metrics updates derived by variational means', Mathematics of Computation, Vol. 24, 1970, pp. 23-26

[158] Shanno, D., 'Conditioning of quasi-Newton methods for function minimization', Mathematics of Computation, Vol. 24, 1970, pp. 647-656

[159] Levenberg, K., 'A method for the solution of certain problems in least squares', Quart. Apl. Math., Vol. 2, 1944, pp. 164-168

[160] Marquardt, D., 'An algorithm for least-squares estimation of nonlinear parameters', SIAM J. Appl. Math., Vol. 11, 1963, pp.431-441

[161] Golub, G., Pereyra, V., 'The differentiation of pseudo-inverses and nonlinear least squares problems whose variables separate', SIAM JNA, Vol. 10, Nº 2, 1973, pp. 413-432

[162] Kaufman, L., 'A variable projection method for solving separable nonlinear least squares problems', BIT, 15, 1975, pp.49-57

[163] Bates, D., Lindstrom, M. , 'Nonlinear least squares with conditionally linear parameters', Proc. of the Statistical Computing Section, New York: American Statistical Association, , 1986, pp. 152-157

[164] Paugam-Moisy, H., 'A spy of parallel neural networks', Technical Report 90-27, Ecole Normale Supérieure de Lyon, France, 1990

[165] Paugman-Moisy, H., 'Parallelizing multilayer neural networks', Esprit PCA Workshop (Bonn-Germany), May 1991

[166] Singer, A., 'Implementations of artificial neural networks on the connection machine', Parallel Computing, Nº 14, 1990, pp. 305-316

[167] Zhang, X., McKenna, M., Mesirov, J. and Waltz, D., 'The backpropagation algorithm on grid and hypercube architectures', Parallel Computing, Nº 14, 1990, pp. 317-327

[168] Petrowski, A., Dreyfus, G., Girault, C., 'Performance of a pipelined backpropagation algorithm on a parallel computer', Esprit PCA Workshop (Bonn-Germany), May 1991

[169] Burns, A., 'Programming in Occam 2', Addison-Wesley, 1988

[170] Gallettly, J., 'Occam 2', Pitman Publishing, 1990

[171] Bowler, K., Kenway, R., Pawley, G. and Roweth, D., 'An introduction to Occam 2 programming', Chartwell-Bratt Ltd., 1987

[172] Stewart, G., 'Introduction to matrix computations', Academic Press, 1973

[173] Modi, J., 'Parallel algorithms and matrix computation', Oxford University Press, 1988

[174] 'MCP 1000 Technical reference manual', Transtech Devices Limited, 1990

[175] Inmos, 'The T9000 transputer manual', Inmos, 1991

[176] Ruano, A., Jones, D., Fleming, P., 'A neural network controller', Proc. of the First IFAC Workshop on 'Algorithms and Architectures for Real-Time Control', September 1991 (to be published)

[177] Franklin, G., Powell, J., Workman, M., 'Digital control of dynamical systems', 2nd ed., Addison-Wesley, 1990

[178] Savant, C., 'Fundamentals of the Laplace transformation', The Maple Press Company, 1962

[179] Ruano, A., Jones, D. and Fleming P., ‘A new formulation of the learning problem for a neural network controller’, submitted to IEEE Transactions on Automatic Control

[180] Harper, D., ‘An introduction to Maple’, Liverpool University Computer Laboratory, University of Liverpool, U.K., 1989

Appendix A

Derivation of the Identification Measures

The identification measures used in the neural PID autotuner may be obtained from integral measures of the step response using simple Laplace transform properties. For completeness, the derivation of these measures is presented in this Appendix.

A.1 Background

Considering a real function x(t), which satisfies the conditions:

i) $x(t) = 0 \qquad t < 0$

ii) $\lim_{t \to 0} f(t) = f(0) \qquad t > 0$

its Laplace transform will be denoted by the symbols X(s) or L[x(t)]. The inverse Laplace transform operator will be denoted as $L^{-1}[.]$. The transform pair will be presented, as standard practice, by:

$$x(t) \leftrightarrow X(s) \tag{A.1}$$

The derivations performed on the next sections are based on two properties which can be obtained from well known theorems of the Laplace transform [178]:

i) Final value theorem:

$$\lim_{t \to \infty} x(t) = \lim_{s \to 0} sX(s) \tag{A.2}$$

provided that the derivative of x(t) is transformable and all the singularities of sF(s) lie in the left half plane.

ii) Complex translation:

$$e^{at}x(t) \leftrightarrow X(s - a) \qquad a \in \Re \tag{A.3}$$

iii) Real integration:

$$\int_0^t x(\tau)d\tau \leftrightarrow \frac{X(s)}{s} \tag{A.4}$$

By combining theorems (i) and (iii), property I may be obtained:

Property I:

$$\int_0^\infty x(\tau)d\tau = \lim_{s \to 0} X(s) \tag{A.5}$$

Proof: Let us denote:

$$y(t) = \int_0^t x(\tau)d\tau \tag{A.6}$$

Then:

$$\int_0^\infty x(\tau)d\tau = \lim_{t \to \infty} \int_0^t x(\tau)d\tau = \lim_{t \to \infty} y(t) \tag{A.7}$$

Using theorem (i), (A.7) can be given as:

$$\int_0^\infty x(\tau)d\tau = \lim_{s \to 0} sY(s) \tag{A.8}$$

But, according to theorem (iii), Y(s) is given as:

$$Y(s) = \frac{X(s)}{s} \tag{A.9}$$

which, when replaced in (A.8) gives (A.5).

Finally, by combining (A.5) with theorem (ii), it is easy to show that the following property is obtained:

Property II:

$$\int_0^\infty e^{-\alpha\tau} x(\tau)d\tau = \lim_{s \to 0} X(s + \alpha) \tag{A.10}$$

Using the two defined properties, we can derive the identification measures. We shall start with the open loop case.

A.2 Open loop

Using (3.77) and (3.72), and assuming that an input step of amplitude B is applied to the plant, eq. (3.78) can be given as:

$$S(\alpha) = \int_0^{\infty} e^{-\tau\alpha} L^{-1}\left[\frac{k_p B}{s} - \frac{B}{s}\left(k_p e^{-Ls}\frac{\prod_{i=1}^{n_z}(1+T_{z_i}s)}{\prod_{i=1}^{n_p}(1+T_{p_i}s)}\right)\right] d\tau$$

$$= B k_p \int_0^{\infty} e^{-\tau\alpha} L^{-1}\left[\frac{\prod_{i=1}^{n_p}(1+T_{p_i}s) - e^{-Ls}\prod_{i=1}^{n_z}(1+T_{z_i}s)}{s\prod_{i=1}^{n_p}(1+T_{p_i}s)}\right] d\tau \qquad \text{(A.11)}$$

This last equation is in the form of (A.10). In this case X(s) is given by:

$$X(s) = \frac{\prod_{i=1}^{n_p}(1+T_{p_i}s) - e^{-Ls}\prod_{i=1}^{n_z}(1+T_{z_i}s)}{s\prod_{i=1}^{n_p}(1+T_{p_i}s)} \qquad \text{(A.12)}$$

The scaling factor T_T can be obtained using Property I. Using L'Hospital's rule to evaluate the limit of X(s) as s goes to 0, we arrive at:

$$S(0) = B k_p\left(\sum_{i=1}^{n_p} T_{p_i} + L - \sum_{i=1}^{n_p} T_{z_i}\right), \qquad \text{(A.13)}$$

which, after simple manipulation, gives (3.79).

Evaluating now $\lim_{s\to 0} X(s+\alpha)$, we can determine the identification measures $F(\alpha)$:

$$\lim_{s\to 0} X(s+\alpha) = B k_p\left|\frac{1}{\alpha} - e^{-L\alpha}\frac{\prod_{i=1}^{n_z}(1+T_{z_i}\alpha)}{\alpha\prod_{i=1}^{n_p}(1+T_{p_i}\alpha)}\right| = S(\alpha) \qquad \text{(A.14)}$$

This last expression, after suitable manipulation, gives eq. (3.80). This concludes the derivations needed for the open loop case.

A.3 Closed loop

For the derivation of the identification measures in closed loop the PID compensation depicted in Fig. 3.14 is assumed. Considering that a step of amplitude B is applied to the reference input, we can obtain, after some manipulation, U(s), the Laplace transform of the control input u(t):

$$U(s) = \frac{Bk_c}{s}F(s) \tag{A.15}$$

where

$$F(s) = \frac{(1+sT_i)(1+sT_f)\prod_{i=1}^{r}(1+T_{p_i}s)}{(sT_i)(1+sT_f)\prod^{n_p}(1+T_{p_i}s) + k_ck_pe^{-Ls}(1+sT_i)(1+sT_d)\prod^{n_z}(1+T_{z_i}s)} \tag{A.16}$$

Using (A.15) and (3.82), eq. (3.83) can be expressed in the same form as (A.10), X(s) being given as:

$$X(s) = \frac{B}{sk_p}\frac{N(s)}{D(s)} \tag{A.17}$$

where N(s) and D(s) are given by eq. (A.18) and (A.19) respectively.

$$N(s) = (sT_i - k_ck_p(1+sT_i))(1+sT_f)\prod_{i=1}^{\cdot}(1+T_{p_i}s) + \\ + k_ck_pe^{-Ls}(1+sT_i)(1+sT_d)\prod^{n_z}(1+T_{z_i}s) \tag{A.18}$$

(A.19)

Evaluating the limit of X(s) as s goes to 0 using L'Hospital's rule, we get:

$$S_u(0) = \frac{B}{k_p}\left(\frac{T_i}{k_ck_p} + T_d - T_f + \sum_{i-1}^{n_z}T_{z_i} - \sum_{i-1}^{n_p}T_{p_i} - L\right) \tag{A.20}$$

which, after convenient manipulation, gives the scaling factor (3.84).

To derive (3.85), we evaluate the $\lim_{s \to 0} X(s+\alpha)$, X(s) being given by (A.17). After some manipulation, (A.21) can be obtained:

$$S_u(\alpha) = \frac{B}{\alpha k_p} \frac{(\alpha T_i - k_c k_p(1+\alpha T_i))(1+\alpha T_f) + k_c k_p (1+\alpha T_i)(1+\alpha T_d)F(\alpha)}{\alpha T_i(1+\alpha T_f) + k_c k_p (1+\alpha T_i)(1+\alpha T_d)F(\alpha)} \qquad (A.21)$$

This expression, with further manipulation, gives eq. (3.85).

Finally, it will be shown how the identification measures can be obtained from the output signal. Considering again a step reference of amplitude B, Y(s), the Laplace transform of the output, is given by:

$$Y(s) = \frac{B}{s} \frac{k_c k_p e^{-Ls}(1+sT_i)(1+sT_f)\prod_{i=1}^{n_z}(1+sT_{z_i})}{sT_i(1+sT_f)\prod_{i=1}^{n_p}(1+sT_{p_i}) + k_c k_p e^{-Ls}(1+sT_i)(1+sT_d)\prod_{i=1}^{n_z}(1+sT_{z_i})} \qquad (A.22)$$

Expressing (3.86) in the form of (A.10), X(s) is then given by:

$$X(s) = B \frac{T_i(1+sT_f)\prod_{i=1}^{n_p}(1+sT_{p_i}) + k_c k_p e^{-Ls}(T_d - T_f)(1+sT_i)\prod_{i=1}^{n_z}(1+sT_{z_i})}{sT_i(1+sT_f)\prod_{i=1}^{n_p}(1+sT_{p_i}) + k_c k_p e^{-Ls}(1+sT_i)(1+sT_d)\prod_{i=1}^{n_z}(1+sT_{z_i})} \qquad (A.23)$$

Using property II, $S_y(\alpha)$ can be obtained as:

$$S_y(\alpha) = \lim_{s \to 0} X(s+\alpha) = B \frac{T_i(1+\alpha T_f) + k_c k_p (T_d - T_f)(1+\alpha T_i)F(\alpha)}{\alpha T_i(1+\alpha T_f) + k_c k_p (1+\alpha T_i)(1+\alpha T_d)F(\alpha)} \qquad (A.24)$$

By rearranging the terms in (A.24), eq. (3.87) is given.

Appendix B

The Reformulated Criterion: Theoretical Proofs

Two main proofs compose this appendix: primarily, it will be shown that all the Jacobian matrices of the reformulated learning criterion, referred to in Chapter 4, produce the same gradient vector; after it will be shown that the Jacobian matrix proposed by Ruano et al. [137] can be obtained from Kaufman's Jacobian matrix.

B.1 Background

Golub and Pereyra [161] introduced, in the context of general nonlinear regression problems, the reformulated criterion (4.60), which, for convenience, is reproduced here:

$$\psi = \frac{\left\|\mathbf{t} - \mathbf{A}\mathbf{A}^{+}\mathbf{t}\right\|_2^2}{2} = \frac{\left\|\mathbf{P}_{\mathbf{A}_\perp}\mathbf{t}\right\|_2^2}{2} \ddagger \tag{B.25}$$

These authors also proved that the derivative of the pseudo-inverse of $\mathbf{A}$, w.r.t. the nonlinear parameters ($\mathbf{v}$), is:

$$(\mathbf{A}^{+})_{\mathbf{v}} = -\mathbf{A}^{+}(\mathbf{A})_{\mathbf{v}}\mathbf{A}^{+} + \mathbf{A}^{+}\mathbf{A}^{+\mathrm{T}}(\mathbf{A})_{\mathbf{v}}^{\mathrm{T}}\mathbf{P}_{\mathbf{A}_\perp} \tag{B.26}$$

where $(\mathbf{A})_{\mathbf{v}}$, a three-dimensional quantity, denotes the derivatives of matrix $\mathbf{A}$ w.r.t. the vector $\mathbf{v}$:

$$(\mathbf{A})_{\mathbf{v}} = \frac{\partial \mathbf{A}}{\partial \mathbf{v}} \tag{B.27}$$

Since the Jacobian matrix of criterion (B.25) can be obtained as:

$$\mathbf{J} = (\mathbf{P}_{\mathbf{A}}\mathbf{t})_{\mathbf{v}} = (\mathbf{A})_{\mathbf{v}}\mathbf{A}^{+}\mathbf{t} + \mathbf{A}(\mathbf{A}^{+})_{\mathbf{v}}\mathbf{t} \quad , \tag{B.28}$$

replacing in this last equation $(\mathbf{A}^{+})_{\mathbf{v}}$ by (B.26), Golub-Pereyra's Jacobian matrix is obtained. This matrix is denoted in Chapter 4 as (4.65), and, for convenience, is reproduced here:

$$\mathbf{J}_{\mathrm{GP}} = \mathbf{P}_{\mathbf{A}_\perp}(\mathbf{A})_{\mathbf{v}}\mathbf{A}^{+}\mathbf{t} + \mathbf{A}^{+\mathrm{T}}(\mathbf{A})_{\mathbf{v}}^{\mathrm{T}}\mathbf{P}_{\mathbf{A}_\perp}\mathbf{t} \tag{B.29}$$

This type of nonlinear least squares problems was investigated also by Kaufman [162]. She observed that the criterion (B.25) can be formulated as:

‡. In this Appendix we assume that the $\mathbf{A}$ has dimensions (m*r)

$$\psi = \frac{\left\|\mathbf{P}_{\mathbf{A}_\perp}\mathbf{t}\right\|_2^2}{2} = \frac{\left\|\mathbf{Q}_2^T\mathbf{t}\right\|_2^2}{2} \tag{B.30}$$

where $\mathbf{Q}_2$ is obtained by performing a QR decomposition of $\mathbf{A}$:

$$\mathbf{A} = \mathbf{QR} = \begin{bmatrix}\mathbf{Q}_1 & \mathbf{Q}_2\end{bmatrix}\begin{bmatrix}\mathbf{R}_{11}\\ 0\end{bmatrix} \tag{B.31}$$

Following this observation, Kaufman arrives at the conjecture:

$$(\mathbf{Q}_2^T\mathbf{t})_{\mathbf{v}_i} = \frac{\partial}{\partial \mathbf{v}_i}\mathbf{Q}_2^T\mathbf{t} = \mathbf{Q}_2^T(\mathbf{A})_{\mathbf{v}_i}\mathbf{A}^+\mathbf{t} - \mathbf{Y}_i\mathbf{Q}_2^T\mathbf{t} \qquad i = 1, \ldots, k, \tag{B.32}$$

where k is the number of nonlinear parameters, and the $\mathbf{Y}$ matrices must satisfy:

$$\mathbf{Y}_i + \mathbf{Y}_i^T = 0 \qquad i = 1, \ldots, k. \tag{B.33}$$

Since the $\mathbf{0}$ matrix satisfies (B.33), Kaufman proposed the following Jacobian matrix (denoted in Chapter 4 as (4.67)):

$$\mathbf{J}_K = \mathbf{P}_{\mathbf{A}_\perp}(\mathbf{A})_{\mathbf{v}}\mathbf{A}^+\mathbf{t} \tag{B.34}$$

To reduce the computational burden incurred in the use of the reformulated criterion for the training of MLPs, we [137] proposed the following Jacobian matrix (denoted in Chapter 4 as (4.63)):

$$\boldsymbol{J} = (\mathbf{A})_{\mathbf{v}}\mathbf{A}^+\mathbf{t}\,, \tag{B.35}$$

which, as shown in [137] and detailed later, can be obtained from (B.32) using suitable values for the $\mathbf{Y}$ matrices.

B.2 Gradient proof

It is well known that, for a least-squares problem, the gradient vector ($\mathbf{g}$) can be obtained from the Jacobian matrix ($\mathbf{J}$) and the error vector ($\mathbf{e}$) using:

$$\mathbf{g} = -\mathbf{J}^T\mathbf{e} \tag{B.36}$$

By replacing $\mathbf{J}$ with our proposed Jacobian matrix (B.35) and $\mathbf{e}$ with $\mathbf{P}_{\mathbf{A}_\perp}\mathbf{t}$, the gradient vector is given by:

$$\mathbf{g} = -\boldsymbol{J}^T\mathbf{P}_{\mathbf{A}_\perp}\mathbf{t} \tag{B.37}$$

If we replace $\mathbf{J}$ in (B.36) with the Golub-Pereyra's Jacobian matrix (B.29), the same

gradient vector is obtained, since $\mathbf{P}_{\mathbf{A}}^{\mathrm{T}}\mathbf{P}_{\mathbf{A}_{\perp}} = 0$.

If we compute the gradient of criterion (B.30) using Kaufman's Jacobian matrix (B.32), to achieve the gradient vector (B.36), the following conditions must be satisfied:

$$\mathbf{t}^{\mathrm{T}}\mathbf{Q}_2\mathbf{Y}_i^{\mathrm{T}}\mathbf{Q}_2^{\mathrm{T}}\mathbf{t} = 0 \qquad i = 1, \ldots, k \tag{B.38}$$

Since $\mathbf{Y}_i$ must satisfy (B.33), then $\mathbf{Y}_i$ can be expressed as:

$$\mathbf{Y}_i = \mathbf{X}_i^{\mathrm{T}} - \mathbf{X}_i, \tag{B.39}$$

$\mathbf{X}_i$ being any matrix of suitable dimensions.

Using a singular value decomposition [14] for $\mathbf{X}_i$:

$$\mathbf{X}_i = \mathbf{U}\mathbf{S}\mathbf{V}^{\mathrm{T}} \tag{B.40}$$

and denoting $\mathbf{V}^{\mathrm{T}}\mathbf{Q}_2^{\mathrm{T}}\mathbf{t}$ by $\mathbf{v}$ and $\mathbf{U}^{\mathrm{T}}\mathbf{Q}_2^{\mathrm{T}}\mathbf{t}$ by $\mathbf{u}$, (B.38) is always satisfied, since

$$\mathbf{u}^{\mathrm{T}}\mathbf{S}\mathbf{v} - \mathbf{v}^{\mathrm{T}}\mathbf{S}^{\mathrm{T}}\mathbf{u} = \sum_{i=1}^{k} \mathbf{u}_j\mathbf{s}_j\mathbf{v}_j - \sum_{i=1}^{k} \mathbf{v}_j\mathbf{s}_j\mathbf{u}_j = 0 \tag{B.41}$$

where the elements $\mathbf{s}_j$ denote the j^{th} singular value of the matrix $\mathbf{X}_i$.

B.3 Jacobian proof

In this section we investigate what conditions must be accomplished by the matrices $\mathbf{J}_i = (\mathbf{Q}_2^{\mathrm{T}}\mathbf{t})_{\mathbf{v}_i}$, so that Kaufman's conjecture is verified. For that purpose, we express (B.32) in the following form:

$$\mathbf{Y}_i\mathbf{Q}_2^{\mathrm{T}}\mathbf{t} = \mathbf{Q}_2^{\mathrm{T}}\mathbf{J}_i - \mathbf{J}_i \qquad i = 1, \ldots, k. \tag{B.42}$$

Denoting $\mathbf{Q}_2^{\mathrm{T}}\mathbf{t}$ by $\mathbf{t}'$, using simple algebraic manipulations, the left-hand-side of (B.42) can be expressed as:

$$\mathbf{Y}_i\mathbf{t}' = \mathbf{L}\mathbf{y}_i \tag{B.43}$$

where $\mathbf{y}_i$ is the $\frac{(m-r)(m-r-1)}{2}$ vector of independent variables of the $\mathbf{Y}_i$ matrix, i.e.

$$\mathbf{y}_i^{\mathrm{T}} = \left[\mathbf{Y}_{i_{1,2}} \ \ldots \ \mathbf{Y}_{i_{1,m-r}} \ \mathbf{Y}_{i_{2,3}} \ \ldots \ \mathbf{Y}_{i_{m-r-1,m-r}}\right] \tag{B.44}$$

The matrix $\mathbf{L}$ in (B.43) can be divided into m-r-1 column partitions, where the z^{th}

partition is given by:

$$\mathbf{L}_z = \begin{bmatrix} \mathbf{E} \\ \mathbf{b} \\ \mathbf{C} \end{bmatrix}, \tag{B.45}$$

and

- **E** is a zero matrix of size (z-1)*(m-r-z) (it exists only when z>1);

- **b** is a row vector with elements $\mathbf{t}'_{z+1} \ldots \mathbf{t}'_{m-r}$;

- **C** is a square diagonal matrix of size (m-r-z) whose diagonal elements are $-\mathbf{t}'_z$.

It was symbolically (using MAPLE [180]) and numerically (using MATLAB [133]) verified that the rank of the **L** matrix is always (m-r-1). Considering, without loss of generality, that the dependent line is the last one, this last row ($\mathbf{L}_d$) can be achieved as a linear combination of the first (m-r-1) independent rows ($\mathbf{L}_i$) of **L**:

$$\mathbf{L}_d = \mathbf{x}^T \mathbf{L}_i \tag{B.46}$$

Again using MAPLE it was found that **x** can be given by:

$$\mathbf{x}^T = -\frac{\begin{bmatrix} \mathbf{t}'_1 & \ldots & \mathbf{t}'_{m-r-1} \end{bmatrix}}{\mathbf{t}'_{m-r}} \tag{B.47}$$

In order to have a solution for (B.42), the linear dependence among the rows of its left-hand side (through **L**) must also be reflected in its right-hand-side, i.e.,

$$[\mathbf{Q}_2^T \boldsymbol{J}_i - \mathbf{J}_i]_{1 \ldots m-r-1, .} = \mathbf{x}^T [\mathbf{Q}_2^T \boldsymbol{J}_i - \mathbf{J}_i]_{m-r, .} \ , \tag{B.48}$$

where **x** is given by (B.47).

Replacing (B.47) in (B.48), and performing some manipulation, we obtain:

$$\mathbf{t}^T \mathbf{P}_{\mathbf{A}_\perp} \boldsymbol{J}_{i, .} = \mathbf{t}^T \mathbf{Q}_2 \mathbf{J}_{i, .} \ , \tag{B.49}$$

which simply states that any matrix which produces the gradient vector (B.37) is a suitable Jacobian matrix for (B.32), and can be obtained using particular values for each one of the k $\mathbf{Y}_i$ matrices. Applying the same reasoning to criterion (B.25), it can be shown that any matrix **J** that satisfies (B.50) is a suitable Jacobian matrix for this criterion.

$$\mathbf{t}^T \mathbf{P}_{\mathbf{A}_\perp} \boldsymbol{J} = \mathbf{t}^T \mathbf{P}_{\mathbf{A}_\perp} \mathbf{J} \tag{B.50}$$

As our proposed Jacobian matrix produces the required gradient vector, we conclude that it is a valid Jacobian, according to Kaufman's conjecture.