NEURAL NETWORKS IN CONTROL ENGINEERING

by W. Trossbach

April 1994

Thesis prepared in fulfilment of the requirements for the degree of M.Sc. in Electrical and Electronic Engineering

ACKNOWLEDGMENTS

Many people contributed either directly or indirectly to this thesis - some I want to mention by name.

My sincere thanks to:

- Associate Professor M. Braae for his supervision of this project.
- DRL for their financial support during 1992.
- R. Anderson (Los Alamos National Lab) for the reference material and pre-prints of papers.
- D. Sbarbaro (University of Glasgow) for the reference material, letters and telephone conversations.
- E. Barnard (University of Pretoria) and D. Weber (University of Stellenbosch) for the useful and stimulating discussions at seminars.
- UCT's Interlibrary Department for their help to locate and obtain copies of the many references that were essential for this study.
- Annabella and Brenda for their continuous support, encouragement and understanding without which this wouldn't have been possible.

TERMS OF REFERENCE

This thesis presents the results of an investigation into the possible utilization of neural networks as part of control structures.

The requirements for this research, as set out by Associate Professor M. Braae, were to :

- Establish an overview of the many different neural network based control strategies suggested in the literature.
- Motivate the selection of a control structure and network topology for the evaluation of this new field.
- Compare the performance of that structure with that of more established control strategies.
- Evaluate the viability of the neural network approach in control engineering.

SYNOPSIS

The purpose of this thesis is to investigate the viability of integrating neural networks into control structures.

These networks are an attempt to create artificial intelligent systems with the ability to learn and remember. They mathematically model the biological structure of the brain and consist of a large number of simple interconnected processing units emulating brain cells. Due to the highly parallel and consequently computationally expensive nature of these networks, intensive research in this field has only become feasible due to the availability of powerful personal computers in recent years. Consequently, attempts at exploiting the attractive learning and nonlinear optimization characteristics of neural networks have been made in most fields of science and engineering, including process control.

The control structures suggested in the literature for the inclusion of neural networks in control applications can be divided into four major classes. The first class includes approaches in which the network forms part of an adaptive mechanism which modulates the structure or parameters of the controller. In the second class the network forms part of the control loop and replaces the conventional control block, thus leading to a pure neural network control law. The third class consists of topologies in which neural networks are used to produce models of the system which are then utilized in the control structure, whilst the fourth category includes suggestions which are specific to the problem or system structure and not suitable for a generic neural network-based-approach to control problems.

Although several of these approaches show promising results, only model based structures are evaluated in this thesis. This is due to the fact that many of the topologies in other classes require system estimation to produce the desired network output during training, whereas the training data for network models is obtained directly by sampling the system input(s) and output(s). Furthermore, many suggested structures lack the mathematical motivation to consider them for a general structure, whilst the neural network model topologies form natural extensions of their linear model based origins.

Since it is impractical and often impossible to collect sufficient training data prior to implementing the neural network based control structure, the network models have to be suited to on-line training during operation. This limits the choice of network topologies for models to those that can be trained on a sample by sample basis (pattern learning) and furthermore are capable of learning even when the variation in training data is relatively slow as is the case for most controlled dynamic systems.

A study of feedforward topologies (one of the main classes of networks) shows that the multilayer perceptron network with its backpropagation training is well suited to model nonlinear mappings but fails to learn and generalize when subjected to slow varying training data. This is due to the global input interpretation of this structure, in which any input affects all hidden nodes such that no effective partitioning of the input space can be

achieved. This problem is overcome in a less flexible feedforward structure, known as regular Gaussian network. In this network, the response of each hidden node is limited to a sphere around its center and these centers are fixed in a uniform distribution over the entire input space. Each input to such a network is therefore interpreted locally and only effects nodes with their centers in close proximity.

A deficiency common to all feedforward networks, when considered as models for dynamic systems, is their inability to conserve previous outputs and states for future predictions. Since this absence of dynamic capability requires the user to identify the order of the system prior to training and is therefore not entirely self-learning, more advanced network topologies are investigated. The most versatile of these structures, known as a fully recurrent network, re-uses the previous state of each of its nodes for subsequent outputs. However, despite its superior modelling capability, the tests performed using the Williams and Zipser training algorithm show that such structures often fail to converge and require excessive computing power and time, when increased in size.

Despite its rigid structure and lack of dynamic capability, the regular Gaussian network produces the most reliable and robust models and was therefore selected for the evaluations in this study.

To overcome the network initialization problem, found when using a pure neural network model, a combination structure in which the network operates in parallel with a mathematical model is suggested. This approach allows the controller to be implemented without any prior network training and initially relies purely on the mathematical model, much like conventional approaches. The network portion is then trained during on-line operation in order to improve the model. Once trained, the enhanced model can be used to improve the system response, since model exactness plays an important role in the control action achievable with model based structures.

The applicability of control structures based on neural network models is evaluated by comparing the performance of two network approaches to that of a linear structure, using a simulation of a nonlinear tank system.

The first network controller is developed from the internal model control (IMC) structure, which includes a forward and inverse model of the system to be controlled. Both models can be replaced by a combination of mathematical and neural topologies, the network portion of which is trained on-line to compensate for the discrepancies between the linear model and nonlinear system. Since the network has no dynamic capacity, former system outputs are used as inputs to the forward and inverse model. Due to this direct feedback, the trained structure can be tuned to perform within limits not achievable using a conventional linear system.

As mentioned previously the IMC structure uses both forward and inverse models. Since the control law requires that these models are exact inverses, an iterative inversion algorithm has to be used to improve the values produced by the inverse combination model. Due to

deadtimes and right-half-plane zeroes, many systems are furthermore not directly invertible. Whilst such unstable elements can be removed from mathematical models, the inverse network is trained directly from the forward model and can not be compensated. These problems could be overcome by a control structure for which only a forward model is required.

The neural predictive controller (NPC) presents such a topology. Based on the optimal control philosophy, this structure uses a model to predict several future outputs. The errors between these and the desired output are then collected to form the cost function, which may also include other factors such as the magnitude of the change in input. The input value that optimally fulfils all the objectives used to formulate the cost function, can then be found by locating its minimum. Since the model in this structure includes a neural network, the optimization can not be formulated in a closed mathematical form and has to be performed using a numerical method. For the NPC topology, as for the neural network IMC structure, former system outputs are fed back to the model and again the trained network approach produces results not achievable with a linear model. Due to the single network approach, the NPC topology furthermore overcomes the limitations described for the neural network IMC structure and can be extended to include multivariable systems.

This study shows that the nonlinear modelling capability of neural networks can be exploited to produce learning control structures with improved responses for nonlinear systems. Many of the difficulties described, are due to the computational burden of these networks and associated algorithms. These are likely to become less significant due to the rapid development in computer technology and advances in neural network hardware.

Although neural network based control structures are unlikely to replace the well understood linear topologies, which are adequate for the majority of applications, they might present a practical alternative where (due to nonlinearity or modelling errors) the conventional controller can not achieve the required control action.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

NOMENCLATURE

NN	- neural network
RBF	- radial basis function (network)
RGN	- regular Gaussian network
IMC	- internal model control
NNIMC	- neural network internal model control
NPC	- neural predictive control
N_1	- minimum output prediction horizon for NPC
N_2	- maximum output prediction horizon for NPC
N_u	- control horizon for NPC
y_s	- desired output
y_p	- system output
y_m	- model output
u	- input
a	- scalar value
$\underline{a}$ or **a**	- vector
A	- matrix
ACC	- American Control Conference
IICNN	- First International Conference on Neural Networks
IJCNN	- First Joint Conference on Neural Networks

1. NEURAL NETWORKS - AN INTRODUCTION AND OVERVIEW

Artificial neural networks and parallel distributed processing are currently receiving significant interest from many scientific disciplines, in the hope that they might help to produce solutions where conventional methods have not been satisfactory.

This introductory chapter starts with a brief historical discussion showing the origin of neural networks as well as their place within the field known as artificial intelligence or AI. After this the underlying biological principle for all these networks is explained, before an overview of the remainder of this study is given.

1.1 HISTORICAL DEVELOPMENTS

Since the early days of modern science there has been considerable interest and effort to understand the underlying principles that constitute intelligence. This curiosity about the brain and the thinking process is to be expected from related fields such as physiology and psychology. In other disciplines of science however, it can be ascribed to the desire of re-creating features such as the abilities to learn and to reason in artificial intelligent systems. Most of the early research work performed in this area, consisted of experiments conducted to understand basic processes such as color perception and recognition as well as the functioning of single nerve cells. This was done in the hope that a thorough knowledge of the elementary principles might help to eventually understand the entire system.

Although this *bottom-up* approach was and is producing valuable results, it soon became clear that it would not necessarily deliver a satisfactory explanation for the storage, retrieval and processing of knowledge. This insight inspired research of a *top-down* nature, in which theories for the above mechanisms were developed with little concern about their implementation in the underlying biological system.

With the advent of the modern computer in the nineteen fifties it became feasible to implement and test some of these hypothesis for the first time. This attracted many non-physiologists and gave a new definition to this field, now well known as artificial intelligence or AI. Although computers provided a platform for testing many of the theories, their sequential execution and symbol manipulation capability also injected some bias into the formulation of new algorithms. Consequently, theories lending themselves towards the implementation on a *von Neumann* type architecture were often favoured above more biologically plausible ones.

The lack of agreement on the appropriate form of knowledge representation raised the fundamental question, that remains unresolved, of whether the purpose of AI is to produce *clever* computer programs or to understand and model human intelligence. It also resulted in a division of the discipline, producing the two sub-disciplines known as symbolic AI and nonsymbolic AI.

1.1.1 SYMBOLIC AI

In this approach to AI, knowledge is represented and stored in the form of symbols, which can be operated on by a set of rules.

The most well known result of this approach is the *expert system*, for which the knowledge of one or more people, considered to be experts on the particular subject, is captured in a knowledge database. This database can be interrogated by a set of rules to produce the desired output. A simplistic view of expert systems is a collection of *if-then-else* rules, which are applied using various mechanisms such as applicability, thresholds and priority.

One of the features that distinguish expert systems from other AI systems is that the knowledge base is designed prior to their use and therefore rigid and nonadaptive. Although the system might *learn* to adapt its response by modifying thresholds and priorities, it can not alter or add information. While this might seem like a deficiency for a supposedly intelligent system, it does have the advantage that the system is guaranteed to work within its capabilities from the outset and does not require further training or development.

Expert systems have been implemented successfully in applications where the necessary knowledge could be extracted and captured in the required format. These applications are in areas where decisions have to be made in accordance with well known rules and constraints. Typical examples of such systems are *expert advisers*, which assist people in fields such as medical diagnosis or technical design.

Despite the fact that the field of symbolic AI produced impressive results relatively quickly, it has become apparent that this approach alone will not be able to deliver the initial promise of *artificial systems with human like intelligence* [1]. This is because the human brain is not a sequential, analytical machine operating on a knowledge base and although the symbolic approach lends itself for implementation on a *von Neumann* architecture, it is not appropriate as a model for the information representation and processing in the brain. In his review on AI and the brain, Smolensky [2] notes that:

- The symbolic approach 'has provided precious little insight into the computational organization of the brain'.
- The 'fine structure of cognition seems to be more naturally described by non-symbolic models'.
- 'The symbolic rules and the logic used to manipulate those rules tend to produce rigid and brittle systems'.

1.1.2 NONSYMBOLIC (CONNECTIONIST) AI

Whilst the early supporters of symbolic AI were trying to create an environment in which knowledge was presented in a format suitable to simulate reasoning on a digital computer, another group of researchers attempted to solve the problem by creating mathematical representations of what was known about the physical brain structure. In these approaches, which became known as connectionist AI, the system generally consists of a large number of highly interconnected simple processing units, much like the brain with all its nerve cells and synaptic links. Researchers in this field of study, also known as connectionism due to the great importance placed on interconnectivity, were thus attempting to create artificial intelligence by modelling the biological structure of the brain, rather than striving to formulate an abstract knowledge representation format suitable for manipulation.

The main difference between these theories and symbolic AI is therefore the representation of knowledge. While the expert system relies on an exact representation of knowledge in the form of a rule operating on symbols, no equivalent format exists in the connectionist approaches. In the *neural network*, which is one such nonsymbolic model, the information is stored in the connection strengths (equivalent to biological synaptic links) between individual nodes and is therefore scattered over part or even the entire network, resulting in the so-called *distributed* representation. Due to this distributed representation it is not possible to program or store all initial knowledge in the network and the system has to learn from a set of input and associated desired output patterns, before it can be used.

In spite of some exciting and promising results such as the perceptron model and associated convergence procedure [3] during the early nineteen sixties, connectionism faded into the background during the seventies. This was due to a number of reasons:

- Although the research had shown some promise, it had failed to deliver anything comparable to the reliable, *seemingly clever* systems the symbolic field was able to produce.
- In a time when the required computing resources where expensive and rare, it was difficult to motivate the use of precious computing time on a research area which is not suited to serial processing and the von Neumann architecture.
- In their book 'Perceptrons' [4] Minsky (by then a well known personality in the field of AI) and Papert proved the limitation of a single layer perceptron model (a neural network model receiving considerable interest at the time) and suggested that an extension to multilayer systems would not produce any worthwhile results; a statement which greatly discouraged funding for further research in the field.

Despite these setbacks, research in the non-symbolic field continued and the availability of cheap computing power in the eighties, together with the realization that symbolic AI might not deliver what it promised [1], generated renewed interest. This revival has resulted in many different connectionist models and learning algorithms. Currently neural

networks, probably the most well known product of the connectionist field, are tested on a wide variety of problems ranging from character - or sound recognition to stock market prediction and in this study the applicability of these networks in the field of process control is investigated.

Presently both, the symbolic and nonsymbolic approaches are continuing to produce promising results, each in their own areas of application. Some scientists believe that only a synthesis of the two methods will result in true artificial intelligence [2].

1.2 NEURAL NETWORK PRINCIPLES (BIOLOGICAL MOTIVATION)

Despite the fact that even amongst connectionists there is no agreement regarding the neural network topology and learning mechanisms, most networks use building blocks very similar to biological neurons. In this subsection the close link between these building blocks and their biological counterparts in the brain is clarified. The elementary processing unit is illustrated in figure 1.1.

STRUCTURE OF ELEMENTARY NEURAL NETWORK PROCESSING UNIT

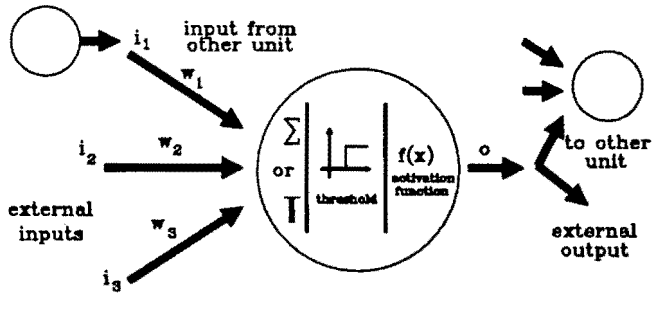


Figure 1.1 - Elementary Processing Unit

As shown, the unit has a number of inputs which might be outputs from other units (i_1) or external inputs (i_2,i_3) from the environment. Each of these inputs is scaled according to its weight or importance (w_1,w_2,w_3) for this unit. The scaled inputs are then summed or multiplied to form a scalar activation value for the unit. If this activation value exceeds the threshold value of the particular unit, it is passed through the nonlinear activation function f(x) to produce an output (o). The output of the particular unit may in turn form an input to other units and/or be used as an external output as indicated. In order to create a neural network, a number of these processing units are linked together. Training for such a system

entails finding the optimal set of connection weights and thresholds such that the network produces the desired external outputs for each set of external inputs.

From the above discussion and illustration it can be seen that the elementary processing unit for neural networks is a mathematical representation of the biological neuron with its cell body, activation threshold, dendrites and axons, as well as synaptic links of varying strengths with other neurons. Although the adjustment of weights can furthermore be likened to the known biological process of establishing and modifying synaptic links between neurons during learning, the underlying mechanism for these adaptations in living systems is not yet fully understood. Consequently many different weight-update theories and algorithms exist, each claiming to be more biologically plausible than others.

Even though most artificial neural networks utilize processing units very similar to the one illustrated above, there exist many ways of interconnecting these units (network topologies) and finding the optimal set of parameters for the network (training algorithms). A discussion of several network topologies and training algorithms is included in later chapters of this thesis, where several networks are evaluated. An overview of the different types of networks and their origins can be found in the discussions by Grossberg [5], Lippmann [6] as well as Rumelhart et al [7].

1.3 OVERVIEW OF THE REMAINING CHAPTERS

The remainder of this study deals with an investigation into the use of neural networks in the field of process control.

- Work previously performed in this area is included in the literature review in chapter 2, which also contains an explanation and motivation for the choice of control structures and networks to be studied.
- In chapters 3 and 4, different types of network topologies are evaluated to determine their suitability for the chosen control algorithms.
- In chapter 5 two neural network based control strategies and their linear system counterparts are compared.
- Finally conclusions as to the suitability of neural networks as part of control structures and algorithms are drawn.

REFERENCES

1.....Dreyfus H.L and Dreyfus S.E., "*KÜNSTLICHE INTELLIGENZ - VON DEN GRENZEN DER DENKMASCHINE UND DEM WERT DER INTUITION*", orig. "*MIND OVER MATTER*", New York, The Free Press, 1986.

2.....Smolensky P., "*CONNECTIONIST AI, SYMBOLIC AI, AND THE BRAIN*", Artificial Intelligence Review, 1987, pp. 95-109.

3.....Rosenblatt F., "*PRINCIPLES OF NEURODYNAMICS: PERCEPTRONS AND THE THEORY OF BRAIN MECHANISMS*", Washington, DC: Sparta Books, 1962.

4.....Minsky M., Seymour P., "*PERCEPTRONS: AN INTRODUCTION TO COMPUTATIONAL GEOMETRY*", Cambridge, MA: MIT Press, 1969.

5.....Grossberg S., "*NONLINEAR NEURAL NETWORKS: PRINCIPLES, MECHANISMS, AND ARCHITECTURES*", Neural Networks, Vol. 1, 1988, pp. 17-61.

6.....Lippmann R.P., "*AN INTRODUCTION TO COMPUTING WITH NEURAL NETS*", IEEE ASSP Magazine, 1987, pp. 4-22.

7.....Rumelhart D.E., McClelland J.L. and the PDP research group, "*PARALLEL DISTRIBUTED PROCESSING - EXPLORATION IN THE MICROSTRUCTURE OF COGNITION, Volume 1: Foundations, ch. 1: The Appeal of PDP*", Cambridge, MA: MIT Press, 1986.

2. NEURAL NETWORKS FOR PROCESS CONTROL

Due to the predominantly discrete nature of the first networks, most of the early connectionist efforts looked at utilizing networks for pattern recognition and classification problems, such as machine vision and speech analysis. The change to continuous nonlinearities (e.g. the perceptron with sigmoidal transfer function) enabled a mathematical analysis of the approximation or learning problem and led to several different formulations of the so-called *existence theorem*. These proofs generally show that, provided the network contains sufficient hidden nodes, it can approximate any mapping with any desired accuracy.

One of the earliest and better known of these theorems was not new, but rather a re-phrasing of Kolmogorov's 1957 solution to the 13th problem of Hilbert[1] by Hecht-Nielsen [2]. Since the *rediscovery* of Kolmogorov there have been a host of other theorems [3], [4] and today almost every neural network publication refers to one of them. The detail of a more general version, as suggested by Kreinovich [5], is presented in Appendix I.

Once proven, the functional approximation capabilities of neural networks caused considerable interest from many fields, such as process control where the modelling and estimation of unknown and possibly nonlinear systems or functions forms an important part of the design.

This chapter provides an overview of recent attempts to utilize neural networks in the field of control engineering. After this the choice of networks and topologies to be studied in the remainder of this thesis is explained and motivated. Finally a set of requirements and criteria for these networks is established.

2.1 NEURAL NETWORK APPLICATIONS IN PROCESS CONTROL

The fact that connectionist networks have been developed as mathematical models of the human brain invokes expectations of unsupervised, self-organizing control structures with human-like performance. However, until the principle for learning in the brain has been fully understood and models of the size and capacity of the human brain can both be developed and managed (the human brain contains in excess of 10 billion neurons whilst even large application artificial neural networks seldom contain more than a few hundred units), such implementations will remain impractical and unreliable.

Due to the above reason, most practical efforts towards the integration of artificial neural networks into process control problems, implement the network in a well defined role in

order to exploit its nonlinear approximation and learning abilities, rather than attempting to reproduce *human-like* controllers. This section provides an overview of such attempts to incorporate neural networks as part of a control structure or algorithm. Due to the fast changing nature and amount of neural network research, the discussion is not intended as exhaustive, authoritive study of the field but rather as background for the remainder of this thesis.

Although each of the approaches described is essentially different, an attempt has been made to group them into applications which:

- Utilize a neural network as part of an adaptive mechanism, external to the control loop.
- Use a neural network in the position of the conventional controller.
- Include a forward or inverse neural network model in the control structure.
- Use a unique structure not fitting any of the above.
- Are very specific to the problem studied.

Since these categories are not mutually exclusive, some methods fit more than one description and in such cases the dominant characteristic is used for classification.

2.1.1 NEURAL NETWORKS IN ADAPTIVE MECHANISMS

In these applications the neural network is not situated in the direct path of the control loop, but forms part of an external adaptive mechanism. The network is usually used to either switch or modulate the parameters of a fixed control law as illustrated in figure 2.1.

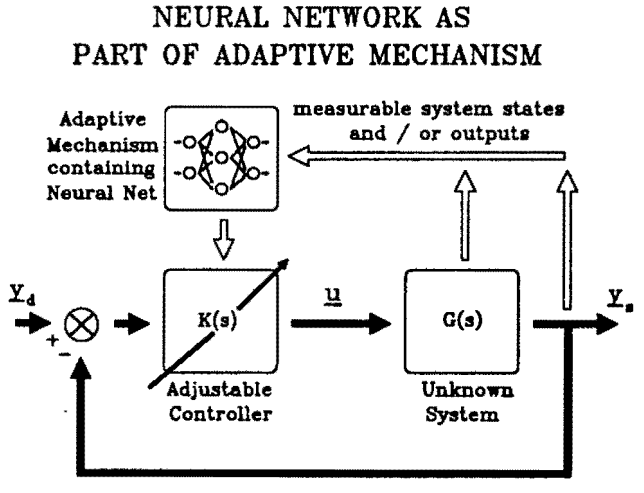


Figure 2.1 - Adaptive Structure

Guez et al [6] for example, employ a neural estimator in a 'model reference adaptive control' or MRAC configuration to modify the parameters of the controller depending on the current basin of attraction in the optimization space. This is achieved by implementing

a Cohen-Grossberg type network which adjusts the feedback gains in the controller according to the estimation of the unknown coefficients for a process of known dynamic behavior.

In a more recent paper Kumar and Guez [7] propose the use of an adaptive resonance theory (or ART-II) network to implement a pole placement algorithm. The network is used as a nearest neighbor type classifier, which categorizes the underlying system using preprocessed step response information. The network output is then utilized in the pole placement algorithm to modify the controller parameters.

In a similar type of application Cooper et al [8] utilize a Kohonen type network to implement pattern based control, in which the gain of an existing control algorithm is updated by analyzing disturbance patterns in the output of a controlled system. The network is employed to classify the disturbance patterns as true load disturbances (which warrant a change in the control law) and purely oscillatory disturbances (which should be ignored).

Another adaptive mechanism is proposed by Cheok and Smith [9], who suggest the training of a network to recall the parameters of a number of discrete pre-designed controllers as a function of one or more system parameters. Due to the interpolation property of neural networks, the final adaptive mechanism should then provide one smooth, continuous and adjustable control algorithm from all these controllers.

2.1.2 NEURAL NETWORKS IN PLACE OF CONVENTIONAL CONTROLLERS

Figure 2.2 depicts the control loop structure for the algorithms included in this subset. As illustrated, the neural network occupies the place of the conventional controller and is an integral part of the feedback loop.

NEURAL NETWORK IN
CONVENTIONAL CONTROLLER POSITION

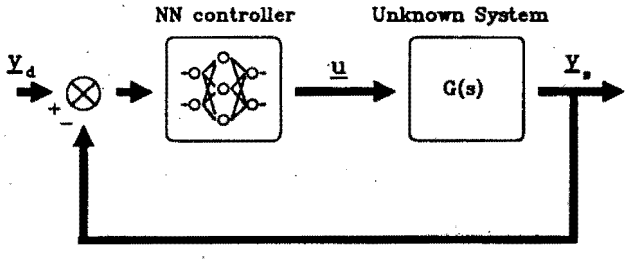


Figure 2.2 - NN Controller Structure

In the configuration shown, the network needs to be trained in such a way that it produces the required input signal(s) $\underline{u}$ to ensure that the system output(s) $\underline{y}_s$ reach(es) the desired state $\underline{y}_d$. Since these required inputs are not known *a priori* for an unknown system, the basic difference between algorithms included in this subsection is the method used to generate or estimate an error signal to train the network. The approaches incorporated in this category can therefore be further classified according to the type of training algorithm and network used.

(a) Multilayer Perceptrons and Backpropagation

The main obstacle with this gradient descend algorithm (discussed in detail in chapter 3) is that the error is only measurable at the system output and would have to be *backpropagated* through the unknown system to allow updating of the neural network weights during training.

One possible method to overcome this problem is to train the network to emulate an existing pre-designed control law. This approach, included in the discussions by Hampo et al [10] and Karsai [11], might be considered where the existing controller is in fact a person regulating the system or where the copied neural network provides a more cost-efficient alternative than the existing system.

A similar approach was adopted by Kong and Kosko [12]. In their comparative study of a fuzzy and neural network controller they use estimation and the outputs of the fuzzy control law to generate the ideal network outputs during training.

Another possibility suggested by Hampo and Marko [10] is to generate the network error, by utilizing the system output error in a cost function. Cramer and Womack [13] take this approach one step further and use the sum of errors between desired and actual plant states, directly, to train the neural network controller.

In their study Cui and Shin [14] overcome the problem by extending the conventional back propagation algorithm to allow training of the controller using the error between desired and actual output(s). The approach is novel since only the direction of the plant response is required as a priori knowledge.

Narendra et al [15] and Troudet et al [16] present a further alternative for training a multilayer perceptron network in the controller position. They use an MRAC structure in which a filter or reference model generates the desired closed system responses and implement a pre-trained neural network model in place of the system during training. This allows conventional error backpropagation through this 'system model' to the neural controller.

The approach is also adopted by Nguyen and Widrow [17] in their famous example of the *truck backer-upper* in which they first train a forward neural network model, which is then used to backpropagate the final position error to the controller network during training.

(b) Non-gradient Search Algorithms

Hsuing et al [18] as well as Moore [19] avoid the problem of error backpropagation through an unknown system by utilizing reinforcement learning as credit assignment algorithm for their multilayer perceptron networks. With reinforcement learning the search for the *best* parameters is guided by evaluating the effect of exploratory steps in the weight space. Hsuing et al point out that although this optimization method is slower than most gradient algorithms, it makes provision for a larger variety of objectives.

Similar non-gradient algorithm based approaches are discussed by Cotter et al [20] who compare a biased search with simulated annealing for training a recursive network. Both of these approaches do not require the calculation of an error gradient and are therefore suitable for this structure.

In another study, Anderson and Vemuri [21] use their chemotaxis search algorithm, based on the movement of bacteria in a liquid with varying concentrations, to show that neural networks can be trained to generate time-optimal control signals in an open loop configuration.

(c) Genetic Algorithms

Genetic algorithms present a further possible technique for training networks in the conventional controller position. This optimization technique, based on the process of evolution, uses concepts such as crossover, mutation and inversion to develop *offspring* with a higher degree of *fitness* as determined by some objective function, which selects the parents.

In their paper Ichikawa and Sawa [22] implement a modified version of this procedure to train a feedforward neural network controller. The genetic approach is also adopted by Wieland [23] in his study on the use of recurrent networks to control unstable systems.

2.1.3 CONTROL STRUCTURES CONTAINING FORWARD OR INVERSE NEURAL NETWORK MODELS

Approaches included in this subsection differ from those discussed previously, in that they utilize a neural network model of the forward or inverse plant characteristic as part of the control structure.

A frequently used method of this type is to train a network to estimate the inverse of the plant characteristic. In other words, the network would be exercised to reproduce the input(s) from the resulting system output(s). Once trained such a network forms the *ideal* control law since it predicts the required inputs for the system to generate the desired outputs.

In one of the earlier studies regarding the use of neural networks in control engineering, Psaltis et al [24] investigate this option. Their investigation focuses on several possible inverse system training architectures which have since been adopted in other investigations [25].

The concept of an inverse neural network model has also been adopted in an open loop control structure in several other studies. Typical examples are the work of Steck et al [26] on distillation columns as well as Levin et al [27] who investigate the use of a delayed-input delayed-state network in this configuration.

A control structure, including a direct neural network model of the system, is presented by Willis et al [28], [29]. They suggest the use of a neural network model to predict future outputs for a DMC-like optimization technique, which determines future inputs to minimize the deviations between neural network predicted and desired system outputs.

In their attempts, Hoskins et al [30] also implement a direct model of the system, together with an iterative inversion algorithm, to generate the required control signals.

Another control structure containing a neural network to estimate part of a forward model, is suggested by Spall and Cristion [31]. They examine the use of a stochastic approximation technique based on *simultaneous perturbation*, rather than *gradient estimation*, to estimate the unknown component of the transfer function in order to implement an adaptive control law.

Cui and Shin [32] also utilize a forward neural network predictor in their multi-system coordinator, which uses the output of the model to synchronize the interaction between several linked systems, such as two robot arms holding one object.

The configuration proposed by Wu et al [33] contains both a forward and inverse model in the control loop and uses the former to allow backpropagation training of the inverse or controller network. The difference between this and similar approaches discussed earlier, is that both models remain in the system and are trained on-line.

In their employment of neural networks in an internal model control (IMC) framework, Hunt and Sbarbaro [25] suggest yet another structure containing both a forward and an inverse model.

2.1.4 OTHER TOPOLOGIES

This section includes attempts in which the suggested control structure does not fit into any of the above classifications.

The integration of several neural networks into one control structure has been the topic of several research studies. Narendra and Mukhopadhyay [34] for example, suggest the use of a two level neural network based controller, where the higher level detects the current plant configuration and activates the required lower level control law. In another study Narenda and Levin [35] show that a multiple-network approach might be superior for the regulation of nonlinear dynamic systems with multiple equilibrium states.

Another multi-network approach is suggested by Jacobs and Jordan [36], who use a *gating* network to mediate the competition between several *expert* networks (which learn the training patterns) in a gain scheduling methodology.

The idea of several control surfaces is also supported by Barto et al [37], who suggest the use of a layered associative network to transform a nonlinear control problem into a presentation which can then be solved linearly.

An entirely different control structure is described in the work by Iiguni et al [38] in which the integration of neural networks into the classical linear optimal regulator structure is investigated. The networks are included to overcome the slight discrepancies and uncertainties between the actual system and the model used to develop the optimal control law.

Moore and Naidu [39] present yet another alternative by showing that a finite-horizon linear quadratic regulator problem can be transformed into a nonlinear programming problem, which can then be solved by a Hopfield type neural network.

A further novel approach is suggested by Berenji [40]. He suggests an integration of fuzzy systems and neural networks, to allow the storage of *a priori* knowledge, whilst retaining the capability to learn and discusses the suitability of the new model for several space applications.

The work by Sznaier and Damborg [41] shows another dimension of utilizing neural networks in process control. They exploit the speed of an analog neural network circuit to perform on-line optimization for constrained linear systems.

2.1.5 PROBLEM SPECIFIC APPROACHES

Approaches included in this subsection are very particular to the problem studied and therefore not applicable to other control problems, without modification.

Examples of studies which are restricted to systems with a certain structure are the work of Chen [42] as well as Guez et al [43]. Both studies restrict themselves to a certain transfer function format and are hence only applicable to a small subset of control problems.

Other investigations are completely specific to the system under investigation. Typical examples in this group are the CMAC network for biped walking by Miller et al [44]; a visual servo system which learns to move the manipulator into the correct position to grab an object by Hashimoto et al [45]; vibration cancellation by Bozich and MacKay [46] and the application of a neural network in a military flight control system by Steinberg and DiGirolamo [47].

Of the many applications presented in the previous sections, only a subset are suitable for a generic neural network controller. The selection of the topologies investigated in this study, is motivated in the following section, which also introduces a set of criteria and requirements for the type of neural network to be used in these approaches.

2.2 CHOICE OF CONTROL TOPOLOGIES AND NETWORK REQUIREMENTS

Although the list of possible implementations, presented in the previous section, contains many promising and novel possibilities, the remainder of this study will be concentrated on applications containing direct and inverse neural network models.

There are two reasons for this focus:

- Firstly, the creation of valid and accurate models of dynamic systems is in itself an important area within the field of process control. Using the model based approach allows an examination of the suitability of neural networks as dynamic models.
- Secondly, the model based topologies (examined later in chapter 5) are based on well known control structures and laws, the mathematical validity of which has been established even for nonlinear systems. This is not the case for many of the other approaches.

The selection of model based control structures affects the choice of neural network topologies and training algorithms. In the remainder of this section certain preferences for the networks to be investigated are explained and a set of requirements for neural network based dynamic models is established.

2.2.1 DISCRETE OR CONTINUOUS NETWORKS

Although the continuous nature of the systems to be modelled calls for networks with continuous outputs, it is possible to implement a discrete input/output network with analog-to-digital and digital-to-analog converters as depicted in figure 2.3. An additional hamming network may also be included between the network outputs and D/A, in order to force the output vector into the nearest of the corners in the n-dimensional output space hypercube, before reconverting.

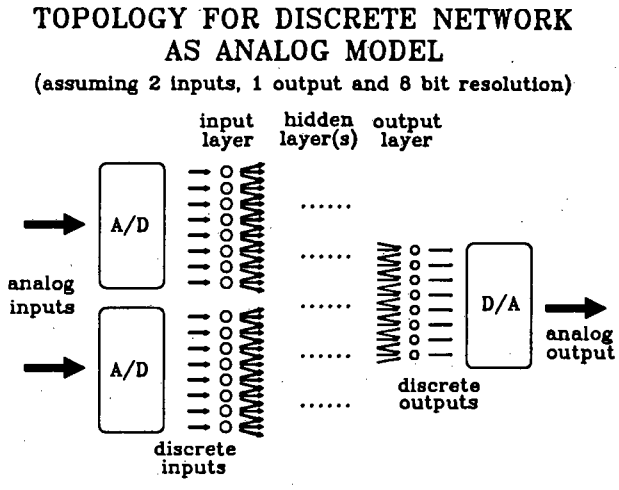


Figure 2.3 - Discrete NN Model

Such an arrangement is favoured by the fact that most modern control loops contain a digital computer, in which the continuous signals do already exist as digitized numbers. A further advantage of the depicted arrangement is the possibility of utilizing typical classification networks (such as Grossberg's ART and ART-II), otherwise not suitable for these applications.

The use of discrete networks also entails certain disadvantages. Both the network size and input/output dimensionality is much higher than for continuous networks. Assuming common twelve bit resolution for the converters, a two-input single-output continuous network for example, would result in a twentyfour-input, twelve-output discrete network. A further disadvantage of the discrete implementation is the importance or weight associated with the higher order bits in a digital representation. This importance, together with the fact that the output of each discrete node is interpreted to be in either one of only two states, makes errors in these higher order output bits both likely and costly. For these reasons, it was decided to concentrate this study on continuous output networks.

2.2.2 LEARNING AND TRAINING REQUIREMENTS FOR NEURAL NETWORK MODELS

Since the networks investigated in this study are to model the forward or inverse characteristics of a dynamic system, they are trained in supervised mode. This method of training is performed by presenting the network with an input vector, to which the desired output is known, and then adjusting the network parameters to minimize the difference between network and desired response.

For the models of a dynamic system these input/desired output pairs are actual samples of the systems input(s) and output(s), which have to be collected during operation. Assuming that it is possible to collect sufficient data from the system initially, the network models could be trained off-line prior to their implementation in the control loop.

Such an initial collection of data is not unique to this method since most conventional controller design methods require an initial identification process, where a mathematical model is either derived from, or validated against, actual process data.

Since the model has to be representative for the entire operating space of the process, it is essential that the collection of system data includes sufficient samples from all regions within this space. Adequate coverage of the operating space can however only be achieved, if the system input is perturbed in a sufficiently turbulent manner. Such harsh regulation of the input variables is in most cases unacceptable both for the equipment involved, as well as the underlying process. Hence, a more controlled method consisting of a set of regulated step changes in the system input(s), to observe their effect on the output(s), is generally employed.

Figure 2.4 below shows a typical set of such step perturbations for a laboratory system, in which the level in a tank with constant inflow is regulated both above and below an overflow pipe, using an automatic control valve in the tank's outlet.

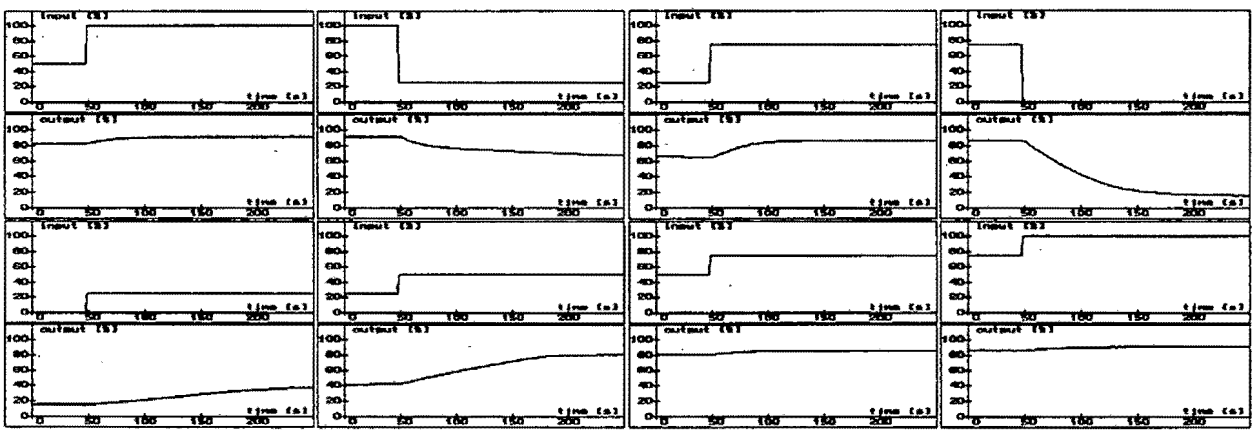


Figure 2.4 - Step Test Results

It is important to note that the range of perturbations chosen for the above set of steptests covers almost the entire input space for the process. Such generous steps are generally not

permissible in an industrial environment, where inputs usually have to remain within a few percent of their normal operating ranges.

Although the above set of steptest results would be more than sufficient to develop a linear model for the system, this is not necessarily true for a neural network model. Figure 2.5 shows the distribution of the collected samples in a normalized input space (assuming a first order model i.e. y[n+1] = f(u[n],y[n])) and illustrates that the distribution of samples is very localized in spite of these liberal input perturbations.

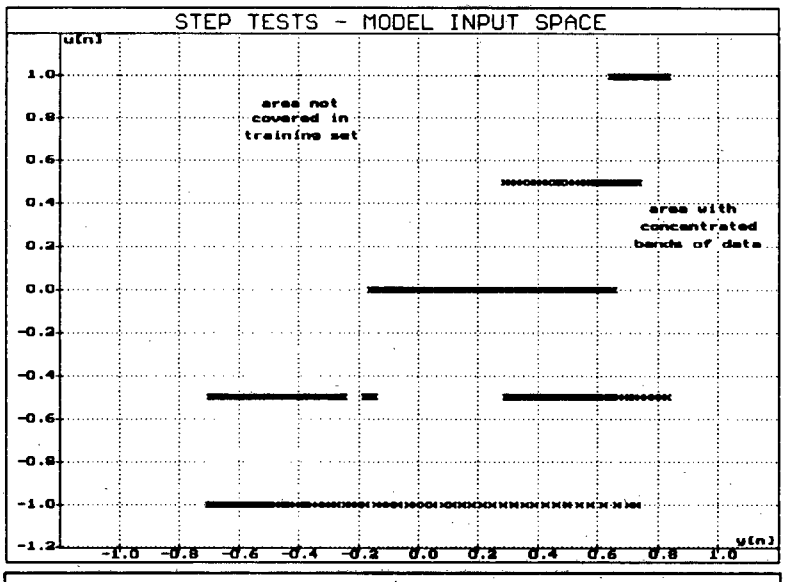


Figure 2.5 - Input Space

As expected the samples are confined to narrow bands of constant input values and the space includes many areas with few or no data points. This poor distribution of samples, even in an laboratory environment, highlights a number of learning and training requirements for neural network models of dynamic systems.

(a) Generalization Capability (Interpolation and Extrapolation)

The training of neural network models for dynamic systems has to be performed using discrete data points collected from the system to be modelled.

As shown, the number of training samples is generally limited and often local in nature (i.e. due to practical constraints the training data only covers certain regions of the system's operating sufficiently). The ability of the network to interpolate between these discrete points and localized regions, as well as to extrapolate into the areas with few or no training samples, is therefore one of the important requirements for neural network models.

(b) On-line or Off-line Training

The previous example illustrates that although theoretically possible, the collection of sufficient training data for off-line training prior to using the network, is not feasible in practice.

Due to the localized and scattered nature of the training data, even a network with excellent interpolation and extrapolation capability can not be trained to sufficient accuracy using only sampled system data. This is particularly true for regions in the operating space, which the system may never enter in an open-loop configuration, but might be forced into, in a demanding closed-loop configuration. Such regions would consequently not be covered adequately in the sampled data and only an on-line training algorithm could ensure an accurate model.

On-line training of neural network models, incorporated in control structures, is also attractive as it facilitates compensation for slow time-varying processes. In other words, on-line training ensures that the control system will remain optimal even for changing processes (such as chemical reactions involving catalysts), provided that the system change is slow in comparison to the network learning rate.

This requirement, of training the network models in an on-line configuration, favours certain types of training philosophies as discussed in the following subsection.

(c) Pattern Learning or Batch Learning

Much of the early and current neural network research looks at classification problems, where a sufficiently large and representative set of training data is available. Consequently, many of the training algorithms utilize the entire training set during learning and evaluate the error over all samples between each weight update. These *batch-learning* algorithms are not suited for on-line training, since some method of managing the training set would be required to prevent uncontrollable sizes.

Pattern-learning algorithms on the other hand, use only the current input/output pair to determine the error and weight changes and are therefore preferable for on-line training.

(d) Storing and Extracting Knowledge

The motivation for employing a non-parametric modelling technique, such as a neural network, is usually that the underlying system is unknown and difficult to capture or describe, using more conventional methods.

In spite of this, the designer often has some knowledge of the process behavior. Capturing this information prior to training would be beneficial for both, the speed of learning as well as interpolation and extrapolation into regions not covered by training data.

Similarly, the extraction of knowledge from a network can be useful for judging the training performance, identifying areas with insufficient training data, assessing the network topology and size, obtaining information about the modelled process and integrating neural networks with other methods such as fuzzy logic.

(e) Computational Effort

Since the neural network forward and/or inverse model are to be implemented as part of the control structure and are to be trained on-line, both the network output and training algorithms have to be such, that they can be carried out between successive samples without compromising the quality of the control loop.

(f) Ease of Use

One of the motivations for integrating neural networks into control structures is that these networks are self-adjusting and learning. Such new control structures can only be considered viable alternatives to existing algorithms, if they are easy to apply. The network initialization and training should therefore not require any in-depth knowledge or understanding of neural network theory.

The preceding discussion and other network or algorithmic specific criteria are used to evaluate the suitability of a number of neural network structures and training algorithms in chapters 3 and 4.

REFERENCES

1.....Kolmogorov A.N., *"ON THE REPRESENTATION OF CONTINUOUS FUNCTIONS OF MANY VARIABLES BY SUPERPOSITION OF CONTINUOUS FUNCTIONS OF ONE VARIABLE AND ADDITION"*, Dokl. Akad. Nauk USSR, 114, 1957, pp. 935-956.

2.....Hecht-Nielsen R., *"KOLMOGOROV'S MAPPING NEURAL NETWORK EXISTENCE THEOREM"*, IEEE International Conference on Neural Networks 1987, pp. III11-III14.

3.....Hornik K., Stinchcombe M. and White H., *"MULTILAYER FEEDFORWARD NETWORKS ARE UNIVERSAL APPROXIMATORS"*, Neural Networks, Vol. 2, 1989, pp. 359-366.

4.....Cybenko G., *"APPROXIMATION BY SUPERPOSITION OF A SIGMOIDAL FUNCTION"*, Mathematics of Control, Signals, and Systems, Vol. 2, 1989, pp. 303-314.

5.....Kreinovich V. Y., *"ARBITRARY NONLINEARITY IS SUFFICIENT TO REPRESENT ALL FUNCTIONS BY NEURAL NETWORKS: A THEOREM"*, Neural Networks, Vol. 4, 1991, pp. 381-383.

6.....Guez A., Eilbert J.L. and Kam M., *"NEURAL NETWORK ARCHITECTURE FOR CONTROL"*, IEEE Control Systems Magazine, April 1988, pp. 23-25.

7.....Kumar S.S. and Guez A., *"ART BASED ADAPTIVE POLE PLACEMENT FOR NEUROCONTROLLERS"*, Neural Networks, Vol. 4, 1991, pp. 319-335.

8.....Cooper D.J., Megan L. and Hinde R.F., *"A NEURAL PATTERN ANALYZER FOR ADAPTIVE PROCESS CONTROL"*, ACC 1991, pp. 2794-2799.

9.....Cheok K.C. and Smith J.C., *"ADAPTIVE NEURAL NETWORK CONTROL WITH FREQUENCY-SHAPED OPTIMAL OUTPUT FEEDBACK"*, International Joint Conference on Neural Networks 1991, pp. II741-II746.

10....Hampo R. and Marko K., *"NEURAL NETWORK ARCHITECTURES FOR ACTIVE SUSPENSION CONTROL"*, IJCNN 1991, pp. II765-II770.

11....Karsai G., *"LEARNING TO CONTROL: SOME PRACTICAL EXPERIMENTS WITH NEURAL NETWORKS"*, IJCNN 1991, pp. II701-II707.

12....Kong S.G. and Kosko B., *"ADAPTIVE FUZZY SYSTEMS FOR BACKING UP A TRUCK-AND-TRAILER"*, IEEE Transactions on Neural Networks, Vol. 3, No. 2, pp. 211-223, 1992.

13....Cramer J.E. and Womack B.F., *"ADAPTIVE CONTROL USING NEURAL NETWORKS"*, ACC 1991, pp. 681-686.

14....Cui X. and Shin K.G., *"DESIGN OF AN INDUSTRIAL PROCESS CONTROLLER USING NEURAL NETWORKS"*, ACC 1991, pp. 508-513.

15....Narendra K.S. and Parthasarathy K., *"IDENTIFICATION AND CONTROL OF DYNAMICAL SYSTEMS USING NEURAL NETWORKS"*, IEEE Transactions on Neural Networks, Vol. 1, No. 1, 1990, pp. 4-27.

16....Troudet T., Garg S., Mattern D. and Merril W., *"TOWARDS PRACTICAL CONTROL DESIGN USING NEURAL COMPUTATION"*, IJCNN 1991, pp. II675-II681.

17....Nguyen H. and Widrow B., *"NEURAL NETWORKS FOR SELF-LEARNING CONTROL SYSTEMS"*, IEEE Control Systems Magazine, April 1990, pp. 18-23.

18....Hsuing J.T. and Himmelblau D.M., *"DEVELOPMENT OF CONTROL STRATEGIES VIA ARTIFICIAL NEURAL NETWORKS AND REINFORCEMENT LEARNING"*, ACC 1991, pp. 2326-2330.

19....Moore K.L., *"A REINFORCEMENT-LEARNING NEURAL NETWORK FOR THE CONTROL OF NONLINEAR SYSTEMS"*, ACC 1991, pp. 21-22.

20.....Cotter N.E., Guillerm T.M., Soller J.B. and Conwell P.R, "*PREJUDICIAL SEARCHES AND THE POLE BALANCER*", IJCNN 1991, pp. II689-II694.

21.....Anderson R.W. and Vemuri V., "*NEURAL NETWORKS CAN BE USED FOR OPEN-LOOP, DYNAMICAL CONTROL*", preprint, to appear in International Journal of Neural Networks, 1991.

22.....Ichikawa Y. and Sawa T., "*NEURAL NETWORK APPLICATION FOR DIRECT FEEDBACK CONTROLLERS*", IEEE Transactions on Neural Networks, Vol. 3, No. 2, March 1992, pp. 224-231.

23.....Wieland A.P., "*EVOLVING NEURAL NETWORK CONTROLLERS FOR UNSTABLE SYSTEMS*", IJCNN 1991, pp. II667-II673.

24.....Psaltis D., Sideris A. and Yamamura A., "*NEURAL CONTROLLERS*", International Conference on Neural Networks 1987, pp. IV551-IV558.

25.....Hunt K.J. and Sbarbaro D., "*NEURAL NETWORKS FOR NONLINEAR INTERNAL MODEL CONTROL*", IEE Proceedings-D, Vol. 138, No. 5, 1991, pp. 431-438.

26.....Steck J., Krishnamurthy K., McMillin B. and Leiniger G., "*NEURAL MODELING AND CONTROL OF A DISTILLATION COLUMN*", IJCNN 1992, pp. II771-II774.

27.....Levin E., Gewirtzman R. and Inbar G.F., "*NEURAL NETWORK ARCHITECTURE FOR ADAPTIVE SYSTEM MODELING AND CONTROL*", Neural Networks, Vol. 4, 1991, pp. 185-191.

28.....Willis M.J., Di Massimo C., Montague G.A., Tham M.T. and Morris A.J., "*ARTIFICIAL NEURAL NETWORKS IN PROCESS ENGINEERING*", IEE Proceedings-D, Vol. 138, No. 3, 1991, pp. 256-266.

29.....Willis M.J., Di Massimo C., Montague G.A., Tham M.T. and Morris A.J., "*NON-LINEAR PREDICTIVE CONTROL USING OPTIMIZATION TECHNIQUES*", ACC 1992, pp. 2788-2793.

30.....Hoskins D.A., Hwang J.N. and Vagners, "*ITERATIVE INVERSION OF NEURAL NETWORKS AND ITS APPLICATION TO ADAPTIVE CONTROL*", IEEE Transactions on Neural Networks, Vol. 3, No. 2, 1992, pp. 292-301.

31.....Spall J.C. and Cristion J.A., "*EFFICIENT WEIGHT ESTIMATION IN NEURAL NETWORKS FOR ADAPTIVE CONTROL*", ACC 1991, pp. 16-20.

32.....Cui X. and Shin K.G., "*INTELLIGENT COORDINATION OF MULTIPLE SYSTEMS WITH NEURAL NETWORKS*", ACC 1991, pp. 481-486.

33.....Wu Q.H., Hogg B.W. and Irwin G.W., "A *NEURAL NETWORK REGULATOR FOR TURBOGENERATORS*", IEEE Transactions on Neural Networks, Vol. 3, No. 1, 1992, pp. 95-100.

34.....Narendra K.S. and Mukhopadhyay S., "*INTELLIGENT CONTROL USING NEURAL NETWORKS*", ACC 1991, pp. 1069-1074.

35.....Narendra K.S. and Levin A.U., "*REGULATION OF NONLINEAR DYNAMICAL SYSTEMS USING MULTIPLE NEURAL NETWORKS*", ACC 1991, pp. 1609-1614.

36.....Jacobs R.A. and Jordan M.I., "*A MODULAR CONNECTIONIST ARCHITECTURE FOR LEARNING PIECEWISE CONTROL STRATEGIES*", ACC 1991, pp. 1597-1602.

37.....Barto A.G., Anderson C.W. and Sutton R.S., "*SYNTHESIS OF NONLINEAR CONTROL SURFACES BY A LAYERED ASSOCIATIVE SEARCH NETWORK*", Biological Cybernetics, 43, 1982, pp. 175-185.

38.....Iiguni Y., Sakai H. and 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.

39.....Moore K.L. and Naidu S., "*LINEAR QUADRATIC REGULATION USING NEURAL NETWORKS*", IJCNN 1991, pp. II735-II739.

40.....Berenji H.R., "*ARTIFICIAL NEURAL NETWORKS AND APPROXIMATE REASONING FOR INTELLIGENT CONTROL IN SPACE*", ACC 1991, pp. 1075-1080.

41.....Sznaier M. and Damborg M.J., "*AN ANALOG "NEURAL NET" BASED SUBOPTIMAL CONTROLLER FOR CONSTRAINED DISCRETE-TIME LINEAR SYSTEMS*", Automatica, Vol. 28, No. 1, 1992, pp. 139-144.

42.....Chen F.C., "*BACK-PROPAGATION NEURAL NETWORKS FOR NONLINEAR SELF-TUNING ADAPTIVE CONTROL*", IEEE Control Systems Magazine, April 1990, pp. 44-47.

43.....Guez A. and Piovoso M., "*CUSTOM NEUROCONTROLLER FOR A TIME DELAY PROCESS*", ACC 1991, pp. 1592-1596.

44.....Miller W.T., Latham P.J. and Scalera S.M., "*BIPEDAL GAIT ADAPTION FOR WALKING WITH DYNAMIC BALANCE*", ACC 1991, pp. 1603-1608.

45.....Hashimoto H., Kubota T., Kudou M. and Harashima F., "*SELF-ORGANIZING VISUAL SERVO SYSTEM BASED ON NEURAL NETWORKS*", ACC 1991, pp. 2262-2267.

46.....Bozich D.J. and MacKay H.B., "*VIBRATION CANCELLATION AT MULTIPLE LOCATIONS USING NEUROCONTROLLERS WITH REAL-TIME LEARNING*", IJCNN 1991, pp. II775-II780.

47.....Steinberg M. and DiGirolamo R., "*APPLYING NEURAL NETWORK TECHNOLOGY TO FUTURE GENERATION MILITARY FLIGHT CONTROL SYSTEMS*", presented at IJCNN 1991.

3. FEEDFORWARD NETWORKS

In the previous chapter the framework for the remainder of this study was established and a set of requirements for neural networks to be used as forward or inverse models of dynamic models was derived. These requirements and other criteria are used in this chapter to evaluate the suitability of feedforward networks.

3.1 GENERAL TOPOLOGY

The general topology of feedforward neural networks is depicted in figure 3.1 below.

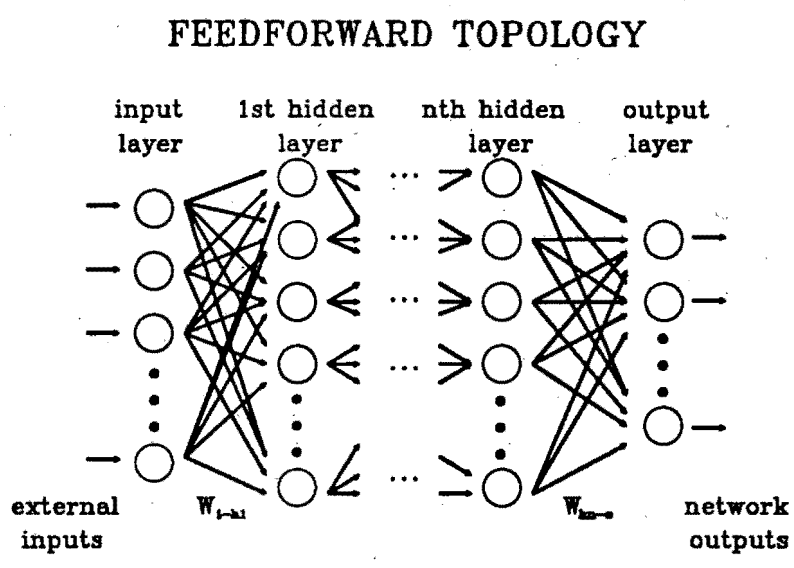


Figure 3.1 - Topology of Feedforward Networks

As shown, these networks consist of an input layer, one or more hidden layers and an output layer, each consisting of one or more nodes. The nodes in the input layer usually perform no processing and just serve to redistribute the external inputs to the first hidden layer, via a set of connection weights ($W_{i\text{-}h1}$). At each of the hidden nodes the scaled inputs are then processed (e.g. summed or multiplied) before they are passed through an activation function, to form the outputs of that layer. These outputs are then re-scaled before they become the inputs of the next hidden or output layer. The nodes in the final or output layer again collect all their inputs scaled via $W_{hn\text{-}o}$ and produce the network's output vector by passing these scalars through an activation function.

Feedforward networks derive their name from the fact that the outputs of each layer only feed into the next layer and not to nodes within the same or previous layers. In other words the network generates an output vector by propagating the input vector, in a forward direction, until it appears at the network outputs.

Learning or optimization of such a structure consists of adjusting the weights associated with the connections between layers, as well as other activation function particular parameters, in order to minimize the error between desired and network outputs for all training data samples.

Networks of the feedforward type are further classified according to the type of activation function used, the number of hidden layers utilized and the kind of training algorithm employed. In the following two subsections multilayer perceptrons and radial basis function feedforward networks are examined in detail.

3.2 MULTILAYER PERCEPTRON

This type of network, with its origin in some of the earliest work performed in the field [1], was made famous by Rumelhart and McClelland [2] and is today probably the most well known and most often applied neural network.

3.2.1 TOPOLOGY

Figure 3.2 depicts the topology of the multilayer perceptrons investigated in this study.

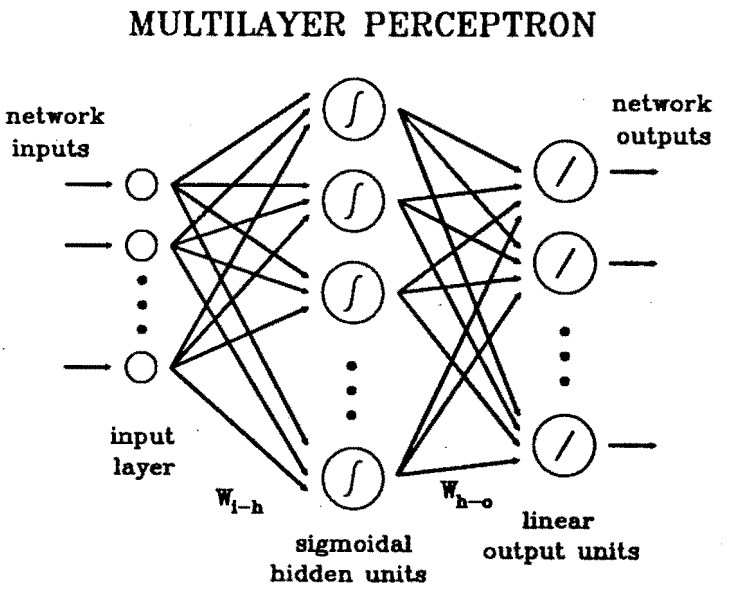


Figure 3.2 - Multilayer Perceptron Topology

As shown, the network uses a single hidden layer and employs a logistic type activation function in its hidden nodes. Both choices are explained in detail below.

In the remainder of this subsection multilayer perceptrons, of the topology shown, will be denoted as l-m-n networks, where l, m and n present the number of nodes in the input, hidden and output layers respectively.

Although many multilayer perceptron studies include a node with constant output in their hidden layer to compensate for any dc offset, it was found that the inflexibility of such a constant node limits the training rates achievable before instability. Using one of the standard hidden nodes to learn any constant offset instead, allows higher learning rates and accelerates the training process.

(a) Number of Hidden Layers

As indicated in the general topology in figure 3.1, feedforward networks (and hence multilayer perceptrons) can have any number of hidden layers.

In spite of the fact that networks with sufficient nodes in a single hidden layer have been proven capable of approximating any continuous function with any desired accuracy [3], many applications (in particular classification problems) still use two-hidden layer networks. The motivation for the two-layer topology is that the first layer is expected to perform some pre-processing on the raw data, before presenting it in a more manageable format for the second hidden layer, to complete the classification or identification task.

In a recent paper comparing single and two-hidden layer networks for classification problems however, de Villiers and Barnard [4] conclude that:

- there is no statistically significant difference between the optimal performance of the two networks
- single-hidden layer networks do better classification on average
- two-hidden layer networks are more prone to the *local minima problem* during training.

According to the research, the use of more than one hidden layer thus offers no advantages. Consequently, only single-hidden layer perceptrons are investigated in this study.

(b) Type of Activation Function

Figure 3.3 depicts the logistic activation function (suggested by Rumelhart and McClelland) employed in the hidden nodes for all multilayer perceptrons used in this evaluation.

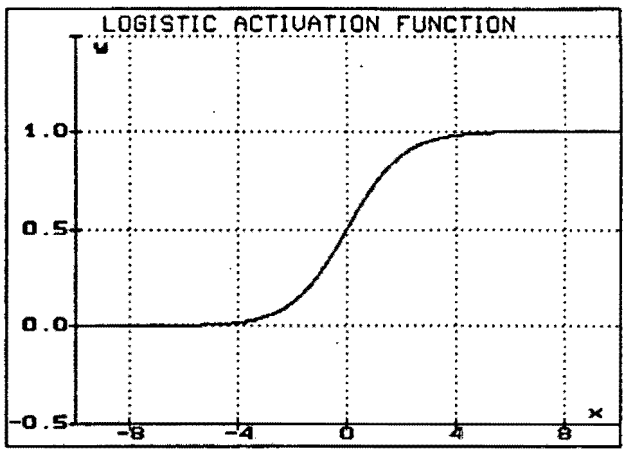


Figure 3.3 - Sigmoidal Activation Function

The sigmoidal shape of the function highlights its origin as a continuous alternative for the hard-limiter in the pattern classification field, where the output of each unit is expected to be either on or off. For such classification applications, where the output of each node is trained to be in either of the two binary states, other similarly shaped functions, with larger derivatives, have been shown to produce faster learning rates for gradient descent training algorithms [5]. Since the nonlinear portions of the sigmoid, rather than its saturated regions, are used in function approximation problems, the above accelerated shifting is not desirable. In their Taylor series analysis of the approximation property of neural networks, Wray and Green [6] also suggest that for approximation applications, different activation functions only vary the way individual weights contribute to the coefficients in the polynomial series representation.

The above function is furthermore attractive since the ability of approximating any continuous function by superpositions of a sigmoidal function has been proven mathematically [7] and since its derivative can be evaluated in a convenient closed form for gradient descent training, as shown in the following subsection.

3.2.2 TRAINING ALGORITHMS

As with all networks, training a multilayer perceptron consists of finding a set of connection weights and other activation function specific parameters, such that the error between desired and actual outputs is sufficiently small for all input/output pairs used during training.

Training is therefore an n-dimensional, non-linear optimization problem where n presents the number of unknown network parameters to be determined.

Since the mathematical format of the underlying function and hence also the error surface is unknown, the optimization has to be performed using numerical search or optimization algorithms which rely solely on the evaluation of this function. Due to the absence of one guaranteed numerical solution to optimization problems of this nature, there have been

many different attempts, using widely differing algorithms. All these algorithms perform a search and can usually be classified as either gradient descent or directed search methods. Whilst the former employ an estimation of the error function gradient to direct their search, the latter are usually guided by modelling some physical or biological optimization process, such as the cooling of a metal or the movement of bacteria in a liquid.

(a) Gradient Descent Search Algorithms

As mentioned previously, algorithms of this type direct their search along the actual or approximated gradient of the error or cost function. *Backpropagation* or the *generalized delta rule*, popularized by Rumelhart et al [2] is currently the most well known and most applied supervised neural network training algorithm.

The following equations summarize both the forward and backpropagation rules for a multilayer perceptron. A detailed mathematical treatment of the algorithm can be found in [2].

Writing the equations governing the forward propagation for each node in the network as follows,

$$x_{pj} = \sum_i (w_{ij} o_{pi})$$

$$o_{pj} = f(x_{pj}, \theta)$$

where - j - the layer of this node
i - the previous layer
x_{pj} - the input to node j for pattern p
$f()$ - the activation function
o_{pj} - the output of node j for pattern p
w_{ij} - the weights connecting the outputs of the previous layer to this node
θ - an activation function dependent parameter

the change in weight according to the generalized delta rule is given by

$$\Delta_p w_{ij} = \eta \delta_{pj} o_{pj}$$

with

$$\delta_{pj} = (t_{pj} - o_{pj}) f'(x_{pj}) \quad \text{if } j \text{ is a node in the output layer}$$

where t_{pj} is the target output for pattern p

and

$$\delta_{pj} = f'(x_{pj}) \sum_k (\delta_{pk} w_{kj}) \quad \text{for a node in a hidden layer}$$

Since this initial format, there have been many attempts at improving the above or finding better gradient training algorithms. Most of these efforts aim to accelerate the learning rate using methods such as the inclusion of second order derivatives [8][9], re-scaling of variables during training [10][11][12], or adding other features to the algorithm [13][14][15][16].

Due to the need to estimate the gradient, all these algorithms do restrict the choice of network topology and type of activation function. This limitation, together with the fact that the generalized delta rule, or any other gradient descent algorithm, are unlikely candidates for biological learning in the brain, inspired some researchers to look for non-gradient methods.

(b) Directed Search Algorithms

These methods usually model a physical or biological optimization process in their search for the optimal set of network parameters. The *chemotaxis* and *simulated annealing* methods included below, provide some insight into this type of algorithm. It should be noted that, although these methods are presented as alternatives for backpropagation, they are not restricted to multilayer perceptrons and feedforward networks, since they do not assume a fixed network topology or activation function.

(i) Chemotaxis

This optimization method, developed by Bremermann and Anderson [17] and [18], resembles bacterial movement in a medium with varying concentration of chemoattractants. In such a medium, bacteria generate random directions until they detect an increase in concentration. Movement along that direction is then sustained until there is no more increase, at which stage they stop and repeat their search for a new direction.

For optimization purposes, the algorithm thus implements intermittent random searches with sustained movements along declining slopes, once found.

The actual steps of computation for the chemotaxis algorithm are shown below.

---

1. Initialize all network parameters to small random values.

2. Present all p input patterns i_p and calculate the corresponding output patterns o_p.

3. Determine the 'energy' (error) of the objective function E_1 over the whole data set.

   - typically $E_1 = \sum_{p}\sum_{k}(t_k - o_k)^2$ where : t_k are target outputs
     o_k are network outputs
     k is the number of outputs
     p is the number of training samples

4. Generate random weight adjustment vector Δw from Gaussian distribution with mean $\mu=0$ and std. deviation $\sigma=1$.

5. Increment weights with random vector
   i.e. $w = w + H\Delta w$ where H is an adjustable step size.

6. Calculate energy E_2 with new set of weights.

7. If $E_2 < E_1$ then
   retain the modified steps
   set E_1 to E_2
   go to step 5
   if $E_2 > E_1$ then
   retain the old weights
   go to step 4

During these iterations the stepsize H can be increased if successive moves along the same direction yield better energy values or decreased if several attempts do not produce a better set of weights.

As the above equations show, the algorithm does not implement any gradient estimation and is therefore completely independent of the network topology and choice of activation functions. This flexibility was exploited by Willis et al [19] to include adjustable first order responses for each node in a multilayer perceptron, a modification not easily possible with backpropagation.

(ii) Simulated Annealing

This training algorithm, based on the work of Kirkpatrick [20] and used extensively by Hopfield and Tank [21] is an analogy to the way metals cool and anneal to a state of minimal internal energy.

As with the chemotaxis algorithm, the weights are adjusted by a random amount between each evaluation of the cost or energy function. Whilst only steps which reduce the overall energy (error) are implemented in the chemotaxis algorithm however, the simulated annealing method sometimes allows parameter variations which increase the error, depending on the temperature of the system. This is achieved by using the so-called Boltzmann probability distribution, to ensure that parameter changes resulting in a lower state of energy are always followed, whilst those resulting in a higher error might occur at high enough temperatures. This facility of temporarily moving to a state of higher energy provides the system with the flexibility to escape local minima and move to another basin of attraction, provided that the temperature is sufficiently high to allow such a change.

A general description of the equations governing the simulated annealing algorithm are shown below.

---

1. Initialize all network parameters to small random values.

2. Present all p input patterns i_p and calculate the corresponding output patterns o_p.

3. Determine the 'energy' (error) of the objective function E_1 over the whole data set.

- typically $E_1 = \sum_{p}\sum_{k}(t_k - o_k)^2$ where : t_k are target outputs
  o_k are network outputs
  k is the number of outputs
  p is the number of training samples

4. Generate random weight adjustment vector Δw
   from Gaussian distribution with mean $\mu=0$ and std. deviation $\sigma=1$.

5. Increment weights with random vector
   i.e. $w = w + H\Delta w$ where H is an adjustable step size.

6. Calculate energy E_2 with new set of weights.

7. Calculate probability of retaining new weights.

   - typically $\rho = e^{-(E_1 - E_2)/kT}$ where k is Boltzmans constant
     T is the system temperature

     if $\rho>1$ then $\rho=1$ (i.e. $E_2 < E_1$)

   if $\rho>0.5$ then retain new weights irrespective whether $E_2 > E_1$
     else keep previous weights

   Reduce temperature T.

   Go to step 4.

Although the above description assumes that the optimal solution can be reached by gradually reducing the temperature only once, this is not necessarily the case. In their discussion, Ingman and Merlis [22] suggest that the system be melted and solidified several times, each time escaping from the local basin of attraction and moving closer to the global minimum.

Even though both non-gradient methods discussed above offer attractive features such as increased flexibility in the choice of network topologies and activation functions, they both require a comprehensive set of training patterns and are not suited for on-line learning. In other words, both these algorithms are of the batch learning type (where the error is evaluated over the entire set of training samples) and are therefore inappropriate for the applications studied in this work, as explained in the previous chapter.

Despite the fact that the backpropagation algorithm, discussed in section (a), should also be used in a batch learning configuration (i.e. all weight adjustments are collected and only implemented once the entire set of training patterns has been presented) to ensure true gradient descent, Rumelhart et al [2] claim that the pattern learning version (weight updates are effected after each pattern) presented above, will produce the same results, provided the training rate is kept small. This is confirmed by the work of Qin et al [23], in which they show that the pattern learning generalized delta rule for feedforward networks is in fact a first order approximation of the equivalent batch learning rule and therefore equivalent for small learning rates. Qin et al also show that pattern learning is stable for

higher learning rates than batch learning. They ascribe this to the fact that batch learning *overshoots* the minimum for high learning rates, whilst the continuous updating of weights prevents this in pattern learning.

Of all algorithms discussed only the generalized delta rule is suitable for pattern and hence on-line training. All training in the following evaluation of the multilayer perceptron was therefore done using backpropagation.

3.2.3 NONLINEAR MODELLING CAPABILITY

In this subsection the function approximation capability of multilayer perceptron networks with a single hidden layer and using the logistic function nonlinearity is illustrated. Although this ability can and has been proven mathematically [24], an understanding of the underlying mechanism is useful for the evaluation of these networks.

As mentioned earlier, training modulates the weights and activation function parameters in order to obtain the best possible fit for all training patterns. In figure 3.4, the modelling capability of a single hidden node with a single input is demonstrated.

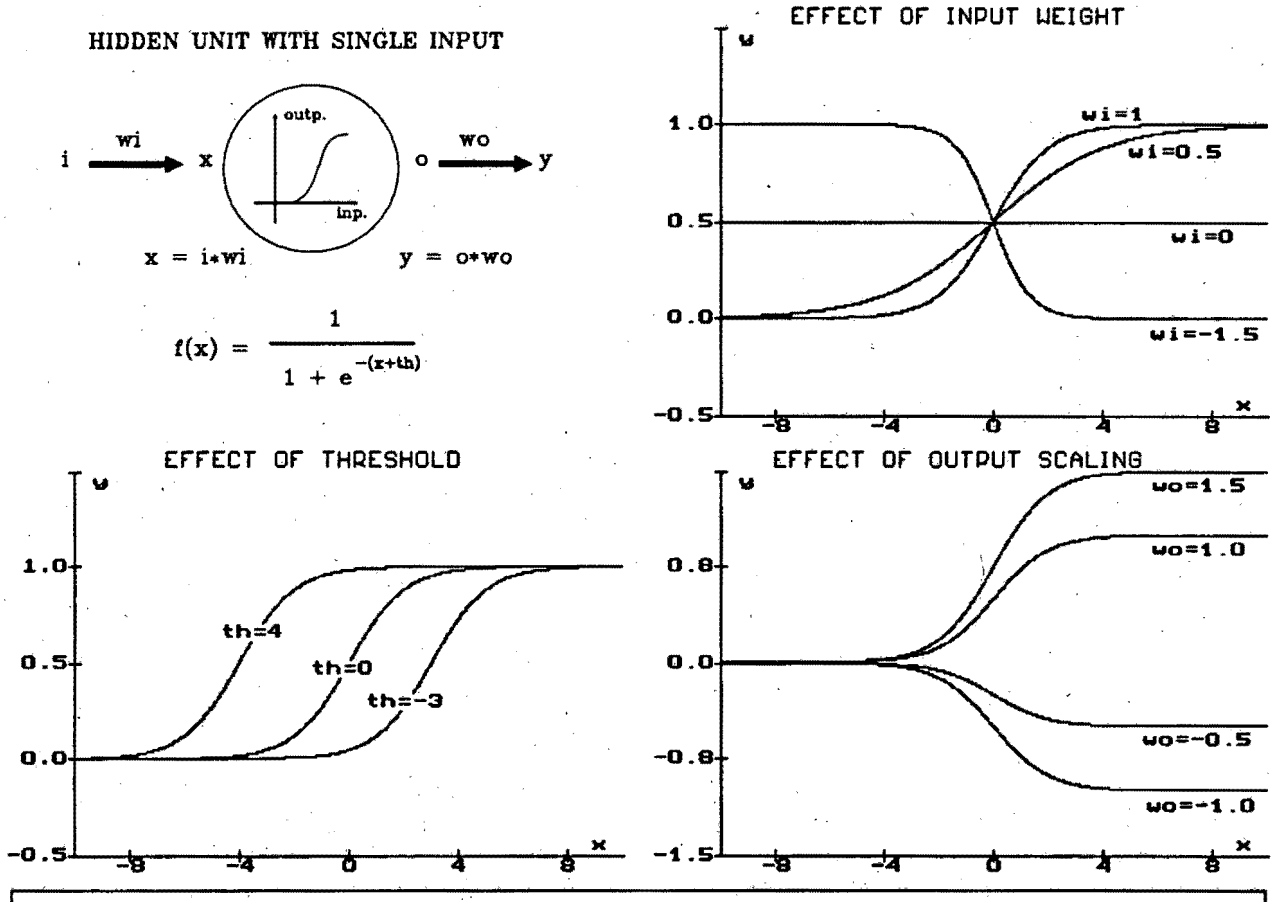


Figure 3.4 - Modelling Capability of Single Hidden Node

The figure illustrates that the output of a hidden node can be scaled and shifted to any desired size and position by varying the input scaling, threshold and output scaling.

Whilst threshold and output scaling have the same effect for nodes with more than one input, it is worthwhile to picture the re-organizing effect of modifying the input weights in a multi- dimensional space, rather than in one dimension using the scalar input to the unit.

Figure 3.5 shows that for such nodes, scaling of the individual inputs changes the orientation of the sigmoidal surface in the input space.

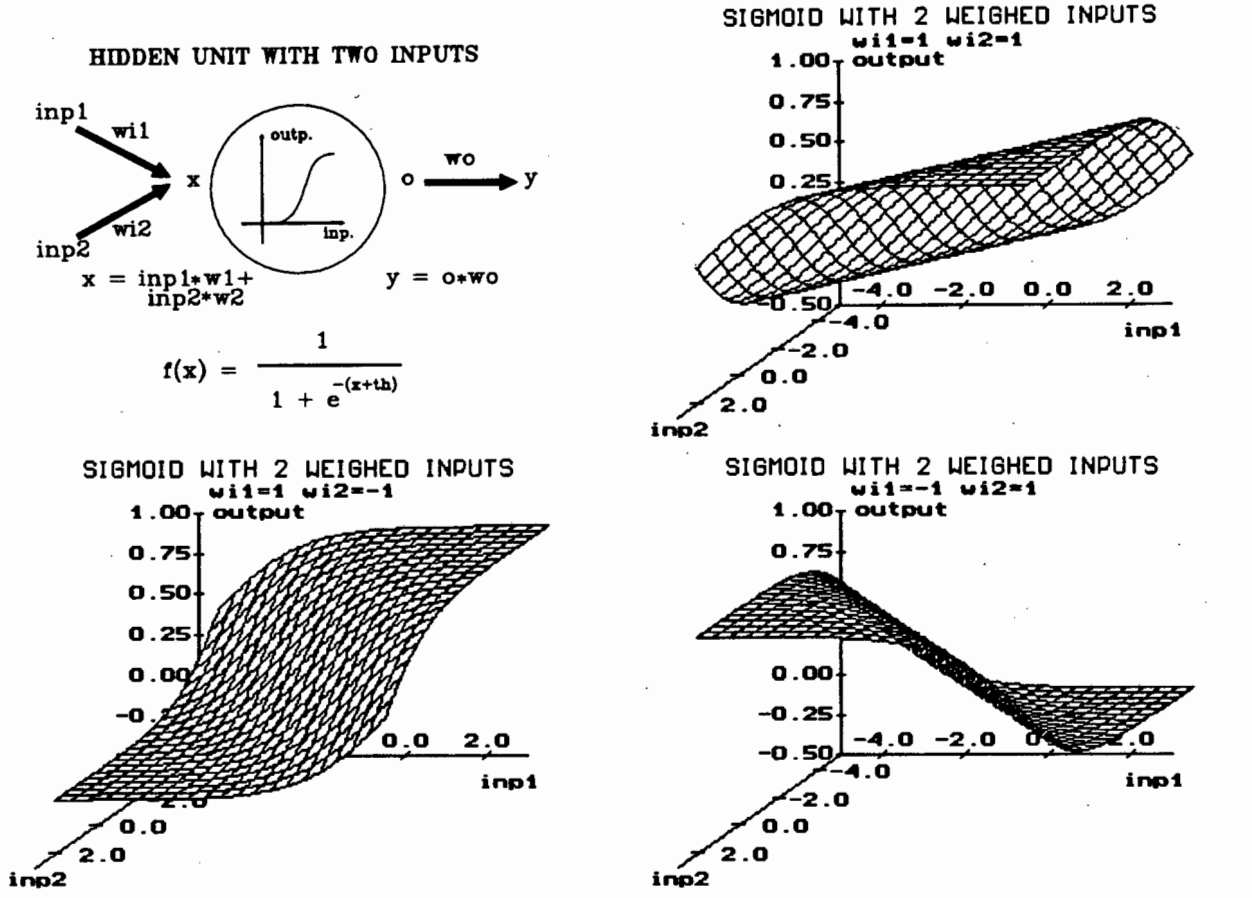


Figure 3.5 - Orientation of Sigmoidal Surface in 2-dimensional Input Space

By adding the outputs of a number of these scaled hidden node outputs, a multilayer perceptron can approximate any nonlinear function. This is illustrated in figures 3.6(a) and 3.6(b), which depict the learning process of a a 2-5-1 multilayer perceptron modelling a paraboloid, as well as the contribution of each of the hidden nodes in the trained network.

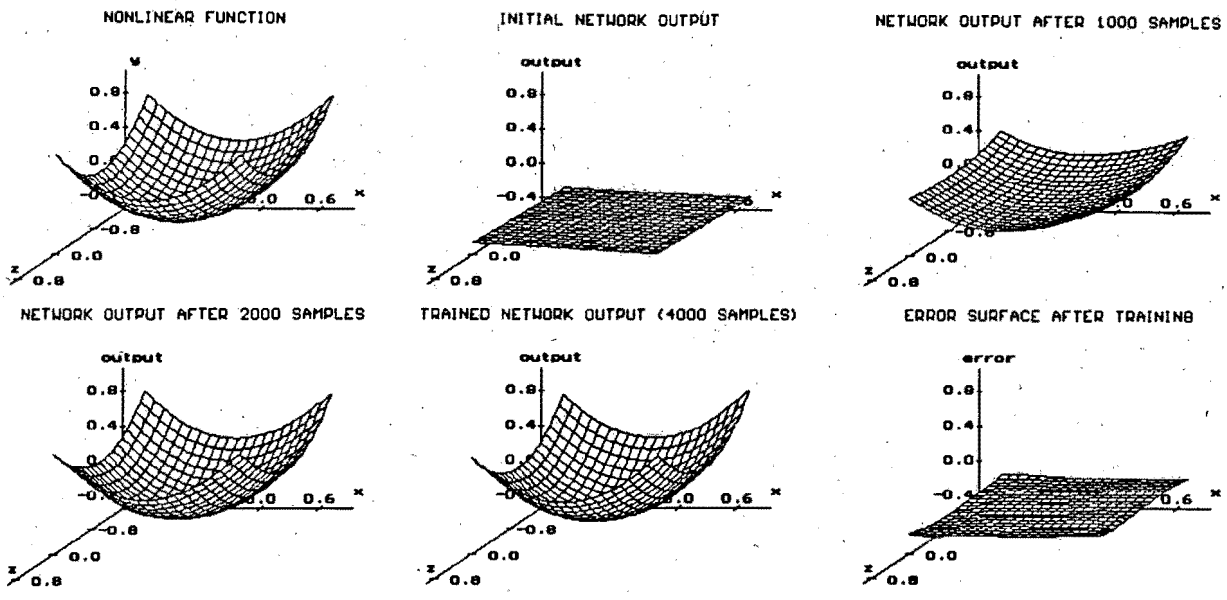


Figure 3.6(a) - Multilayer Perceptron Model of a Paraboloid

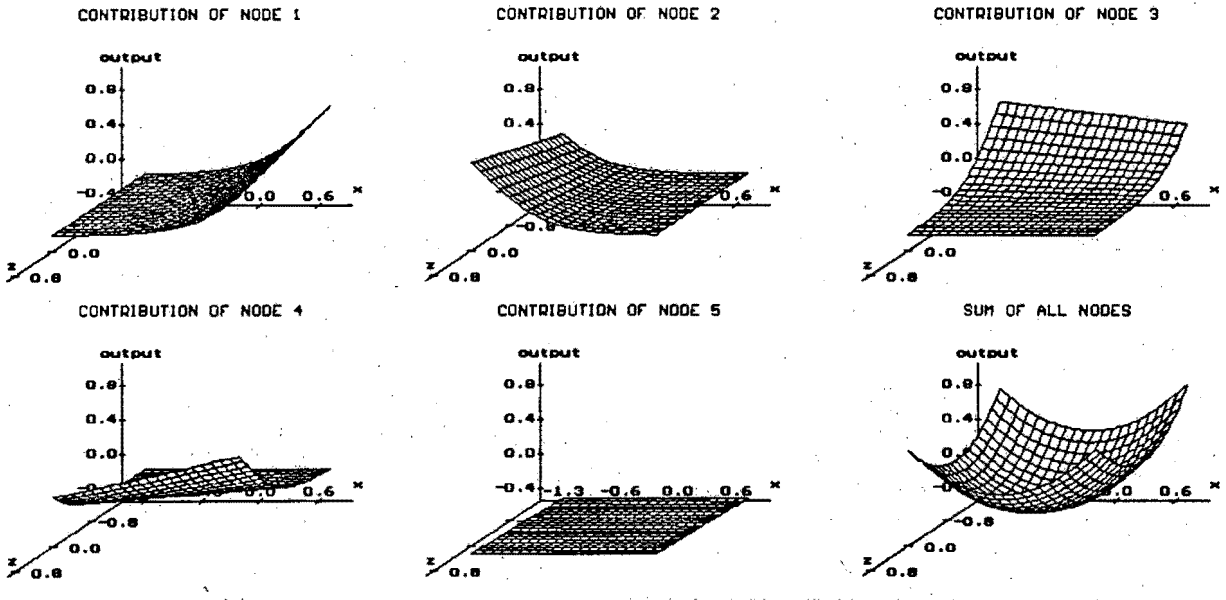


Figure 3.6(b) - Contribution of Individual Nodes in Trained Network

In the above example node 5 has been trained to provide the constant offset required, as described earlier in this chapter.

3.2.4 EVALUATION

This subsection contains an evaluation of the suitability of multilayer perceptrons and backpropagation training for nonlinear modelling. The analysis focuses on the set of criteria established at the end of chapter 2 as well as other network and training algorithm specific problems, some of which were first presented in Durban in 1991 [25].

Since this examination serves to assess the multilayer perceptron as one of the possible networks for dynamic modelling, it highlights possible problem areas, rather than focus on strong points, such as noise filtering, exhibited by most networks.

(a) Test Setup

Although multilayer perceptrons can approximate functions of several variables as shown in the previous subsection, the evaluation criteria are illustrated for single variable functions to allow easier and clearer presentation.

Due to the on-line (or pattern) learning requirement discussed in chapter 2, the networks are not trained using a fixed set of training samples. Instead samples are generated during training by choosing an input and calculating the respective output, much like an on-line system for which both the input and output are sampled at given intervals. Unless otherwise specified, the input samples are chosen at random with uniform distribution within the input range on which training is performed. Where required, the random number generator was started with the same seed value in consecutive runs, to ensure that the same training samples would be used.

Wherever displayed, the root-mean-squared (rms) error was calculated by evaluating the function and network output for several hundred equally spaced input values over the permissible input range.

All programs used in the tests were written in Turbo Pascal ver. 6.0 by Borland and a listing of the neural network algorithms is included in Appendix II.

(b) Extrapolation and Interpolation Capability

As illustrated in chapter 2, the data collected from step perturbations of a dynamic system is often very localized and does not necessarily cover the entire input space sufficiently. This localized nature of the training data also exists during on-line training, where the underlying dynamic system is usually only operated within a well defined region.

One of the desired properties for a neural network model is therefore that it should be able to interpolate between and extrapolate from these localized islands of training data, to cover the entire operating space.

(i) Extrapolation

This refers to the ability of the network to anticipate the behavior of the underlying function into regions which contain no training data.

As shown above, the multilayer perceptron model is a sum of scaled sigmoidal functions, which display highly nonlinear behavior on both ends, before saturation. Since usually only one of the two nonlinear ends is used in the approximation, the sigmoids are scaled such that the other tail is just beyond the training data range. This implies that a multilayer perceptron model does not exhibit any useful extrapolation quality unless the underlying function tapers off in a similar manner.

Figure 3.7 illustrates this poor capacity to extrapolate. The figure presents the results for a 1-3-1 network trained to model an offset parabola on the interval [-1,1].

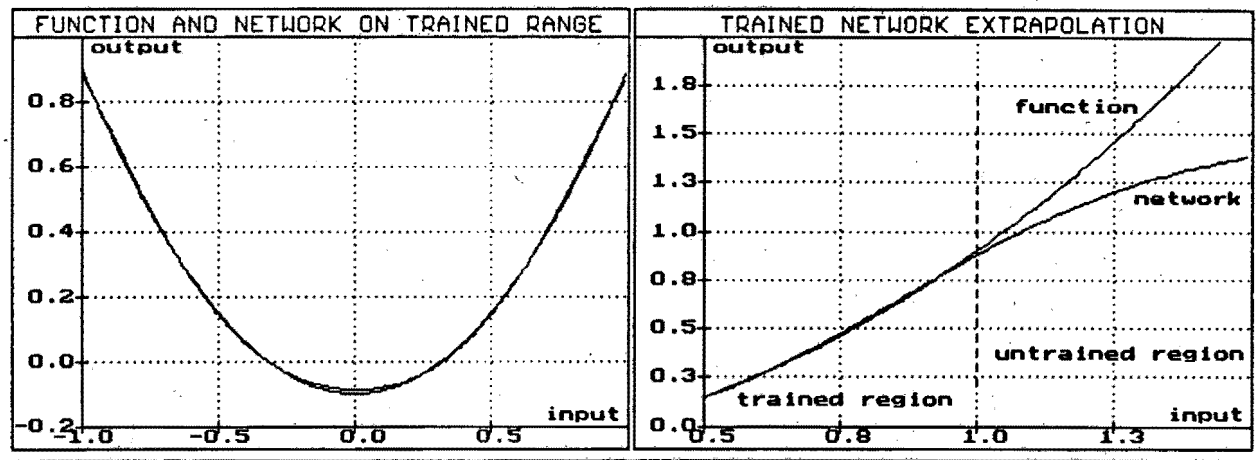


Figure 3.7 - Perceptron Extrapolation Capability

As shown the trained network is an accurate model for the range on which it was trained, but fails to predict the parabola's behavior even for small excursions beyond the training limits.

(ii) Interpolation

Whilst extrapolation refers to the ability to predict into areas with no data, interpolation pertains to the estimation between groups of localized data. As with extrapolation, the ability of multilayer perceptrons to perform such estimation depends on the behavior of the underlying function and on the distribution of the training data.

Figure 3.8(a) below shows a 1-3-1 network, as before trained to model the same offset parabola, but only using data on the intervals [-1.0,-0.8], [-0.1,0.1] and [0.8,1.0].

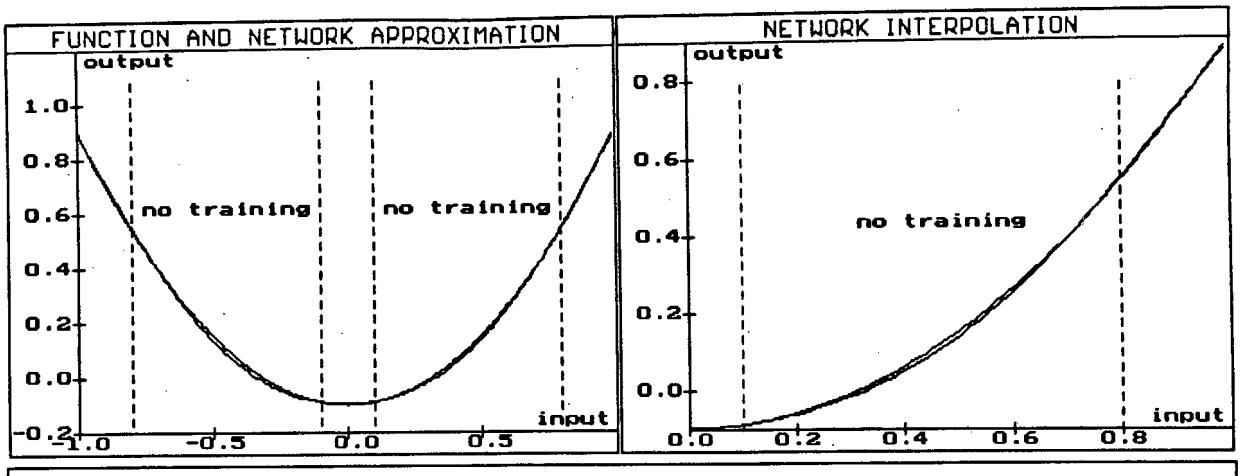


Figure 3.8(a) - Perceptron Interpolation Capability

As illustrated on the left, the network still learns to model the function and also interpolates very well between the islands of training data shown in the detailed view on the right.

Acceptable interpolation performance can however only be achieved if the opening between these islands is small enough for the interpolation to be performed by a single node and where the behavior of the underlying function in these gaps is sufficiently similar to that of the sigmoid, as in the above example. This is illustrated in figure 3.8(b), which shows the same 1-3-1 network trained to model the identical function, but this time only utilizing training data from intervals [-0.1,0.0] and [0.8,1.0].

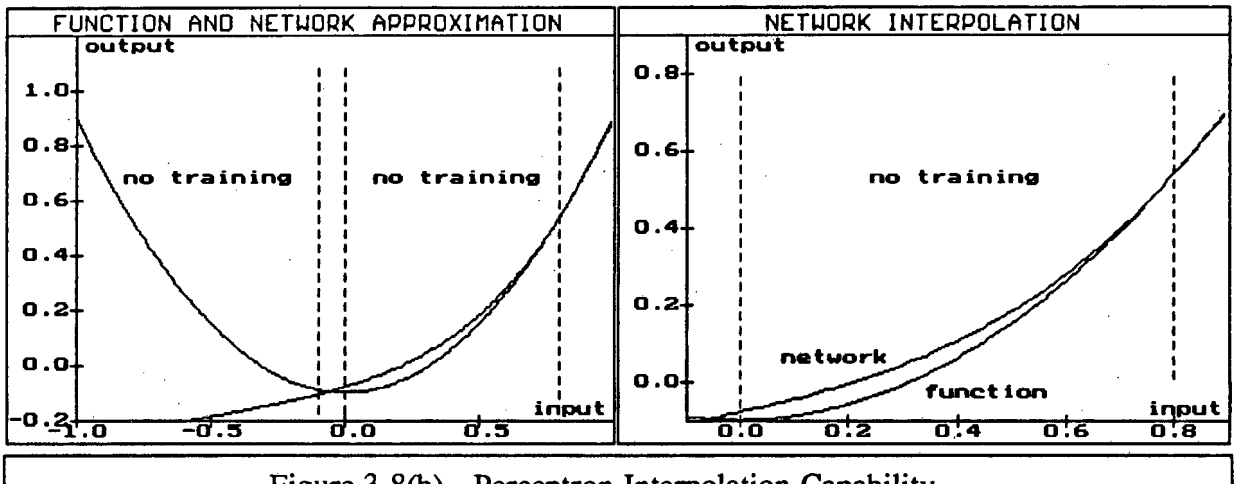


Figure 3.8(b) - Perceptron Interpolation Capability

Whilst previously using two nodes to model each of the parabola halves (and one to produce the constant offset), the new training intervals are such that the network can produce a sufficiently close match inside these, with only one node. Between the training intervals the approximation now deviates significantly from the underlying function, thus resulting in very poor interpolation.

(c) On-line Training : Data Presentation and Interpretation

Although the generalized delta rule is suitable for pattern learning and thus also for on-line training, as illustrated in the previous examples, it still requires the training samples to be presented in a random fashion. In other words consecutive training samples have to be significantly different to ensure sufficient excitation for the network to escape from local minima and to *learn* and *remember*.

(i) Random Data Presentation

This problem is illustrated in figure 3.9, where instead of choosing the input samples in a random fashion, a sinusoidal excitation signal is used to drive the inputs of both the system (parabola) and the network.

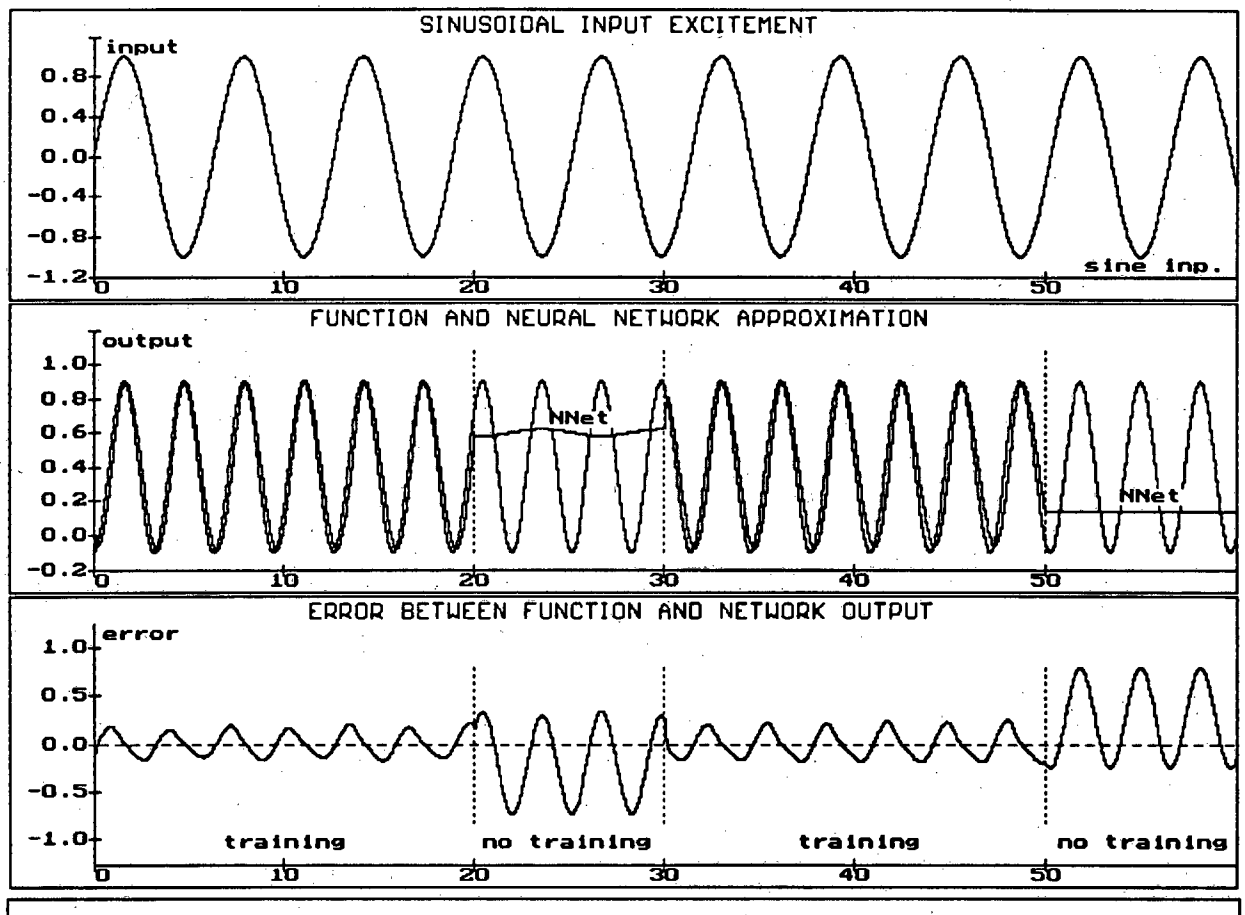


Figure 3.9 - Non-random Excitation

In spite of the fact that the network output follows the desired function output very accurately almost immediately, the network does not learn as can be seen from the poor response in the intervals where training is stopped. As mentioned above this is because consecutive training samples do not differ enough to excite the weights sufficiently in order to escape from their initial local minimum. The above problem was also experienced by Narendra and Parthasarathy [26] in their investigation.

(ii) Global Input Interpretation

Besides the need for randomization of the training data, the multilayer perceptron with its semi-infinite sigmoids is also prone to *forget* or *re-learn* when not presented with data from all areas of the input space at all times. In other words, due to the global nature of the sigmoidal function each hidden node is affected by any input in the entire input space and inputs are thus interpreted globally.

The resulting problem is illustrated in figure 3.10. The left part of the figure shows how a 1-3-1 network, due to its insufficient number of hidden nodes to model the underlying function accurately, learns to generalize as required, if the training data samples are selected from the entire input space. When later presented with training data restricted to a subsection of the input space however, the previously trained network relearns and in the process destroys *knowledge* about the area now omitted as illustrated in the right hand plot.

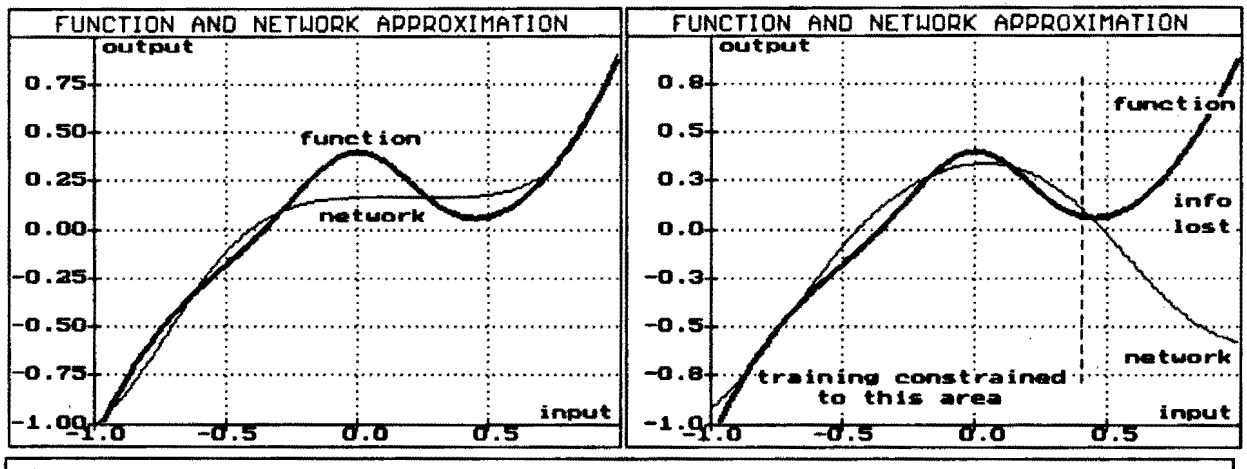


Figure 3.10 - Global Input Interpretation

Although the problem was illustrated using an abstract single variable example, it applies to most dynamic systems, where the data obtained is usually localized and where, due to the effect of disturbances and fast poles, the network has to generalize without inappropriately destroying previously captured information.

Since the inputs to and outputs from most dynamic systems are slow varying signals and often localized for considerable periods of time, the above two phenomena imply that multilayer perceptrons with backpropagation are not suited for direct on-line training and would require some form of data capturing and preprocessing.

Even though the above data presentation problems was illustrated for a network containing too few hidden nodes to accurately model the underlying system, it is also likely to occur for structures with more hidden nodes, if poor initial parameters are selected.

(d) Initial Weight and Training Rate Choices

Both, the set of initial network parameters and the training rate have to be chosen prior to training the network. The magnitude of these parameters can have significant impact on the network's learning performance.

(i) Initial Network Parameters

All adjustable weights and thresholds of a multilayer perceptron are usually initialized to small random values prior to training. Although this initialization procedure is generally satisfactory and does not significantly influence the rate of convergence for approximation networks (larger initial weights generally imply faster initial learning but slower fine tuning and vice versa), an unfortunate choice of initial weights might result in either too much or too little generalization.

This effect of initial weight choices is illustrated in figure 3.11. The graph on the left depicts how a 1-5-1 network with small initial weights learns to generalize. If all the initial weights are scaled by a factor of ten, the same network fails to generalize and learns the exact underlying function as shown on the right.

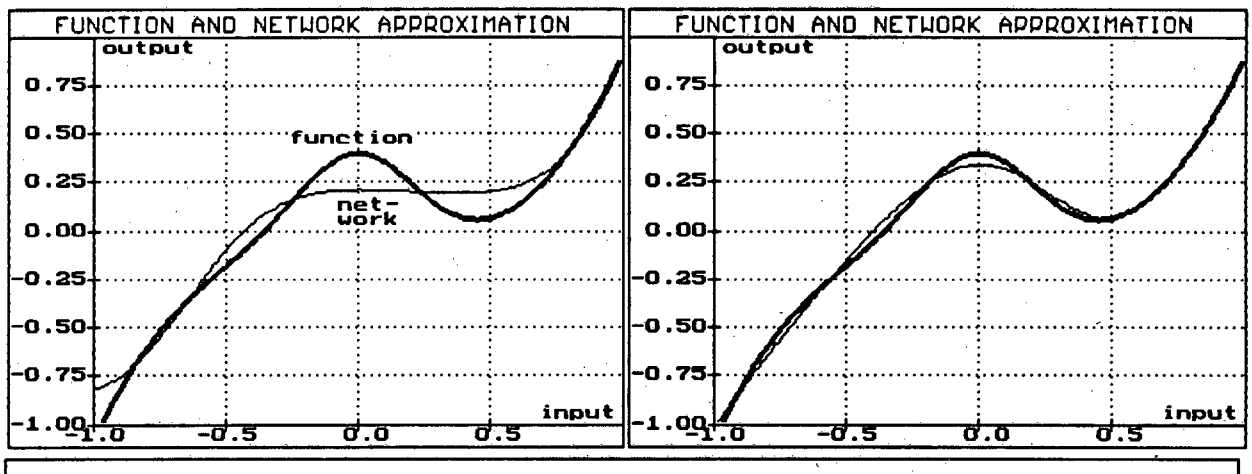


Figure 3.11 - Effect of Initial Weight Choices

Depending on whether the small *humps* present an important part of the model or just an unwanted disturbance, either of the two resulting neural network models could be unsatisfactory. Some a priori knowledge of the underlying function is therefore required in order to choose the correct magnitude for the initial network parameters, such that the model is sufficiently accurate but ignores all undesirable disturbances.

(ii) Training Rate

Figure 3.12 depicts the evolution of the rms error for the same 1-3-1 network with identical initial parameters but varying training rates. For simplicity the momentum term,

which has a filtering effect and generally improves the learning rate, was omitted in the example.

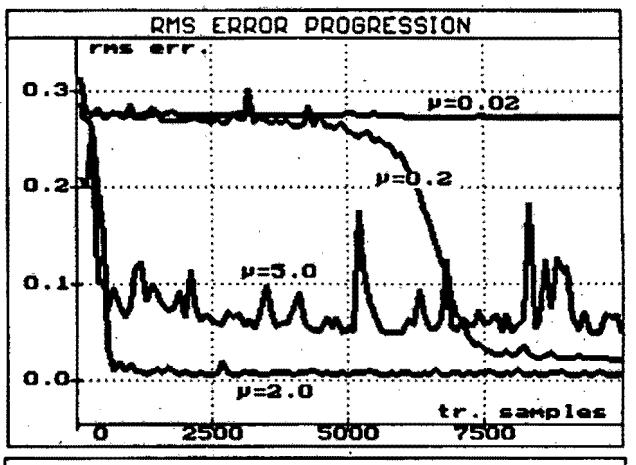


Figure 3.12 - Effect of Training Rate

The choice of 0.02 is too small and never allows the network parameters to escape from their local minimum. For a training rate of 0.2 on the other hand, convergence is initially slow but then accelerates and the rms error settles at an acceptable value. The slow convergence during the early stages of learning, also known as premature saturation [27], is due to the initial re-orientation of the sigmoidal outputs of the hidden nodes. In the problem chosen here, a training rate of 2.0 is close to optimal, producing fast and efficient convergence. Although a larger training rate of 5.0 also produces fast initial convergence, the first oscillations and signs of instability are noticeable, indicating that this training rate is too high for the specific problem.

Whilst the examples in this subsection are not meant to suggest specific choices of parameters for all approximation problems, they illustrate that poor network performance might not only be the result of network size or the quality of the training data, but also due to poor choices of parameters and/or learning rates.

(e) Number of Hidden Nodes

Despite the many algorithms that exist to prove that a single hidden layer containing a sufficient number of nodes is adequate to model any function, no accepted method of determining this number exists, particularly for problems where the underlying function is unknown.

Whilst increasing the number of hidden nodes in a network is generally believed to improve their performance and, due to the distributed representation, create some form of backup for classification networks, this is not the case for modelling networks. Since each of the utilized hidden nodes models part of the underlying function, any additional nodes that cannot significantly improve the accuracy of the model are not applied and do not improve the overall performance of the network.

This is depicted in figure 3.13 below, where the contribution of individual nodes of a 1-5-1 multilayer perceptron network, trained to model a parabola, are depicted. Only two of the four nodes contribute to the actual model and one of the remaining three nodes would suffice to provide the necessary offset.

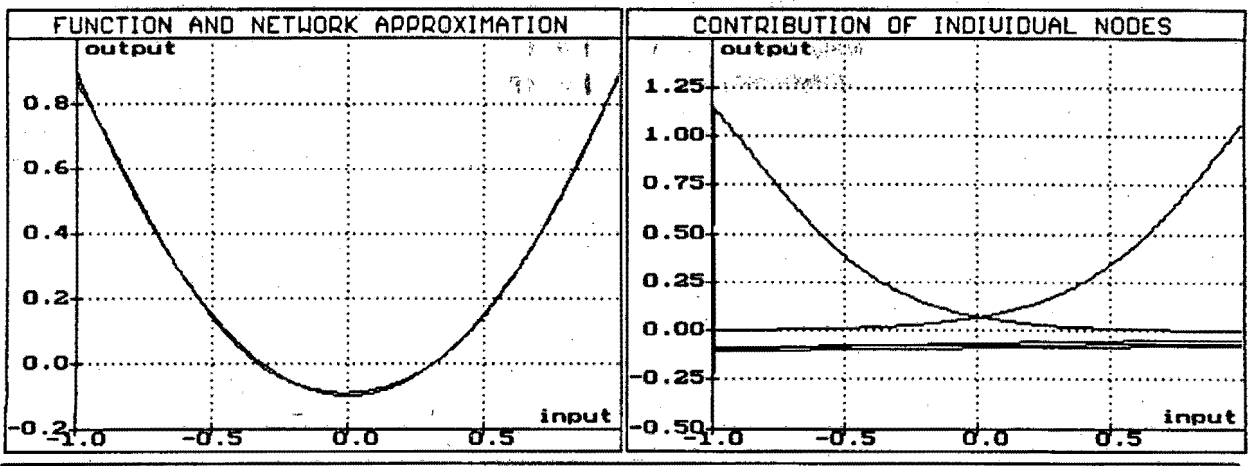


Figure 3.13 - Redundant Nodes

Although this condition does not impact on the performance in the example shown above, it implies that modelling multilayer perceptrons do not exhibit true distributed representation as their classification counterparts. These networks can therefore also not be expected to *degrade gracefully*, if one or more of the utilized hidden nodes are damaged.

In section (d) above, it was shown that the number of nodes utilized in the network model not only depends on the underlying function, but is furthermore influenced by the initial choice of parameters. In other words, even if the network could utilize more of its hidden nodes, it might not do so due to poor initial weight choices.

(f) Network Interpretation

This division deals with the possibilities of storing knowledge in the weights of a multilayer perceptron prior to training, as well as the interpretation and extraction of information stored in the weights of a trained network.

(i) Storing a priori Knowledge

Due to the global input interpretation discussed previously, any attempt to store a priori knowledge in a multilayer perceptron network would require the anticipation of the final position of and scaling for each hidden node, to ensure that the stored information will remain and not be destroyed by subsequent learning.

The problem is compounded by the sensitivity of these networks to initialization errors as discussed in (d). Even if the initial knowledge could be stored, the weights chosen might restrict further learning of the network and hence limit its approximation potential.

(ii) Extracting Data

Due to the high flexibility of multilayer perceptrons, the knowledge stored in a trained network model can generally not be interpreted or extracted directly from the network

parameters. The idea of using Taylor series representations for each of the hidden node suggested by Wray and Green [6] might produce useful results for small networks but becomes impractical for larger structures.

(g) Local Minima

The *local minima* problem refers to the condition, where due to a basin of attraction in the weight space, the search for the optimal set of parameters remains trapped in this region of suboptimal choices. As illustrated in the above examples, this might be due to a number of reasons, such as poor choices for initial parameters or training rates, the number of hidden nodes utilized or even the order of presenting the training data.

This problem of not converging or only converging to a local minimum, has not yet been resolved and remains one of the main arguments against multilayer perceptrons and backpropagation training.

(h) Computational Effort and Ease of Use

(i) Computational Effort

Both the forward and backpropagation algorithm for the multilayer perceptron are of medium complexity and moderately sized networks of this type are therefore suitable for on-line implementation and training.

(ii) Ease of Use

In spite of the suitability from a computational point of view and the fact that a multilayer perceptron is easy to configure and program, the evaluation shows that the performance of the network depends largely on appropriate choices of network parameters, the number of hidden nodes and the presentation of the training data. This type of network and training algorithm is therefore not well suited for a generic approach, where the designer may have little or no knowledge of neural networks.

The assessment of the multilayer perceptron and the backpropagation training algorithm, demonstrated that although these networks are flexible and capable of modelling almost any nonlinear function, they have several shortcomings. The fact that they are not suited for direct on-line training without some processing or re-arranging of the training data and furthermore require a *skilled* trainer to select the parameters and *steer* the training, are the more serious for the type of applications considered in this thesis.

3.3 RADIAL BASIS FUNCTION (RBF) FEEDFORWARD NETWORKS

Although not yet as well-known as the multilayer perceptron, this type of feedforward network is receiving increased interest as a viable alternative, which might overcome some of the limitations highlighted in the previous section.

As in the analysis of multilayer perceptrons, this section starts with an discussion of the network topology, before several training methods are presented. After this the selected network and algorithm are evaluated.

3.3.1 TOPOLOGY

Figure 3.14 depicts the general topology of feedforward RBF networks.

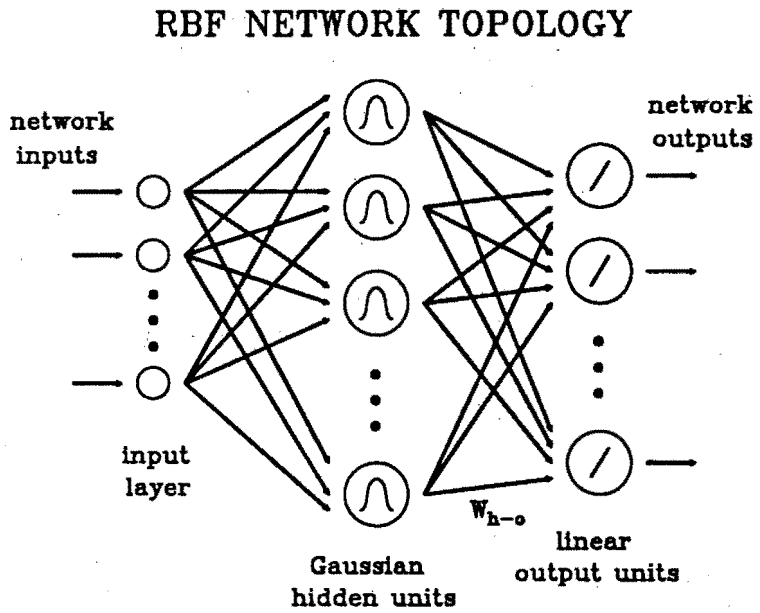


Figure 3.14 - Feedforward RBF Network Topology

As shown, the structure of this type of network does not differ significantly from that of the multilayer perceptron studied in the previous section. Both networks have a single hidden layer and only utilize nonlinear activation functions in their hidden nodes. The main difference between the two topologies is the type of activation function applied in the hidden nodes. Although this difference might not seem significant, it is probably the main reason why RBF networks do not exhibit some of the problems illustrated earlier for multilayer perceptrons.

The radial basis function approach has also been motivated mathematically. Girosi and Poggio [28] demonstrate that these networks have the *best approximation property* (i.e. the

set of RBF networks contains at least one approximation with minimum norm from the function to be approximated) and that multilayer perceptrons do not possess this property.

The following divisions motivate and discuss the *radial basis type* activation function, the absence of weights between input and hidden layer and the use of a single hidden layer for these networks.

(a) Radial Basis Activation Function

Most RBF networks employ the Gaussian function depicted in figure 3.15 as the activation function in the hidden nodes. The local nature is the most significant difference between this RBF and the sigmoidal type of activation function. In other words, whilst the logistic function is semi-infinite, the RBF functions only produces an output on a restricted, local sphere of influence. This localized response is due to the fact that the activation of a RBF function depends on the distance of the input vector from its center, rather than the absolute values of the inputs. The output of such a function is therefore maximal, if the input vector coincides with its center and gradually decreases to zero for inputs further away, thus resulting in a spherical space of reaction around the center of the node.

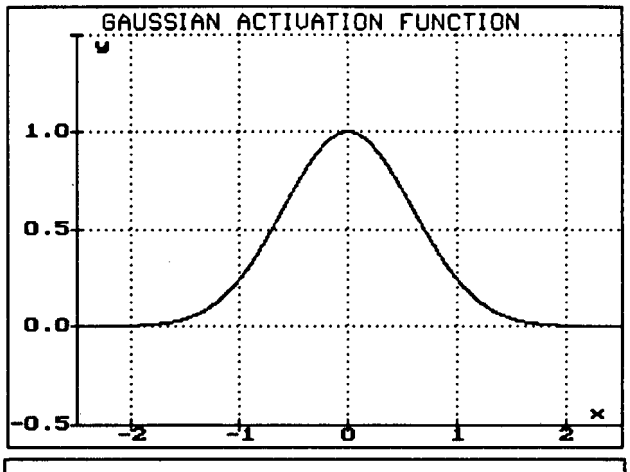


Figure 3.15 - Gaussian Activation Function

The advantage of hidden nodes with such local response is not only beneficial for modelling applications (as shown later in this chapter) but also for classification problems. In their attempt to utilize neural networks for process fault classification, Leonard and Cramer [29] note that RBF networks overcome the problem of non-intuitive, non-robust decision surfaces produced by multilayer perceptrons and backpropagation.

(b) Weights from Input to Hidden Layer

The structure in figure 3.14 shows no weight matrix for the connections between input and hidden layer. Again this is due to the radial basis type activation function, where the equivalent of input scaling is performed by adjusting the bandwidth of each node for each input dimension. In other words, internal parameters (equivalent to the standard deviation) can be modified to widen or narrow the region of influence for each of the inputs to each of the nodes.

Although possible, most RBF applications do not use different bandwidth parameters for different input dimensions and do not include parameters cross-linking these dimensions, since the resulting ellipsoid type of activation region, not aligned with the major directions of the input dimensions, would require extensive optimization for each node. Single bandwidth parameters without cross-linking terms are thus used in most RBF applications.

(c) Number of Hidden Layers

Although some multilayer perceptron applications utilize two hidden layers, this is not the case for RBF networks. This is partly because, as for multilayer perceptrons, algorithms exist to prove that a single hidden layer suffices. However, the main reason for a single hidden layer in RBF networks is their structure. The underlying idea of utilizing radial basis type functions is that the network will partition the input space, so that each of the hidden nodes has a well defined, local sphere of influence, on which it models the underlying function. The addition of further layers in such a structure would therefore not improve the modelling capability of the network and the additional nodes should rather be used in the first layer to perform a finer division of the input space.

3.3.2 TRAINING ALGORITHMS

As with multilayer perceptrons, there exist a large variety of training algorithms to determine the tunable parameters for RBF networks. These parameters, to be determined prior to or during training include:

- the number of hidden nodes
- the center for each of the hidden nodes (mean for Gaussians)
- the sphere of influence for each hidden node (standard deviation)
- the weight between each hidden and output node

In the overview of training methods presented below, the algorithms appear in order of network flexibility. Whilst the first algorithm determines all the above network parameters, the third only adjusts the connection weights between hidden and output layers and requires that the remaining variables are determined by other means prior to training.

(a) Hierarchically Self-Organizing Learning

As mentioned above this RBF training algorithm suggested by Lee and Kil [30] determines all the parameters of an RBF network.

An overview of the steps involved in generating and training the network using this algorithm are presented below:

The network starts with no hidden nodes and an output of zero. The following algorithm is then applied to generate all the hidden nodes and the necessary connection weights.

1. Start with no hidden nodes and set the effective radius r_i large enough to cover the entire input space.

2. Invoke new learning cycle (for all patterns) as described below :
   (i) get next training pattern
   (ii) if the output of the network differs by more than the error margin from the desired output then
      - if the input vector falls within the effective radius of an existing hidden node then update the mean, standard deviation and weight associated with that hidden node to accommodate the training pattern
      - else create a new hidden node with its center at the input vector, its weight equal to the desired output, its standard deviation to small initial values and its effective radius to the current value
3. Reduce the effective radius of each hidden unit by a pre-determined value or using a measure of the error gradient to judge when 'saturation of learning' occurs for the current number of hidden nodes.
4. Go back to step 2 until the difference between network - and desired output is within the error margin for all training patterns.

The algorithm thus grows the network, starting with one node, covering the entire input space and then slowly shrinking its sphere of influence whilst adding more nodes as required.

(b) Clustering Type Algorithms

For this type of algorithm the required number of hidden units has to be estimated prior to training. During optimization the remaining parameters are then usually determined in three phases. First the center for each of the units is located, then the bandwidth of each is adjusted and finally the connection weights with each of the output units are established.

Due to the many variations of this type of algorithm published in literature [31][32][33], the discussion below includes more than one method of determining the parameters, where applicable.

Phase 1 - Locating the Centers

If all training patterns are available, this is usually done using standard k-means clustering:

(i) The centers are set to randomly chosen training pattern inputs.
(ii) Each training sample is then assigned to the unit with its center nearest to its input.
(iii) When all units have been assigned, the center of each cluster is moved to the average position of all points in the cluster.
(iv) Go to step (ii) until convergence (the centers don't move significantly).

For more on-line orientated algorithms:

(i) Choose centers randomly distributed in the input space.
(ii) Present the next pattern with input $\underline{x}_p$ to the network.
(iii) Find the unit with its center $\underline{x}_h$ closest to the input of this pattern and move it fractionally towards the input $\underline{x}_h(t+1) = \underline{x}_h(t) + \eta(\underline{x}_p - \underline{x}_h(t))$ where η is the learning rate which decreases with time.
(iv) Go to step (ii) until there is no more significant improvement.

To ensure a more global optimization, the updating might be applied to all centers in the topological neighborhood, which decreases with time much like the effective radius in the above algorithm.

Phase 2 - Determining the Bandwidth (interpolation)

- For both, batch and pattern learning a P-nearest neighbor heuristic is usually employed to find the bandwidth for each hidden node i.e.

$$\sigma_h = \sqrt{\frac{1}{P}\sum_j^P \|\underline{x}_h - \underline{x}_j\|^2}$$

where $\underline{x}_j$ are the P nearest neighbors of the center. $\underline{x}_h$

- Other iterative algorithms to determine these parameters exist [30].

Phase 3 - Training the Hidden-Output Weights

- Since all other parameters have been fixed at this stage, finding the weights between the hidden and output units (which only sum all their inputs to produce a network output), presents a linear optimization problem, where the least mean square solution is required.

- This solution can be obtained in one pass for batch type problems, using a pseudo inverse matrix of the hidden node activation vectors for all training patterns **A** and multiplying this with the matrix of target outputs **T**.

$$w = TA^T(AA^T)^{-1}$$

- For pattern or on-line learning, the solution has to be obtained by adjusting the weights proportional to the contribution of the node and the error between desired and actual output

  i.e. $w_{ij}(t+1) = w_{ij}(t) + \alpha K_i err$ where - α is the learning rate
  - K_i is the output of the hidden unit
  - err is the difference between desired and actual output.

  Training continues until the error is acceptably small for all patterns.

Although the algorithms discussed are attractive as far as network flexibility is concerned, they are not suited for on-line training. This is because these algorithms are of the batch learning type and require the entire training set prior to training. Even the pattern learning type clustering algorithm adjusts the centers, bandwidths and weights in stages and therefore requires that during each of these, the training patterns used are representative for the entire operating space.

A less flexible network structure, more suited for true on-line training, was therefore used in this study.

(c) Regular Gaussian RBF Network Learning

This type of simplified network structure and learning algorithm, suggested by Hunt and Sbarbaro [34], is a simplification of the above method. Instead of utilizing actual input patterns during the first two phases, the available number of nodes are distributed uniformly over the entire input space and the bandwidth of all nodes is predetermined to ensure smooth interpolation. The steps below describe and motivate the procedures used to generate and train such a regular Gaussian network.

The discussion below assumes the following network notations and equations:

- The output of hidden node j for input pattern p is given by

  $$K_j = e^{-d(\underline{x},\underline{x}p,\underline{\Delta})}$$

  where $d(\underline{x},\underline{x}_p,\underline{\Delta}) = (\underline{x}-\underline{x}_p)^T\underline{\Delta}(\underline{x}-\underline{x}_p)$

  with $\underline{x}$ - the center of the unit
  $\underline{x}_p$ - the input vector
  $\underline{\Delta}$ - a diagonal matrix of the bandwidth value Δ.

- Each of the o outputs for a network with N hidden units is then determined as follows :

$$y_o = \sum_h^N w_{ho} K_h \qquad \text{where } w_{jo} \text{ is the weight connecting hidden node j to this output.}$$

Using this notation, the network is initialized and trained as follows :

1. Distribute the available nodes uniformly in an n-dimensional unit cube (all inputs to the RBF are normalized)
   - where n presents the number of inputs to the network (each input dimension receives the same number of units to allow the use of a single bandwidth parameter per unit).

2. Calculate the bandwidth.
   - Due to the uniform distribution the number of nearest neighbors depends on the input dimension (2 for 1 input, 4 for 2 inputs etc.).
   - It was however found that even for multi-dimensional problems the following fixed bandwidth produced the best interpolation results,

$$\Delta = \frac{2}{(\text{distance betw. nearest nodes})^2}$$

Figure 3.16 shows the result of this initialization procedure for a 1-7-1 regular Gaussian network where the hidden to output weight for each of the nodes was set to 0.5. The figure highlights the trade-off between smooth interpolation and inter-node interference in choosing the bandwidth for hidden nodes.
The network output shown, furthermore illustrates the poor interpolation near the edge, which explains why the input range of [-1..1] is re-mapped to [-0.75..0.75] in the feedforward algorithm.

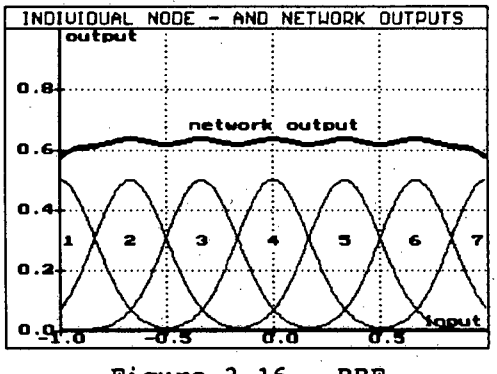


Figure 3.16 - RBF Initialization

3. Initialize all network weights w_{ho} to small random values.

4. During training update the weights connecting each of the j hidden nodes to each of the k output nodes using the least squares regression algorithm described above

$$w_{jk}(t+1) = w_{jk}(t)\ \alpha K_j(y_{k\text{-desired}}(t+1) - y_{k\text{-net}}(t+1))$$

Although the regular Gaussian network sacrifices a significant amount of the standard RBF network flexibility, the resulting structure ensures a homogeneous distribution of nodes and hence modelling capability over the entire input space and can be trained on-

line. Both are important requirements for neural network models and warrant an evaluation of this type of network as an alternative to the multilayer perceptron.

3.3.3 NONLINEAR MODELLING CAPABILITY

As described above, the flexibility of individual hidden nodes in the regular Gaussian network structure is limited to the strength of their connection weight(s) with the output(s). The only adjustment for individual nodes is thus their height as illustrated for single and two-dimensional input nodes in figure 3.17.

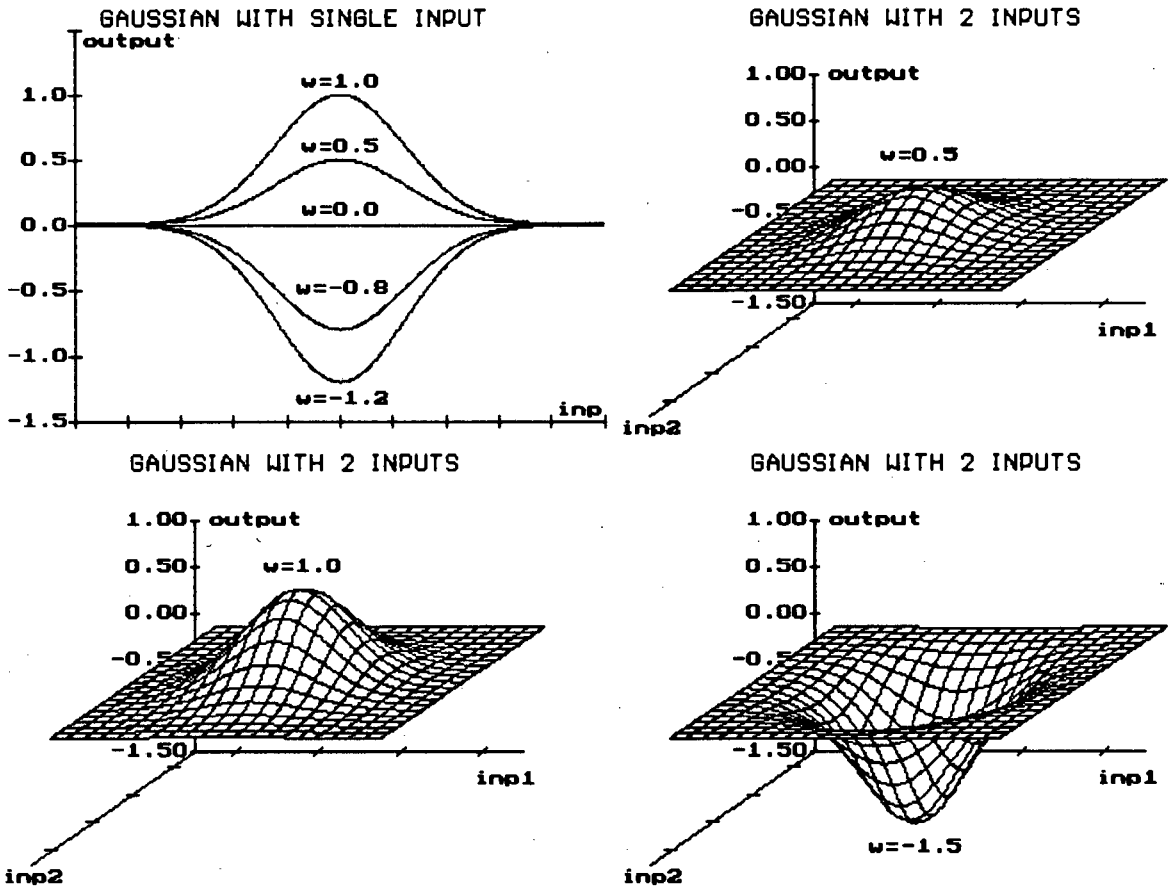


Figure 3.17 - Modelling Capability of Hidden Nodes in Regular Gaussian Networks

As with the multilayer perceptron, the summation of a number of these scalable nodes in a network can be trained to model a nonlinear function. This is depicted in figures 3.18(a) and (b), which show the generation of a 2-9-1 regular Gaussian network model for a paraboloid and the contribution of each of the nine hidden nodes to the trained network output.

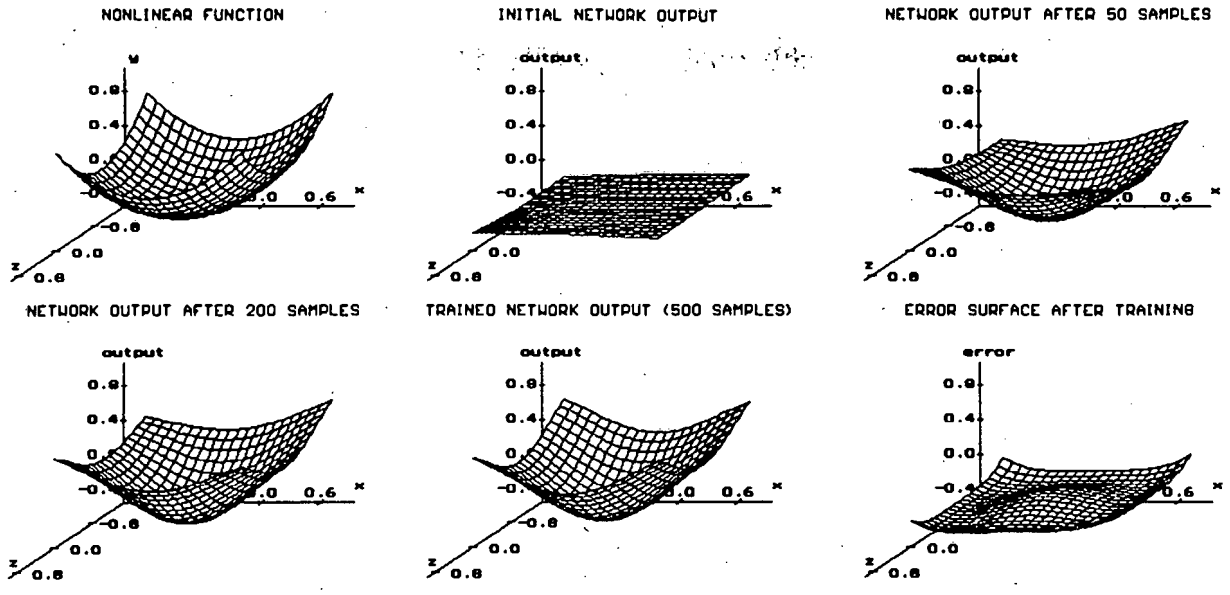


Figure 3.18(a) - Regular Gaussian Network Model of a Paraboloid

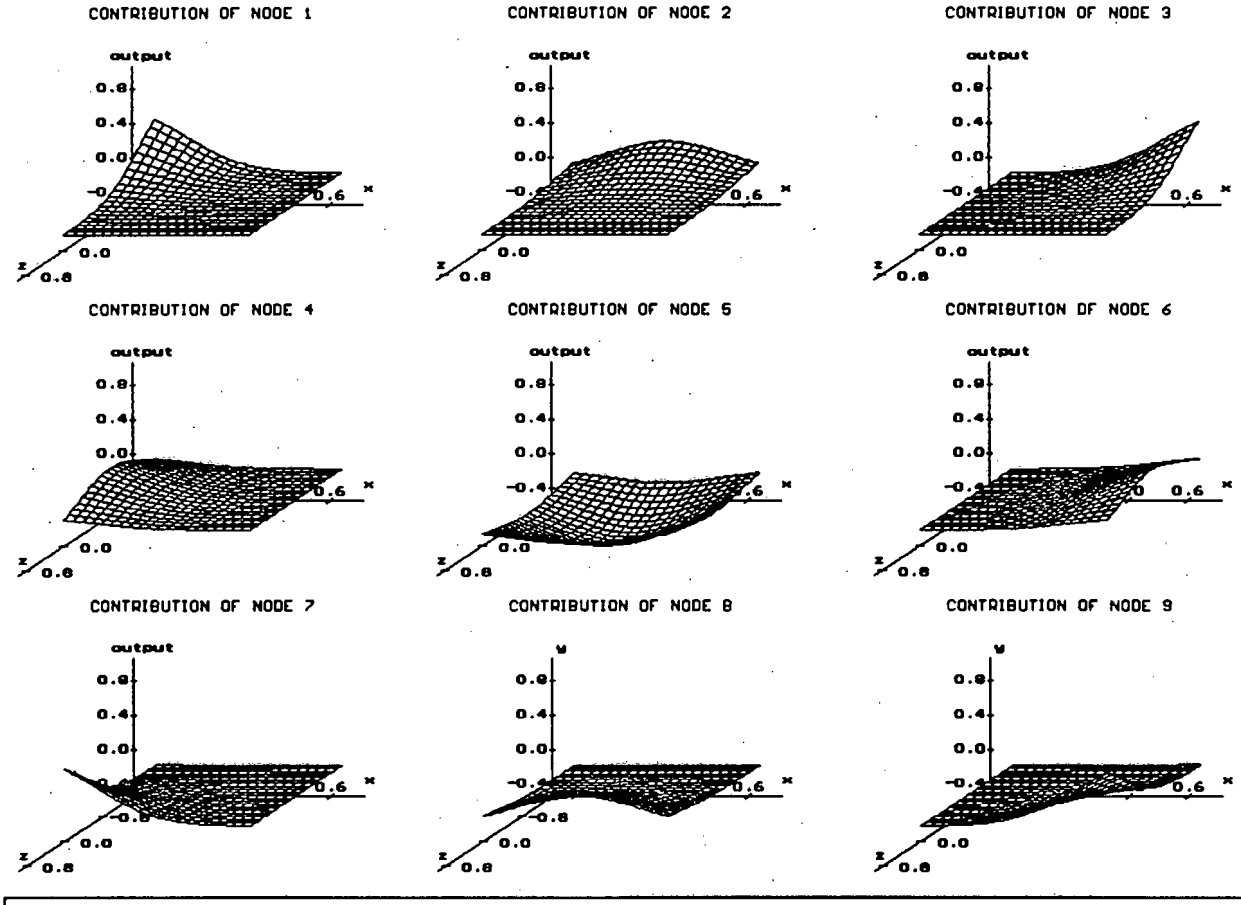


Figure 3.18(b) - Contribution of Hidden Nodes

It is worth noting that although the regular Gaussian network requires nearly twice the number of hidden nodes when compared with the multilayer perceptron discussed previously, it converges using far less training samples.

From the error surface for the trained network it is clear that due to the rigid structure of the network (fixed centers and bandwidth), some of the nodes have to compromise, resulting in a less than optimal model. This is particularly evident for the nodes on each of the corners as well as the node at the center of the training space.

These and other advantages and disadvantages of the regular Gaussian network are discussed in the following evaluation.

3.3.4 EVALUATION

As for the multilayer perceptron, this section investigates the suitability of the network and learning method for nonlinear modelling. In order to compare the applicability of regular Gaussian networks with that of multilayer perceptrons, particular attention is paid to the problems highlighted for the latter in the previous section.

(a) Test Setup

The test arrangement used in this assessment is identical to the one utilized earlier; problems are illustrated for single input networks; training patterns are generated at random during run-time unless otherwise specified; the root-mean-squared (rms) error is calculated by evaluating function and network output for several hundred equally spaced values, on the permissible input range.

Again all the required software was developed in Turbo Pascal ver. 6.0 and all the relevant algorithms are included in Appendix II.

(b) Extrapolation and Interpolation Capability

As discussed earlier, the clustered nature of data collected from dynamic systems due to their localized method of operation, demands some form of extrapolation from and interpolation between these islands of training data.

(i) Extrapolation

Since all inputs to regular Gaussian networks have to be normalized to within the permissible [-1..1] interval, a 1-5-1 network was trained to model an offset parabola, using a restricted training interval [-0.5..0.5] and was then examined between 0.5 and 1.0 in order to judge the extrapolation capability of this type of network. Figure 3.19 depicts the results of this test.

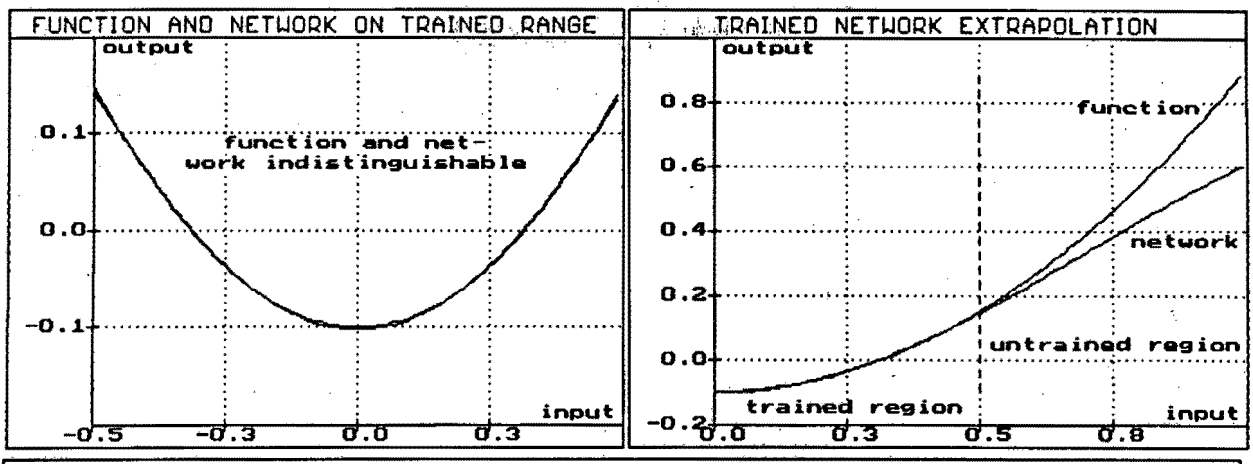


Figure 3.19 - Regular Gaussian Extrapolation Capability

As shown, the network has learned to model the underlying function accurately on the trained interval (left), but fails to predict function values for inputs beyond the limits. This poor extrapolation capability is expected for RBF type networks where, due to the local nature of each hidden unit, nodes beyond the interval receive no or little training and are therefore not modified.

Although this behavior is disadvantageous from an extrapolation point of view, it is the very same quality that prevents a regular Gaussian network from *forgetting* like the perceptron, if trained in a local fashion. The advantage of this *local learning* is illustrated later in this assessment.

(ii) Interpolation

As with the multilayer perceptron, the ability of regular Gaussian networks to interpolate between localized regions of training data depends on the number of units and the behavior of the underlying function between the data islands.

If the number of nodes is sufficiently small as in the 1-3-1 network trained on intervals [-1.0..-0.8],[-0.1..0.1] and [0.8..1.0] shown in figure 3.20(a) (left) below, the resulting network model is acceptable and predicts the function values adequately in the untrained interval (right). This is due to the fact that despite the small intervals used, each of the hidden nodes is receiving sufficient training input.

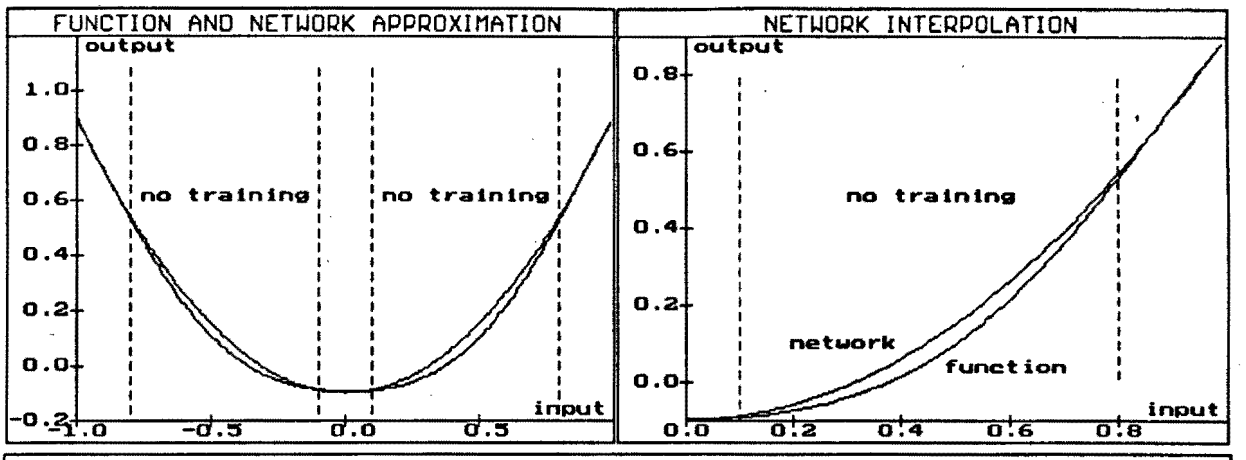


Figure 3.20(a) - Regular Gaussian Interpolation Capability

The networks interpolation capability is reduced considerably if the number of hidden nodes between the two training intervals is increased, such that the active input region for some of these nodes lies entirely outside the training intervals (i.e. the node receives no training). This is illustrated in figure 3.20(b) which shows the results of training a 1-9-1 network, using the same intervals as above.

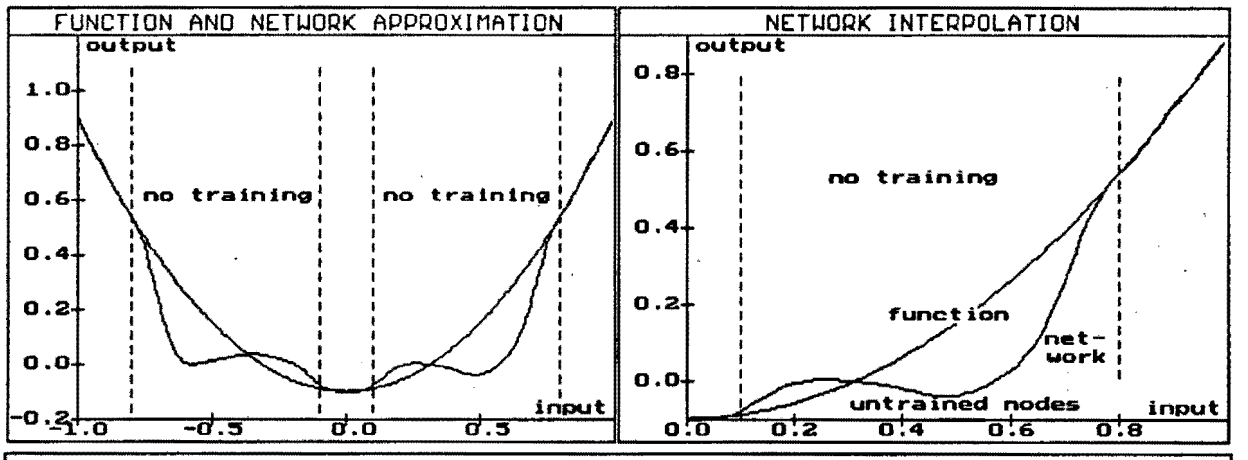


Figure 3.20(b) - Regular Gaussian Interpolation Capability

Although the network model is very accurate on the trained intervals, the interpolation between these is extremely poor. Due to interpolation, units with their centers close to the trained intervals receive one-sided training and are therefore distorted in order to improve the model in the trained interval (e.g. node at 0.2), whilst nodes further inside the interval receive no training at all.

These extrapolation and interpolation results for regular Gaussian networks highlight the need for sufficiently dense and distributed training data and/or some form of network initialization, to ensure adequate performance in regions with little or no training.

(c) On-Line Training : Data Presentation and Interpretation

In the evaluation of multilayer perceptrons, the need for random presentation of and sufficient variation in the training data was highlighted as one of the major drawbacks of the backpropagation training algorithm. Due to the local response of the activation function used in their hidden units, regular Gaussian networks can overcome these problems.

(i) Training Data Presentation

Although random presentation of training patterns allows higher training rates and hence accelerated learning in regular Gaussian networks, it is possible to train such a network with a slow varying input signal. This is illustrated in figure 3.21, where a sinusoidal input signal is used to drive both, the parabola to be learned as well as the input to the 1-10-1 regular Gaussian network model.

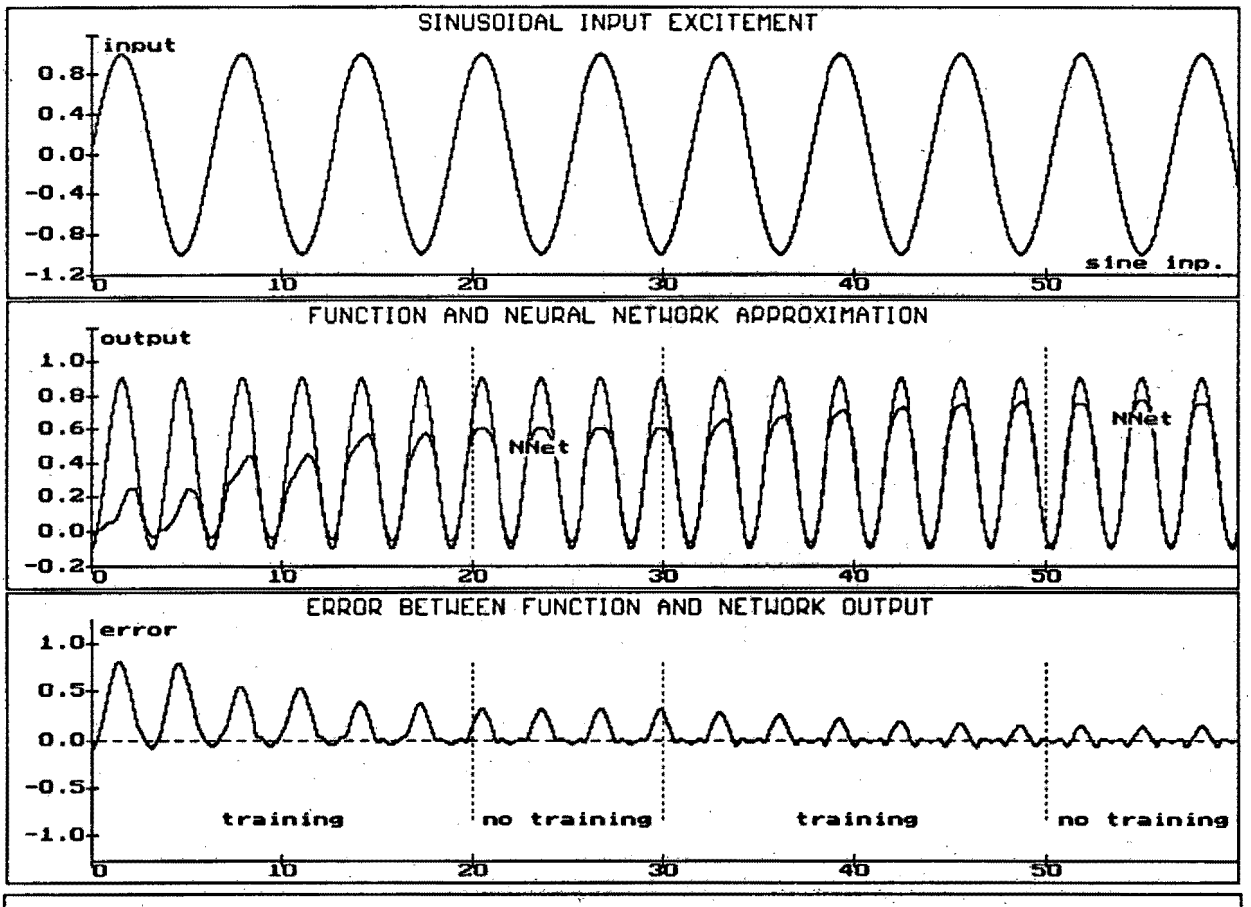


Figure 3.21 - Sinusoidal Excitation for Regular Gaussian Network

The *larger than required* number of hidden nodes was chosen to provide a very fine partitioning of the input space and a low training rate was used to allow slow convergence of the weights and prevent overshoot. As shown, the network learns gradually and accurately predicts the lower half of the model when learning is stopped for the first time [20..30]. After further training, the network performance continues to

improve and when training is stopped again [50..60], the network is an accurate model for all but a very small region at one end of the input space.

(ii) Input Interpretation

The global input interpretation and resulting loss of information when trained locally, was highlighted as one of the reasons why multilayer perceptrons are not suited for true on-line training. Again the local response of the nodes in a regular Gaussian network help to overcome this limitation as illustrated in figure 3.22 below.

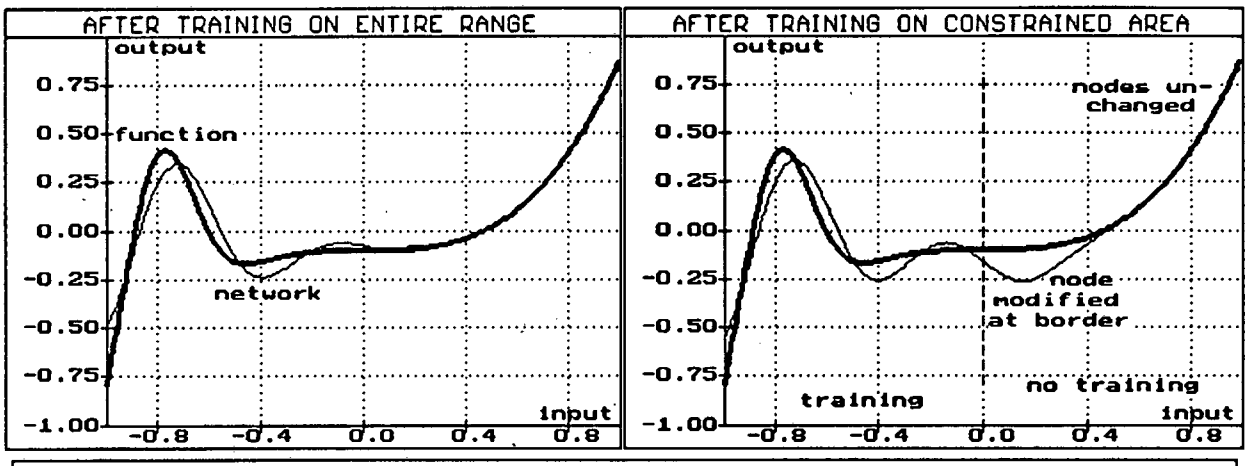


Figure 3.22 - Local Training for Regular Gaussian Network

The graph on the left shows the output of a 1-10-1 regular Gaussian network model together with the underlying function after training on the entire input range. The network was then trained further on a reduced interval [-1..0] and as shown on the right, even after considerable training, only the output of one node directly adjacent to this area is affected. Nodes sufficiently far away [0.4..1.0] are not influenced and the network thus *remembers* the information previously learned for that input region, provided the network contains a sufficient number of hidden nodes.

(d) Initial Weight and Training Rate Choices

As with all neural networks both, the initial network parameters and the training rate have to be chosen prior to training.

(i) Initial Network Parameters

The only adjustable network parameters for a regular Gaussian network are the weights connecting each of the hidden nodes to the outputs. For most applications these are initialized to small random numbers. Due to the well defined, localized area of response pre-assigned to each of the hidden nodes, the initial value of these weights does not determine the performance of nodes in the trained network, as in the case of the

multilayer perceptron. In other words, the initial choice of parameters does not influence the modelling capability of the network.

(ii) Training Rate

Figure 3.23 shows the convergence of a 1-5-1 multilayer perceptron with identical initial conditions for different training rates.

As shown the rms error declines steadily to the final minimum for a training rate of 0.02. The network does however learn much faster for a value of 0.2, for which the final minimum is reached after one tenth of the number of training samples. As illustrated, a further increase of the training rate to 2.0 leads to oscillations i.e. the error does not converge and the system borders on instability.

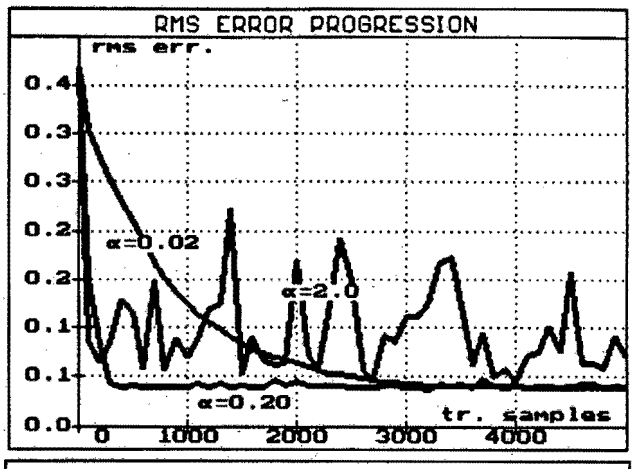


Figure 3.23 - Effect of Training Rate

As before, this example was not included to recommend an optimal training rate, since this depends on the underlying function as well as the format in which the training data is presented (for slow varying input signals only small parameters lead to convergence), but rather to show that the network converges for all, except very large training rates.

Both, the initial choice of weights as well as the training rate are thus not as critical for regular Gaussian networks as they are for multilayer perceptrons.

(e) Number of Hidden Nodes

In the evaluation of multilayer perceptrons it was shown that such a structure only utilizes the number of hidden nodes *required* to approximate the underlying function and that the addition of further nodes does not improve the model.

This is not the case for regular Gaussian networks, where each node is assigned a certain input partition prior to training and hence all hidden nodes are employed in the model. Figures 3.24(a) and (b) below show the resulting approximation and node contribution for regular Gaussian networks containing five and ten hidden nodes respectively.

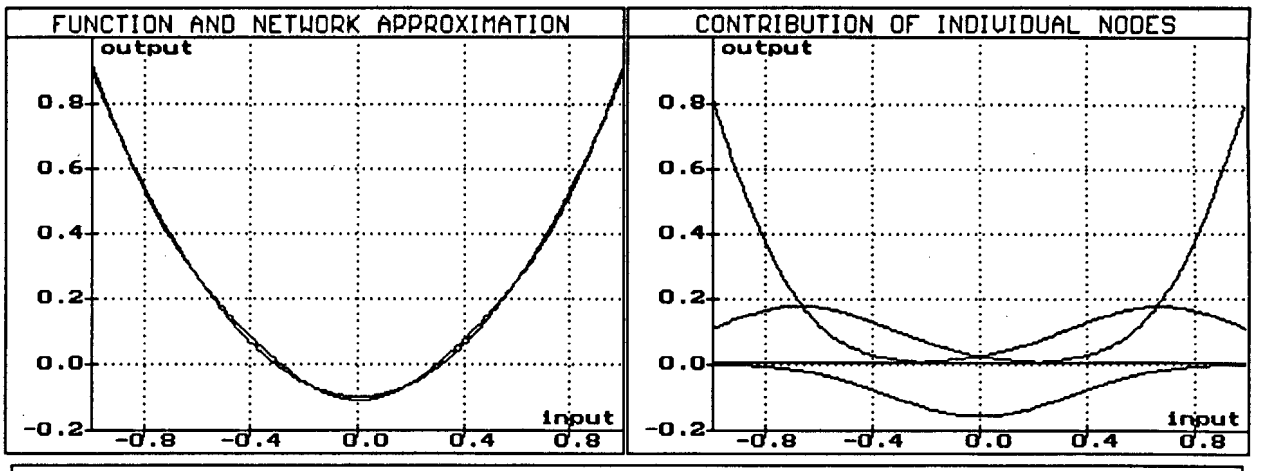


Figure 3.24(a) - Five Node Approximation

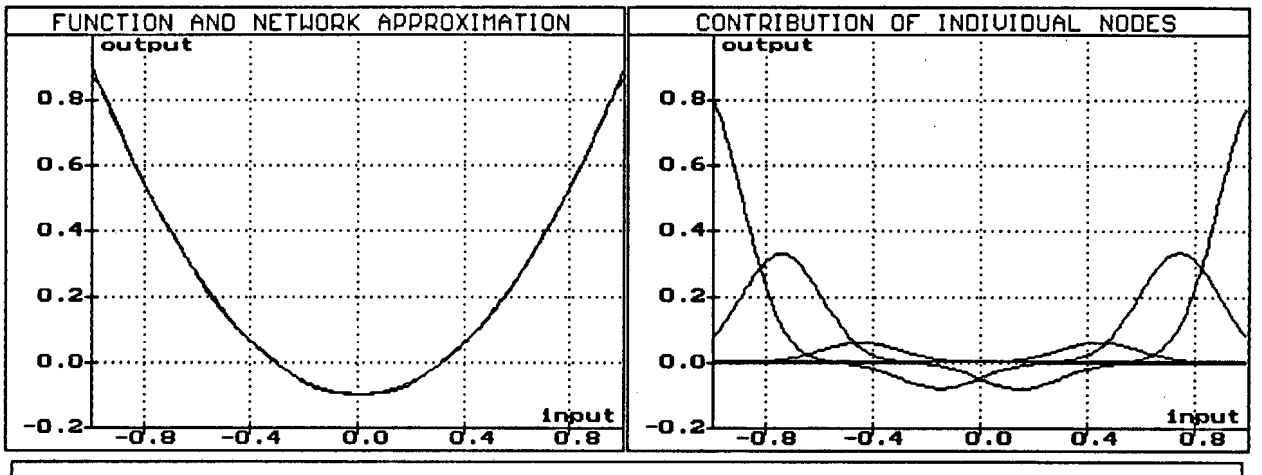


Figure 3.24(b) - Ten Node Approximation

As shown, all nodes are utilized in both networks and the only difference is a finer resolution of the input space for the 'ten hidden node' network.

(f) Network Inflexibility and Size

Although many of the advantages of regular Gaussian networks over multilayer perceptrons, are due to predetermined fixed positions and bandwidths of the hidden nodes, this rigid structure also has drawbacks. One of the main disadvantages associated with this inflexible topology is that the fixed position and width of the Gaussians might not be optimal for the underlying function, hence forcing the network to compromise even though a better approximation could be achieved with the same number of hidden nodes.

This is illustrated in figure 3.25 below, where the output of a trained 1-8-1 network is shown together with the underlying function.

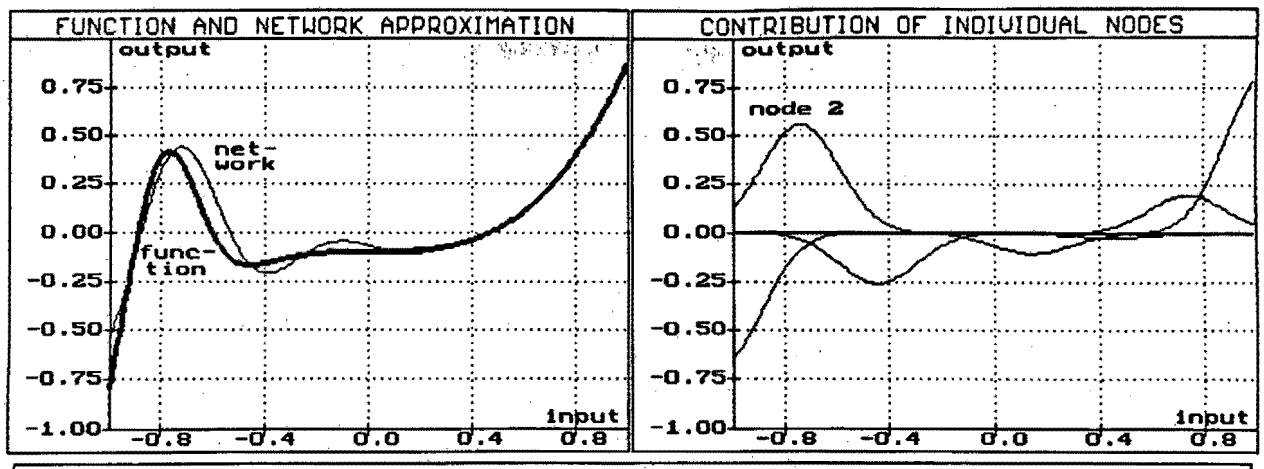


Figure 3.25 - Regular Gaussian Network Inflexibility

The poor performance near -0.8 could easily be resolved if the network had the flexibility to shift the center and modify the bandwidth of the node labeled *node 2* on the right hand plot. Due to the fixed arrangement of centers and bandwidths for these networks, such a solution is not feasible and more hidden nodes would have to be employed, if a more accurate model is required.

Whilst this approach, of increasing the number of nodes until satisfactory accuracy is achieved, is acceptable for single input systems, it becomes a problem for networks with more than one dimension. Since for regular Gaussian networks the number of nodes required to achieve a certain resolution per input dimension grows exponentially as a function of the number of inputs, higher dimension structures have to be limited to few nodes per input dimension in order to keep the number of nodes within manageable limits. This restriction might in turn compromise the modelling performance of the network as illustrated in the above example.

(g) Network Interpretation

Unlike the multilayer perceptron for which both, the storage of a priori knowledge and extraction of information from the network were not feasible, the regular Gaussian network is relatively easy to initialize and interpret.

Again this is due to the well defined partitioning of the input space. The weight between each hidden node and the output(s) is a direct reflection of the modelled function output at the center of this node and can be used to both, initialize an untrained network and obtain information about the modelled function in a certain region after training (provided the interpolation of neighboring nodes is taken into consideration).

A further benefit of the well defined structured topology of regular Gaussian networks is their similarity with fuzzy inference systems. This link between self-adjusting, learning and rule-based systems is currently receiving considerable interest [35] and might in future provide other, more appropriate methods for storing and extracting knowledge.

(h) Computational Effort and Ease of Use

(i) Computational Effort

Although the forward propagation algorithm for regular Gaussian networks is not complex, it is computationally intensive since the distance from the center of each hidden node, rather than the actual input vector is used. The problem is particularly evident for multi-input systems. The training algorithm on the other hand, is very fast and robust and compensates somewhat for the slow forward propagation, rendering the combined system suitable for on-line implementation on real-time systems.

(ii) Ease of Use

In the above evaluation of regular Gaussian networks it was shown that these networks are very robust, converge for almost any choice of parameters and are suitable for true on-line learning. In addition these networks utilize all available hidden nodes and store information in an accessible format, allowing easy interpretation and interrogation of the network. This type of network is hence easy to use and does not require an in-depth understanding of neural networks.

The above assessment of regular Gaussian networks illustrated that despite several drawbacks, these networks are capable of nonlinear modelling and overcome some of the problems experienced with multilayer perceptrons discussed in the previous section.

3.4 DISCUSSION

The suitability of two typical feedforward networks for nonlinear modelling was analyzed in this chapter. This subsection summarizes the main results of this evaluation and highlights one further drawback of all feedforward networks.

3.4.1 A COMPARISON OF MULTILAYER PERCEPTRONS AND REGULAR GAUSSIAN NETWORKS

Both types of network perform poor extrapolation and only accomplish satisfactory interpolation if the local islands of training data can be incorporated into one node.

Due to the local interpretation of inputs in the regular Gaussian network, this structure is more suited for true on-line learning, does not *forget* when trained in localized regions, utilizes all its available nodes (independent of the initial parameter choices) and lends itself for storing and extracting knowledge.

Despite these advantages the inflexible regular Gaussian topology can lead to suboptimal results and becomes impractical for high resolution multi-input systems due to the large number of hidden nodes required for such a system. A more adaptable structure, such as that of the multilayer perceptron, may therefore be more appropriate for such applications.

The tests performed in the assessment of the two structures do suggest that the loss in flexibility in a regular Gaussian network is compensated for by its robust performance and that these networks are therefore more suitable for the application considered in this study.

3.4.2 THE LACK OF DYNAMICS IN FEEDFORWARD NETWORKS

This section highlights one common limitation shared by all feedforward neural network models of dynamic systems. Since feedforward neural networks propagate an input vector in one direction through each of the layers to produce an output, these structures possess no internal dynamics. In other words, the network can only utilize information contained in the current input vector to generate the output and has no access to previous inputs, outputs or internal values.

For models of dynamic systems, in which the output(s) is (are) a function of current and previous inputs as well as previous outputs, this implies that the order of the system (number of previous inputs and outputs required) has to be known prior to training and that all these previous values have to be included in the network input vector.

The lack of dynamics in these networks also implies that the network can only predict one step ahead and that feedforward neural network models are therefore not suited as off-line models, unless previous network predictions are re-utilized as network inputs. Such a feedforward structure with feedback as well as several other attempts to introduce dynamic behavior into neural networks, are investigated in the following chapter.

REFERENCES

1.....Rosenblatt F., *"PRINCIPLES OF NEURODYNAMICS: PERCEPTRONS AND THE THEORY OF BRAIN MECHANISMS"*, Washington, DC: Sparta Books, 1962.

2.....Rumelhart D.E., McClelland J.L. and the PDP Research group, *"PARALLEL DISTRIBUTED PROCESSING - Exploration in the Microstructure of Cognition, Volume 1: Foundations"*, Cambridge, MA: MIT Press, 1986.

3.....Hornik K., Stinchcombe M. and White H., "*MULTILAYER FEEDFORWARD NETWORKS ARE UNIVERSAL APPROXIMATORS*", Neural Networks, Vol. 2, 1989, pp. 359-366.

4.....De Villiers J. and Barnard E., "*BACKPROPAGATION NEURAL NETS WITH ONE AND TWO HIDDEN LAYERS*", IEEE Transactions on Neural Networks, Vol. 4, No. 1, 1992, pp. 136-141.

5.....Srender K.K., "*EFFICIENT ACTIVATION FUNCTIONS FOR THE BACK-PROPAGATION NEURAL NETWORK*", presented at IJCNN 1991.

6.....Wray J. and Green G.G.R. , "*HOW NEURAL NETWORKS WORK: THE MATHEMATICS OF NETWORKS USED TO SOLVE STANDARD ENGINEERING PROBLEMS*", ACC 1991, pp. 2311-2313.

7.....Cybenko G., "*APPROXIMATION BY SUPERPOSITIONS OF A SIGMOIDAL FUNCTION*", Mathematics of Control, Signals, and Systems, Vol. 2, 1989, pp. 303-314.

8.....Watrous R. L., "*LEARNING ALGORITHMS FOR CONNECTIONIST NETWORKS: APPLIED GRADIENT METHODS OF NONLINEAR OPTIMIZATION*", IICNN 1987, pp. II619-II627.

9.....Parker D.B. , "*OPTIMAL ALGORITHMS FOR ADAPTIVE NETWORKS: SECOND ORDER BACKPROPAGATION, SECOND ORDER DIRECT PROPAGATION, AND SECOND ORDER HEBBIAN LEARNING*", IICNN 1987, pp. II593-II600.

10.....Cater J.P., "*SUCCESSFULLY USING LEARNING RATES OF 10 (AND GREATER) IN BACK-PROPAGATION NETWORKS WITH THE HEURISTIC LEARNING ALGORITHM*", IICNN 1987, pp. II645-II651.

11.....Weir M.K., "*A METHOD FOR SELF-DETERMINATION OF ADAPTIVE LEARNING RATES IN BACK PROPAGATION*", Neural Networks, Vol. 4, 1991, pp. 371-379.

12.....Rigler A.K., Irvoine J.M. and Vogl T.P., "*RESCALING OF VARIABLES IN BACK PROPAGATION LEARNING*", Neural Networks, Vol. 4, 1991, pp. 225-229.

13.....Dahl E.D., "*ACCELERATED LEARNING USING THE GENERALIZED DELTA RULE*", IICNN 1987, pp. II523-II530.

14.....Shoemaker P.A., Carlin M.J. and Shimabukuro R.L., "*BACK PROPAGATION LEARNING WITH TRINARY QUANTIZATION OF WEIGHT UPDATES*", Neural Networks, Vol. 4, 1991, pp. 231-241.

15.....Stornetta W.S. and Huberman B.A., "*AN IMPROVED THREE-LAYER BACK PROPAGATION ALGORITHM*", IICNN 1987, pp. II637-II643.

16.....Hirose Y., Yamashita K. and Hijiya S., "*BACK-PROPAGATION ALGORITHM WHICH VARIES THE NUMBER OF HIDDEN UNITS*", Neural Networks, Vol. 4, 1991, pp. 61-66.

17.....Bremermann H.J. and Anderson R.W., "*AN ALTERNATIVE TO BACK-PROPAGATION: A SIMPLE RULE OF SYNAPTIC MODIFICATION FOR NEURAL NET TRAINING AND MEMORY*", Internal Report, Center for Pure and Applied Mathematics, University of California, Berkeley, 1990.

18.....Bremermann H.J. and Anderson R.W., *"HOW THE BRAIN ADJUSTS SYNAPSES - MAYBE"*, preprint, to appear in *"FESTSCHRIFT FOR WOODY BLEDSOE"*.

19.....Willis M.J., Di Massimo C., Montague G.A., Tham M.T. and Morris A.J., *"ARTIFICIAL NEURAL NETWORKS IN PROCESS ENGINEERING"*, IEE Proceedings-D, Vol. 138, No. 3, 1991, pp. 256-266.

20.....Kirkpatrick S., Gelatt C.D. and Vecchi M.P., *"OPTIMIZATION BY SIMULATED ANNEALING"*, Science, Vol. 220, 1983, pp. 671-680.

21.....Hopfield J.J. and Tank D.W., *"NEURAL COMPUTATION OF DECISIONS IN OPTIMIZATION PROBLEMS"*, Biological Cybernetics, Vol. 52, 1985, pp. 141-152.

22.....Ingman D. and Merlis Y., *"LOCAL MINIMUM ESCAPE USING THERMODYNAMIC PROPERTIES OF NEURAL NETWORKS"*, Neural Networks, Vol. 4, 1991, pp. 395-404.

23.....Qin S., Su H. and McAvoy T., *"COMPARISON OF FOUR NEURAL NET LEARNING METHODS FOR DYNAMIC SYSTEM IDENTIFICATION"*, IEEE Transactions on Neural Networks, Vol. 3, No. 1, 1992, pp. 122-130.

24.....Cybenco G., *"APPROXIMATION BY SUPERPOSITION OF A SIGMOIDAL FUNCTION"*, Math. Control Signals & Systems, Vol. 2, 1989, pp. 303-314.

25.....Trossbach W. and Braae M., *"INVESTIGATION INTO NEURAL NETWORKS FOR CONTROL OF NON-LINEAR PROCESSES"*, presented at SACAC Tutorial and Workshop on Neural Networks, Durban University, July 1991.

26.....Narendra K.S. and Parthasaraty K., *"IDENTIFICATION AND CONTROL OF DYNAMICAL SYSTEMS USING NEURAL NETWORKS"*, IEEE Transactions on Neural Networks, Vol. 1, No. 1, 1990, pp. 4-27.

27.....Youngjik L., Sang-Hoon O. and Myung W.K., *"THE EFFECT OF INITIAL WEIGHTS ON PREMATURE SATURATION IN BACK-PROPAGATION LEARNING"*, IJCNN 1991, pp. I765-I770.

28.....Girosi F. and Poggio, *"NETWORKS AND THE BEST APPROXIMATION PROPERTY"*, Biological Cybernetics, 63, 1990, pp. 169-176.

29.....Leonard J.A. and Kramer M.A., *"RADIAL BASIS FUNCTION NETWORKS FOR CLASSIFYING PROCESS FAULTS"*, IEEE Control Systems Magazine, April 1991, pp. 31-38.

30.....Lee S. and Kil R.M., *"MULTILAYER FEEDFORWARD POTENTIAL FUNCTION NETWORK"*, pre-print, submitted to Neural Networks September 1989.

31.....Moody T.J. and Darken C.J., *"FAST LEARNING IN NETWORKS OF LOCALLY TUNED PROCESSING UNITS"*, Neural Computation, Vol. 1, 1989, pp. 151-160.

32.....Sbarbaro D. and Gawthrop P.J., *"SELF-ORGANIZATION AND ADAPTION IN GAUSSIAN NETWORKS"*, 9th IFAC/IFOR Symposium on Identification and System Parameter Estimation, 1991.

33.....Chinrungrueng C. and Sequin C.H., *"OPTIMAL ADAPTIVE K-MEANS ALGORITHM WITH DYNAMIC ADJUSTMENT OF LEARNING RATE"*, IJCNN 1991, pp. I855-I862.

34.....Hunt K.J. and Sbarbaro D., *"NEURAL NETWORKS FOR NONLINEAR INTERNAL MODEL CONTROL"*, IEE Proceedings-D, Vol. 138, No. 5, 1991, pp. 431-438.

35.....Roger Jang J.S. and Sun C.T., *"FUNCTIONAL EQUIVALENCE BETWEEN RADIAL BASIS FUNCTION NETWORKS AND FUZZY INFERENCE SYSTEMS"*, IEEE Transactions on Neural Networks, Vol. 4, No. 1, 1993, pp. 156-159.

4. NETWORKS WITH DYNAMIC MODELLING CAPACITY

The discussion of feedforward networks in the previous chapter illustrated their ability to copy nonlinear input/output mappings but also accentuated their lack of dynamic response as one of the drawbacks for their use as models of dynamic systems. Since this shortcoming is inherent to the feedforward topology, more complex network structures are required for dynamic modelling.

This chapter starts with an overview of such networks, before introducing and evaluating one of the better known topologies and training algorithms.

4.1 OVERVIEW

Due to the static behavior of feedforward networks the number of previous input(s) and output(s), required to predict the next output(s), have to be identified prior to training and the prediction horizon is furthermore limited to one sample interval. The former can be overcome either if the system to be modelled is well defined and understood or by adopting a trial and error approach in which these numbers are varied until a satisfactory result is obtained. The limitation to one prediction implies however, that feedforward networks can not be used as *stand alone* models without modifying their topology. Despite the fact that both these reasons do not prevent the use of feedforward neural networks as part of a control structure, they warrant an investigation into other topologies which overcome these limitations.

The lack of dynamic behavior in feedforward networks is due to the fact that information is only propagated in a forward direction and hence none of the internal or external outputs are retained and re-used when the next output is calculated. In order to include dynamic behavior in these structures, some form of information storage is required. This is generally accomplished using some form of feedback within the network i.e. the outputs of nodes are fed back to be re-utilized as node inputs, thus ensuring that the network output is a function of both the external inputs as well as the previous state of the network.

These feedback connections, included for dynamic response, are time delayed (the previous output is used to calculate the next output) and the resulting topology, hence also known as *time-lag* recurrent structure, therefore produces a valid output vector, which incorporates the current inputs as well as the previous state of the network, for each iteration.

The type of recurrent structure, referred to above, is not to be mistaken for the fully connected *associative memory* topology (such as the Hopfield network), where all nodes are

interconnected and transmission along all connections is supposed to occur simultaneously. The fact that the equations governing such structures have to be solved iteratively before converging to the final stable network state (which is then used as valid output) and thus produce a series of output vectors for each new input, is an undesirable side-effect due to the sequential nature of digital computers, rather than a desired consequence of the network topology.

As with feedforward networks, many different topologies and optimization algorithms exist and the choice of network again requires a careful examination. The following subsections introduce some of the approaches to date.

4.1.1 EXTERNAL RECURRENT NETWORKS

One approach to overcome the *one step ahead* limitation of feedforward networks is to utilize time delayed versions of the network estimations, rather than the system outputs, in the input vector. Once trained, such a structure could then be used as a stand alone model for long term predictions since only the system input values have to be supplied.

(a) Topology

Figure 4.1 below shows this external recurrent topology, in which theoretically any type of feedforward network can be utilized in the position indicated.

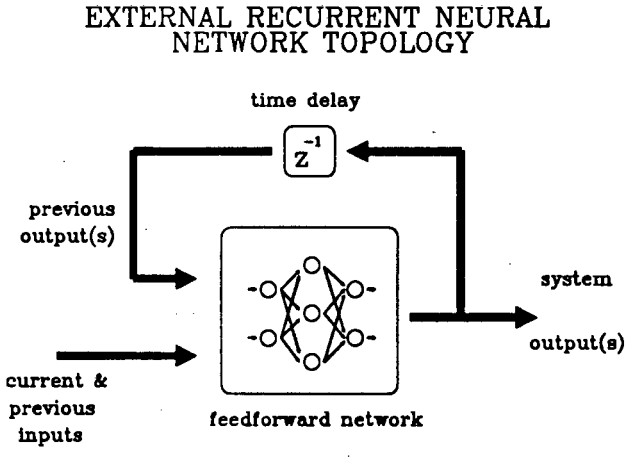


Figure 4.1 - External Recurrent Topology

As illustrated, the output(s) of the feedforward network are fed back via a time delay, to form part of the network's input vector. Although a single time delay is depicted in figure 4.1 above, many training methods permit multiple delays per output if required.

(b) Network Training

Despite the fact that the external recurrent topology is constructed around a standard feedforward network, the algorithms used to train the feedforward network are not applicable, without modification. This is due to the fact that a new source of error has been introduced and even though the network is still expected to learn the same input/ouput mapping as in the standard feedforward topology, convergence is no longer guaranteed.

Training algorithms for feedforward networks presume a *correct* input vector i.e. the algorithms are based on the assumption that any error between actual and desired output(s) is solely due to network parameters and not due to inaccuracies in the input vector. This does not apply to the topology shown in figure 4.1, where any error in the network outputs becomes an error in the input vector for the subsequent iteration and therefore produces an *invalid* input/output pair, which distorts the network mapping and hence complicates learning.

Several modifications to conventional feedforward training methods and algorithms have been suggested to overcome this problem:

Narendra et al [1] for example, propose that a multilayer perceptron be trained for use in the external recurrent topology using standard backpropagation but that such training be performed in two stages. Initially the standard *series-parallel* structure, in which true system outputs are utilized as network inputs, is employed. Once the network has converged sufficiently, further training is performed in the *parallel* configuration in which the network outputs, rather than the system outputs, are employed.

A mathematically more correct method, in which the network is trained entirely in the external recurrent configuration, is known as *backpropagation through time*. This method has its origins in the work by Rumelhart et al [2], who unfold the recurrent structure into a feedforward network growing by an extra layer for each sample interval; a method only suitable for finite time series. The algorithm was formalized for the general external recurrent structure depicted in figure 4.1 by Werbos [3], using his *ordered derivative* approach. In their study of neural network models for a biological wastewater treatment plant, Su and McAvoy [4] employ this training algorithm to compare the performance of an external recurrent network with that of a feedforward structure.

(c) Discussion

The external recurrent structure can overcome the one step ahead limitation of feedforward networks as is evident from the work of Su and McAvoy mentioned above, in which they show that the recurrently trained structure performs better for long term prediction. The approach does however, not eliminate the need to identify the number of

delayed outputs that have to be included in the input vector. In other words, the designer still has to identify the number of feedback requirements before initializing and training the network.

In their comparison Su and McAvoy furthermore show that for one step ahead prediction, a recurrently trained network can not match the accuracy of the feedforwardly trained structure. Since most control structures require high accuracy and are only evaluated sample by sample, this suggests that external recurrent networks might find more use as off-line models, rather than as part of a control loop.

4.1.2 NETWORKS WITH DYNAMIC NODES

In the external recurrent topology described previously, the feedback connections extend from the network outputs back to its inputs and hence only the final network output values are preserved for subsequent iterations. This subsection introduces an alternative approach in which the output of each individual node is dynamic and each node thus has a time delayed feedback connection to itself.

(a) Topology

Figure 4.2 below depicts the structure of a feedforward network with dynamic nodes.

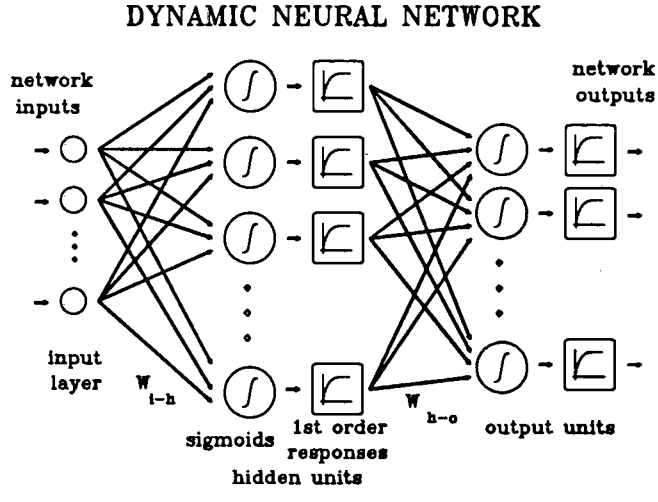


Figure 4.2 - Networks with Dynamic Nodes

This type of network, suggested by Willis et al [5] is in effect a multilayer perceptron, in which each node in all the hidden layers and the output layer is augmented by a first order response, for which the time constant has to be determined as part of the training

process. The output of this type of network is therefore the sum of a number of nonlinear first order responses.

Even though only a single hidden layer is shown in figure 4.2, the network's modelling ability can be increased further by adding more layers with dynamic nodes. Willis et al for example, usually implement a network with two hidden layers in their studies [5][6][7] in which they employ this topology to model various highly nonlinear processes such as biomass concentration in a fermenter and product composition in a distillation column.

(b) Network Training

Since the filter time constant for each of the nodes has to be determined as part of the training process, the backpropagation algorithm is not suitable for these modified perceptrons. In their studies, Willis et al use the chemotaxis algorithm (discussed in the previous chapter), which implements a directed search optimization, independent of any network equation or gradient and hence also independent of the network topology.

(c) Discussion

Although the idea of compounding each of the nodes in a feedforward network with a first order response is biologically plausible and provides the system with dynamic capability, the approach still has several shortcomings.

The information retained in the network for subsequent iterations is not shared amongst all nodes and does not reappear as network inputs. In other words, the previous state of a node is only available to *that* node, for subsequent iterations and hence none of the retained information is shared directly. The above topology is therefore not truly *recurrent* but rather a feedforward type network with some dynamic capability.

An examination of this structure furthermore raises the question, whether such an attempt of modelling the system as a number of nonlinear first order responses could not be improved by extending the capability of each node to include dead time, as well as oscillatory response (i.e. a complex pole).

The most serious restriction of the above topology is nevertheless the lack of a pattern training algorithm which would allow on-line optimization of these networks. As pointed out in chapter three, the chemotaxis optimization technique is not suited for pattern learning since the effect of a change in parameters has to be evaluated over the entire training set.

4.1.3 FULLY RECURRENT NETWORKS

In the approaches discussed, the information retained in the network was either limited to a few outputs (external recurrent structure) or restricted to a subsection of the network (dynamic node network). For the *fully recurrent* topology on the other hand, the output of each node is re-used in subsequent iterations and forms a scaled input to *all* other nodes. This implies, that the complete state of the network can be re-used by each node in the subsequent iteration.

(a) Topology

The structure of a node for a fully recurrent network is illustrated in figure 4.3 below.

NODE STRUCTURE FOR FULLY RECURRENT TOPOLOGY

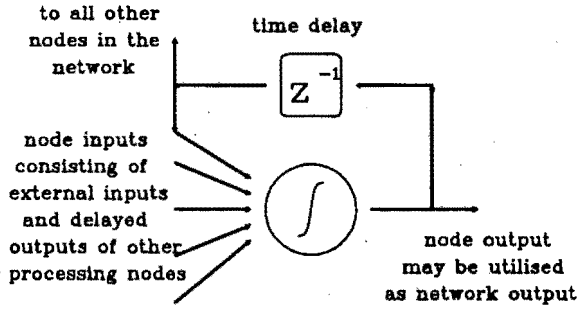


Figure 4.3 - Fully Recurrent Network Node Structure

As shown, the interconnections of the network are such that each node receives the previous outputs from all other nodes, together with all external network inputs, as its inputs. In such a structure the network can no longer be separated into distinct layers and nodes are classified as *input* if they serve to distribute an external input signal, as *output* if their output is used as one of the system outputs and as *hidden* otherwise. The output nodes are a subset of the processing nodes which implies that the structure must contain at least as many nodes as outputs required.

Since each of the processing nodes receives all inputs as well as the previous outputs of all processing nodes, the input vector is a combination of external inputs as well as the

previous state of the network. This is known as the fully interconnected or *fully recurrent* structure in which the entire network state can be re-used in the subsequent iteration. Due to the fact that the recurrent connections are furthermore adjustable, the network can be trained to *select or learn* the required delayed output values and internal states required for the next iteration. The fully recurrent structure is therefore extremely flexible and requires little designer input.

(b) Network Training

Due to the high degree of feedback or recurrent connectivity, the fully recurrent structure requires an optimization method, which includes and compensates for the iterative nature, in which the error of previous outputs influences the result of subsequent iterations.

Although most early recurrent network investigations employed search based algorithms (which are only suitable for batch and hence off-line training) a number of gradient based training methods have been developed in recent years. Of these, the method suggested by Williams and Zipser [8] is probably the best known although other algorithms such as Tsung [9] and Sun et al [10] are also being used [11]. These algorithms apply a *gradient descent* type optimization and can therefore be applied on-line, in a pattern learning structure, much like the gradient based algorithms for feedforward networks. The William and Zipser algorithm is discussed in detail later in this chapter.

(c) Discussion

The time-lag, fully recurrent structure is clearly the most versatile and suitable of the three topologies introduced. It theoretically allows a *black box* approach in which the network is merely supplied with the current system input(s) and desired output(s) from which it then establishes all the required feedback connections, thus becoming a true dynamic model. Due to the internal feedback connections such a trained network can then be utilized as both an on-line model within a control structure and for stand alone, off-line simulation.

The fully recurrent structure could therefore overcome the one-step ahead limitation and eliminate the need to identify the number of delayed input(s) and output(s) required. These features, together with the fact that algorithms exist to train these networks on-line, warrant a more detailed evaluation of this type of network.

4.2 EVALUATION OF A FULLY RECURRENT NETWORK

The topology and training algorithm suggested by Williams and Zipser [8] is evaluated in this subsection. Although all the criteria established for the assessment of feedforward topologies still apply and are included in the discussion, the aim is to establish whether these structures are more suitable as models of dynamic systems than the feedforward networks introduced in the previous chapter. Printouts of all relevant programs are included in Appendix II.

4.2.1 NETWORK TOPOLOGY

The fully recurrent structure as per Williams and Zipser, shown in figure 4.4 below, is very similar to the general topology described previously (figure 4.3).

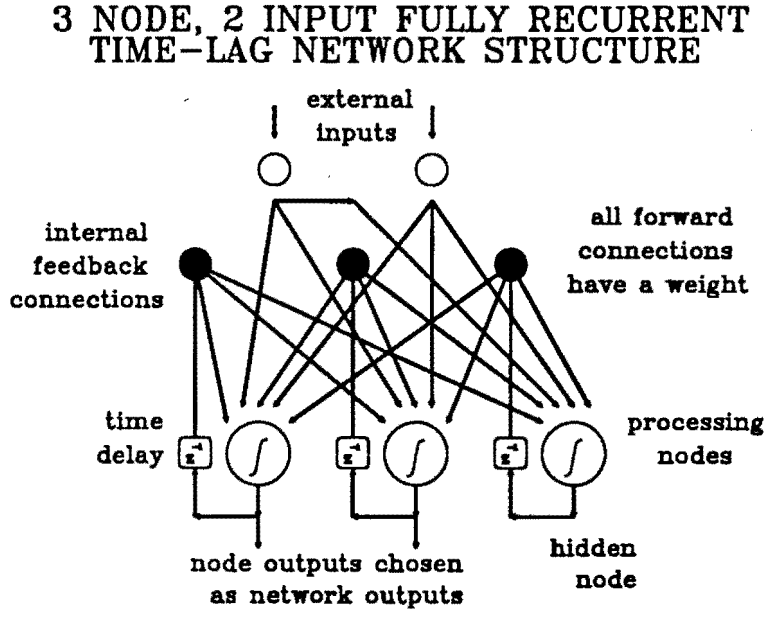


Figure 4.4 - Williams & Zipser Recurrent Network Structure

As indicated, all processing nodes utilize the logistic squashing function that is also used in the multilayer perceptron. Although other semi-linear functions can be employed, the clamping nature of the sigmoidal function has an important stabilizing effect in this structure with possible positive feedback connections. The resulting similarity with the multilayer perceptron is mentioned in the following evaluation.

Since the weights of the feedback connections in the above structure have to be established as part of the training, this topology requires a training algorithm in which the effect of re-iterated errors are either compensated for or eliminated.

4.2.2 TRAINING ALGORITHM

This subsection introduces the gradient descent learning algorithm, as suggested by Williams and Zipser, for the fully recurrent structure above . As mentioned earlier, it is one of the few optimization methods for these networks, suitable for pattern learning and therefore applicable in an on-line configuration.

The description below shows the equations governing the network output generation and training. A detailed mathematical treatment can be found in [8].

The equations below assume a network with m external inputs and n processing units.

1. Standard Training Algorithm

- Using the notation as per Williams et al where
  $\underline{y}(t)$ - the n-element output vector at time t
  $\underline{x}(t)$ - the m-element external input vector at time t
  $\underline{z}(t)$ - a concatenation of $\underline{y}(t)$ and $\underline{x}(t)$ i.e.

$$z_k(t) = \begin{cases} y_k(t) & \text{if } k \in U \\ x_k(t) & \text{if } k \in I \end{cases}$$

(The order of inputs and outputs was changed from that suggested by Williams and Zipser in order to simplify the calculation of indices in the software but is not significant for the functioning of the algorithm.)

- The weights connecting each of the units with each other and each of the inputs to each of the units can then be collected in a single n × (n+m) matrix W.
- A bias or offset for each unit is provided by including an external input which is always 1.
- Also let U denote the set of indices k for which z_k is the output of a processing unit, I for which z_k is an external input and T the subset of U for which z_k is furthermore an external output.

- The forward propagation of the network can then be calculated as follows: the input to each processing unit k is given by

$$s_k(t) = \sum_l w_{kl} z_l(t) \qquad l \in U \cup I$$

from which the next output y_k is then generated by

$y_k(t+1) = f_k(s_k(t))$ where $f_k()$ is the activation function for that unit. (In this evaluation only sigmoids are used.)

- Letting $d_k(t)$ denote the desired output of the kth unit at time t, the error vector $\underline{e}(t)$ is given by

$$\underline{e}(t) = \begin{cases} d_k(t) - y_k(t) & \text{if } k \in T(t) \\ 0 & \text{otherwise} \end{cases}$$

where T is a function of time thus providing the possibility of varying the processing units chosen as external outputs.

- Defining the total network error at time t as

$$J(t) = \tfrac{1}{2}\sum_k (e_k(t))^2 \qquad k \in U$$

Williams and Zipser derive a weight update rule which implements a gradient decent i.e.

$$\Delta w_{ij}(t) \propto -\frac{dJ(t)}{dw_{ij}}$$

This is achieved by defining variables p_{ij}^k for $k \in U$, $i \in U$, $j \in U \cup I$ where

$$p_{ij}^k(t+1) = f_k'(s_k(t))\left[\sum_l w_{kl} p_{ij}^l(t) + \delta_{ik} z_j(t)\right] \qquad l \in U$$

where δ_{ik} denotes the Kronecker delta.

From this the weight update is then calculated as follows

$$\Delta w_{ij}(t) = \alpha \sum_{k \in U} e_k(t) p_{ij}^k(t)$$

- As with the backpropagation algorithm, all the weight updates should be collected and only implemented once all training samples have been presented in order to implement a *true gradient decent*, but can be applied after each iteration provided the training rate α is sufficiently small.

- The above method describes a training algorithm in which the effect of re-iterated errors is compensated for.
- Another possibility is to eliminate such errors by using the desired rather than the actual output for subsequent iterations as described below.

2. Teacher-Forced Training

- For this variation of the above algorithm the original $\underline{z}$ is replaced by

$$z_k(t) = \begin{cases} d_k(t) & \text{if } k \in T(t) \\ y_k(t) & \text{if } k \in U \notin T(t) \\ x_k(t) & \text{if } k \in I \end{cases}$$

This modification ensures that any errors in the external output units are not re-iterated and hence the corresponding dynamic learning variables for these nodes are zero i.e. the update rule becomes.

$$p_{ij}^k(t+1) = f_k'(s_k(t))\left[\sum_l w_{kl} p_{ij}^l(t) + \delta_{ik} z_j(t)\right] \qquad l \in U \notin T(t)$$

In order to select one of the two training algorithms for the evaluation, their *dynamic modelling ability* was examined using data generated by a recurrent network with known parameters. A small network with a single processing/output node and one input was employed for the evaluation. To generate the training and test data, this network was then excited using random step perturbations and the resulting input and output sequences are shown in figure 4.5. Of the 1000 point data sequence, only the first half was used during training, whilst the entire series was employed to evaluate the accuracy of the trained network model.

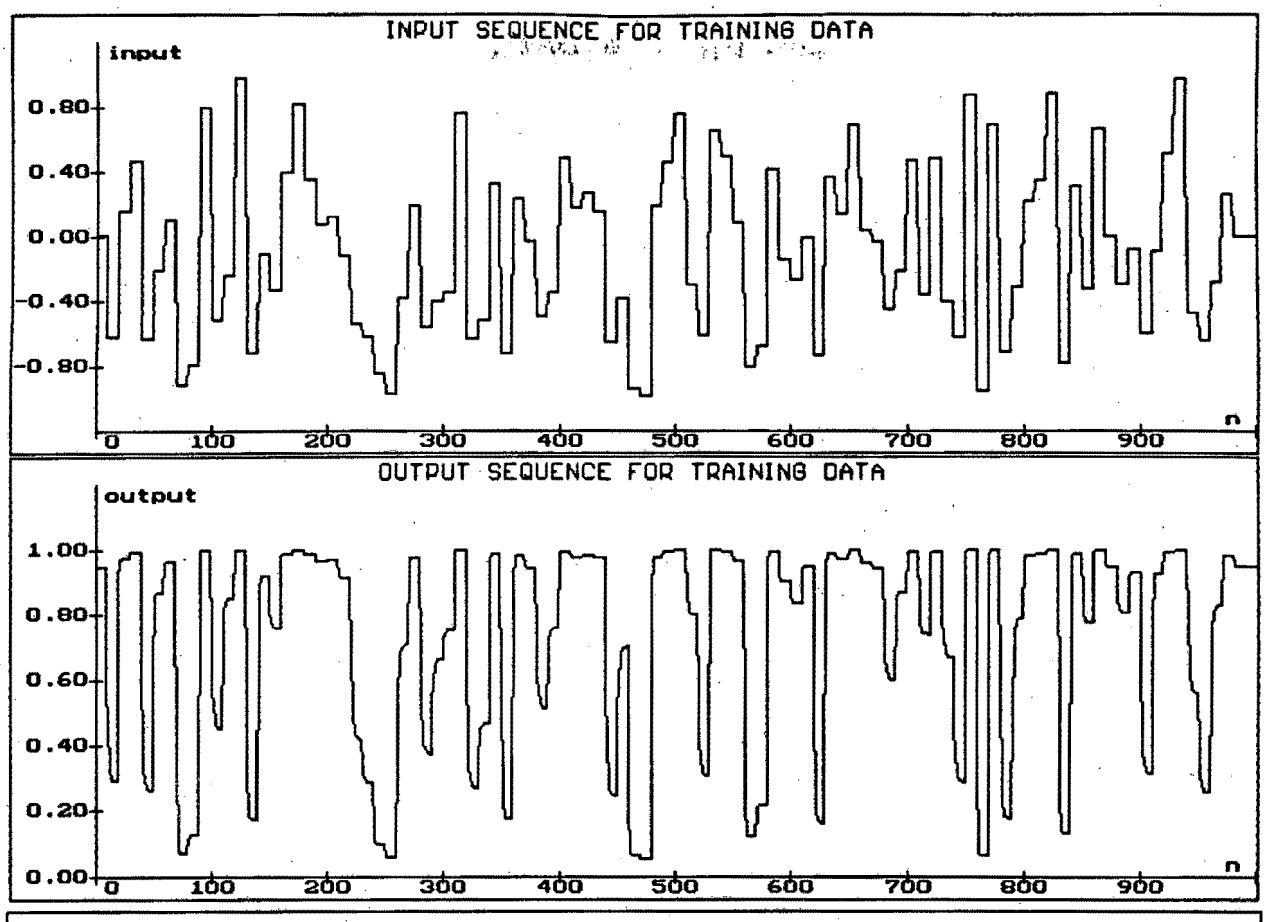


Figure 4.5 - Recurrent Network Training Data

One of the problems experienced, particularly with the teacher-forced version of the training algorithm, was that when using a fixed training rate, the rms error decreased initially as expected but then started to increase, before settling to a new value above the previously achieved minimum. An example of this profile showing the rms error versus training iterations is included in figure 4.6 below.

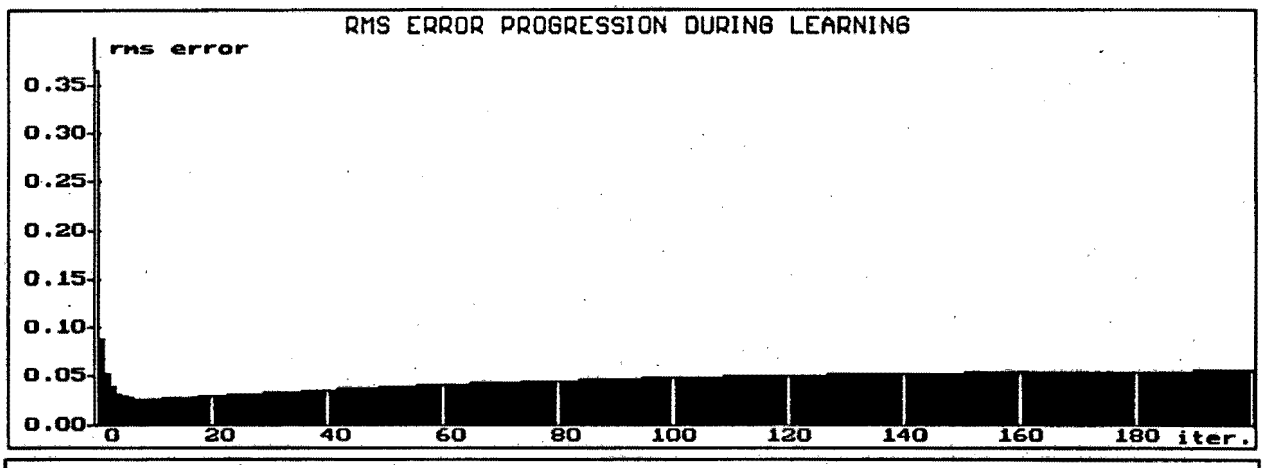


Figure 4.6 - RMS Progression for Teacher Forced Learning with Fixed Learning Rate

Although this increase in the rms error for fixed learning rates can also be observed with the normal learning algorithm, it generally only occurs much later during the training cycle and is of negligible magnitude. The type of profile shown, suggests that the training rate is too high, thus causing an overshoot and preventing the algorithm from settling closer to the actual minimum. To overcome this problem, a smaller fixed training rate as well as an automatic training rate adjustment procedure (in which the training rate is halved whenever the rms error increases and new weights are only accepted if they result in a lower rms) were tried. Only the *automatic training rate adjustment* produced better results and it was applied for all teacher force learning.

The two versions of the algorithm were then used to train a network of the same structure as that used to generate the training data. The training was performed for five different sets of initial conditions (weights) and in each case training was terminated after one hundred iterations at which stage the algorithms had usually converged. The desired and network output for one such training run is depicted in figure 4.7. The figure shows these signals prior to training (a), after standard training (b) and after teacher-force training (c).

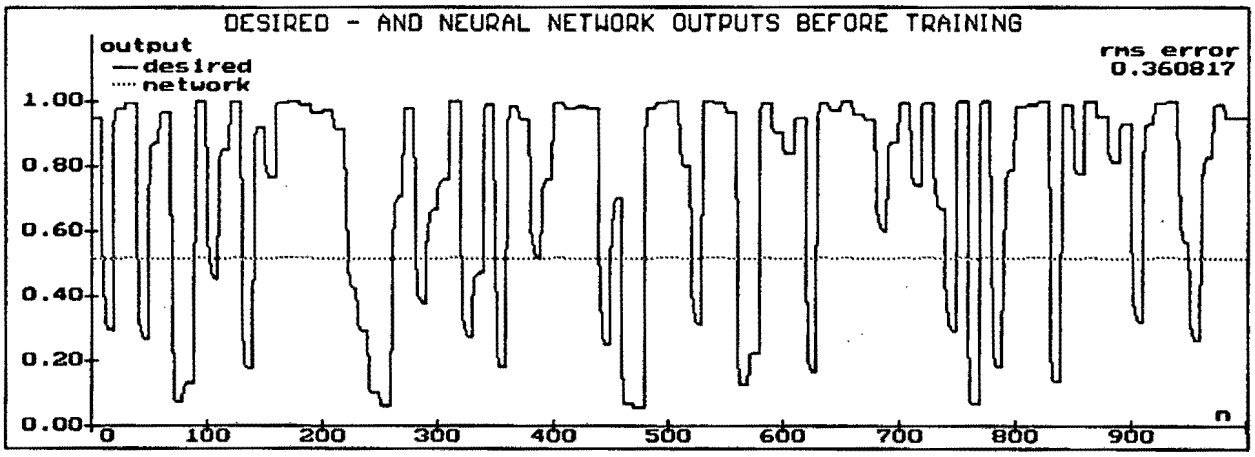


Figure 4.7(a) - Desired and Network Outputs before Training

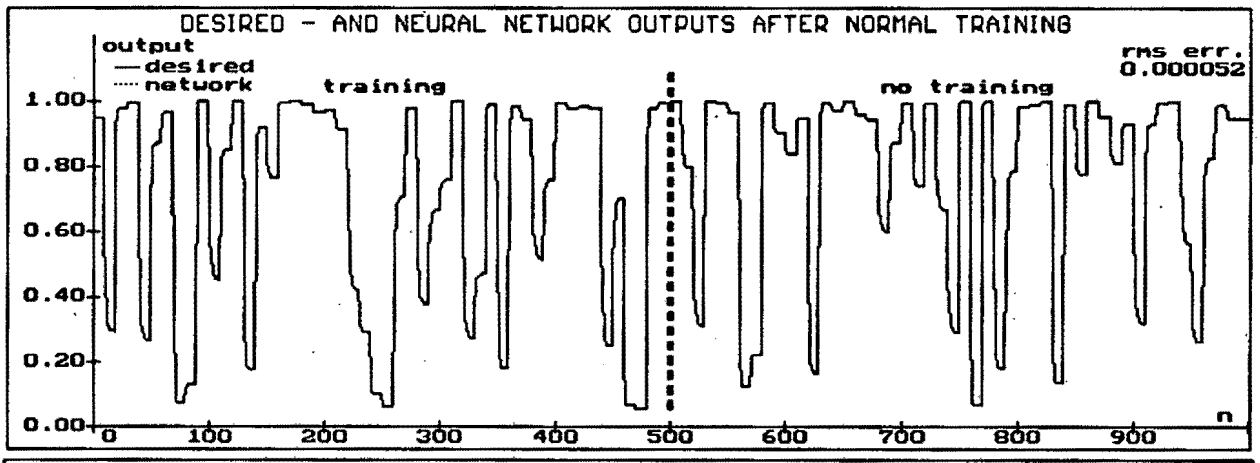


Figure 4.7(b) - Desired and Network Output after Normal Training (Indistinguishable)

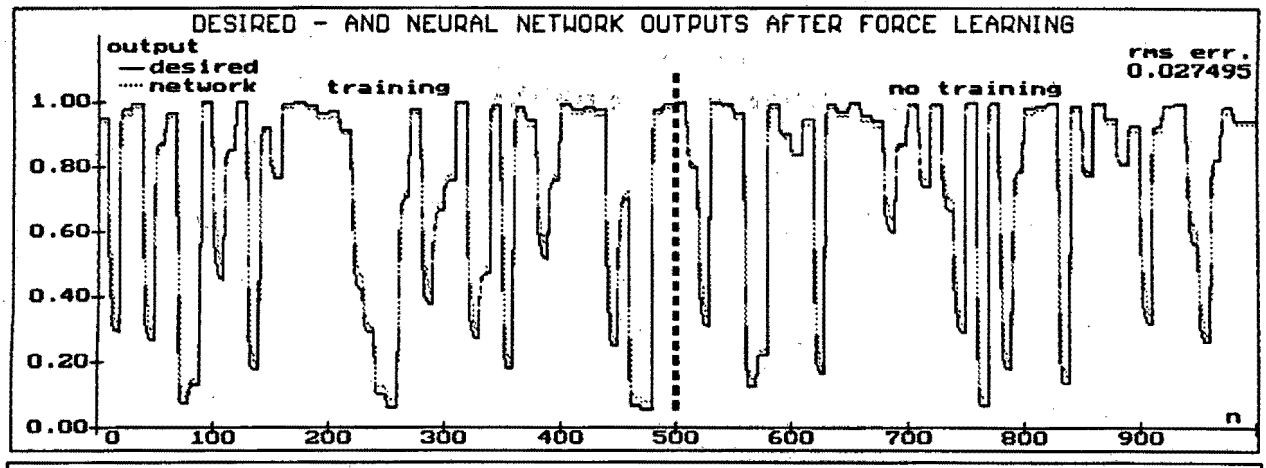


Figure 4.7(c) - Desired and Network Outputs after Teacher-Force Training

As shown both algorithms produce a model which follows the desired output, but whilst desired and network outputs are still distinguishable for the teacher-force trained network [4.7(c)], they are practically identical after standard training [4.7(b)]. This outcome was the same for all five training runs and despite the different starting points and various training rates, the teacher force version never converged to the same level of accuracy as the standard version of the algorithm. Figure 4.8 below shows the network parameters used to generate the training data, together with the estimates produced by the two versions for each of the five training runs.

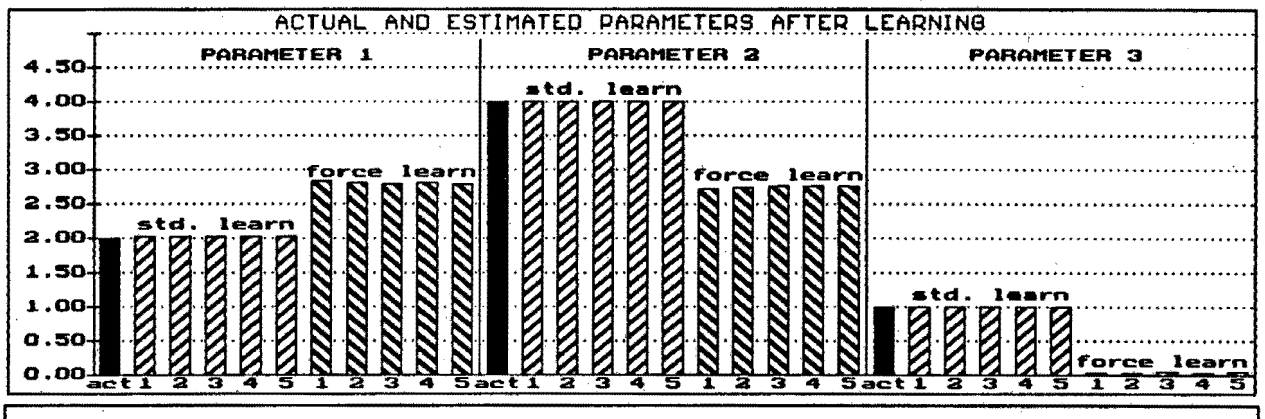


Figure 4.8 - Actual Parameters and Estimates

As indicated, the standard training algorithm converged towards the *actual (act)* parameters i.e. the *global minimum* for each run, whereas the teacher force method got stuck in a *local minimum* for each training cycle, despite the differing initial conditions.

Whilst these results are not intended to discredit the validity and usefulness of the teacher force training method, they suggest that although the former might be useful in situations where the network has to learn marginally stable behavior such as to model an oscillator [8], the standard version of the training algorithm is more suited to the type of dynamic modelling investigated in this study. All training in the evaluation was therefore performed using standard recurrent learning.

4.2.3 NONLINEAR MODELLING CAPABILITY - A COMPARISON WITH FEEDFORWARD NETWORKS

The evaluation of recurrent networks (as per Williams and Zipser) in this subsection focuses on a comparison with feedforward networks, in particular regular Gaussian networks, rather than assessing their capability in isolation.

Due to the use of the same logistic squashing function as in the multilayer perceptron, the recurrent structure furthermore shares many of the problems discussed for the perceptron in the previous chapter. Among these are the inability to interpolate and extrapolate, to learn without destroying previous knowledge and to utilize all the nodes provided. The complex structure of the recurrent structure is furthermore difficult to analyze, which makes the initialization and/or interpretation of these networks even more complex than for the multilayer perceptron.

Since the recurrent topology requires training data to be presented in chronological order the random/non-random training discussed for feedforward networks is not applicable.

(a) Test Setup

The structure of a fully recurrent network with its internal feedback connections would be underutilized in a *single-valued mapping* investigation as performed for the feedforward structures in the previous chapter and hence all tests are performed using data from dynamic systems, such that the network can utilize these feedbacks.

The desired outputs were furthermore scaled to fall into the linear interval of the logistic squashing function. This was done to prevent the output node of the recurrent structure from performing any nonlinear modelling, which could lead to saturation and compromise the network performance. In other words, the output node performs as a *linear combiner* of all other nodes, much like the output nodes in feedforward networks. Since the regular Gaussian network (RGN) utilizes linear output nodes, the range of outputs is not significant and hence the same scaled values were utilized.

In the RGN topology input values are not re-scaled internally but are expected to lie within the unit cube. Any output, included in the input vector for these networks, was therefore re-scaled prior to use in order to ensure full utilization of the RGN network.

The data for each test was generated by performing random step perturbations of the input variable every 10 or 20 time steps. In each case a sequence of 1000 data samples was produced. Of these, only the first half was used for training, whilst the entire set was used for testing. The rms error shown, was calculated for the entire set of samples.

(b) Learning the Required Feedback Connections

As mentioned earlier, one of the main advantages fully recurrent structures enjoy over the feedforward topology is their ability to *learn* the required feedback combinations. In other words, the basic framework includes all the required connections and weights for feedback and hence unlike feedforward networks, no estimation of the number of previous inputs and outputs is required. In order to illustrate this advantage, a recurrent and RGN network were trained to approximate a simple linear system with complex pole (i.e. the two previous output values are required to produce the next estimate).

The two-node recurrent network, which has the capability to twice delay the output within its own structure, *learns* the required connections to become a near perfect model as shown in figure 4.9(a) below.

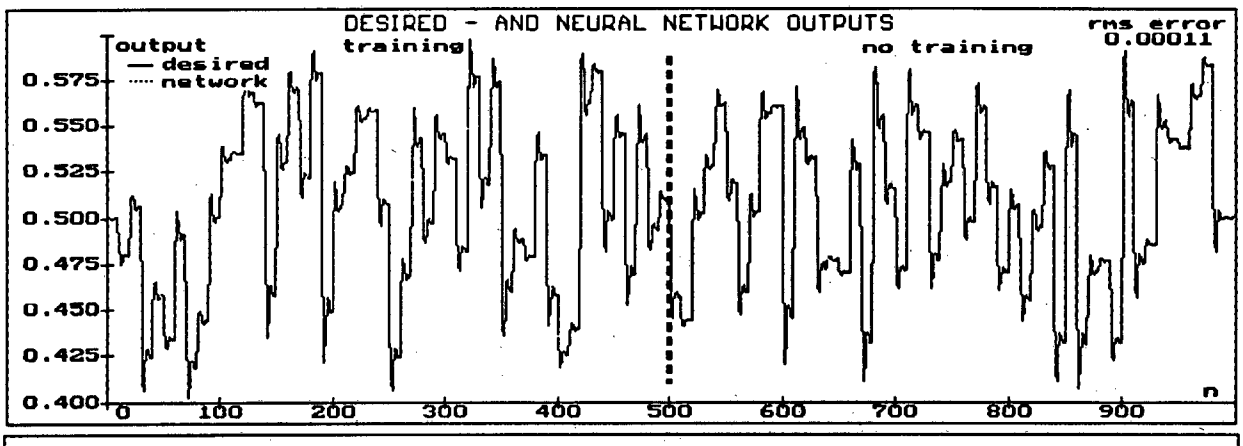


Figure 4.9(a) - Recurrent Model for Linear Process (Indistinguishable)

Unlike the recurrent structure above, the RGN network has no form of feedback and hence requires that the essential number of previous inputs and outputs (to be included in the input vector) are estimated. Any mistakes in these estimations can seriously impede the network's modelling capability as illustrated in figure 4.9(b), which shows the approximation produced by a regular Gaussian network in which only the most recent output value is included as input (i.e. the network can only perform a *first order* approximation).

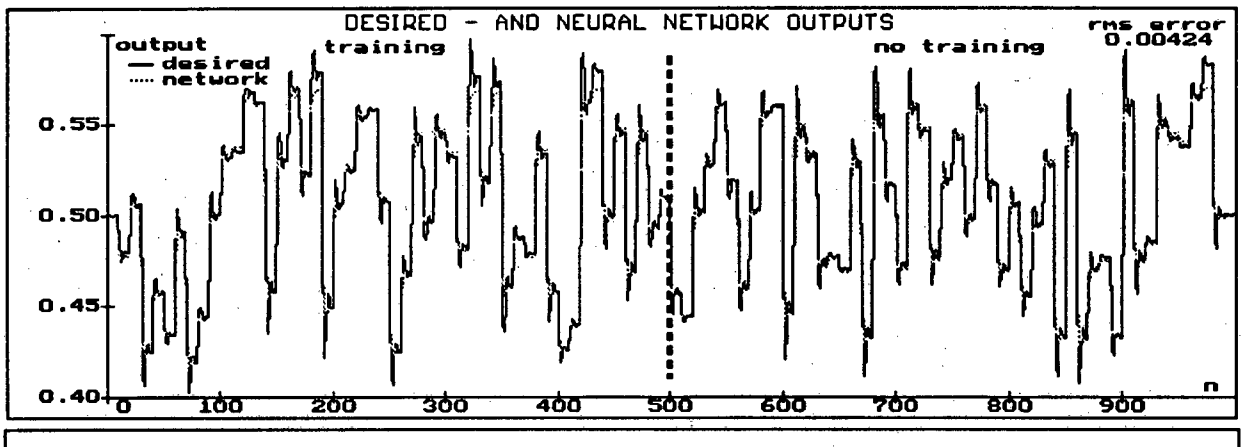


Figure 4.9(b) - First Order RGN Model for Linear Process

The restricted modelling capacity is particularly evident in the transient phases and can only be overcome by also including the twice-delayed output in the input vector. The approximation produced by this arrangement is depicted in figure 4.9(c).

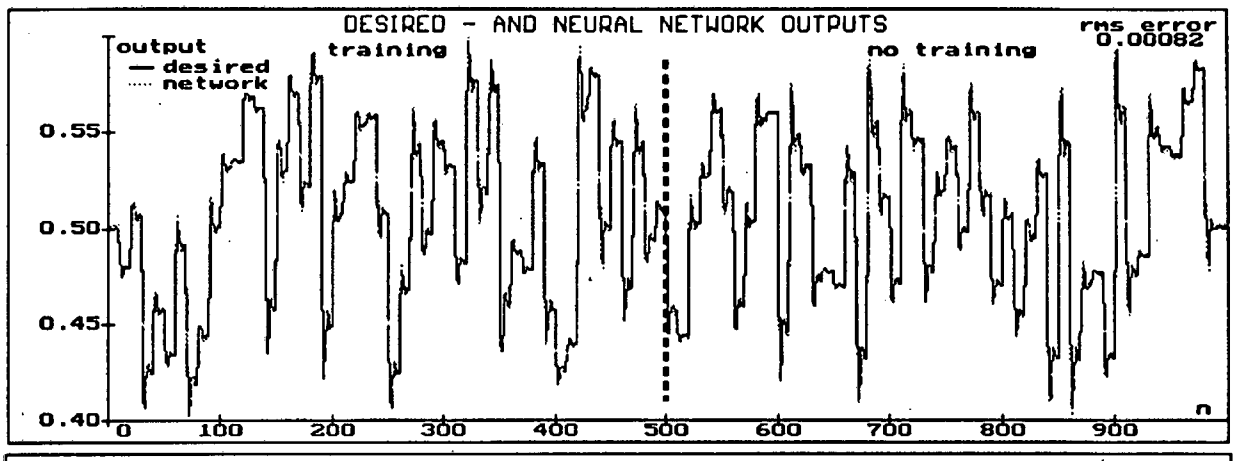


Figure 4.9(c) - Second Order RGN Model for Linear Process

As shown, this configuration produces a more accurate model. The fact that the rms error is not quite as low as for the recurrent network is due to the fixed RGN structure, which is less suited to model exact linear relationships than the flexible recurrent topology.

Besides the ability to *learn* the required feedback connections, recurrent networks are also more suitable as stand alone models. This is because these structures are trained in the so called *parallel* mode where only the system input and desired output are utilized during training. The recurrent network therefore needs no information from the actual system and can be expected to perform with the same accuracy if used as a stand alone model. For the RGN on the other hand, the *serial-parallel* structure, for which all previous output values are taken from the actual system rather than the network estimations, is used. Since these networks are not trained to operate in the *parallel* mode, their performance usually deteriorates if used in the *external recurrent configuration* discussed earlier, since any errors are re-introduced as incorrect inputs for subsequent iterations. For the RGN (figure 4.9(c)) the rms error more than doubles, if the network is employed in a stand alone (external recurrent) configuration.

Despite the above advantage, recurrent networks have a number of drawbacks as discussed in the following subsections.

(c) Local Minima and Insufficient Convergence

One of the main problems with fully recurrent networks is their incapability to consistently convergence. This problem is not unique to the Williams and Zipser training algorithm but rather a common problem with most recurrent structures. Su and McAvoy for example, employ random search techniques as part of the training in their study of external recurrent networks.

The lack of convergence is illustrated in the following example in which both, a fully recurrent and a regular Gaussian network were trained to approximate a nonlinear system governed by the following equation.

$$y_{n+1} = \frac{1}{1 + (y_n)^2} + (u_n)^3$$

As before, the first half of a 1000 point sequence was generated to train and test the networks and again the output was scaled into the linear range of the sigmoid.

Figure 4.10(a) below shows the system output together with the estimation produced by a four node recurrent network.

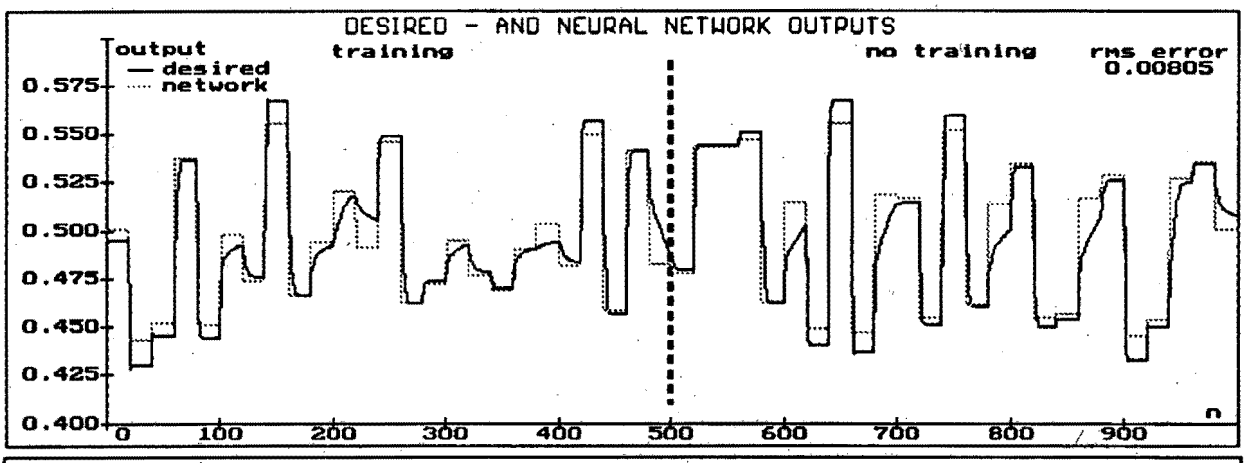


Figure 4.10(a) - Recurrent Network for Nonlinear System

As shown, the network follows the overall trend of the system but fails to capture its nonlinear and dynamic characteristics. This lack of convergence for the recurrent structure could not be improved by changing the initial conditions, allowing more training iterations or even increasing the number of processing nodes to more than double.

A RGN with two inputs and nine hidden nodes on the other hand, rapidly learns to approximate the same system as show in figure 4.10(b) below.

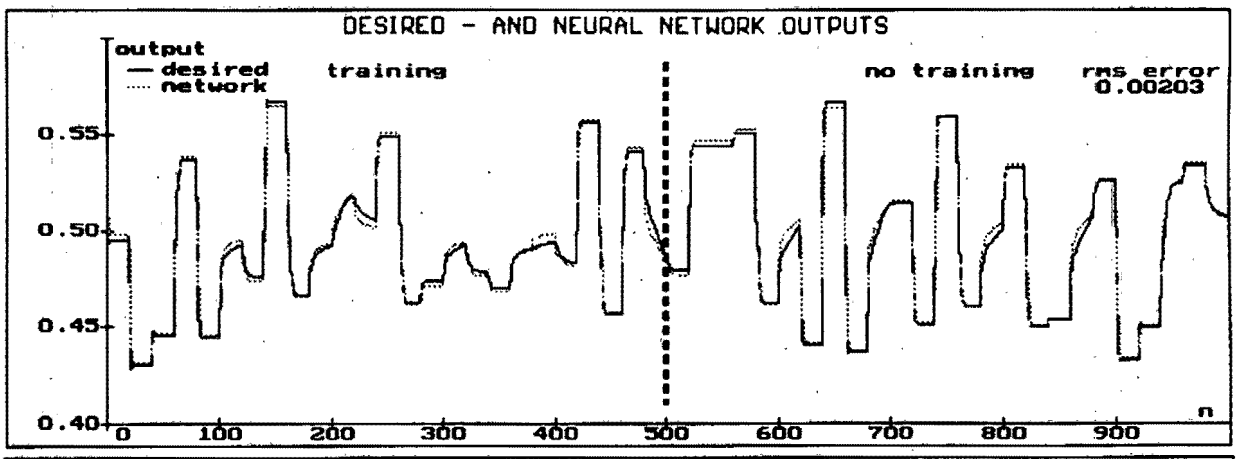


Figure 4.10(b) - RGN Model for Nonlinear System

These results suggest that despite its limiting effects, the rigid structure of regular Gaussian networks does ensure a certain level of convergence, whilst the much more flexible recurrent topology is more likely to get trapped in local minima and not to converge sufficiently.

Although an increase in the number of nodes and more training iterations did not produce a better model in this example, it might still be argued that a sufficiently large recurrent network with sufficiently long training will produce more satisfactory results. Due to the high computing requirements of recurrent training algorithms such an approach is however not practical, as shown in the following subsection.

(d) Timing Considerations

As mentioned earlier, any training algorithm for fully recurrent structures has to include and compensate for feeding back errors via its internal feedback connections. Although this is possible, as illustrated by the Williams and Zipser algorithm, it greatly increases the computational requirements of these training algorithms, when compared with those for feedforward topologies. Whilst such considerations might not be important in other applications, timing requirements are critical in sampled control systems, where the next input has to be calculated in a fraction of the sample period if the control algorithm is to perform as expected.

In an attempt to highlight the difference between the fully recurrent and regular Gaussian networks, the time for one training iteration is shown against the number of nodes in the network. The training iteration referred to includes the assigning of input(s) as well as desired output, generating the network output and performing the optimization. All measurements were performed on an IBM compatible 33 MHz 486-DX PC and in order to avoid any inaccuracy due to other interrupts, the time for one iteration was calculated as the average value of one thousand iterations.

Figure 4.11 below depicts the time requirements for a fully recurrent network with one external input and output, using the Williams and Zipser training algorithm.

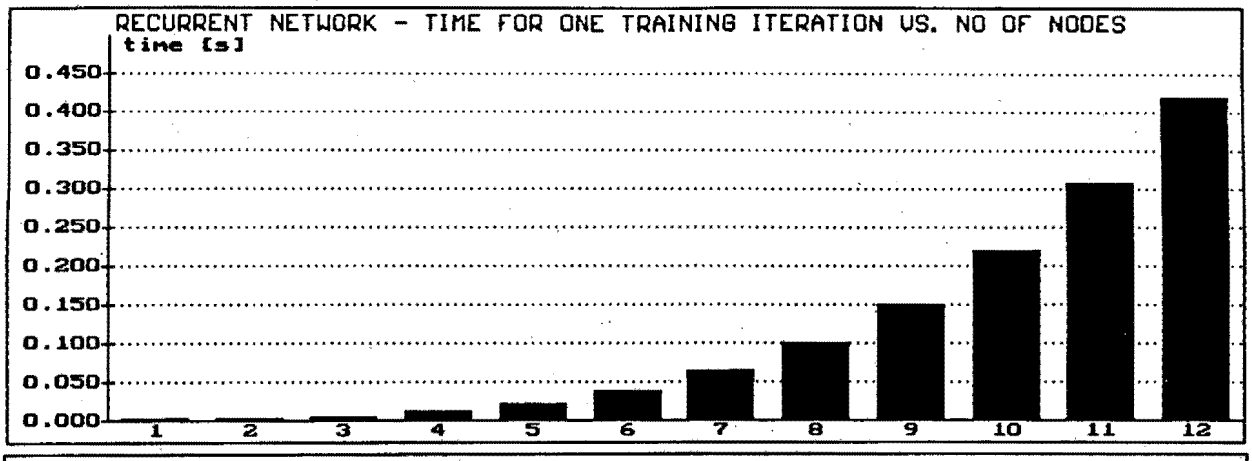


Figure 4.11 - Timing Requirements for Recurrent Network

The bar graph clearly shows the exponential growth of the time as a function of nodes, which is due to the $O(N^4)$ calculations required for each iteration (where N presents the number of nodes). As shown, the time for one iteration exceeds a quarter of a second for more than ten nodes which, when considered with the fact that often many thousand iterations are required during training, renders such structures unfeasible for incorporation into control structures.

The RGN training algorithm, on the other hand, requires O(INO) calculations, where I presents the number of inputs, N the number of hidden nodes and O the number of output nodes. For a fixed number of inputs and outputs, the number of calculations is therefore a linear function of the number of hidden nodes. In order to provide a valid comparison for the recurrent structure above, an RGN with two inputs (allowing for one previous output or feedback) and one output was chosen. The bar graph in figure 4.12 below shows the time for one iteration against the number of nodes per input dimension.

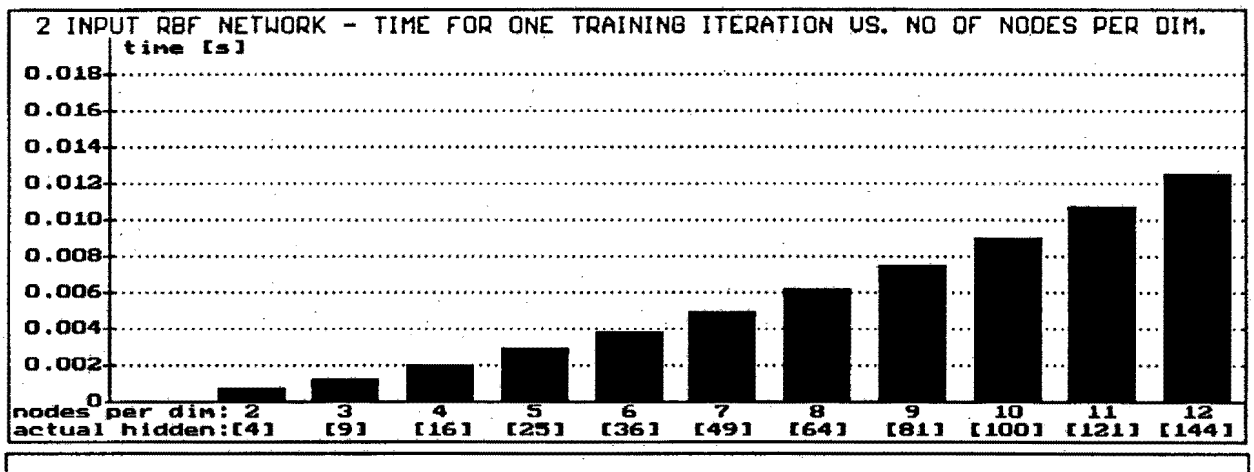


Figure 4.12 - Timing Requirements for Regular Gaussian Network

The more than linear increase is due to the fact that the total number of hidden nodes (shown in brackets) is a function of the number of inputs (here the square). In other words, the linear relationship mentioned above, holds for the total number of hidden nodes shown in brackets. Despite the fact that the RGN with 12 node resolution per input does consist of 144 hidden nodes, it still only requires one thirtieth of the time taken by the twelve node recurrent network.

One significant advantage the recurrent structure enjoys over that of the RGN is that an N node network has the built in capability to re-use the N previous inputs and outputs, whereas each of these has to be included as a separate input in the RGN structure. Since the number of hidden nodes in the RGN structure is an exponential function of the number of inputs, this can very quickly lead to a large number of hidden nodes and consequent high iteration times. This is illustrated in figure 4.13, which shows the iteration times for an RGN with 2 nodes per dimension as a function of the number of inputs. The total number of hidden nodes is again shown in brackets below.

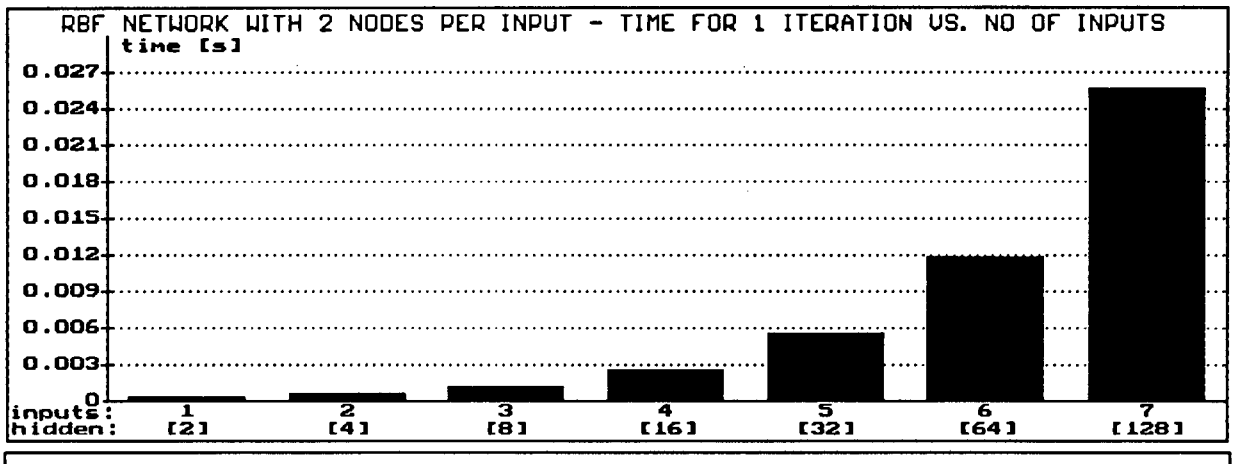


Figure 4.13 - RGN Timing Requirements for Varying Number of Inputs

It is worth noting however, that in spite of this rapid increase in the number of nodes, the training times still compare favorably with those produced by the recurrent structure.

The slow convergence of recurrent structures together with the computationally expensive nature of their training algorithms presents a serious obstacle. In their comparison of several neural network models for a chemical process, Lambert and Hecht-Nielsen [11] recorded training times of two days on a Sun 4 workstation, in order to sufficiently train an eight node recurrent network.

4.2.4 DISCUSSION

The above evaluation of fully recurrent networks showed that whilst these structures do have the ability to *learn* feedback connections and therefore overcome the limitations associated with the feedforward topology, this additional flexibility does have its drawbacks.

Firstly, the recurrent structure (as per Williams and Zipser) investigated in this chapter does not converge consistently. Although the structure produced an extremely accurate model for a linear system, it failed to capture the dynamic behavior of a more complex, nonlinear system. A second drawback of the recurrent structure are its excessive computational requirements which lead to unacceptably large training times.

The less flexible regular Gaussian network, on the other hand, can not learn the required feedback connections but converges reliably and requires acceptable computation and training times. The RGN structure is therefore more suited for on-line implementation in control structures.

It is worth noting that the excessive training time is a limitation imposed by the current state of technology. The rapid growth in the computing field as well as the development of neural network chips, might soon produce the required hardware to make training and on-line implementation of more complex, recurrent structures feasible.

REFERENCES

1.....Narendra K.S. and Parthasarathy K., "*IDENTIFICATION AND CONTROL OF DYNAMICAL SYSTEMS USING NEURAL NETWORKS*", IEEE Transactions on Neural Networks, Vol. 1, No. 1, 1990, pp. 4-27.

2.....Rumelhart D.E., McClelland J.L. and the PDP research group, "*PARALLEL DISTRIBUTED PROCESSING - EXPLORATION IN THE MICROSTRUCTURE OF COGNITION, Volume 1: Foundations, ch. 8: Learning Internal Representations*", Cambridge, MA: MIT Press, 1986.

3.....Werbos P.J., "*GENERALIZATION OF BACKPROPAGATION WITH APPLICATION TO A RECURRENT GAS MARKET MODEL*", Neural Computation, Vol. 1, 1988, pp. 270-280.

4.....Su H. and McAvoy T.J., "*IDENTIFICATION OF CHEMICAL PROCESSES USING RECURRENT NETWORKS*", ACC 1991, pp. 2314-2319.

5.....Willis M.J., Di Massimo C., Montague G.A., Tham M.T. and Morris A.J., "*ARTIFICIAL NEURAL NETWORKS IN PROCESS ENGINEERING*", IEE Proceedings-D, Vol. 138, No. 3 ,1991 , pp. 256-266.

6.....Willis M.J., Montague G.A., Morris A.J. and Tham M.T., "*ARTIFICIAL NEURAL NETWORKS: - AM PANACEA TO MODELLING PROBLEMS?*", ACC 1991, pp. 2337-2342.

7.....Willis M.J., Montague G.A., Di Massimo C., Tham M.T. and Morris A.J., "*NON-LINEAR PREDICTIVE CONTROL USING OPTIMIZATION TECHNIQUES*", ACC 1991, pp. 2788-2793.

8.....Williams R.J. and Zipser D., "*A LEARNING ALGORITHM FOR CONTINUALLY RUNNING FULLY RECURRENT NETWORKS*", Neural Computation, Vol. 1, 1989.

9.....Tsung F.S., "*LEARNING IN RECURRENT FINITE DIFFERENCE NETWORKS*", Connectionist Models Summer School 1991.

10.....Sun G.Z., Chen H.H. and Lee Y.C., "*A FAST ON-LINE LEARNING ALGORITHM FOR RECURRENT NEURAL NETWORKS*", IJCNN 1991, pp. II13-II18.

11.....Lambert J.M. and Hecht-Nielsen R., "*APPLICATION OF FEEDFORWARD AND RECURRENT NEURAL NETWORKS TO CHEMICAL PLANT PREDICTIVE MODELING*", IJCNN 1991, pp. I373-I378.

5. NEURAL NETWORK BASED CONTROL STRUCTURES

The first two chapters of this thesis introduced the concept of neural networks and suggested the possibility of utilizing them as part of a control structure. In chapters three and four the suitability of several network topologies and training algorithms for such an application were investigated. In this final chapter two model-based control structures, in which the model can be replaced by an artificial neural network, are examined. Although both these structures have been suggested elsewhere [1][2], the original configurations had to be modified or augmented in order to accommodate some of the requirements, such as on-line training, outlined in chapter two.

In order to evaluate the performance of these neural network controllers, their performance is compared to that of a linear model control structure, using a simulation of a nonlinear tank system. The first section of this chapter describes the tank system and highlights the discrepancy between the actual system and a typical linear model derived from step-pertubation data. In the subsequent section the internal model control structure is introduced and the performance of a linear model based controller and neural network model based controller are compared. After this an alternative model based topology, known as neural predictive control, is introduced and evaluated. The final section of this chapter summarizes the advantages and disadvantages of the two neural network approaches and highlights some of the practical issues that need to be addressed before these network based control structures can be employed in industrial systems.

A listing of all software procedures and algorithms used to generate the results in this chapter are included in Appendix II.

5.1 NONLINEAR SYSTEM

One of the primary motivations for the interest in nonlinear, self-adjusting control systems (such as neural network based controllers) is that most real systems exhibit some nonlinear behavior. In many cases this is due to non-ideal factors such as friction or non-laminar flow, which cannot be modelled easily, whilst in other circumstances the nonlinear response is a direct result of the physical laws governing the behavior of the system.

The tank system chosen to evaluate the neural network based control structures, is an example of the latter. As shown in figure 5.1, the process consists of a tank with an outflow at the bottom and an inflow which can be regulated via a control valve. For this study it is assumed that the tank forms part of a larger system and that it is desirable to regulate the level of liquid in the tank.

NONLINEAR TANK SYSTEM

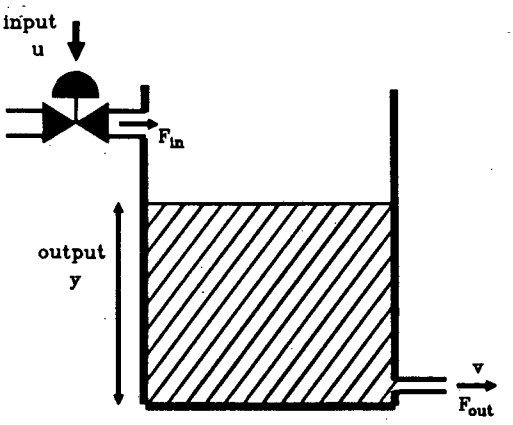


Figure 5.1 - Tank System Schematic

The outflow of liquid in systems such as the above, has a nonlinear dependence on the level in the tank, thus resulting in a nonlinear equation as shown below.

- Denoting the level (or output) as y and the signal to the control valve (or input) as u, the system equation can be derived as follows :

$$\frac{dV(t)}{dt} = F_{in}(t) - F_{out}(t) \quad [i]$$

where V - volume
F_{in} - flow in
F_{out} - flow out
A - cross sect. area of the tank

Assuming that the reaction time of the valve is negligible in comparison to the system response, the inflow is given by

$$F_{in}(t) = k \cdot u(t) \quad [ii]$$ where k - constant

Assuming furthermore that the outflow is homogeneous and laminar and that friction can be ignored, the flow out of the tank is determined as

$$F_{out}(t) = A_{out} \cdot v(t) \quad [iii]$$ where A_{out} - cross sectional area of pipe
v - velocity of liquid

Since $A >> A_{out} \Rightarrow \frac{dy}{dt} << v$ and hence v can be approximated using *Torricelli's* equation [3] i.e.

$$v(t)^2 = 2 \cdot g \cdot y(t) \quad [iv]$$

Substituting [ii],[iii] and [iv] into [i] :

$$\frac{dy(t)}{dt} = \frac{dV(t)}{dt} \cdot \frac{1}{A} = \frac{1}{A} \cdot [k \cdot u(t) - \sqrt{2 \cdot g \cdot y(t)}]$$

$$= c_1 \cdot u(t) - c_2 \cdot \sqrt{y(t)}$$

from which : $sy = c_1 \cdot u(t) - c_2 \cdot \sqrt{y(t)}$ or $y = \frac{1}{s} [c_1 \cdot u(t) - c_2 \cdot \sqrt{y(t)}]$

In the discrete domain, the dynamic eqn. becomes

$$y[n+1] = y[n] + \Delta t \cdot [c_1 \cdot u[n] - c_2 \cdot \sqrt{y[n]}]$$

which is the equation used in all simulations below.

- Since the expression assumes a constant y[n] during Δt, this value has to be kept small to minimize errors and ensure the assumption for [iv].
- The constants c_1 and c_2 were chosen as 0.1 and 0.02 respectively such that for maximum input (u=1.0) and no outflow, the incr. in level would be 0.1 [m/s] and that the maximum level achievable is 25 [m].

Due to the slow response time of the system, a twenty second sampling interval is adequate. The significance of large sampling intervals for neural network based control structures is discussed later. It should be noted that within this twenty second interval the simulation equation is solved on an iterative basis every 0.1 seconds to provide the required accuracy.

Figure 5.2 shows two perspectives of the change of level after one sampling interval (20 seconds) as a function of the previous level and input. Both surfaces clearly indicate the nonlinear nature of the system.

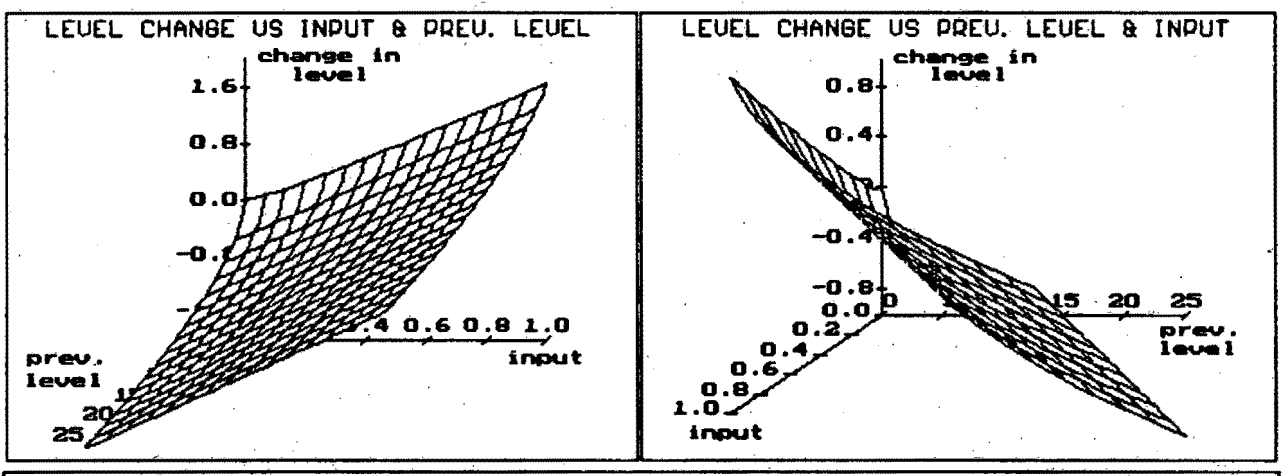


Figure 5.2 - Change in Level vs. Previous Level and Input

Of the many methods that exist to derive a linear model for a system with unknown characteristics, the so-called *step test* approach in which the system response to step perturbations in the input is modelled, is one of the most common. This technique was selected to develop the linear model for the above system and the *step test* results are shown in figure 5.3 below. Although the number of tests and their range might seem minimal, this is common in commercial applications where open loop step tests are often expensive or detrimental to product quality and models frequently have to be developed from even less data.

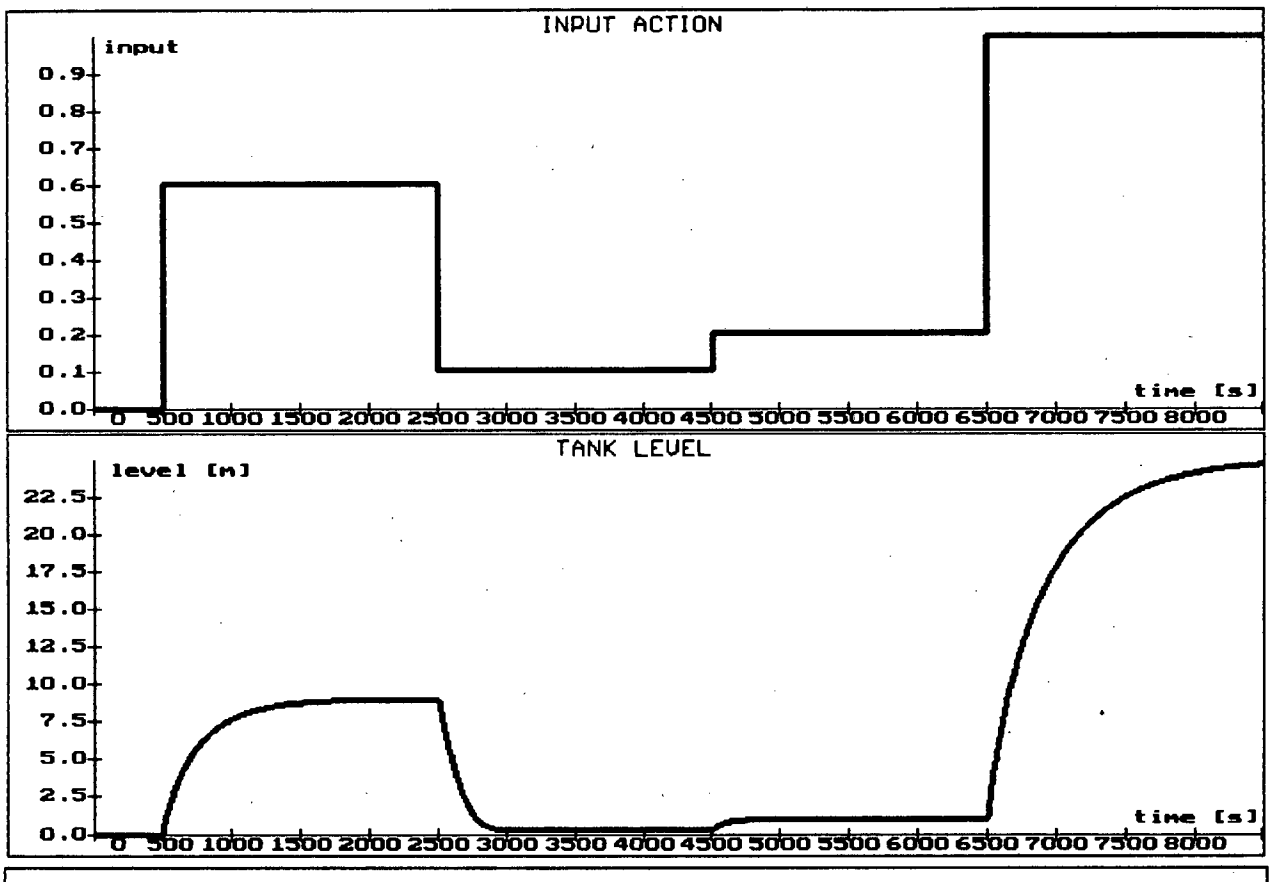


Figure 5.3 - Tank System Step Responses

The nonlinear character of the tank system is reflected in the widely varying gains and time responses. The system does however display a general first order character and normalization of the step responses with respect to the change in input and the final value (shown in figure 5.4) suggest a gain of 15 and a time constant of 250 seconds. It should be noted that the step tests were selected to favour the choice of a low gain and high time constant, to enhance the difference between the linear model and the actual system and thus illustrate the advantages of the neural network approach.

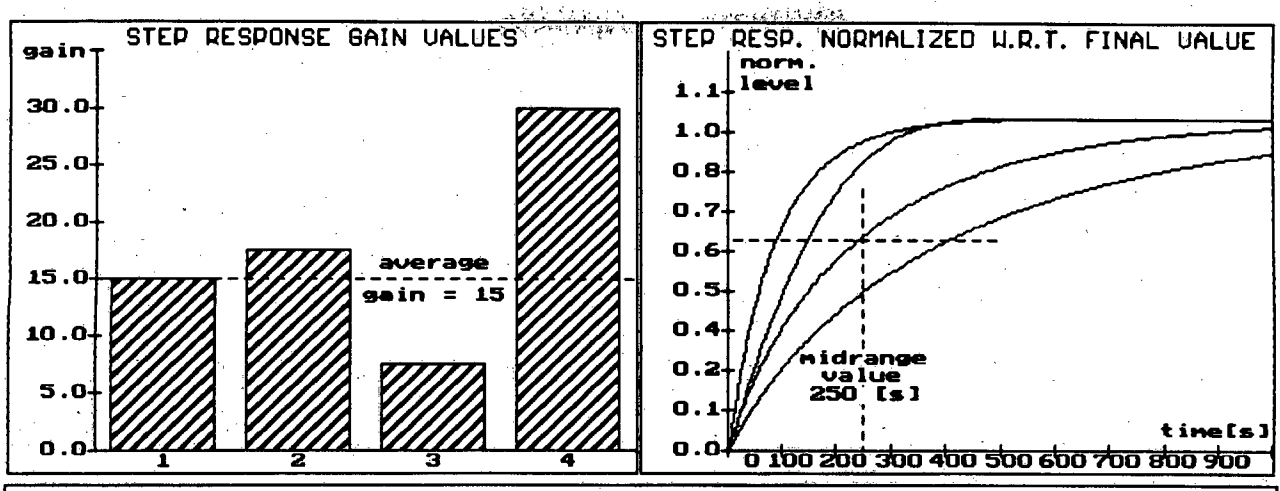


Figure 5.4 - Normalized Step Response

The linear first order model with the above parameters is used throughout the remainder of this chapter in both, the conventional linear control structures and as part of the model/network combination described in the following subsection.

5.2 MODEL/NETWORK COMBINATION

As mentioned in chapter 2, the applications considered in this study are model based control systems in which the linear model may be replaced by a neural network approach. Although it is possible (and common) to utilize a pure network model, the approach selected here is a mathematical model/network combination as shown in figure 5.5 below.

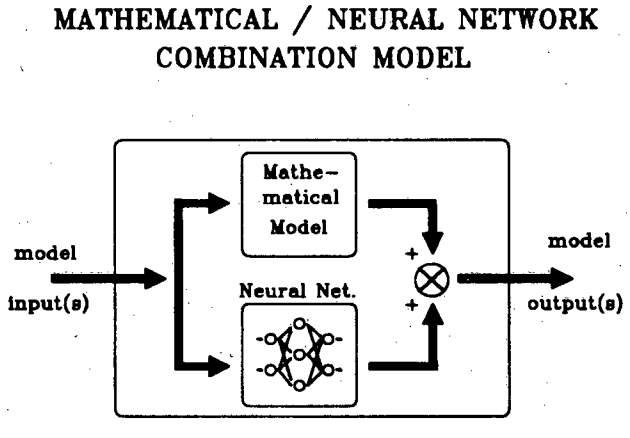


Figure 5.5 - Combination Model

Although the mathematical model shown in the above topology can be any type of equation derived from system data or based on past experience, one of the most obvious choices is a linear model derived from first principles or step test data. The neural network in this structure therefore learns the difference between the mathematical model and the actual system, rather than the entire model. Even though the inclusion of a mathematical model might seem superfluous, this combination enjoys two distinct advantages over the pure network approach:

One of the difficulties with a control structure in which pure neural network models are employed, is the initialization of these networks. The problem can be overcome by attempting to collect adequate system data to train the network off-line, but as shown earlier it is difficult to collect a sufficiently dense set of data to ensure acceptable training. An alternative approach is based on the assumption that the majority of systems can be controlled using linear models, even though the control may be sub-optimal. Based on this, the network can thus be trained prior to use, utilizing data generated by a linear model of the system. Although not ideal, this method is preferable to the former since it allows more uniform training and only requires the subset of data needed to develop the linear model.

Utilizing the above combination, rather than the pure network topology, allows a further simplification since the linear model can now be used as part of the model and the network does not require any prior training. In other words, provided the initial network weights are kept sufficiently small, the model starts as a pure linear model which is then refined and improved by training the network portion on-line.

A further advantage of the combination topology is that the network does not sacrifice any of its accuracy by modelling the general system behavior, which can be captured in the mathematical part of the model. This is particularly pertinent for the regular Gaussian RBF topology which, due to its inflexibility, is not well suited to model linear systems. Regular Gaussian networks therefore produce superior accuracy if used to learn the discrepancy between linear model and system, rather than to model the entire process.

These advantages of the combination model are illustrated in the following subsections, in which the performance of model based control strategies are examined and compared.

5.3 LINEAR AND NON-LINEAR INTERNAL MODEL CONTROL

The renewed interest in this control structure, in which both a forward and inverse model of the process to be controlled are included, can be attributed to the relative ease with which such controllers can be designed as opposed to more conventional approaches. Due to the fact that the principles of this method [4] have since been extended to include non-linear systems [5], it lends itself to the incorporation of neural network based nonlinear models.

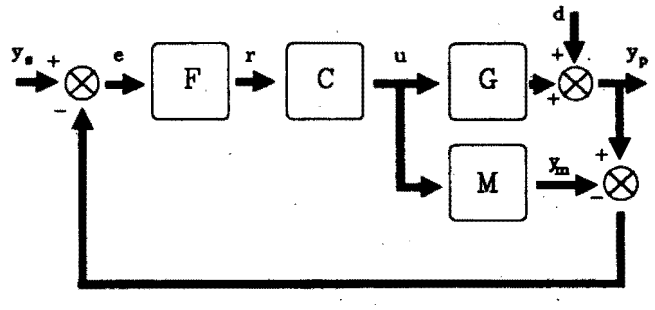
This section starts with an overview of the IMC (internal model control) structure and algorithm. After this the performance of the standard linear IMC control structure for the tank system is evaluated. Since an IMC structure with first order model and filter is equivalent and can be converted to a conventional PI controller with the same functionality, this controller is used for comparison purposes in the remainder of the chapter.

The next subsection then introduces an IMC structure in which the linear models are replaced by the combination model explained in section 5.2 and demonstrates how the neural network portion can be trained in an on-line configuration. The applicability of the control structure is again illustrated using the tank system to allow a comparison with the above PI type control. In the final part of this subsection some of the limitations of the suggested IMC structure are highlighted.

5.3.1 INTERNAL MODEL CONTROL ALGORITHM

Figure 5.6 shows the IMC structure and highlights the key function that both the forward and reverse model play in this control configuration.

INTERNAL MODEL CONTROL STRUCTURE



F – filter C – inverse model G – system M – model

Figure 5.6 - IMC Structure

From the configuration it follows that, if the model is an exact replica of the process (thus

resulting in zero feedback), the system output is the desired (filtered) response to the setpoint since the system dynamics are cancelled by the inverse model. A mathematical analysis of the properties for both linear and nonlinear IMC structure can be found in the work by Morari et al [4][5]. The main results can be summarized as follows:

- For stability and the perfect controller, the model needs to be exact and both the system as well as the controller need to be stable.
- Zero offset for a steady setpoint can be ensured by selecting the controller such that the product of steady state gains of model and controller is unity.
- The filter included in the controller compensates for modelling errors and thus introduces robustness into the IMC structure since it reduces the loop gain.

These factors suggest that although zero offset can be achieved relatively easy using the IMC structure, the exactness of the model plays a key role in determining the quality of the dynamic behavior.

The equations governing the IMC configuration are summarized below:

Using the notations of figure 5.6, the calculations for each sample instance are -

- obtain sample $y_p(t)$ from the system
- calculate model output $y_m(t)$
- using the above together with the setpoint $y_s(t)$ calculate the error e(t) as follows $e(t) = y_s(t) - [y_p(t)-y_m(t)]$
- the error is then fed into the filter to produce r(t) i.e. r(t) = F(e(t))
- the next input u(t) is then calculated using the inverse model C

In linear applications the filter F and inverse model C are often combined to form a single causal block (known as controller).

The above equations are intentionally general since several variations exist between the linear and neural network based structures. These variations are discussed, as they are introduced, in the following subsections in which the performance of linear-model and neural-network-combination-model based IMC structures are compared.

5.3.2 LINEAR IMC

This subsection shows the results obtained using the linear model (derived earlier) and its inverse within the IMC structure shown above. The filter, which presents the desired system response, was chosen as a unity gain, first order system with a time-constant of one hundred seconds.

As mentioned earlier, this particular linear IMC structure can be converted to a conventional PI controller and is therefore representative for conventional well-established linear control methodology. In order to evaluate the performance of this control structure, the system was subjected to a number of step setpoint perturbations. The output together with the setpoint, desired response and input action are shown in figure 5.7.

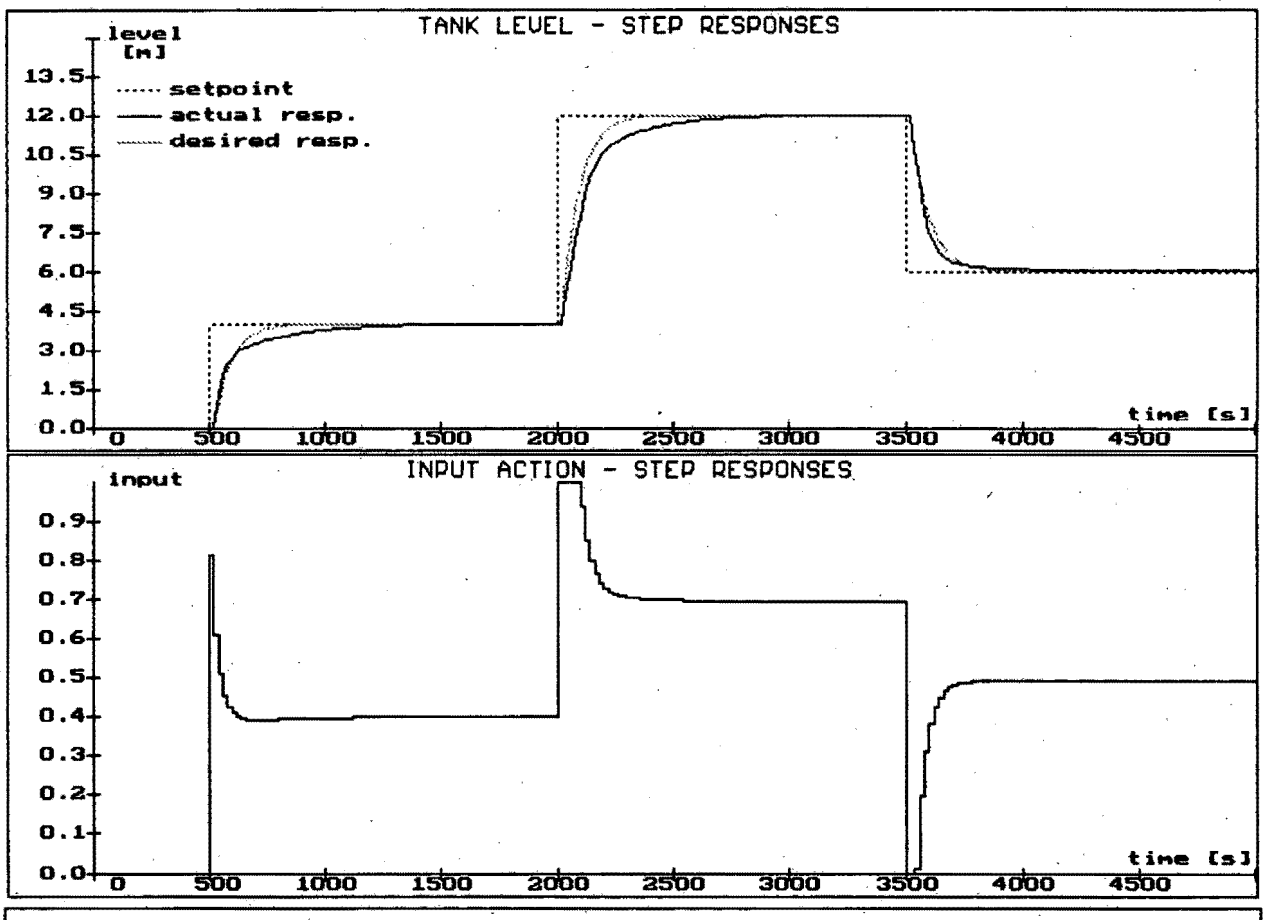


Figure 5.7 - Linear IMC Step Responses

These step tests show that although the system is both stable and exhibits the expected zero offset setpoint tracking, the response for the first two steps is initially sufficiently fast but becomes unacceptably slow in its final approach. Since this poor performance is due to the discrepancy between model and system, the performance of this controller can not be improved by simply increasing the gain. This is also evident from the high initial input spikes, following each of the setpoint step changes, which indicate a high loop gain.

Besides the ability to track and respond sufficiently fast to changes in the setpoint, the capacity to reject disturbances is a further important criterion for any control system. Whilst this ability can be determined analytically as a function of frequency for purely linear systems, this is not possible if the structure includes nonlinear elements. The disturbance rejection capability is therefore evaluated by injecting disturbance signals and observing the response.

Figure 5.8 shows the system response together with the setpoint, disturbance and input signal for constant disturbances and a sinusoidal interference. For convenience the disturbance signal is shown on the tank level graph, centered around the system setpoint, rather than on a separate graph, centered around zero. It should be noted that this line does not present the system response without any control but rather the change in level per sampling interval i.e. a disturbance of 0.5 (shown as 5.5 on the graph) presents a change of level by 0.5m per sampling interval. In other words the effect of the disturbance signal shown is cumulative and the *level plus disturbance* is used in the tank equation, to calculate the subsequent level.

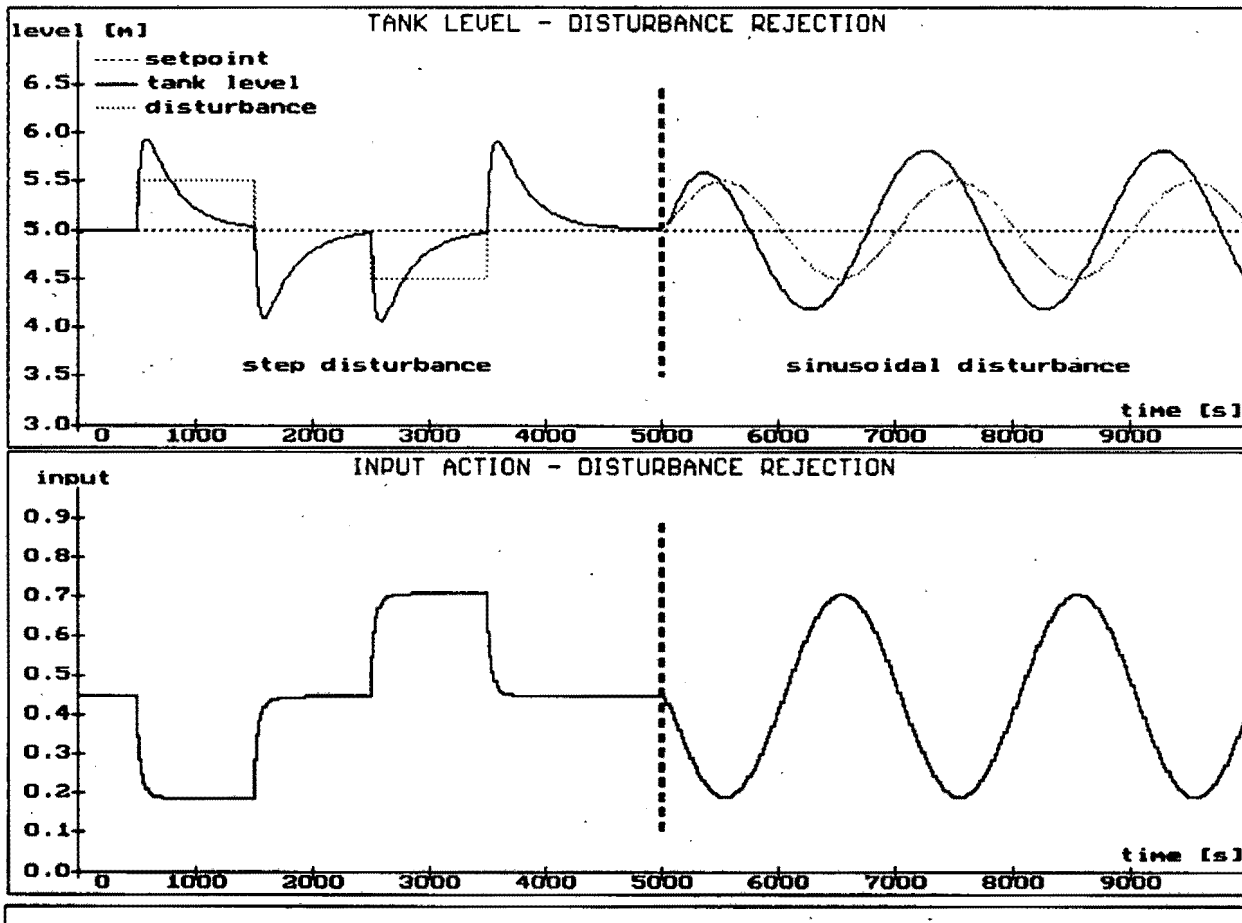


Figure 5.8 - Linear IMC Disturbance Rejection

As with the step responses, the graphs show that the system is stable and does strive to follow the setpoint but again the slow dynamic response results in rather slow and unsatisfactory performance.

In the discussion of the IMC properties, it was mentioned that the quality of the dynamic response is dependent on the exactness of the model. Since a non-optimal, linear representation of the nonlinear system was used in the above simulations, the modelling errors are significant and hence impact negatively on the overall performance. In order to improve the response of this structure a more accurate model is therefore required.

5.3.3 NON-LINEAR NEURAL NETWORK IMC

In the previous subsection the performance of a linear model based IMC controller for a nonlinear process was evaluated, emphasizing that although such a structure is stable and does ensure setpoint tracking, it fails to produce the desired dynamic response. Since this is mainly due to the error between the model and system, a more accurate representation should produce improved results. This, together with the fact that the IMC structure is suitable for nonlinear control structures, motivated Hunt and Sbarbaro [1] to utilize neural networks in place of the forward and inverse model.

One of the problems with the implementation suggested by Hunt and Sbarbaro is that although they use regular Gaussian RBF networks (as recommended in chapter four), they assume that both networks can be trained prior to use and therefore do not implement any on-line optimization. Besides the difficulty of obtaining a sufficiently dense set of training data for a dynamic system (discussed in chapter two), such an implementation deprives a neural network based structure of one of its most significant attractions, namely on-line adaptability. The lack of on-line training thus renders such a system as susceptible to non-optimal parameter choices, due to limited training data, as a linear model. The utilization suggested in this study is a modification of the one proposed by Hunt and Sbarbaro, allowing for on-line training and optimization. This subsection starts with a discussion of the modifications to the standard IMC structure, before showing the responses of a neural network based controller to the same step and disturbance perturbations as the linear structure.

(a) Modifications and Network Training

Using RBF or other feedforward type networks to replace the forward and inverse model requires a modification to the standard IMC structure. Due to the lack of dynamic capability of these networks, they have to be supplied with the required previous input(s) and output(s) of the actual system or else be implemented in an external recursive configuration. Since the latter does not produce satisfactory accuracy as shown in the previous chapter, both networks are supplied with past values from the actual system. For the tank system this implies that both models receive the previous tank level, rather than their past output as one of the inputs. Although this modification is strictly only valid if the model is exact (a requirement for most IMC properties) it still produces a forward and inverse model as required by the IMC structure, thus ensuring a zero offset control law even when the model is not precise.

Since neural networks are not exact mathematical expressions, some residual error between the forward and reverse models is expected, even after exhaustive training and irrespective of network size. In order to eliminate these discrepancies, which oppose one of the key requirements for the IMC structure, Hunt and Sbarbaro employ a successive substitution algorithm [6] to improve the accuracy of the values produced by the inverse model. This recursive algorithm (shown below) can be used to find the inverse at any operating point and although it could replace the inverse model entirely, this model can still help to supply initial estimates, thereby limiting the number of iterations required and saving valuable calculation time. Although other methods such as Newton's Method might converge faster, they are generally not directly applicable since they require differentiation and integration of the model which includes a neural network in the structure suggested here.

Given a function (or model) $f(x)$, the following iterative inversion algorithm can be used to find the value x for which $f(x) = y_{des}$

$$x_{n+1} = x_n + \gamma[y_{des} - f(x_n)]$$

The small gain theorem shows that this algorithm shown in figure 5.9 is stable if

$$\|I\| \, \|I-\gamma f\| \leq 1$$

ITERATIVE INVERSION ALGORITHM
BLOCK DIAGRAM

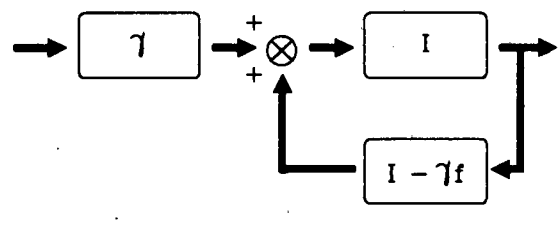


Figure 5.9 - Iterative Inversion Block Diagram

The above algorithm is used to improve the initial estimate provided by the inverse model. Although the iterative process should ideally only be terminated when successive iterations do not change the result significantly, the real-time requirements of a control structure demand that a maximum allowable number of iterations also be imposed. The accuracy of the inverse model, which supplies the initial estimates, therefore plays an important part in this structure.

As mentioned previously, Hunt and Sbabaro suggest off-line training of both the forward and inverse model. In their configuration the forward model is trained using sampled system data and although the same data could be used to teach the inverse, it is more appropriate to generate its training data using the already converged forward model since this helps to ensure that the two models are exact inverses of each other. This training is performed using both specialized and general learning structures adopted from Psaltis et al [7]. Despite the fact that these training methods facilitate adequate training of the inverse, they do rely on the assumption that the forward model has been trained sufficiently which implies that sufficient training data could be collected prior to training. Due to the fact that it is seldom possible to obtain such a sufficient set of

training data from any dynamic system and since such a pre-trained structure furthermore restrains the flexibility of a neural network configuration, a more practical solution with on-line learning is suggested in this application. Rather than attempting to collect an exhaustive set of training data, the system is initially controlled using linear models, which can be derived from a few step tests as shown previously. Once controlled, both models can then be trained on-line to optimize the controller. The initial linear models can be obtained by training two pure network models prior to using them, a method employed successfully for the control of a laboratory tank system [8], or more efficiently by using the combination type model described in section 5.2, in which the linear model is used together with a regular Gaussian RBF network, the weights of which are initially negligible.

Figure 5.10 below shows the step responses of the tank system using an IMC control structure in which both the forward and reverse combination model consist of the linear models, used previously, together with untrained 100 node RGN networks.

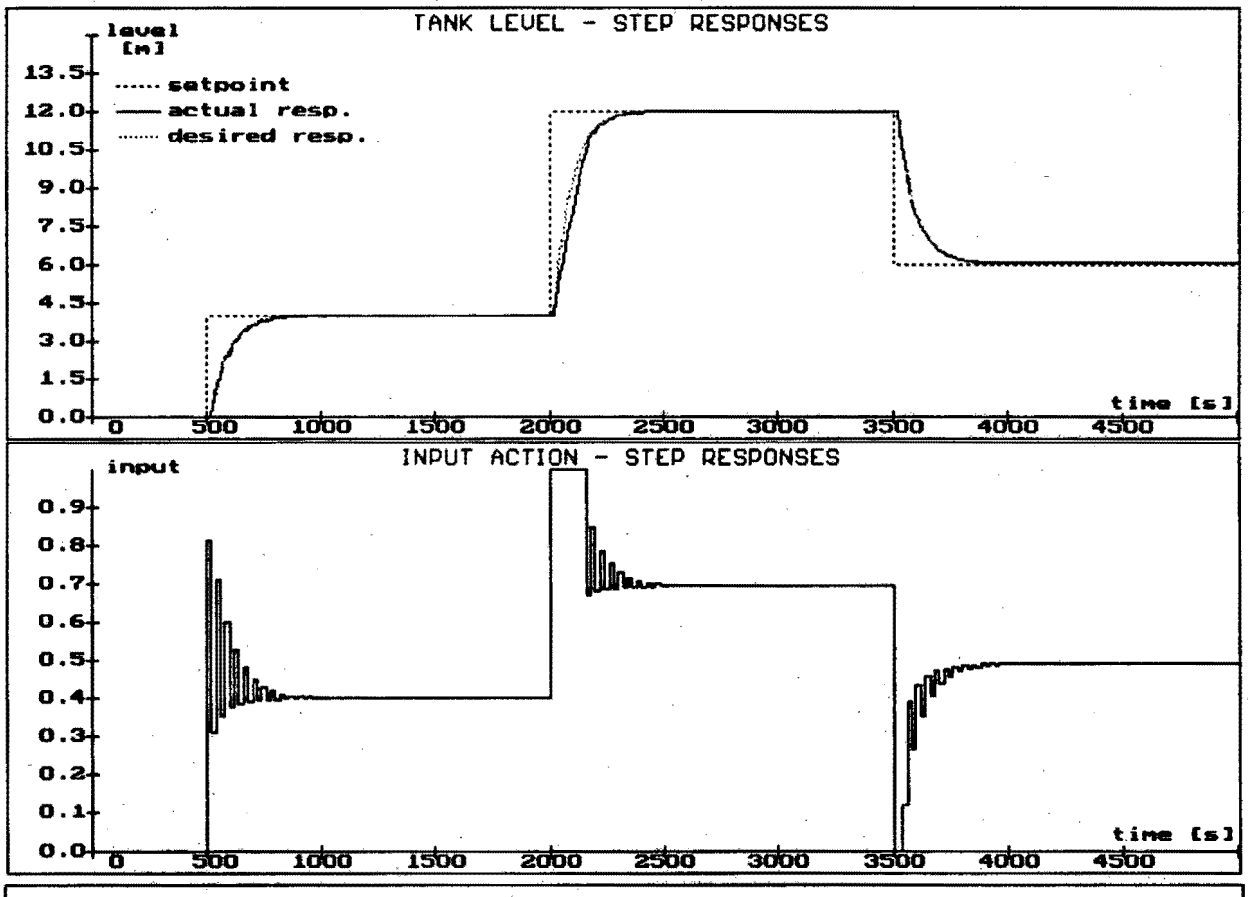


Figure 5.10 - Neural Network IMC before Training

Although the response time of the system has improved significantly from that of the standard linear IMC structure, the input action shows oscillations unacceptable for most practical applications. These oscillations are due to the discrepancy between the untrained model (which is identical to the linear model used earlier) and the actual system. Since the past system output is used as input in both models, these errors are fed

back and effect the output of these blocks in the next iteration. This direct feedback thus ensures that the system does follow the desired response but due to the large discrepancy between model and system, this response can only be achieved using unacceptable input modulation. In order to eliminate the above problem it is necessary to improve the accuracy of both models, which can be achieved by further on-line training. Since the input signals shown above are undesirable for any real system, the controller first has to be detuned for initial training. This detuning can be realized by increasing the filter time-constant or using a filtered version of the setpoint to introduce additional robustness. In this application the latter option was selected and for simplicity the desired output, which is a filtered version of the setpoint, was utilized. The response of this detuned, untrained neural network IMC controller is shown in figure 5.11 below.

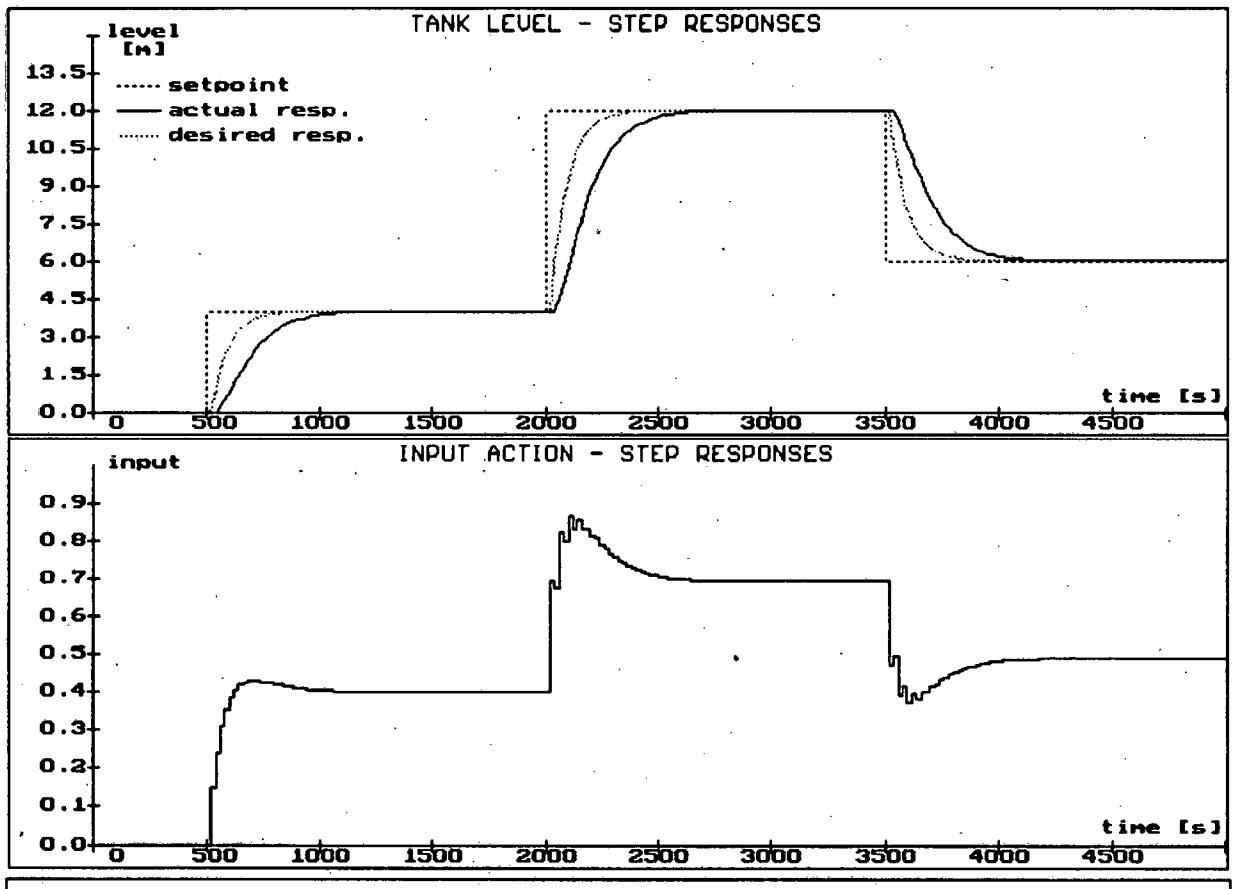


Figure 5.11 - Modified Neural Network IMC before Training

As shown, the system has lost some of its responsiveness but is more robust in terms of discrepancies between system and model. The input signal generated with this dampened controller is more acceptable and although the response is slower than before, it still compares favorably with that of the standard linear IMC controller. The above structure is therefore acceptable as an initial controller and can be used in this form for further training. Once the initial control law has been configured as above, the IMC structure can be enhanced by training the networks contained in both models, thus providing an on-line optimization method which caters for both imminent inconsistencies between the linear models and the system, as well as future changes in the system's behavior.

Whilst the training samples for the forward model are obtained directly as a result of sampling the system response, the improved value obtained from the iterative inversion algorithm provides the desired output for the inverse. Since there is generally a delay between generating the current input and taking the next sample in sampled control systems, this time can be utilized to perform the network training. Due to the fact that training is performed on-line, successive training samples may vary little when compared with off-line training, where the samples can be selected at random. Although a sufficiently small training rate is generally sufficient, more elaborate precautions such as limiting the number of training iterations per setpoint interval or complementing the training with off-line training methods between successive samples, may be required in some cases. The data for such training could be selected at random from a FIFO list which is updated regularly to ensure that slow system changes are captured. Such intricate methods are however only required for systems in which the setpoint is varied infrequently, but uniform training is still required. Other systems might only operate around a fixed setpoint and for such processes adequate training in that operating region is sufficient.

In order to illustrate the learning capability of this structure the system chosen for this example was subjected to setpoint step changes at regular intervals and hence none of the above training enhancing methods were required. Figure 5.12 below shows the evolution of the neural network portion of the forward model in this training.

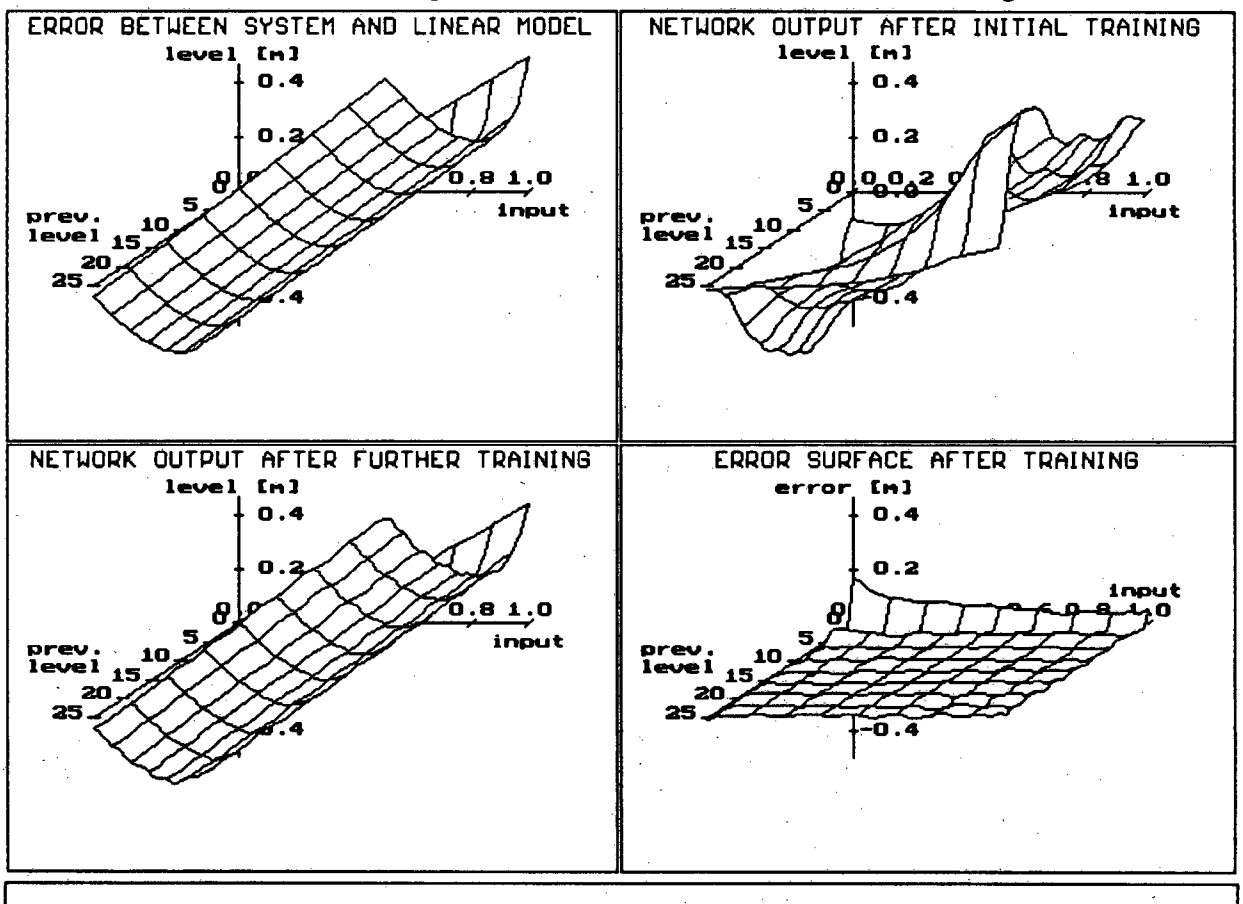


Figure 5.12 - Forward Model Network Training

The final error surface highlights to what extent the inclusion of the network reduces the error between model and system. Due to the fact that the setpoint remains constant between the step changes the training is localized, as is evident from the network surface after some initial training.

Since the iterative equation used in the simulation can not be inverted, the development of the inverse model and network is not shown. The network was however trained with the same localized data and developed in a similar fashion. It should be noted that although the clamping of the input could be incorporated into the inverse network (i.e. it could be trained to produce clamped values), such abrupt discontinuities usually compromise the accuracy of networks near these boundaries and hence the network was trained using unbounded values.

Once sufficiently trained, the additional robustness introduced previously (such as the additional setpoint filtering in this example) can be removed to enhance the system performance.

(b) Results for Neural Network IMC Controller

The response of a trained IMC structure without additional filtering is shown in figure 5.13 below.

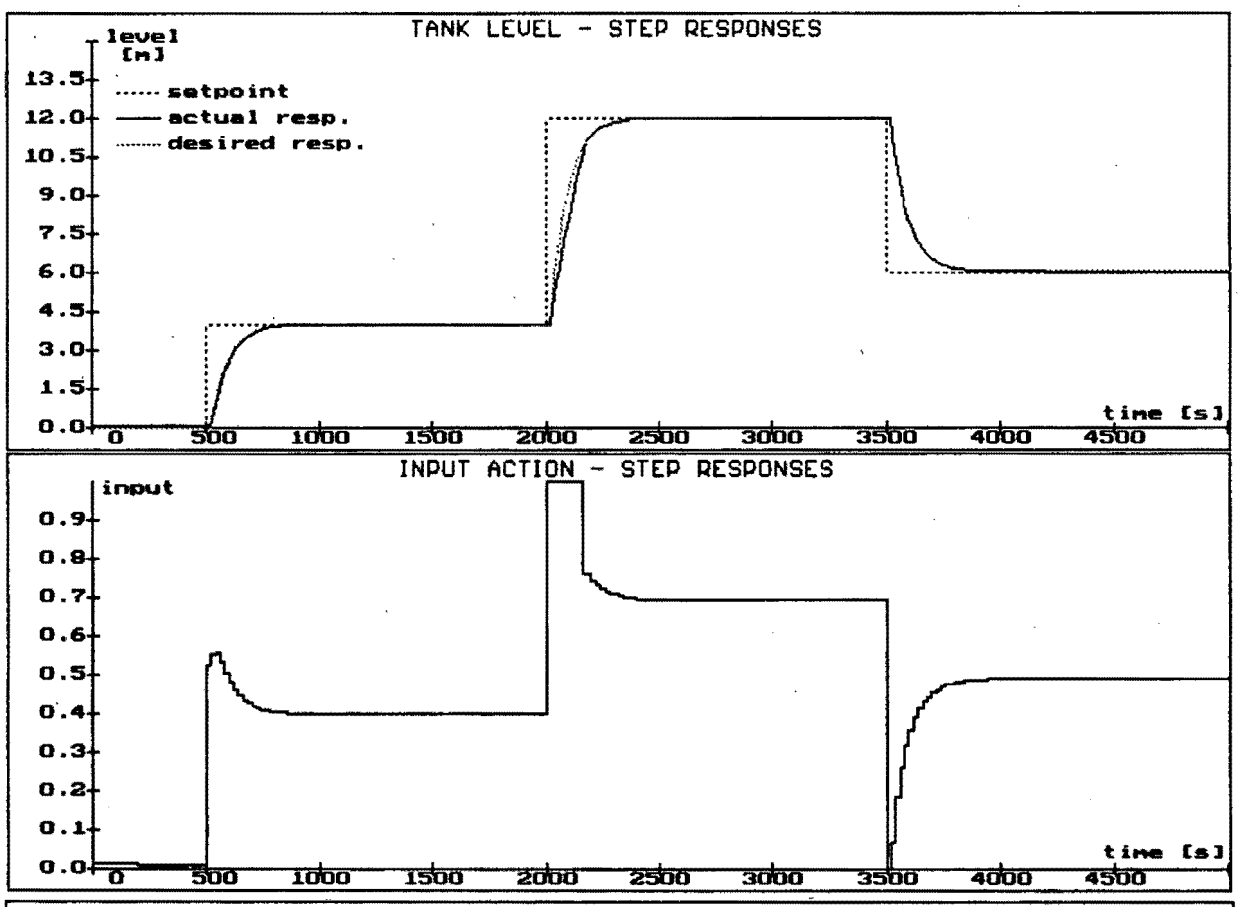


Figure 5.13 - Trained Neural Network IMC Step Responses

The system behavior is in most cases indistinguishable from the desired response and uses a realistic and acceptable input signal. In addition to the superior performance of this structure for the above step tests, it also shows a significant improvement when subjected to the same disturbances used previously, as illustrated in figure 5.14.

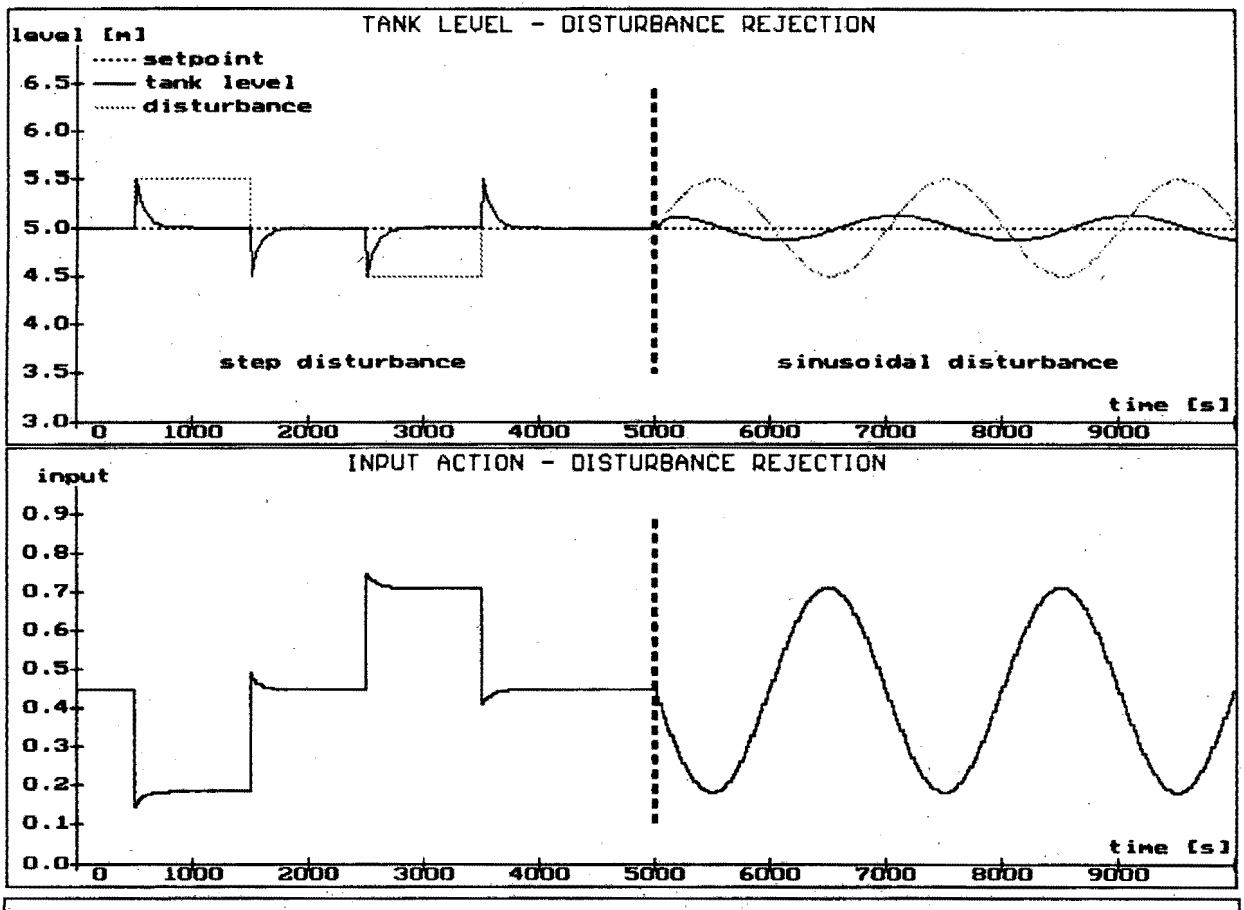


Figure 5.14 - Trained Neural Network IMC Disturbance Rejection

As shown, the controller significantly reduces the magnitude as well as the time taken to recover from the step disturbances, whilst also reducing the amplitude of the sinusoidal disturbance to a far more acceptable level than the previous linear IMC structure.

It is important to note that the improved response of this structure is not due to the enhanced accuracy of the models but rather the direct feedback of the system output to these models. The more accurate models are however essential in this modified structure, to produce a stable and robust controller which generates acceptable input requirements.

5.3.4 LIMITATIONS DUE TO INVERSE MODEL

Despite the encouraging results illustrated above, the need for an inverse model in this approach presents a serious limitation since many systems can not simply be inverted. In their discussion of the IMC structure for linear systems Morari and Garcia [4] exclude all

time delays and zeros outside the unit circle since these would require prediction and result in unstable poles respectively, if they were to be inverted. Whilst such a distinction is feasible in an analytical design, the numerical and neural network inversion operate directly on the model and do therefore not distinguish between feasible and non-invertible components.

The problem becomes more apparent for multi-variable systems for which even the analytical inversion of linear systems is often non-trivial. In their treatment of nonlinear IMC structures Economou and Morari [4] note that the analytical construction of nonlinear inverses uses higher order derivatives thus making it too sensitive to noise and other errors to be used in practical cases and therefore suggest that only an iterative numerical inversion technique is used.

Morari and Economou furthermore note that no guidelines exist for the design of the filter and that it is to be expected that nonlinear filters could lead to distinct advantages. Since the filter together with the inverse model represent the *controller* portion in the IMC structure, the uncertainties in the design of both these blocks for nonlinear systems suggests that an alternative approach may be more appropriate.

The neural predictive control structure introduced in the next subsection is such an approach in which the filter and inverse are replaced by a more manageable, nonlinear controller.

5.4 NEURAL NETWORK PREDICTIVE CONTROL

This algorithm, suggested by Willis et al [2], is based on the optimal control philosophy where a cost function is minimized in order to determine the most effective input. Since a system model is required to predict the output(s) for various inputs in order to formulate the cost function, this structure again lends itself towards the incorporation of a nonlinear and in particular a neural network model. The subsection starts with an introduction of the NPC control structure which is then implemented and tested using the nonlinear tank system.

5.4.1 NEURAL PREDICTIVE CONTROL ALGORITHM

Despite the fact that this algorithm does not fit the standard block diagram structure, it can be represented as shown in figure 5.15.

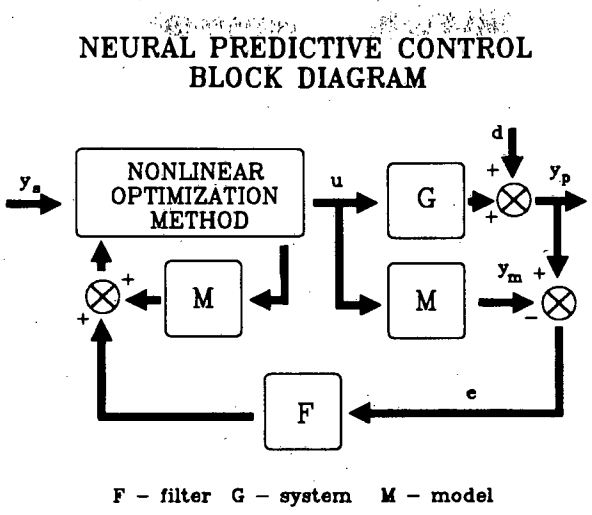


Figure 5.15 - NPC Structure

Although shown in two places, the model block M is in effect one and the same forward model which is used to calculate the error as well as to generate the cost function. As mentioned earlier, the controller implements the well-known predictive control strategy for which the future deviations between system output and setpoint are minimized. In order to achieve this, the squares of these differences are collected as a function of the input(s) to form the so-called cost function which can then be minimized to find the optimal input or input sequence. One significant variation from more common predictive control structures is the use of a neural network based non-linear model to perform the predictions required for the cost function. Unlike the standard approaches the minimum of the cost function can therefore not be found analytically and a numerical optimization method has to be employed. This optimization is indicated as a separate block, receiving the desired output(s) [y_s] and using the model [M] to produce the next input(s) [u] in the above diagram. As in the IMC structure, the function of the feedback loop with filter [F] is to ensure setpoint tracking in the presence of model-system mismatches or disturbances.

The calculations to be performed for the NPC algorithm at each sampling instance are summarized below:

For completeness the steps below show the extensions for a multivariable system:

(i) Take samples of system output(s) $y_p(t)$.

(ii) Generate model predictions $y_m(t)$ using the current input(s) u(t) and previous plant output(s) $y_p(t-T_s)$.

(iii) Calculate the error(s) $e(t) = y_p(t) - y_m(t)$ and filter if required.

(iv) Using the nonlinear optimization algorithm find the minimum of the following cost function

$$J = \sum_{i=1}^{n=N_{1,i}} \{ \sum^{N_1 N_{2,i}} [y_{si}(t+n\cdot Ts) - y_{mi}(t+n\cdot Ts)]^2 + \sum_{k=0}^{N_{u,i}} [\mu_i \cdot \Delta u(t+i_k)] \}$$

where : N_1 - the number of control loops (controlled variables)
$N_{1,i}$ - the minimum output prediction horizon for each loop
$N_{2,i}$ - the maximum output prediction horizon for each loop
$N_{u,i}$ - the control horizon for each loop
μ_i - weighting to penalize excessive input changes for each input
y_s - desired output(s)
y_m - predicted output(s) generated by the model and corrected using the error from (iii)

(v) Implement the first of the calculated input(s) and return to (i).

Although the controller structure provides the facility to calculate several future input values, only the first is used in a *receding horizon* fashion to avoid errors due to long-term extrapolation.

As mentioned earlier, a nonlinear optimization algorithm has to be used to find the minimum of the cost function, since the predictions used are generated by a nonlinear neural network based model. Both, the optimization routine as well as the network configuration and training are therefore vital parts of this control structure.

(a) Nonlinear Optimization Algorithm

Although very similar to the problem of finding the best set of weights for a neural network, the optimization problem considered here is limited to fewer dimensions (the number of input moves for all loops) and hence more computationally intensive algorithms can and must be employed to find the minimum with the required accuracy in the short period available for this calculation. As with network training algorithms however, these optimization methods perform a search in the multidimensional variable space and can be classified into gradient and gradient-free methods.

Whilst the use of the actual or estimated derivative often accelerates the convergence, these methods are not feasible due to the presence of measurement noise and model uncertainties which might produce discontinuities in the objective functions. In this application the so-called Nelder and Mead algorithm [9] was selected for its robustness and because it is a self-contained method which makes no assumptions about the shape of the surface it has to minimize. Other non-gradient techniques, particularly Powell's method [10] may produce more efficient searches and have also been implemented successfully [11]. The significance of the convergence speed in this type of application is discussed later in the chapter.

The Nelder-Mead optimization algorithm accomplishes its search in the n-dimensional space by guiding a (n+1) simplex using contraction, reflection and expansion in order to find the minimum. In this application the previous optimal input value is supplied as the initial estimate and thus forms one corner of the starting simplex. To ensure that the minimum found is global, the algorithm is restarted from this value and must re-converge to the same neighborhood for the algorithm to terminate. Some of the other more specific tuning choices for its application in a control structure are discussed with the program listing in Appendix II.

(b) Neural Network Implementation and Training

In their applications Morris et al [2],[10] use *feedforward networks with dynamic nodes* as introduced in the previous chapter and utilize the chemotaxis algorithm for training. Although feasible, this again implies that the networks have to be trained off-line prior to their implementation. In an attempt to overcome this limitation, the combination type model with a regular Gaussian RBF network used in the nonlinear IMC structure, was employed in this structure. As before, the linear model derived from step test results can be employed immediately, together with an untrained network to produce an initial controller for the plant. The network is again evolved during on-line operation, thus learning to compensate for the discrepancy between the linear model and the system. Most precautions mentioned earlier for the on-line training of networks in the IMC structure also apply to this configuration.

One significant difference from the IMC structure is that a number of future predictions may be required in this configuration. This implies that previous model outputs have to be re-used, leading to an external recursive topology which might impact negatively on the accuracy, as shown in the previous chapter. Since other dynamic network structures often fail to converge and/or are not suited for on-line training however, the regular Gaussian RBF topology still presents the most reliable option.

In their investigation Morris et al [10] suggest an alternative training method in order to improve the long-term prediction accuracy for feedforward networks. Rather than training the network for one-step ahead prediction, they allow the network to generate the required number of iterations before updating the weights. Although this approach might improve the accuracy in a pure network model structure, it does not train the network for the on-line requirements where the control horizon is generally less than the prediction horizon and inputs are applied in a receding horizon fashion. In other words, whilst the optimization routine employs the model to predict a number of steps ahead with a constant input, the actual inputs to the plant vary for each sampling instance and are therefore not a true reflection of the prediction requirements. Despite this, the method has merit and could be included as additional optimization technique between sampling intervals if required.

Another noteworthy difference from the IMC application is that in this topology the network can be trained on-line without effecting the controller performance. In the IMC topology the desired output of the inverse model is generated by using the forward model in an iterative algorithm. To learn, both these models therefore have to utilize their network portion whilst being used to control the plant. Since this structure only uses a forward model, the system can be operated using only the linear part of the model whilst the network part is being trained in a parallel, non-contributing configuration.

Although this discussion does not exhaust all aspects of the NPC structure it provides the framework for its implementation, which is illustrated in the following subsection.

5.4.2 NPC RESULTS FOR TANK SYSTEM

This subsection shows the results of an implementation of the NPC control algorithm for the nonlinear tank system. All the results presented were obtained using N1=N2=5 (smaller N1 produce more input action but do not improve the control), Nu=1 and a first order filter with a time-constant of 60 seconds.

As for the IMC approach, the linear model was first implemented with an untrained 100 node regular Gaussian RBF network. The response of this structure to the step perturbations is shown in figure 5.16.

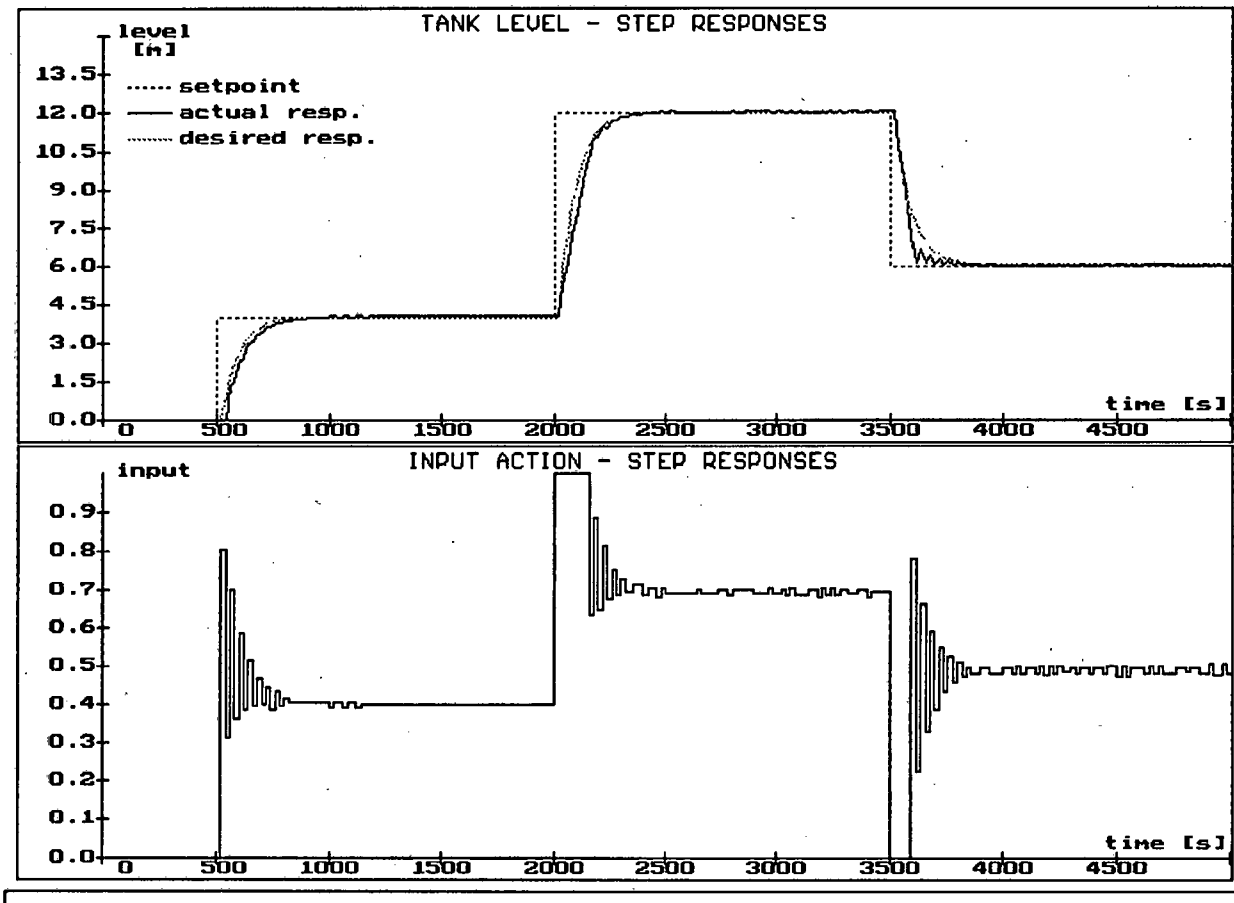


Figure 5.16 - Untrained NPC Step Responses

From this response it is evident that the discrepancies between model and system cause unacceptable modulation of the system input. The system therefore has to be detuned to allow satisfactory operation until the network has been trained sufficiently.

This detuning can be accomplished in several ways in the NPC structure. Since the oscillations are due to large modelling errors, one solution would be to increase the filter time-constant. The use of a filtered setpoint as for the IMC structure presents another correction. Since the NPC algorithm does however also include a term to penalize excessive control moves such as shown in figure 5.16, the most obvious solution is to increase the weighting of this factor. Figure 5.17 shows the response of the identical untrained system in which the weighting of this factor has been increased from 0 to 100.

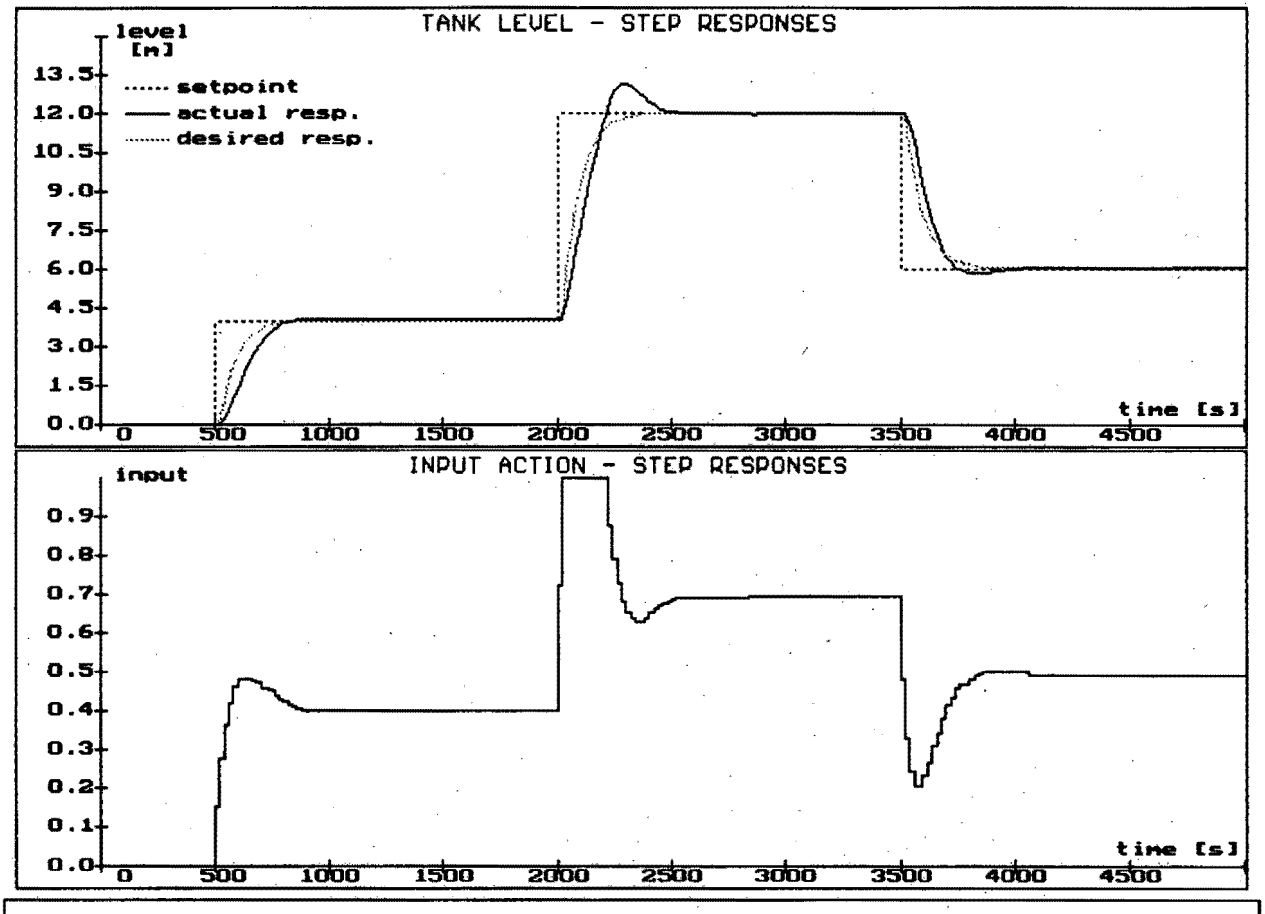


Figure 5.17 - Modified Untrained NPC Step Responses

As shown, the system reacts somewhat slower than before but utilizes reasonable input action to achieve the desired output. The system is now controllable and can be used in this configuration, whilst the network portion of the model is trained. As mentioned earlier, one of the advantages that the NPC structure enjoys over the IMC topology is that the network can be trained without being used in the model i.e. it can be trained separately without impacting on the system performance.

In this example the training of the network portion of the model was achieved by performing steps of random magnitude as in the IMC application. The progression of the network is therefore very similar to that shown previously and is hence not repeated here. As the model accuracy improves due to the network's contribution, the penalization of excessive input moves can be reduced. Figure 5.18 below shows the responses of the NPC structure with a fully trained network and no input penalization.

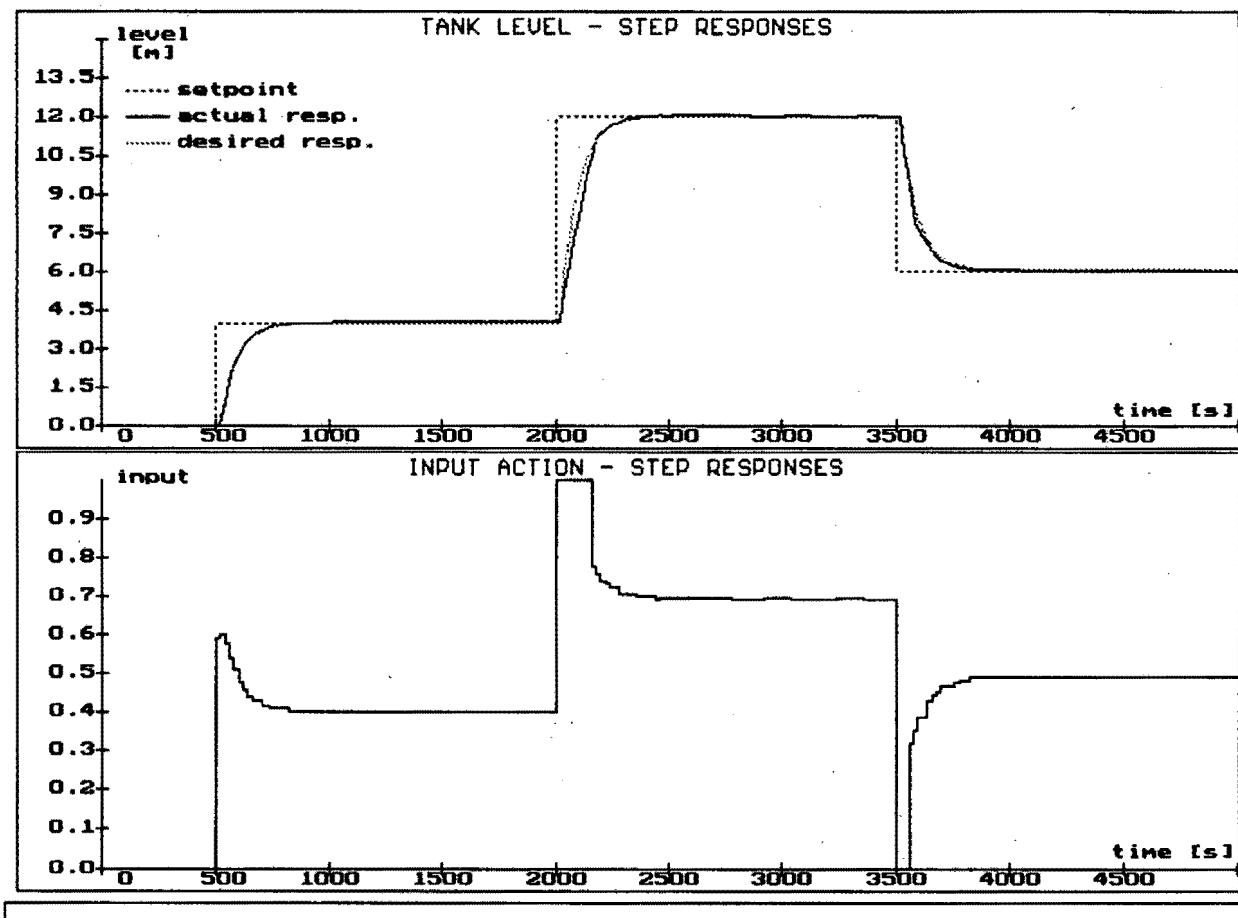


Figure 5.18 - Trained NPC Step Responses

As shown, the control structure achieves the desired response times (taken from the IMC investigation) with moderate input modulation and no oscillations.

The trained NPC configuration was also subjected to the disturbance used to evaluate the two previous controllers and the resulting input and output patterns are shown in figure 5.19 overleaf. The controller responds quickly and effectively to the step disturbances and also attenuates the sinusoidal interference significantly.

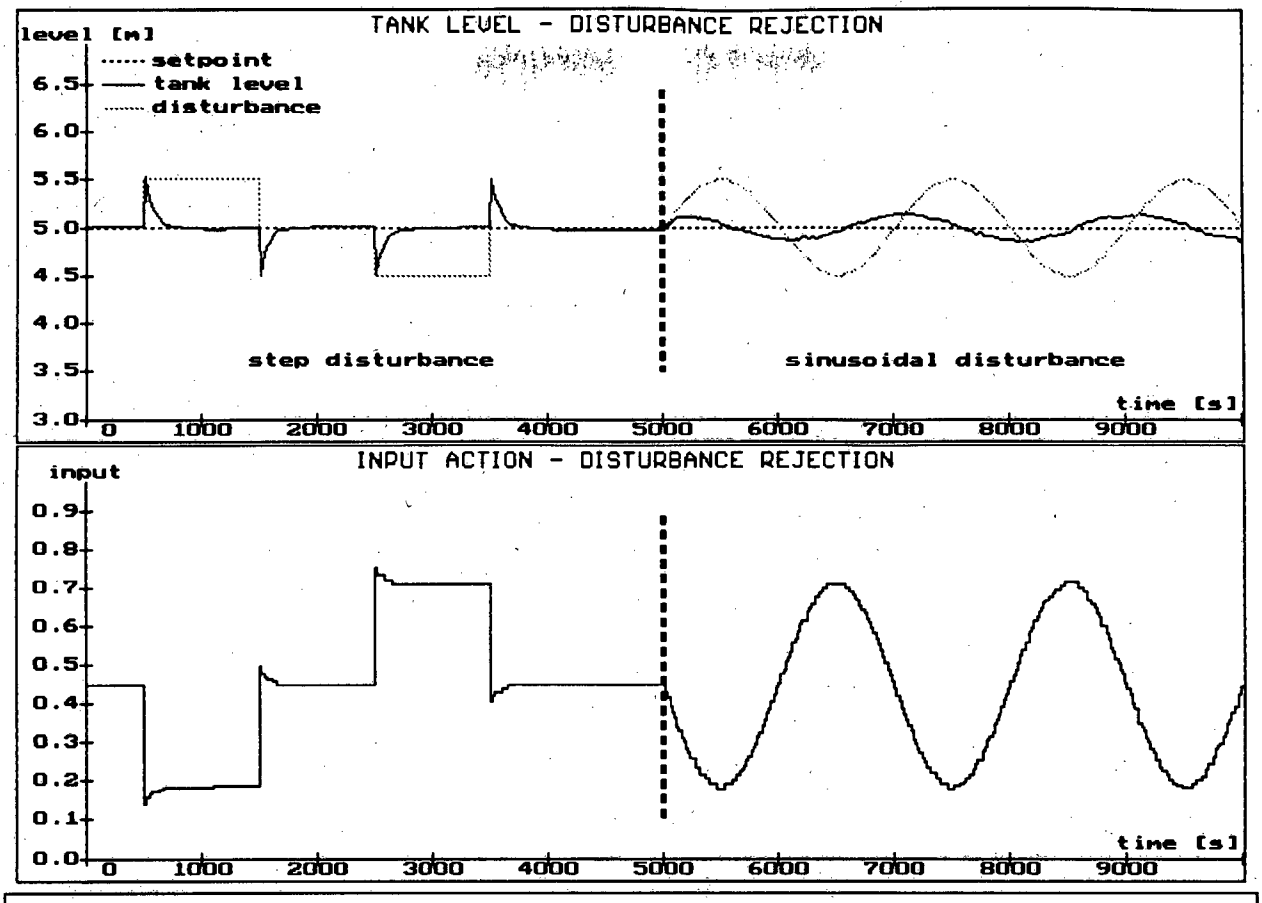


Figure 5.19 - Trained NPC Disturbance Rejection

The NPC control structure introduced in this subsection is not only a feasible alternative to the neural network based IMC approach, but also includes several attractive features not available in the latter. The final part of this chapter highlights some of the differences between the two methods and also raises some general concerns about the viability of these structures for practical applications.

5.5 DISCUSSION

In this chapter two neural network based control structures were introduced and applied to a simulated nonlinear tank system. Both methods use combination type models and can be initiated in a suboptimal or detuned structure with only the mathematical part of the model complete, thus allowing on-line training of the neural network portion in a controlled system.

Although the application, chosen to illustrate the capability of these methods, could possibly be controlled adequately by a carefully designed linear controller, the example highlights typical practical constraints when a linear model is developed from insufficient or non-

representative data and the design is kept conservative, resulting in a sub-optimal system. In both methods the neural network portion of the model is trained on-line in order to correct modelling discrepancies and once trained allows far superior control than achievable if only the linear model is employed. Both techniques are thus *learning* control laws in which the system is activated using a priori knowledge and is then optimized in a controlled state as the model(s) *learn* and become more exact. The advantages of such control laws for systems with nonlinear characteristics or slow changing behavior are evident.

This section starts with a brief discussion of the advantages and disadvantages of the two approaches and then highlights some of the common problems that need to be addressed if such structures are to be implemented in practical systems.

5.5.1 A COMPARISON OF THE NEURAL NETWORK IMC AND NPC APPROACHES

Despite the fact that there is little difference between the performance of the final control laws generated by either method, the two approaches do differ in certain aspects.

One of the possible problem areas of the IMC structure is the need for the inverse model, which may not exist or be unstable for certain systems. Although the inversion can in such cases be performed, using only the iterative inversion algorithm, such a search does in fact still attempt to find the inverse of the system at that point. The optimization algorithm of the NPC structure on the other hand, does not endeavor to find the inverse of the system but rather the optimum input to minimize the cost function, which in turn only relies on forward predictions and can be formulated in such a way that possible obstacles are avoided. Due to the above, the NPC algorithm can be extended to multivariable systems and has been implemented successfully on a simulation of a distillation column [10], whilst the inversion of multivariable systems required for the IMC structure is often non-trivial.

One drawback of the NPC structure is that the network has to be implemented in an external recursive connection which is not ideal for feedforward networks such as the regular Gaussian topology. The problem is lessened by the use of a combination model and on-line training but could still impact on the performance if a large number of iterations are required. Since only one-step ahead predictions are employed in the IMC structure, this difficulty does not arise.

A difficulty shared by both methods is the exponential growth of regular Gaussian networks as a function of inputs, which suggests that despite inferior modelling and a lack of on-line learning capacity other network topologies might have to be considered for higher order and multivariable systems.

A further common problem, linked to the network sizes, is the time required to generate the next input, once the outputs have been sampled. Although this problem is more pronounced in the NPC structure, which generally requires more network evaluations to

find the next input, even the network IMC topology is not suited for high speed applications as shown in the following subsection.

5.5.2 COMPUTATIONAL POWER AND TIMING CONSIDERATIONS

For any sampled system, the time between sampling the output(s) and generating the next input(s) has to be kept to a minimum and should be considerably less than the sampling period as illustrated in figure 5.20.

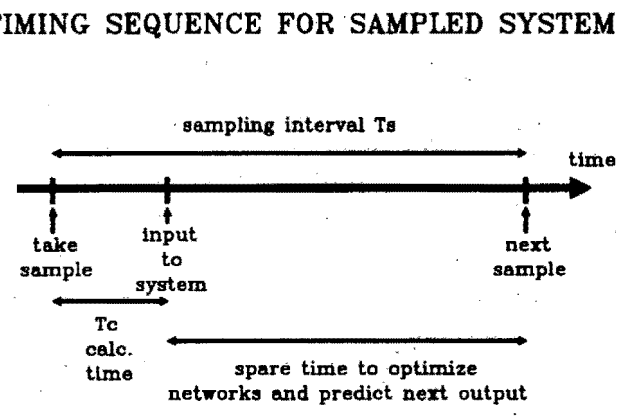


Figure 5.20 - Timing Diagram

Whilst this requirement is easily met for control laws in which the next input(s) are calculated using some pre-determined closed form mathematical equation, this is not the case for the neural network controllers suggested in this study. Both the neural network IMC and NPC methods utilize an optimization algorithm in order to determine the next input(s) and hence the calculation time can easily become excessive and impact on the overall system performance.

The effect of an increased calculation time is shown in figure 5.21 which illustrates the response of the trained NPC structure to the same step perturbation for various calculation times. These responses show the impact on both the system response and input once the calculation time becomes comparable to the sampling period.

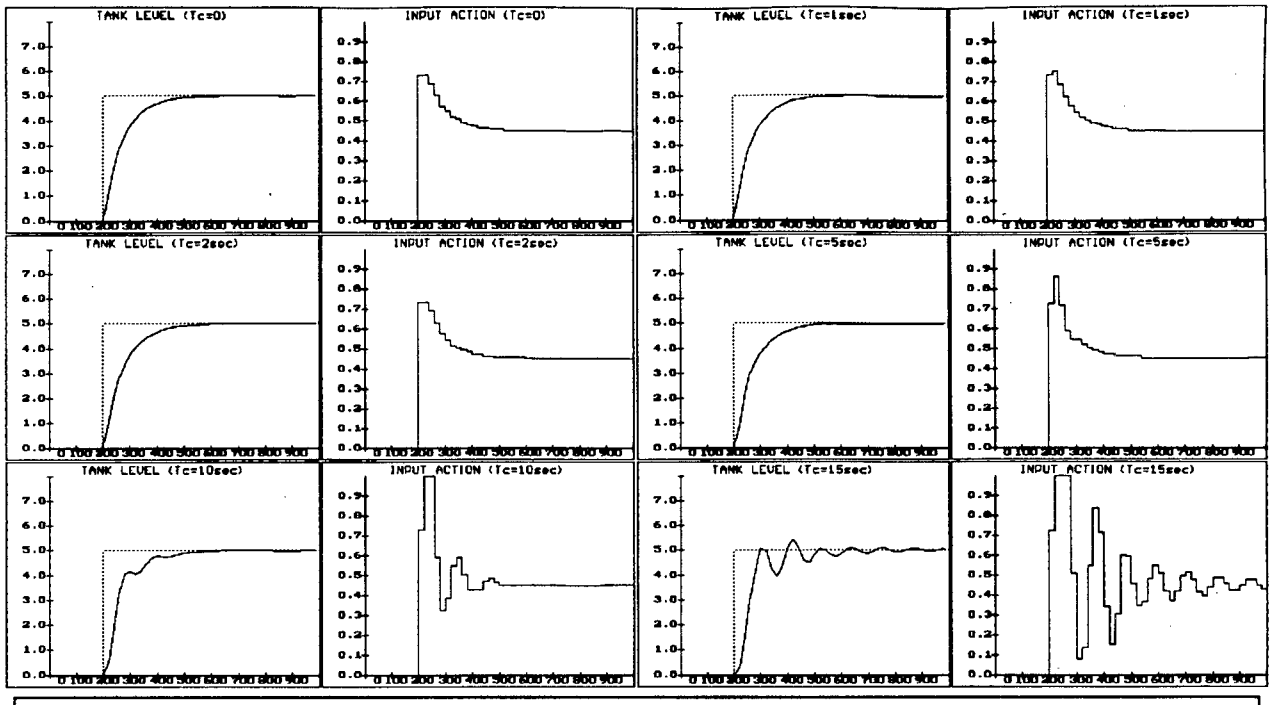


Figure 5.21 - Effect of Calculation Time

The aim of the results shown for both structures in the previous sections was to illustrate their ability to control the system rather than to perform a real-time simulation and hence the calculation time was not incorporated i.e. a calculation time of zero was assumed in both cases. Due to the large sampling period (20 seconds) the average calculation times of 180 and 580 milli seconds measured on a 33 MHz 486DX personal computer for IMC and NPC respectively, do seem negligibly small. Individual calculation times might nevertheless be of the order of seconds, indicating that these algorithms do require a certain minimum amount of computation time and are therefore not suitable for high speed applications in their current format.

Since the high calculation times are a direct consequence of the number of model evaluations required by the optimization methods, a reduction in either the computation time of the model or the number of evaluations is beneficial. Although the latter can be achieved by increasing the tolerance and reducing the number of iterations in the search algorithms, there exists a limit beyond which the decrease in accuracy impacts negatively on the controller performance.

Whilst a further reduction in calculation time can be achieved by optimizing the code and/or algorithms used as well as by increasing the computing power of the platform on which these are realized, the implementation of neural network based control structures on a large scale might be delayed until neural network chips become available to produce the required speed at a reasonable price.

5.5.3 STABILITY AND ROBUSTNESS

One of the main objections levelled against nonlinear and non-standard *learning* approaches such as these, is that they cannot be proven to be stable and robust on a system-by-system basis and are therefore not as reliable as their linear counterparts. Whilst this is true, the question remains as to how valid such a proof for a linear system is, if the underlying system is in fact highly nonlinear or exhibits significant changes over time.

This final chapter illustrated the utilization of neural networks within two model based control structures. In both cases the network was introduced in an untrained state and was optimized during on-line operation. The network was used to minimize modelling discrepancies between a linear model and the system. The improved model could then in turn be utilized to enhance the controller performance. Despite the fact that the two trained control structures performed equally well and that the NPC configuration is computationally more intensive, its flexibility and the ease with which it can be extended to other systems make it the preferred solution. The excessive computing power and long calculation times of neural network based control topologies currently limits the applicability of these structures to slow systems.

REFERENCES

1.....Hunt K.J. and Sbarbaro D., "*NEURAL NETWORKS FOR NONLINEAR INTERNAL MODEL CONTROL*", IEE Proceedings-D, 1991, Vol. 138, No. 5, pp. 431-438.

2.....Willis M.J., Di Massimo C., Mantague G.A., Tham M.T. and Morris A.J., "*ARTIFICIAL NEURAL NETWORKS IN PROCESS ENGINEERING*", IEE Proceedings-D, 1991, Vol. 138, No. 3, pp. 256-266.

3.....Nelkon M. and Parker P., "*ADVANCED LEVEL PHYSICS* : Chapter 4 - Static Bodies. Fluids", Heinemann Educational Books, London, 1974.

4.....Garcia C.E. and Morari M., "*INTERNAL MODEL CONTROL. 1. A UNIFYING REVIEW AND SOME NEW RESULTS*", Ind. Eng. Chem. Process Des. Dev. 1982, 21, pp. 308323.

5.....Economou C.G., Morari M. and Palsson B.O. , "*INTERNAL MODEL CONTROL. 5. EXTENSION TO NONLINEAR SYSTEMS*", Ind. Eng. Chem. Process Des. Dev. 1986, 25, pp. 403411.

6.....Desoer C.A. and Vidyasagar M., "*FEEDBACK SYSTEMS: INPUT-OUTPUT PROPERTIES*", Academic Press, London, 1975.

7.....Psaltis D., Sideris A. and Yamamura A.A., "*A MULTILAYERTED NEURAL NETWORK CONTROLLER*", IEEE Control System Mag., 1988, 8, pp. 1721.

8.....Trossbach W. and Braae M., "*NEURAL NETWORKS FOR REAL-TIME, NON-LINEAR MODELLING AND CONTROL*", presented at 'Artificial Intelligence in Process Engineering' colloquium, University of Stellenbosch, April 1992.

9.....Press W.H., Flannery B.P., Teukolsky S.A. and Vetterling W.T., '*NUMERICAL RECIPES*', Cambridge University Press, 1988.

10.....Powell M.J.D., '*AN EFFICIENT METHOD FOR FINDING THE MINIMUM OF A FUNCTION OF SEVERAL VARIABLES WITHOUT CALCULATING THE DERIVATIVES*', Computer Journal, 1964, 7, pp. 155-162.

11.....Willis M.J., Montague G.A., Di Massimo C., Tham M.T. and Morris A.J., '*NON-LINEAR PREDICTIVE CONTROL USING OPTIMIZATION TECHNIQUES*' ACC 1991, pp. 2788-2793.

6. CONCLUSIONS

This thesis presents the results of an investigation into the possible use of neural networks as part of control structures.

Initially an overview of neural network based control structures was provided, together with the motivation for selecting a subset of these for this investigation. After this a framework of prerequisites for neural networks to be used in these structures was introduced. The subsequent section of the thesis focussed on the concept of feedforward networks and evaluated the suitability of two topologies, using the above prerequisites. This analysis of network structures was then extended to include topologies which incorporate dynamic capability. Finally the performance of two *network based* control structures was evaluated using a simulation of a nonlinear tank system.

The conclusions pertaining to the information presented in this study are discussed below.

6.1 CHOICE OF NEURAL NETWORK BASED CONTROL STRUCTURES

As shown in chapter 2, neural networks have been used in various control laws and structures. These can be grouped into applications which use a network in an adaptive mechanism, in place of a conventional controller or as a model in the control structure and other problem specific utilizations.

Although many of the adaptive and conventional approaches are theoretically plausible, they lack the mathematical foundation to ensure convergence and often require the estimation of system parameters or gradients to generate the training samples. Such systems are therefore not suited for application to practical problems. Since the problem specific approaches lack the generic suitability required, only *model based* structures are examined in this study.

The networks in such structures *learn* to model the underlying system and hence the training data is obtained directly as system samples and needs no advanced processing such as the estimation of unknown parameters and gradients or even the backpropagation of errors through unknown systems required for some of the other approaches.

Model based structures are furthermore mathematically motivated as extensions of their linear origins, assuming that a more accurate improved model might produce better control, particularly for nonlinear systems for which the linear model is not adequate.

6.2 SELECTION OF TRAINING ALGORITHMS AND TOPOLOGIES FOR NETWORK MODELS IN CONTROL STRUCTURES

6.2.1 TRAINING ALGORITHMS

The training of any neural network structure is in effect an optimization problem in n dimensions, where n presents the number of adjustable parameters for that network. Depending on whether an adjustment of parameters is evaluated using a complete, pre-defined set of training data or only the current sample, the algorithms utilized to perform this optimization can be classified as either *batch* or *pattern* learning algorithms

Due to the difficulty of obtaining a representative set of training data from any dynamic system and since the network model is furthermore required to *learn* whilst operating on-line, only pattern learning algorithms are suitable for the applications considered in this study.

6.2.2 FEEDFORWARD NETWORKS

Multilayer perceptrons and regular Gaussian networks are both feedforward structures suitable for on-line training and incorporation as models into a control structure.

Although the multilayer perceptron is more flexible and adaptable in an ideal learning environment, the local nature and fixed positions of the hidden nodes in the regular Gaussian, radial basis function (RBF) structure produce a more robust and faster learning configuration which is better equipped for the non-ideal training circumstances within a control structure.

6.2.3 NETWORKS WITH DYNAMICS

One of the main drawbacks of feedforward networks as models for dynamic systems is their lack of dynamic capability, which is due to the unidirectional flow of information.

More complex structures in which previous outputs and network states are re-utilized do exist and although the *Williams and Zipser* structure and training algorithm are suitable for on-line training, the evaluation shows that these structures often fail to converge and require excessive computing times when increased in size.

Of all the topologies investigated, the regular Gaussian RBF structure is therefore currently the best suited for implementation in a control structure and on-line training, despite its rigid structure and lack of dynamic capability.

6.3 IMPLEMENTATION

6.3.1 COMBINATION MODEL

The combination type model, in which a neural network is used in parallel with a mathematical model, permits the implementation of the controller without any prior training and is therefore superior to the pure network model.

6.3.2 COMPARISON OF NEURAL NETWORK AND STANDARD LINEAR CONTROL

Even prior to training both the neural network based internal model control (NNIMC) and neural predictive control (NPC) produce controllers of equivalent standard as the linear IMC control law.

After sufficient training the improved model allows further tuning of these topologies to produce control action not possible using standard linear approaches and the inadequate model.

Due to the recursive algorithms used in both network approaches, the time required to calculate the next input far exceeds that of linear control laws and therefore currently limits these approaches to applications on slow systems.

6.3.3 COMPARISON OF NEURAL NETWORK IMC AND NPC

Although the final control laws generated by these approaches produce almost identical results, there are a number of significant differences.

Since the NPC algorithm usually requires prediction for several sampling intervals (prediction horizon) the network has to be utilized in an external recursive topology which might compromise the accuracy. The larger number of network evaluations furthermore implies an increased calculation time, thus rendering this approach less applicable for fast systems.

Despite these drawbacks the flexible and transparent nature of the NPC algorithm makes it easier to tune, allows the incorporation of many external constraints and can be extended to multivariable systems without modification. Since the timing limitations, mentioned above, might furthermore be overcome due to the rapid advances in computer technology or the availability of neural network chips, the clear NPC structure is preferable to the IMC topology.

6.4 CONCLUDING REMARKS

The results presented in this thesis show that it is feasible to exploit the modelling and learning capability of neural networks for process control.

Whilst the neural network approach is however unlikely to replace the well known and understood linear controllers which are adequate for the majority of control problems, it might become a reality for highly nonlinear or *difficult to model* problems for which the linear algorithms fail to deliver the required response.

Although the approaches suggested are currently still computationally expensive and slow, the rapid advances in the field of computer and neural network technology are likely to improve this considerably.

BIBLIOGRAPHY

Anderson R.W. and Vemuri V., "*NEURAL NETWORKS CAN BE USED FOR OPEN-LOOP, DYNAMICAL CONTROL*", preprint, to appear in International Journal of Neural Networks, 1991.

Barto A.G., Anderson C.W. and Sutton R.S., "*SYNTHESIS OF NONLINEAR CONTROL SURFACES BY A LAYERED ASSOCIATIVE SEARCH NETWORK*", Biological Cybernetics, 43, 1982, pp. 175-185.

Berenji H.R., "*ARTIFICIAL NEURAL NETWORKS AND APPROXIMATE REASONING FOR INTELLIGENT CONTROL IN SPACE*", ACC 1991, pp. 1075-1080.

Bozich D.J. and MacKay H.B., "*VIBRATION CANCELLATION AT MULTIPLE LOCATIONS USING NEUROCONTROLLERS WITH REAL-TIME LEARNING*", IJCNN 1991, pp. II775-II780.

Bremermann H.J. and Anderson R.W., "*AN ALTERNATIVE TO BACK-PROPAGATION: A SIMPLE RULE OF SYNAPTIC MODIFICATION FOR NEURAL NET TRAINING AND MEMORY*", Internal Report, Center for Pure and Applied Mathematics, University of California, Berkeley, 1990.

Bremermann H.J. and Anderson R.W., "*HOW THE BRAIN ADJUSTS SYNAPSES - MAYBE*", preprint, to appear in "*FESTSCHRIFT FOR WOODY BLEDSOE*".

Cater J.P., "*SUCCESSFULLY USING LEARNING RATES OF 10 (AND GREATER) IN BACK-PROPAG*ATION *NETWORKS WITH THE HEURISTIC LEARNING ALGORITHM*", IICNN 1987, pp. II645-II651.

Chen F.C., "*BACK-PROPAGATION NEURAL NETWORKS FOR NONLINEAR SELF-TUNING ADAPTIVE CONTROL*", IEEE Control Systems Magazine, April 1990, pp. 44-47.

Cheok K.C. and Smith J.C., "*ADAPTIVE NEURAL NETWORK CONTROL WITH FREQUENCY-SHAPED OPTIMAL OUTPUT FEEDBACK*", International Joint Conference on Neural Networks 1991, pp. II741-II746.

Chinrungrueng C. and Sequin C.H., "*OPTIMAL ADAPTIVE K-MEANS ALGORITHM WITH DYNAMIC ADJUSTMENT OF LEARNING RATE*", IJCNN 1991, pp. I855-I862.

Cooper D.J., Megan L. and Hinde R.F., "*A NEURAL PATTERN ANALYZER FOR ADAPTIVE PROCESS CONTROL*", ACC 1991, pp. 2794-2799.

Cotter N.E., Guillerm T.M., Soller J.B. and Conwell P.R, "*PREJUDICIAL SEARCHES AND THE POLE BALANCER*", IJCNN 1991, pp. II689-II694.

Cramer J.E. and Womack B.F., "*ADAPTIVE CONTROL USING NEURAL NETWORKS*", ACC 1991, pp. 681-686.

Cui X. and Shin K.G., "*DESIGN OF AN INDUSTRIAL PROCESS CONTROLLER USING NEURAL NETWORKS*", ACC 1991, pp. 508-513.

Cui X. and Shin K.G., "*INTELLIGENT COORDINATION OF MULTIPLE SYSTEMS WITH NEURAL NETWORKS*", ACC 1991, pp. 481-486.

Cybenko G., "*APPROXIMATION BY SUPERPOSITION OF A SIGMOIDAL FUNCTION*", Mathematics of Control, Signals, and Systems, Vol. 2, 1989, pp. 303-314.

Dahl E.D., "*ACCELERATED LEARNING USING THE GENERALIZED DELTA RULE*", IICNN 1987, pp. II523-II530.

De Villiers J. and Barnard E., "*BACKPROPAGATION NEURAL NETS WITH ONE AND TWO HIDDEN LAYERS*", IEEE Transactions on Neural Networks, Vol. 4, No. 1, 1992, pp. 136-141.

Desoer C.A. and Vidyasagar M., "*FEEDBACK SYSTEMS: INPUT-OUTPUT PROPERTIES*", Academic Press, London, 1975.

Dreyfus H.L and Dreyfus S.E., "*KUENSTLICHE INTELLIGENZ - VON DEN GRENZEN DER DENKMASCHINE UND DEM WERT DER INTUITION*", orig. "*MIND OVER MATTER*", New York, The Free Press, 1986.

Economou C.G., Morari M. and Palsson B.O. , "*INTERNAL MODEL CONTROL. 5. EXTENSION TO NONLINEAR SYSTEMS*", Ind. Eng. Chem. Process Des. Dev. 1986, 25, pp. 403411.

Garcia C.E. and Morari M., "*INTERNAL MODEL CONTROL. 1. A UNIFYING REVIEW AND SOME NEW RESULTS*", Ind. Eng. Chem. Process Des. Dev. 1982, 21, pp. 308323.

Girosi F. and Poggio, "*NETWORKS AND THE BEST APPROXIMATION PROPERTY*", Biological Cybernetics, 63, 1990, pp. 169-176.

Grossberg S., "*NONLINEAR NEURAL NETWORKS: PRINCIPLES, MECHANISMS, AND ARCHITECTURES*", Neural Networks, Vol. 1, 1988, pp. 17-61.

Guez A. and Piovoso M., "*CUSTOM NEUROCONTROLLER FOR A TIME DELAY PROCESS*", ACC 1991, pp. 1592-1596.

Guez A., Eilbert J.L. and Kam M., "*NEURAL NETWORK ARCHITECTURE FOR CONTROL*", IEEE Control Systems Magazine, April 1988, pp. 23-25.

Hampo R. and Marko K., "*NEURAL NETWORK ARCHITECTURES FOR ACTIVE SUSPENSION CONTROL*", IJCNN 1991, pp. II765-II770.

Hashimoto H., Kubota T., Kudou M. and Harashima F., "*SELF-ORGANIZING VISUAL SERVO SYSTEM BASED ON NEURAL NETWORKS*", ACC 1991, pp. 2262-2267.

Hecht-Nielsen R., "*KOLMOGOROV'S MAPPING NEURAL NETWORK EXISTENCE THEOREM*", IEEE International Conference on Neural Networks 1987, pp. III11-III14.

Hirose Y., Yamashita K. and Hijiya S., "*BACK-PROPAGATION ALGORITHM WHICH VARIES THE NUMBER OF HIDDEN UNITS*", Neural Networks, Vol. 4, 1991, pp. 61-66.

Hopfield J.J. and Tank D.W., "*NEURAL COMPUTATION OF DECISIONS IN OPTIMIZATION PROBLEMS*", Biological Cybernetics, Vol. 52, 1985, pp. 141-152.

Hornik K., Stinchcombe M. and White H., "*MULTILAYER FEEDFORWARD NETWORKS ARE UNIVERSAL APPROXIMATORS*", Neural Networks, Vol. 2, 1989, pp. 359-366.

Hoskins D.A., Hwang J.N. and Vagners, "*ITERATIVE INVERSION OF NEURAL NETWORKS AND ITS APPLICATION TO ADAPTIVE CONTROL*", IEEE Transactions on Neural Networks, Vol. 3, No. 2, 1992, pp. 292-301.

Hsuing J.T. and Himmelblau D.M., "*DEVELOPMENT OF CONTROL STRATEGIES VIA ARTIFICIAL NEURAL NETWORKS AND REINFORCEMENT LEARNING*", ACC 1991, pp. 2326-2330.

Hunt K.J. and Sbarbaro D., "*NEURAL NETWORKS FOR NONLINEAR INTERNAL MODEL CONTROL*", IEE Proceedings-D, Vol. 138, No. 5, 1991, pp. 431-438.

Ichikawa Y. and Sawa T., "*NEURAL NETWORK APPLICATION FOR DIRECT FEEDBACK CONTROLLERS*", IEEE Transactions on Neural Networks, Vol. 3, No. 2, March 1992, pp. 224-231.

Iiguni Y., Sakai H. and 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.

Ingman D. and Merlis Y., "*LOCAL MINIMUM ESCAPE USING THERMODYNAMIC PROPERTIES OF NEURAL NETWORKS*", Neural Networks, Vol. 4, 1991, pp. 395-404.

Jacobs R.A. and Jordan M.I., "*A MODULAR CONNECTIONIST ARCHITECTURE FOR LEARNING PIECEWISE CONTROL STRATEGIES*", ACC 1991, pp. 1597-1602.

Karsai G., "*LEARNING TO CONTROL: SOME PRACTICAL EXPERIMENTS WITH NEURAL NETWORKS*", IJCNN 1991, pp. II701-II707.

Kirkpatrick S., Gelatt C.D. and Vecchi M.P., "*OPTIMIZATION BY SIMULATED ANNEALING*", Science, Vol. 220, 1983, pp. 671-680.

Kolmogorov A.N., "*ON THE REPRESENTATION OF CONTINUOUS FUNCTIONS OF MANY VARIABLES BY SUPERPOSITION OF CONTINUOUS FUNCTIONS OF ONE VARIABLE AND ADDITION*", Dokl. Akad. Nauk USSR, 114, 1957, pp. 935-956.

Kong S.G. and Kosko B., "*ADAPTIVE FUZZY SYSTEMS FOR BACKING UP A TRUCK-AND-TRAILER*", IEEE Transactions on Neural Networks, Vol. 3, No. 2, pp. 211-223, 1992.

Kreinovich V. Y., "*ARBITRARY NONLINEARITY IS SUFFICIENT TO REPRESENT ALL FUNCTIONS BY NEURAL NETWORKS: A THEOREM*", Neural Networks, Vol. 4, 1991, pp. 381-383.

Kumar S.S. and Guez A., "*ART BASED ADAPTIVE POLE PLACEMENT FOR NEUROCONTROLLERS*", Neural Networks, Vol. 4, 1991, pp. 319-335.

Lambert J.M. and Hecht-Nielsen R., "*APPLICATION OF FEEDFORWARD AND RECURRENT NEURAL NETWORKS TO CHEMICAL PLANT PREDICTIVE MODELING*", IJCNN 1991, pp. I373-I378.

Lee S. and Kil R.M., "*MULTILAYER FEEDFORWARD POTENTIAL FUNCTION NETWORK*", pre-print, submitted to Neural Networks September 1989.

Leonard J.A. and Kramer M.A., "*RADIAL BASIS FUNCTION NETWORKS FOR CLASSIFYING PROCESS FAULTS*", IEEE Control Systems Magazine, April 1991, pp. 31-38.

Levin E., Gewirtzman R. and Inbar G.F., "*NEURAL NETWORK ARCHITECTURE FOR ADAPTIVE SYSTEM MODELING AND CONTROL*", Neural Networks, Vol. 4, 1991, pp. 185-191.

Lippmann R.P., "*AN INTRODUCTION TO COMPUTING WITH NEURAL NETS*", IEEE ASSP Magazine, 1987, pp. 4-22.

Miller W.T., Latham P.J. and Scalera S.M., "*BIPEDAL GAIT ADAPTION FOR WALKING WITH DYNAMIC BALANCE*", ACC 1991, pp. 1603-1608.

Minsky M., Seymour P., "*PERCEPTRONS: AN INTRODUCTION TO COMPUTATIONAL GEOMETRY*", Cambridge, MA: MIT Press, 1969.

Moody T.J. and Darken C.J., "*FAST LEARNING IN NETWORKS OF LOCALLY TUNED PROCESSING UNITS*", Neural Computation, Vol. 1, 1989, pp. 151-160.

Moore K.L. and Naidu S., "*LINEAR QUADRATIC REGULATION USING NEURAL NETWORKS*", IJCNN 1991, pp. II735-II739.

Moore K.L., "*A REINFORCEMENT-LEARNING NEURAL NETWORK FOR THE CONTROL OF NONLINEAR SYSTEMS*", ACC 1991, pp. 21-22.

Narendra K.S. and Levin A.U., "*REGULATION OF NONLINEAR DYNAMICAL SYSTEMS USING MULTIPLE NEURAL NETWORKS*", ACC 1991, pp. 1609-1614.

Narendra K.S. and Mukhopadhyay S., "*INTELLIGENT CONTROL USING NEURAL NETWORKS*", ACC 1991, pp. 1069-1074.

Narendra K.S. and Parthasaraty K., "*IDENTIFICATION AND CONTROL OF DYNAMICAL SYSTEMS USING NEURAL NETWORKS*", IEEE Transactions on Neural Networks, Vol. 1, No. 1, 1990, pp. 4-27.

Nelkon M. and Parker P., "*ADVANCED LEVEL PHYSICS* : Chapter 4 - Static Bodies. Fluids", Heinemann Educational Books, London, 1974.

Nguyen H. and Widrow B., "*NEURAL NETWORKS FOR SELF-LEARNING CONTROL SYSTEMS*", IEEE Control Systems Magazine, April 1990, pp. 18-23.

Parker D.B. , "*OPTIMAL ALGORITHMS FOR ADAPTIVE NETWORKS: SECOND ORDER BACKPROPAGATION, SECOND ORDER DIRECT PROPAGATION, AND SECOND ORDER HEBBIAN LEARNING*", IICNN 1987, pp. II593-II600.

Powell M.J.D., "*AN EFFICIENT METHOD FOR FINDING THE MINIMUM OF A FUNCTION OF SEVERAL VARIABLES WITHOUT CALCULATING THE DERIVATIVES*", Computer Journal, 1964, 7, pp. 155-162.

Press W.H., Flannery B.P., Teukolsky S.A. and Vetterling W.T., "*NUMERICAL RECIPES*", Cambridge University Press, 1988.

Psaltis D., Sideris A. and Yamamura A., "*NEURAL CONTROLLERS*", International Conference on Neural Networks 1987, pp. IV551-IV558.

Psaltis D., Sideris A. and Yamamura A.A., "*A MULTILAYERTED NEURAL NETWORK CONTROLLER*", IEEE Control System Mag., 1988, 8, pp. 1721.

Qin S., Su H. and McAvoy T., "*COMPARISON OF FOUR NEURAL NET LEARNING METHODS FOR DYNAMIC SYSTEM IDENTIFICATION*", IEEE Transactions on Neural Networks, Vol. 3, No. 1, 1992, pp. 122-130.

Rigler A.K., Irvoine J.M. and Vogl T.P., "*RESCALING OF VARIABLES IN BACK PROPAGATION LEARNING*", Neural Networks, Vol. 4, 1991, pp. 225-229.

Roger Jang J.S. and Sun C.T., "*FUNCTIONAL EQUIVALENCE BETWEEN RADIAL BASIS FUNCTION NETWORKS AND FUZZY INFERENCE SYSTEMS*", IEEE Transactions on Neural Networks, Vol. 4, No. 1, 1993, pp. 156-159.

Rosenblatt F., "*PRINCIPLES OF NEURODYNAMICS: PERCEPTRONS AND THE THEORY OF BRAIN MECHANISMS*", Washington, DC: Sparta Books, 1962.

Rumelhart D.E., McClelland J.L. and the PDP research group, "*PARALLEL DISTRIBUTED PROCESSING - EXPLORATION IN THE MICROSTRUCTURE OF COGNITION, Volume 1: Foundations*", Cambridge, MA: MIT Press, 1986.

Sbarbaro D. and Gawthrop P.J., "*SELF-ORGANIZATION AND ADAPTION IN GAUSSIAN NETWORKS*", 9th IFAC/IFOR Symposium on Identification and System Parameter Estimation, 1991.

Shoemaker P.A., Carlin M.J. and Shimabukuro R.L., "*BACK PROPAGATION LEARNING WITH TRINARY QUANTIZATION OF WEIGHT UPDATES*", Neural Networks, Vol. 4, 1991, pp. 231-241.

Smolensky P., "*CONNECTIONIST AI, SYMBOLIC AI, AND THE BRAIN*", Artificial Intelligence Review, 1987, pp. 95-109.

Spall J.C. and Cristion J.A., "*EFFICIENT WEIGHT ESTIMATION IN NEURAL NETWORKS FOR ADAPTIVE CONTROL*", ACC 1991, pp. 16-20.

Srender K.K., "*EFFICIENT ACTIVATION FUNCTIONS FOR THE BACK-PROPAGATION NEURAL NETWORK*", presented at IJCNN 1991.

Steck J., Krishnamurthy K., McMillin B. and Leiniger G., "*NEURAL MODELING AND CONTROL OF A DISTILLATION COLUMN*", IJCNN 1992, pp. II771-II774.

Steinberg M. and DiGirolamo R., "*APPLYING NEURAL NETWORK TECHNOLOGY TO FUTURE GENERATION MILITARY FLIGHT CONTROL SYSTEMS*", presented at IJCNN 1991.

Stornetta W.S. and Huberman B.A., "*AN IMPROVED THREE-LAYER BACK PROPAGATION ALGORITHM*", IICNN 1987, pp. II637-II643.

Su H. and McAvoy T.J., "*IDENTIFICATION OF CHEMICAL PROCESSES USING RECURRENT NETWORKS*", ACC 1991, pp. 2314-2319.

Sun G.Z., Chen H.H. and Lee Y.C., "*A FAST ON-LINE LEARNING ALGORITHM FOR RECURRENT NEURAL NETWORKS*", IJCNN 1991, pp. II13-II18.

Sznaier M. and Damborg M.J., "*AN ANALOG "NEURAL NET" BASED SUBOPTIMAL CONTROLLER FOR CONSTRAINED DISCRETE-TIME LINEAR SYSTEMS*", Automatica, Vol. 28, No. 1, 1992, pp. 139-144.

Trossbach W. and Braae M., "*INVESTIGATION INTO NEURAL NETWORKS FOR CONTROL OF NON-LINEAR PROCESSES*", presented at SACAC Tutorial and Workshop on Neural Networks, Durban University, July 1991.

Trossbach W. and Braae M., "*NEURAL NETWORKS FOR REAL-TIME, NON-LINEAR MODELLING AND CONTROL*", presented at 'Artificial Intelligence in Process Engineering' colloquium, University of Stellenbosch, April 1992.

Troudet T., Garg S., Mattern D. and Merril W., "*TOWARDS PRACTICAL CONTROL DESIGN USING NEURAL COMPUTATION*", IJCNN 1991, pp. II675-II681.

Tsung F.S., "*LEARNING IN RECURRENT FINITE DIFFERENCE NETWORKS*", Connectionist Models Summer School 1991.

Watrous R. L., "*LEARNING ALGORITHMS FOR CONNECTIONIST NETWORKS: APPLIED GRADIENT METHODS OF NONLINEAR OPTIMIZATION*", IICNN 1987, pp. II619-II627.

Weir M.K., "*A METHOD FOR SELF-DETERMINATION OF ADAPTIVE LEARNING RATES IN BACK PROPAGATION*", Neural Networks, Vol. 4, 1991, pp. 371-379.

Werbos P.J., "*GENERALIZATION OF BACKPROPAGATION WITH APPLICATION TO A RECURRENT GAS MARKET MODEL*", Neural Computation, Vol. 1, 1988, pp. 270-280.

Wieland A.P., "*EVOLVING NEURAL NETWORK CONTROLLERS FOR UNSTABLE SYSTEMS*", IJCNN 1991, pp. II667-II673.

Williams R.J. and Zipser D., "*A LEARNING ALGORITHM FOR CONTINUALLY RUNNING FULLY RECURRENT NETWORKS*", Neural Computation, Vol. 1, 1989.

Willis M.J., Di Massimo C., Mantague G.A., Tham M.T. and Morris A.J., "*ARTIFICIAL NEURAL NETWORKS IN PROCESS ENGINEERING*", IEE Proceedings-D, 1991, Vol. 138, No. 3, pp. 256-266.

Willis M.J., Di Massimo C., Montague G.A., Tham M.T. and Morris A.J., "*NON-LINEAR PREDICTIVE CONTROL USING OPTIMIZATION TECHNIQUES*", ACC 1992, pp. 2788-2793.

Willis M.J., Montague G.A., Morris A.J. and Tham M.T., "*ARTIFICIAL NEURAL NETWORKS: - AM PANACEA TO MODELLING PROBLEMS?*", ACC 1991, pp. 2337-2342.

Wray J. and Green G.G.R. , "*HOW NEURAL NETWORKS WORK: THE MATHEMATICS OF NETWORKS USED TO SOLVE STANDARD ENGINEERING PROBLEMS*", ACC 1991, pp. 2311-2313.

Wu Q.H., Hogg B.W. and Irwin G.W., "*A NEURAL NETWORK REGULATOR FOR TURBOGENERATORS*", IEEE Transactions on Neural Networks, Vol. 3, No. 1, 1992, pp. 95-100.

Youngjik L., Sang-Hoon O. and Myung W.K., "*THE EFFECT OF INITIAL WEIGHTS ON PREMATURE SATURATION IN BACK-PROPAGATION LEARNING*", IJCNN 1991, pp. I765-I770.

APPENDIX I - GENERAL EXISTENCE THEOREM

This Appendix presents one of the many network mapping theorems. The proof included here is taken directly from Kreinovich [1] and was selected as an example due to its generality, rather than to validate the mapping competence of all networks discussed in this work.

Preliminary Remarks and Definitions

A neural network approximates the function $f(x_1,x_2...x_m)$ (defined on $[-X,X]^m$) with precision $e>0$, if for every vector $\underline{x}\in[-X,X]^m$, $|y - f(\underline{x})| < e$ where y is the output of the network for input $\underline{x}$.

Since mappings $f:[-X,X]^m \rightarrow \Re^n$ $n>1$ are effectively n functions $f_k(x_1,x_2...x_m)$, $k=1,2...n$ with each function mapping the dependence of the k-th coordinate of the output on the input vector $\underline{x}$. Such a mapping can therefore be approximated with precision e if each of the n outputs is approximated as described above.

Main Theorem

Assume that g(x) is an arbitrary smooth (at least three times differentiable function) $\Re\rightarrow\Re$, X and e are positive real numbers, and f is a continuous mapping from $[-X,X]^m$ to $\Re^n$. Then there exists a neural network that consists of linear elements and elements $x\rightarrow g(x)$ that approximates f with precision e.

Proof

1. Since a mapping to $\Re^n$ can be represented by combining n functions, it is sufficient to show that the network can approximate any real-valued function.

2. The Weierstrass approximation theorem shows that any arbitrary continuous function on a cube can be approximated by a polynomial with precision d (where d is any real number > 0).
   It is therefore sufficient to show that neural networks can approximate polynomials with precision $e/2$, since the polynomial can then be used to approximate the function f with the same precision thus resulting in a network which approximates f with precision e.

3. A polynomial of m input variables is obtained by addition and multiplication of these inputs $x_1,x_2...x_m$.
   Since the linear elements included can perform the addition, it is sufficient to show that the network can also implement multiplication.
   (It is worth noting that using a Taylor expansion in networks without

linear elements the addition can be shown to be performed by the nonlinear members.)

4. Since g(x) is at least three times differentiable and *nonlinear* its second derivative exists and has to be nonzero for some x i.e. there exists a point x_o where $g''(x_o)=d\neq 0$
then $$g(x_o+h) = g(x_o) + g'(x_o)h + dh^2 + o(h^2)$$

where $F = o(G)$ means that $F/G \to 0$ as $h \to 0$ i.e. for every e_1 there exists an H such that for $h\in[-H,H]$

$$|g(x_o+h) - g(x_o) - g'(x_o)h - dh^2| < e_1 h^2$$

and hence

$$|g(x+h) - g(x_o) - g'(x_o)h - dh^2| < e_1 H^2$$

or $$|h^2 - a(h)| < e_1 H^2/d \text{ where } a(h) = [g(x_o+h) - g(x_o) - g'(x_o)h]/d$$

Since all elements of a(h) are linear and constants, the expression can be implemented by a neural network which implies that this neural network approximates the squaring function $h \to h^2$ on [-H,H].

For the above to apply to [-X,X] this interval is transformed linearly into [-H,H] .

- substituting for $h = (H/X)x$ and then applying a(h) which is a network and let $y = (X/H)^2 a(h)$ then since a(h) is an approximation to $h^2 = ((H/X)x)^2$ it follows that y is an approximation to $(X/H)^2((H/X)x)^2 = x^2$
- the precision of this approximation is obtained by multiplying both sides of $|a(h) - h^2| < e_1 H^2/d$ by $(X/H)^2$ which, since $(X/H)^2 h^2 = x^2$ and $(X/H)^2 a(h) = y$, results in $|y - x^2| < e_1 X^2$

Choosing e_1 such that $e_1 < c/X^2$ where c is an arbitrary positive real number, then by applying the above network y it is possible to approximate the function x^2 with precision c.

5. Since the network can approximate linear functions and x^2, it can implement multiplication $x,y \to xy$ as $1/4((x + y)^2 - (x - y)^2)$, which in view of 1-3 above completes the proof of the theorem. Q.E.D.

REFERENCES

1.....Kreinovich V., *"ARBITRARY NONLINEARITY IS SUFFICIENT TO REPRESENT ALL FUNCTIONS BY NEURAL NETWORKS: A THEOREM"*, Neural Networks, Vol. 4, 1991, pp. 381-383.

APPENDIX II - SOURCE CODE LISTING

This appendix contains listings of all relevant source code. Rather than including a listing of each of the programs, the appropriate units and examples of how to apply them have been included. All listings shown, were developed and run using Borland's Turbo Pascal version 6.0.

1. FEEDFORWARD NETWORKS

```
UNIT FFNETS;

(*=============================================================================================================*)

INTERFACE

 CONST

   max_inp  = 10;                                    {maximum number of inputs}
   max_hid  = 100;                                   {maximum number of hidden nodes}
   max_outp = 10;                                    {maximum number of outputs}


 TYPE

   Xfer = function(x:real):real;                     {type for Xfer function}

   FFNet = object                                    {abstract feedforward network object}
     inp,hid,
     outp      : integer;                            {actual size parameters}
     InpVec    : array[1..max_inp] of real;          {input vector}
     HidVec    : array[1..max_hid] of real;          {outp. of hidden}
     DesOutp,                                        {desired output}
     OutpVec   : array[1..max_outp] of real;         {output vector}
     inp_Xfer,                                       {transfer function}
     hid_Xfer,                                       {for individ.     }
     outp_Xfer  : Xfer;                              {layers           }
     Wt_hid_outp,                                    {wts from hid to out}
     dW_hid_outp: array[1..max_hid,1..max_outp] of real;  {wt changes        for offset }
     beta,alpha : real;                              {learning and momentum rates}
     fname      : string;                            {directory & filename}
    end;

   FastPerceptron = object(FFNet)                    {perceptron with single hidden layer}
     DeltaHid   : array[1..max_hid] of real;         {delta for hidden l.}
     DeltaOutp  : array[1..max_outp] of real;        {delta for output l.}
     Wt_inp_hid,                                     {wts from inp to hid - add 1 for theta's}
     dW_inp_hid : array[1..(max_inp+1),1..max_hid] of real;  {wt changes        in hidden layer  }
     constructor Init(i,h,o:integer;bt,al:real;nam:string);
     procedure FeedForward; virtual;
     procedure BackPropagate; virtual;
     function SaveNet:integer;
     function LoadNet:integer;
     destructor Done;
    end;

   FastRBF = object(FFNet)                           {RBF network}
     delta : real;                                   {bandwidth}
     irh : integer;                                  { inputh root of hidden - nodes per
                                                       dimens.}
     centres : array[1..max_hid,1..max_inp] of real; {coord. of centres - stored for speed}
     constructor Init(i,ih,o:integer;al:real;nam:string);
     procedure FeedForward; virtual;
     procedure Train; virtual;
     function SaveNet:integer;
     function LoadNet:integer;
     destructor Done;
    end;
```

```
(*-------------------------------------------------------------------------------------------------------------*)
IMPLEMENTATION
(*-------------------------------------------------------------------------------------------------------------*)
 function Linear(x:real):real; far;                                    {linear Xfer function}
  begin
    Linear := x;
  end;
(*-------------------------------------------------------------------------------------------------------------*)
 function Sigmoid(x:real):real; far;                                   {sigmoidal (logistic) Xfer function}
  begin
    if x>10 then Sigmoid := 0.999999999
        else if x<-10 then Sigmoid := 0.00000001
            else Sigmoid := 1/(1+exp(-x));
  end;
(*-------------------------------------------------------------------------------------------------------------*)
 function Gaussian(x:real):real; far;                                  {Gaussian Xfer function}
  begin
    if x<1e-9 then Gaussian := 0.999999999
        else if x>20 then Gaussian := 0.00000001
            else Gaussian := exp(-x);
  end;
(*-------------------------------------------------------------------------------------------------------------*)
 CONST
   lin :Xfer = Linear;                                                 {define Xfer functions as const to pass}
   sigm:Xfer = Sigmoid;                                                {  as parameters}
   Gaus:Xfer = Gaussian;
(*-------------------------------------------------------------------------------------------------------------*)
 constructor FastPerceptron.Init(i,h,o:integer;ht,al:real;nam:string);          {create perceptron}
   var
     q,r : integer;
   begin
     inp := i+1;
     InpVec[inp] := 1;                                                 {constant input for hid. layer thetas}
     hid := h;
     outp := o;
     inp_Xfer  := lin;
     hid_Xfer  := sigm;
     outp_Xfer := lin;
     heta  := ht;
     alpha := al;
     fname := nam;
     Randomize;                                                        {initialize all weights}
     for q:=1 to inp do
       for r:=1 to hid do
         begin
           Wt_inp_hid[q,r] := (random-0.5)*2;
           dW_inp_hid[q,r] := 0;
         end;
     for q:=1 to hid do
       for r:=1 to outp do
         begin
           Wt_hid_outp[q,r] := (random-0.5)*2;
           dW_hid_outp[q,r] := 0;
         end;
   end;
(*-------------------------------------------------------------------------------------------------------------*)
 procedure FastPerceptron.FeedForward;                                 {calculate output with current inputs}
   var
     q,r : integer;
   begin
     InpVec[inp]:=1;
     for q:=1 to hid do
       begin
         HidVec[q] := 0;
         for r:=1 to inp do
```

```
          HidVec[q] := HidVec[q]+Wt_inp_hid[r,q]*InpVec[r];      {hid. layer inp}
        HidVec[q] := Sigmoid(HidVec[q]);                         {hid. layer outp}
      end;
    for q:=1 to outp do
      begin
        OutpVec[q] := 0;
        for r:=1 to hid do
          OutpVec[q] := OutpVec[q]+Wt_hid_outp[r,q]*HidVec[r];   {outp. layer inp.=outp. since lin.}
      end;
  end;

(*-------------------------------------------------------------------------------------------------------*)

 procedure FastPerceptron.BackPropagate;                         {perform backprop. optimiztion}

   var
     q,r : integer;

   begin

     for q:=1 to outp do
       deltaOutp[q] := DesOutp[q]-OutpVec[q];                    {delta outp. for lin. units}

     for q:=1 to hid do
       begin
         deltaHid[q] := 0;
         for r:=1 to outp do
           deltaHid[q] := deltaHid[q] + Wt_hid_outp[q,r]*deltaOutp[r];
         deltaHid[q] := HidVec[q]*(1-HidVec[q])*deltaHid[q];     {delta for sigm. hidden l.}
       end;

      for q:=1 to outp do
        for r:=1 to hid do
          begin
              dW_hid_outp[r,q] := heta*deltaOutp[q]*HidVec[r] + alpha*dW_hid_outp[r,q];
              Wt_hid_outp[r,q] := Wt_hid_outp[r,q] + dW_hid_outp[r,q];     {update weights}
            end;

        for q:=1 to hid do
          for r:=1 to inp do
            begin
              dW_inp_hid[r,q] := heta*deltaHid[q]*InpVec[r] + alpha*dW_inp_hid[r,q];
              Wt_inp_hid[r,q] := Wt_inp_hid[r,q] + dW_inp_hid[r,q];
            end;

   end;

(*-------------------------------------------------------------------------------------------------------*)

 function FastPerceptron.SaveNet:integer;                        {save current network parameters}

   var
     q,r   : integer;
     tfile : text;

   begin
{$I-}
     assign(tfile,fname);
     rewrite(tfile);
     writeln(tfile,'Perceptron Weight File : ',fname);
     writeln(tfile);
     writeln(tfile,inp,'  ',hid,'  ',outp,' - inputs(+const. for th.), hidden, outputs');
     writeln(tfile);
     writeln(tfile,heta,'  ',alpha,' - heta, alpha');
     writeln(tfile);
     writeln(tfile,'weights from inp. to hidden (inp * hid) :');
     for q:=1 to inp do
       begin
         for r:=1 to hid do write(tfile,Wt_inp_hid[q,r],'  ');
         writeln(tfile);
       end;
     writeln(tfile);
     writeln(tfile,'weights from hidden to output :');
     for q:=1 to hid do
       begin
         for r:=1 to outp do write(tfile,Wt_hid_outp[q,r],'  ');
         writeln(tfile);
       end;
     close(tfile);
{$I+}
     SaveNet := IOResult;
   end;

(*-------------------------------------------------------------------------------------------------------*)

 function FastPerceptron.LoadNet:integer;                        {load network parameters from file}

   var
```

```
    q,r   : integer;
    tfile : text;

  begin
{$I-}
    assign(tfile,fname);
    reset(tfile);
    readln(tfile); readln(tfile);
    readln(tfile,inp,hid,outp); readln(tfile);
    readln(tfile,heta,alpha); readln(tfile);
    readln(tfile);
    for q:=1 to inp do
      begin
        for r:=1 to hid do read(tfile,Wt_inp_hid[q,r]);
        readln(tfile);
      end;
    readln(tfile); readln(tfile);
    for q:=1 to hid do
      begin
        for r:=1 to outp do read(tfile,Wt_hid_outp[q,r]);
        readln(tfile);
      end;
    close(tfile);
{$I+}
    LoadNet := IOResult;
  end;
(*-------------------------------------------------------------------------------------------------------*)

 destructor FastPerceptron.Done;                                  {destructor for perceptron}

  begin
  end;

(*-------------------------------------------------------------------------------------------------------*)

 constructor FastRBF.Init(i,ih,o:integer;al:real;nam:string);     {create RBF network}

  var
    space : real;
    q,r,s,t,l : integer;

  begin
    inp := i;
    hid := ih;
    irh := ih;
    for q:=1 to (inp-1) do
      hid := hid*ih;
    outp := o;
    inp_Xfer  := lin;
    hid_Xfer  := Gaus;
    outp_Xfer := lin;
    alpha := al;
    fname := nam;

    Randomize;              {calculate centres & initialize all weights}

    space := 2/(irh-1);

    for q:=1 to hid do
      for r:=1 to inp do
        begin
          s := 1;
          for t:=1 to (r-1) do
            s := s*irh;
          centres[q,r] := -1+(((q-1) div s)mod irh)*space;
        end;

    delta := 2/sqr(space);

    for q:=1 to hid do
      for r:=1 to outp do
        begin
          Wt_hid_outp[q,r] := (random-0.5)/10;
        end;
  end;
(*-------------------------------------------------------------------------------------------------------*)

 procedure FastRBF.FeedForward;                                   {generate ouput}

  var
    q,r : integer;

  begin
    for q:=1 to hid do
      begin
        HidVec[q] := 0;
        for r:=1 to inp do
```

```
          HidVec[q] := HidVec[q]+delta*sqr(0.75*InpVec[r]-centres[q,r]);      {hid. layer inp}
        HidVec[q] := Gaussian(HidVec[q]);                                     {hid. layer outp}
      end;
    for q:=1 to outp do
      begin
        OutpVec[q] := 0;
        for r:=1 to hid do
          OutpVec[q] := OutpVec[q]+Wt_hid_outp[r,q]*HidVec[r];    {outp. layer inp.=outp. since lin.}
      end;
  end;

(*-------------------------------------------------------------------------------------------------------*)

 procedure FastRBF.Train;                                                 {update weights}

   var
     q,r : integer;

   begin
     for q:=1 to hid do
       for r:=1 to outp do
         Wt_hid_outp[q,r] := Wt_hid_outp[q,r] + alpha*HidVec[q]*(DesOutp[r]-OutpVec[r]);  {update weights}
   end;

(*-------------------------------------------------------------------------------------------------------*)

 function FastRBF.SaveNet:integer;                                        {save current network parameters}

   var
     q,r   : integer;
     tfile : text;

   begin
{$I-}
     assign(tfile,fname);
     rewrite(tfile);
     writeln(tfile,'RBF Weight File : ',fname);
     writeln(tfile);
     writeln(tfile,inp,'  ',irh,'  ',outp,' - inputs, i-r-h, outputs');
     writeln(tfile);
     writeln(tfile,alpha,' - alpha');
     writeln(tfile);
     writeln(tfile,'weights from hidden to output :');
     for q:=1 to hid do
       begin
         for r:=1 to outp do write(tfile,Wt_hid_outp[q,r],'  ');
         writeln(tfile);
       end;
     close(tfile);
{$I+}
     SaveNet := IOResult;
   end;

(*-------------------------------------------------------------------------------------------------------*)

 function FastRBF.LoadNet:integer;                                        {load network parameters from file}

   var
     q,r   : integer;
     tfile : text;

   begin
{$I-}
     assign(tfile,fname);
     reset(tfile);
     readln(tfile); readln(tfile);
     readln(tfile,inp,irh,outp); readln(tfile);
     readln(tfile,alpha); readln(tfile);readln(tfile);
     for q:=1 to hid do
       begin
         for r:=1 to outp do read(tfile,Wt_hid_outp[q,r]);
         readln(tfile);
       end;
     close(tfile);
{$I+}
     LoadNet := IOResult;
   end;

(*-------------------------------------------------------------------------------------------------------*)

 destructor FastRBF.Done;                                                 {destructor for RBF net}
   begin
   end;
(*-------------------------------------------------------------------------------------------------------*)

End.

(*=======================================================================================================*)
```

The following listings shows how to generate and use perceptron and RBF networks using the procedures listed above :

```
(*-------------------------------------------------------------------------------------------------------*)

{to create perceptron}

uses dos,
     crt,
     FFNets;                                                    {include unit}

var
  net : ^FastPerceptron;                                        {create occurence - here a pointer was
                                                                 chosen to create the network using heap
                                                                 memory}
.....
          {in main program}
begin
  release(HeapOrg);
  new(net,Init(1,3,1,0.2,0.5,'test.wts'));                      {create perceptron in heap}
...

(*-------------------------------------------------------------------------------------------------------*)

{to use and train perceptron}


  with net^ do
    begin
      InpVec[1] := x;                                           {set input(s)}
      Feedforward;                                              {generate output(s)}
      nn := OutpVec[1];                                         {store output(s)}
      DesOutp[1] := y;                                          {set desired output(s)}
      BackPropagate;                                            {backpropagate}
    end;                                                        {theta and alpha may need changing during
                                                                 operation}
(*-------------------------------------------------------------------------------------------------------*)

{to save and load perceptron}

  writeln('Save current network parameters ...');
  ch := 'N';
  repeat
    write('enter filename : ');
    readln(filename);
    with net^ do
       begin
         fname := filename;
         if SaveNet<>0 then
           begin
             write('saving not successful - try again [y/n] : ');
             ch := UpCase(ReadKey);
           end;
       end;
    until (ch='N');

  writeln('Load existing network parameters ...');
  ch := 'N';
  repeat
    write('enter filename : ');
    readln(filename);
    with net^ do
       begin
         fname := filename;
         if LoadNet<>0 then
           begin
             write('loading not successful - try again [y/n] : ');
             ch := UpCase(ReadKey);
           end;
       end;
    until (ch='N');
(*-------------------------------------------------------------------------------------------------------*)
```

Since RBF networks share the same abstract parent object, they are generated and used in a similar way.

2. RECURRENT NETWORKS

This subsection includes the listing of the recurrent network unit as well as an application example.

```
Unit RECNETS;

(*=============================================================================================================*)

INTERFACE

 CONST
    max_m = 10;      {maximum number of external inputs}
    max_n = 15;      {maximum number of nodes}

 TYPE

   recptr = ^recnet;

   recnet = record
     m,n,t : integer;
     x     : array[1..max_m] of real;  {external inputs}
     yd    : array[1..max_n] of real;  {desired outputs}
     y,s,e : array[1..max_n] of real;  {netw. outp., inputs, errors - use 1..t units as ext. outp.}
     z     : array[1..max_m+max_n] of real;  {netw. inp. and outp.}
     W,dW  : array[1..max_n,1..max_n+max_m] of real; {weights}
     p     : array[1..max_n,1..max_n+max_m,1..max_n] of real;
     alpha : real;
     fname : string[12];
    end;

(*-------------------------------------------------------------------------------------------------------------*)

 PROCEDURE INITRec(p:recptr);
 PROCEDURE Normz(p:recptr);
 PROCEDURE ForceLz(p:recptr);
 PROCEDURE RecOut(p:recptr);
 PROCEDURE CalcdW(p:recptr);
 PROCEDURE UpdateW(p:recptr);
 PROCEDURE ContLearn(p:recptr);
 PROCEDURE ForceLearn(p:recptr);
 PROCEDURE ClearP(p:recptr);
 PROCEDURE SaveWeights(p:recptr);
 PROCEDURE ReadWeights(p:recptr);

(*=============================================================================================================*)

IMPLEMENTATION

(*-------------------------------------------------------------------------------------------------------------*)

 FUNCTION Sigmoid(x:real):real;                                  {sigmoidal (logistic) activation function}

  begin
    if x>10 then
      Sigmoid := 0.99999
        else
          if x<-10 then
            Sigmoid := 0.000001
            else
              Sigmoid := 1/(1+exp(-x));
  end;

(*-------------------------------------------------------------------------------------------------------------*)

 FUNCTION KronDelta(i,j:integer):real;                           {Kronecker delta}

  begin
    if i=j then
      KronDelta := 1
      else KronDelta := 0;
  end;

(*-------------------------------------------------------------------------------------------------------------*)
```

```
PROCEDURE INITRec(p:recptr);                                    {creates network}

  var
    i,j,k : integer;

  begin
    randomize;
    with p^ do
      begin
        for i:=1 to max_n do
          begin
            for j:=1 to (max_m+max_n) do
              begin
                W[i,j] := (1-2*random)/10;
                dW[i,j] := 0;
                for k:=1 to max_n do
                  p[i,j,k] := 0;
              end;
            e[i] := 0;
            y[i] := 0;
          end;
        m := m+1;                                               {add constant node for bias}
      end;
  end;

(*-------------------------------------------------------------------------------------------------------------*)

PROCEDURE Normz(p:recptr);                                      {calculate z for normal learning}

  var
    k : integer;

  begin
    with p^ do
      begin
        for k:=1 to n do
          z[k] := y[k];
        for k:=1 to (m-1) do
          z[k+n] := x[k];
        z[n+m] := 1;                                            {constant input}
      end;
  end;

(*-------------------------------------------------------------------------------------------------------------*)

PROCEDURE ForceLz(p:recptr);                                    {calculate z for force-learning}

  var
    k : integer;

  begin
    with p^ do
      begin
        for k:=1 to n do
          if k<=t then
            z[k] := yd[k]
            else z[k] := y[k];
        for k:=1 to (m-1) do
          z[k+n] := x[k];
        z[n+m] := 1;                                            {constant input}
      end;
  end;

(*-------------------------------------------------------------------------------------------------------------*)

PROCEDURE RecOut(p:recptr);                                     {generate output}

 var
   k,l : integer;

 begin
   with p^ do
     begin
       for k:=1 to n do
         begin
           s[k] := 0;
           for l:=1 to (m+n) do
             s[k] := s[k]+W[k,l]*z[l];
         end;
       for k:=1 to n do
         begin
           y[k] := Sigmoid(s[k]);
           if (k<=t) then                                       {if one of the external outputs}
             e[k] := yd[k]-y[k]
             else e[k]:=0;
         end;
     end;
 end;
```

```
PROCEDURE UpDateP(p:recptr);                                    {intermediate calculation for training}
                                                                {called from training procedure}
  var
    tmp : array[1..max_n] of real;
    i,j,
    k,l : integer;

  begin
    with p^ do
      begin
        for i:=1 to n do
          for j:=1 to (m+n) do
            begin
              for k:=1 to n do
                begin
                  tmp[k]:=0;
                  for l:=1 to n do
                    tmp[k] := tmp[k]+W[k,l]*p[i,j,l];
                  tmp[k] := tmp[k]+KronDelta(i,k)*z[j];
                end;
              for k:=1 to n do
                p[i,j,k] := y[k]*(1-y[k])*tmp[k];
            end;
      end;
  end;

(*-------------------------------------------------------------------------------------------------------------*)

PROCEDURE Calcdw(p:recptr);                                     {calculate change to weights for normal}
                                                                {learning - used if changes are to be}
  var                                                           {monitored}
    tmp : real;
    i,j,
    k   : integer;

  begin
    UpDateP(p);
    with p^ do
      begin
        for i:=1 to n do
          for j:=1 to (n+m+1) do
            begin
              tmp := 0;
              for k:=1 to n do
                tmp := tmp+e[k]*p[i,j,k];
              dW[i,j] := alpha*tmp;
            end;
      end;
  end;

(*-------------------------------------------------------------------------------------------------------------*)

PROCEDURE UpdateW(p:recptr);                                    {perform updates-use with above proc.}

  var
    i,j : integer;

  begin
    with p^ do
      begin
        for i:=1 to n do
          for j:=1 to (n+m) do
            W[i,j] := W[i,j]+dW[i,j];
      end;
   end;

(*-------------------------------------------------------------------------------------------------------------*)

PROCEDURE ContLearn(p:recptr);                                  {standard learning routine}
                                                                {use with Normz}
  var
    tmp : real;
    i,j,
    k   : integer;

  begin
    UpDateP(p);
    with p^ do
      begin
        for i:=1 to n do
          for j:=1 to (n+m) do
            begin
              tmp := 0;
              for k:=1 to n do
                tmp := tmp+e[k]*p[i,j,k];
              W[i,j] := W[i,j]+alpha*tmp;
            end;
```

```
      end;
  end;

(*-------------------------------------------------------------------------------------------------------*)

 PROCEDURE ForceLearn(p:recptr);                          {force-learning algorithm}
                                                          {use with ForceLz}
  var
    tmp : real;
    i,j,
    k   : integer;

  begin
    UpDateP(p);
    with p^ do
      begin
        for i:=1 to n do
          for j:=1 to (n+m) do
            begin
              tmp := 0;
              for k:=1 to n do
                tmp := tmp+e[k]*p[i,j,k];
              W[i,j] := W[i,j]+alpha*tmp;
            end;
        for i:=1 to n do                                  {clear the corresponding p's}
          for j:=1 to (n+m) do                            {after using them          }
            for k:=1 to t do
              p[i,j,k] := 0;
      end;
  end;

(*-------------------------------------------------------------------------------------------------------*)

 PROCEDURE ClearP(p:recptr);                              {clear intermediate storage}

  var
    i,j,
    k   : integer;

  begin
    with p^ do
      for i:=1 to n do                                    {clear all p's}
        for j:=1 to (n+m) do
          for k:=1 to n do
             p[i,j,k] := 0;
  end;

(*-------------------------------------------------------------------------------------------------------*)

 PROCEDURE SaveWeights(p:recptr);                         {save current weights to ASCII file}

  var
    i,j : integer;
    outf: text;

  begin
    with p^ do
      begin
        assign(outf,fname);
        rewrite(outf);
        for i:=1 to n do
          begin
            for j:=1 to (m+n) do
              write(outf,W[i,j],'  ');
            writeln(outf);
          end;
        close(outf);
      end;
  end;

(*-------------------------------------------------------------------------------------------------------*)

 PROCEDURE ReadWeights(p:recptr);                         {load weights from ASCII file}

  var
    i,j : integer;
    inf : text;

  begin
    with p^ do
      begin
        assign(inf,fname);
        reset(inf);
        for i:=1 to n do
          begin
            for j:=1 to (m+n) do
              read(inf,W[i,j]);
            readln(inf);
```

```
          end;
        close(inf);
      end;
  end;

(*-------------------------------------------------------------------------------------------------------*)

End.

(*=======================================================================================================*)
```

The following listing shows how to create and utilize a recurrent network using the above procedures.

```
(*-------------------------------------------------------------------------------------------------------*)
{create recurrent network}

uses dos,
     crt,
     recnets;                                          {include unit}

 var
   M  : recptr;                                        {create pointer (or variable)}

....

 begin

   new(M);                                             {create space on heap}
   with M^ do                                          {set network parameters}
     begin
       m := 1;
       n := 2;
       t := 1;
       alpha := 0.5;
     end;
   InitRec(M);                                         {initialize network}
   ClearP(M);
......

(*-------------------------------------------------------------------------------------------------------*)
{use and train network}

                                {force learning}
  M^.x[1] := input;                                    {set network input(s)}
  M^.yd[1] := out_des;                                 {set desired output(s)}
  ForceLz(M);                                          {generate z for force learning}
  RecOut(M);                                           {generate network output(s)}
  ForceLearn(M);                                       {perform learning/optimization}

                                {normal learning}
  M^.x[1] := input;                                    {set network input(s)}
  M^.yd[1] := out_des;                                 {set desired output(s)}
  Normz(M);                                            {generate z for normal learning}
  RecOut(M);                                           {generate network output(s)}
  ContLearn(M);                                        {perform learning/optimization}

(*-------------------------------------------------------------------------------------------------------*)
{save or load network weights}

      {save}
  write('enter filename : ');
  readln(M^.fname);
  SaveWeights(M);

      {load}
  write('enter filename : ');
  readln(M^.fname);
  ReadWeights(M);

{note - no error checking performed}
(*-------------------------------------------------------------------------------------------------------*)
```

3. SIMULATION AND CONTROL

This subsection includes the listing of units and programs used for the tank simulation and control.

3.1 TANK SIMULATION

```
UNIT TANKLEV;

(*=============================================================================================================*)

INTERFACE
 FUNCTION NextLev(y,u,dt:real):real;

(*-------------------------------------------------------------------------------------------------------------*)

IMPLEMENTATION

(*-------------------------------------------------------------------------------------------------------------*)

 FUNCTION NextLev(y,u,dt:real):real;

 { Calcculates the level after time dt given the current level y and input u. To ensure sufficient accuracy,
   an iterative calculation is performed if dt>0.1.

   inputs : y - current level
            u - current input
            dt - time period over which level is to be evaluated

   outputs : NextLev - level at time t+dt }

  var
    tmplev,                                                    { store intermediate level }
    time   : real;                                             { store cumulative time    }
  begin
    time := 0;
    tmplev := y;
    if (dt>0.1) then                                           { for large dt - use iterative calc. }
      repeat                                                   { to improve accuracy                }
        if tmplev<0 then tmplev:=0;
        tmplev := tmplev + 0.1*(0.1*u-0.02*sqrt(tmplev));
        time := time + 0.1;
      until ((dt-time)<0.1);
    if ((dt-time)>1E-10) then                                  { calculate change for remaining time}
      tmplev := tmplev + (dt-time)*(0.1*u-0.02*sqrt(tmplev));
    NextLev := tmplev;                                         { or small initial dt }
  end;

(*-------------------------------------------------------------------------------------------------------------*)

begin
end.

(*=============================================================================================================*)
```

3.2 OPTIMIZATION ALGORITHM FOR NPC

```
UNIT MINMAX;

(*=============================================================================================================*)

INTERFACE

             { Global Definitions }
 const
   np = 10;                {maximum no. of dimensions - optimization space}
   mp = 11;                {points for n-tuple -> np+1                     }

 type
   point    = array[1..np] of real;       {point in optimization space  }
   pointptr = ^point;
   Func     = Function(p:pointptr):real;  {type of func. to be optimized}

 var
```

```
  Min : point;          {global variable - holds initial guess in call }
                        { -returns the minimum point after optimization}
  MinBound,             { minimum and                                   }
  MaxBound : point;     { maximum values for clamping - if clamp=true  }
  bound : boolean;

(*-----------------------------------------*)

 PROCEDURE NelderMead(F:Func;dim:integer);


(*-------------------------------------------------------------------------------------------------------------*)

IMPLEMENTATION

(*-------------------------------------------------------------------------------------------------------------*)

 PROCEDURE clamp(var val:real;index:integer);                    {clamp value to bounds defined in
                                                                  MaxBound and MinBound}
   begin
     if val>MaxBound[index]
       then val:=MaxBound[index]
       else if val<MinBound[index]
              then val:=MinBound[index];
    end;


(*-----------------------------------------------------------------------*)

 PROCEDURE NelderMead(F:Func;dim:integer);

 { The procedure performs a search in the dim dimensional space to find
   the minimum value of function F, starting with a simplex built around
   the initial user supplied guess in min[1]...min[dim]. If clamping is
   active (clamp set to true), the algorithm contains the search for each
   coordinate to the interval [MinBound[i]..MaxBound[i]].
   Based on algorithm in Num. Methods for Pascal.

   inputs : F - function to be optimized
            - define as : function funcname(p:pointptr):real; far;
                     NB far and use p[1]..p[n] inside
              NOTE : if minimum of F is likely to be close to 0
                     add 1 in F to avoid possible numeric overflow

          dim - dimension of optimization space 1<=dim<=10

   other variables to be initialized :
          min[1]..min[dim] - initialize with first guess
          clamp - set to true for clamping/false for none
          MinBound[1..dim] - if clamping is required
          MaxBound[1..dim] -          "

   outputs : min[1]..min[dim] - coordinates at which F is minimum}

  const
    nfuncmax = 100;                                              {no. of evaluations}
    ftol = 0.001;                                                {tolerances for simplex hi/lo}
    alpha = 1;                                                   {scaling for reflection,contraction etc.}
    beta  = 0.5;
    gamma = 2;

  var
    p      : array[1..mp] of point;                              {vertices of simplex          }
    y      : array[1..mp] of real;                               {store function values        }
    count  : integer;                                            {no. of evaluations           }
    ptry,psum,
    pstore             : point;                                  {temp. storage locations      }
    mpts,inhi,ilo,
    ihi,i,eval         : integer;                                {internal storage variables   }
    ytry,ysave,sum,                                              {              "              }
    rtol,diff          : real;                                   {              "              }

  (*-----------------------------------------*)
  procedure MakeSimplex;                                         {create simplex around point min}
    var
      j : integer;
    begin
      Randomize;
      mpts := dim+1;
      for j:=1 to mpts do                                        {initialize all corners to min}
        p[j] := min;
      for j:=1 to dim do                                         {add random amount to one dim. for each}
        begin                                                    {corner except last}
          p[j][j] := min[j]+(random-0.5);
          if bound then clamp(p[j][j],j);
          y[j] := F(@p[j]);                                      {evaluate function for all corners}
        end;
      y[mpts] := F(@p[mpts]);
```

```
  end;
(*-----------------------------------------*)
 function amotry(hi:integer;fac:real):real;                    {try new point and if better then keep it}
   var
     j : integer;
     fac1,fac2,yn : real;
   begin
     fac1 := (1-fac)/dim;
     fac2 := fac1-fac;
     for j:=1 to dim do
        begin
          ptry[j] := psum[j]*fac1-p[hi][j]*fac2;
          if bound then clamp(ptry[j],j);
        end;
     yn := F(@ptry);
     Inc(count);
     if yn<y[hi] then
       begin
         y[hi] := yn;
         for j:=1 to dim do
             psum[j] := psum[j]+ptry[j]-p[hi][j];
         p[hi] := ptry;
       end;
     amotry := yn;
   end;
(*-----------------------------------------*)
 procedure Amoeba;                                             {finds simplex around minimum}
   var
     i,j : integer;
   begin
     count := 0;                                               {initialisation}
     for j:=1 to dim do
       begin
         psum[j] := 0;
         for i:=1 to mpts do
           psum[j] := psum[j]+p[i][j];
       end;
     repeat                                                    {modulate simplex}
       ilo := 1;
       if y[1]>y[2] then
         begin
           ihi := 1;
           inhi := 2;
         end
         else
           begin
             ihi := 2;
             inhi := 1;
           end;
       for i:=1 to mpts do
         begin
           if y[i] < y[ilo] then
             ilo := i;
           if y[i]>y[ihi] then
             begin
               inhi := ihi;
               ihi := i;
             end
             else if y[i]>y[inhi] then
               if i<>ihi then inhi := i;
         end;
       ytry := amotry(ihi,-alpha);                             {reflection}
       if ytry<=y[ilo] then
         ytry := amotry(ihi,gamma)                             {if better try doubling the distance}
         else if ytry>=y[ihi] then
           begin
             ysave := y[ihi];                                  {point was worse so try 1-dim.}
             ytry := amotry(ihi,beta);                         {contraction}
             if ytry>=ysave then                               {not better -> }
               begin                                           {contract around lowest point}
                 for i:=1 to mpts do
                   if i<>ilo then
                     begin
                       for j:=1 to dim do
                           psum[j] := 0.5*(p[i][j]+p[ilo][j]);
                       p[i] := psum;
                       y[i] := F(@psum);
                     end;
                 Inc(count);
                 for j:=1 to dim do
                   begin
                     psum[j] := 0;
                     for i:=1 to mpts do
                       psum[j] := psum[j]+p[i][j];
                   end;
               end;
           end;
       rtol := 2*Abs(y[ihi]-y[ilo])/(Abs(y[ihi])+Abs(y[ilo]));   {compute tolerance}
```

```
    until ((count>nfuncmax) OR (rtol<ftol));                    {stop if tol ok or too many evaluations}
  end;
(*-----------------------------------------*)
 begin                                                          {Main Loop}
   eval := 0;
   repeat
     pstore := min;                                             {store previous guess}
     MakeSimplex;                                               {create new simplex}
     Amoeba;                                                    {find minimum}
     min := p[ilo];
     diff := 0;
     for i:=1 to dim do
       diff := diff+sqr(pstore[i]-min[i])/dim;                  {calculate dist. from previous minimum}
     diff := sqrt(diff);
     Inc(eval);
   until ((diff<1E-2) OR (eval=5));                             {until succ. point close enough or too}
 end;                                                           {many evaluations}

(*-------------------------------------------------------------------------------------------------------------*)

begin
  bound := false;
end.
(*-------------------------------------------------------------------------------------------------------------*)

{ NOTE :
All tolerances and no. of evaluations effect the time taken by the algorithm and form part of the tuning if
this algorithm is used in a control loop. The required accuracy therefore has to be traded off against the
time required to find the next minimum.
Furthermore, in the case of networks, the cost function may not be smooth enough to allow extremely high
accuracy and repeatability and these constants should therefore be relaxed sufficiently to allow the
algorithm to converge.}
```

3.3 CONTROL

The following listings show the code used to perform the control simulation for linear IMC, neural network IMC and NPC.

```
(*=============================================================================================================*)
{procedures and functions}

{linear and combination model}

function modnet(u,y:real):real;                                 {generate network prediction for }
                                                                {current u,y}
  begin
    with modl^ do
      begin
        if y<0 then y:=0;
        if y>25 then y:=25;
        InpVec[1] := 2*u-1;
        InpVec[2] := y/12.5-1;
        Feedforward;
        modnet := OutpVec[1];
      end;
  end;

function linmod(u,y:real):real;                                 {linear model prediction}

  begin
    linmod := exp(-20/250)*y + 15*u*(1-exp(-20/250));
  end;

function model(u,y:real):real;                                  {combination model output}

  begin
    model := modnet(u,y)+linmod(u,y);
  end;

(*-------------------------------------------------------------------------------------------------------------*)

{inverse linear and combination model}

function invnet(ys,y:real):real;                                {inverse network}

  begin
    with inv^ do
      begin
```

```
        InpVec[1] := ys/12.5-1;
        InpVec[2] := y/12.5-1;
        Feedforward;
        invnet := OutpVec[1];
      end;
  end;

function lininv(ys,y:real):real;                                    {linear inverse model}

  begin
    lininv := (ys - exp(-20/250)*y)/(15*(1-exp(-20/250)));
  end;

function inverse(ys,y:real):real;                                   {combination inverse model}

  begin
    inverse := invnet(u,y)+lininv(ys,y);
  end;

(*-------------------------------------------------------------------------------------------------------------*)

function smallgain(r,u:real):real;                                  {small gain algorithm for NNIMC}

  var
    c : integer;
    y : real;

  begin
    c := 0;
    repeat
      y := model(u,yp);
      u := u + 0.1 * (r - y);
      inc(c);
    until ((Abs(r-y)<0.001)OR(c>1000));
    smallgain := u;
  end;

(*-------------------------------------------------------------------------------------------------------------*)

function cost(p:pointptr):real; far;                                {cost function for NPC}
  var
    yn,tmp : real;
    i  : integer;

  begin
    yn := yp;
    tmp := 0;
    e := 0.72*e + 0.28*(yp-ynn);                                    {filtered error}
    for i:=1 to N2 do
      begin
        yn := model(p^[1],yn) + e;
        if (i>=N1) then
          tmp := tmp + sqr(yn-ys);                                  {add errors - for N1<=i<=N2}
      end;
    cost := tmp + 10*sqr(p^[1]-u) + 1;                              {cost function output}
  end;

(*=============================================================================================================*)

{Parts of Main Program}
{all variables folowed by 'o' hold the value of the previous iteration}

  release(HeapOrg);
  new(modl,Init(2,10,1,0.001,'model'));                             {forward model network}
  new(inv,Init(2,10,1,0.001,'inverse'));                            {inverse model network NNIMC only}

  MinBound[1] := 0;                                                 {boundaries for NPC only}
  MaxBound[1] := 1;
  bound := true;
  min[1] := 0;

(*-------------------------------------------------------------------------------------------------------------*)

        yp := NextLev(ypo,u,20);                                    {generate next level}

        {-----------------------------------LINEAR IMC-----------------------------------------}
        ym := exp(-20/250) * ymo + 15*(1-exp(-20/250))*u;           {model prediction}
        e := ys-(yp-ym);                                            {error}
        r := exp(-20/100) * ro + (1-exp(-20/100)) * e;              {filter}
        u := 1/(15*(1-exp(-20/250))) * (r - exp(-20/250)*ro);       {inverse model}
        if u<0 then u:=0;                                           {clamping}
        if u>1 then u:=1;
```

```
{-----------------------------------NEURAL NETWORK IMC-----------------------------------------}
    ym := model(u,ypo)                                    {combination model prediction}
    e := ys-(yp-ym);
    r := exp(-20/100) * ro + (1-exp(-20/100)) * e;
    u := inverse(r,yp);                                   {inverse comb. model}
    u := smallgain(r,u);                                  {smallgain optimisation}
    if u<0 then u:=0;
    if u>1 then u:=1;

    with modl^ do                                         {forward model training}
      begin
        DesOutp[1] := yp-linmod(uo,ypo);                  {desired output = diff. betw. actual and}
        Train;                                            {linear model}
      end;

    with inv^ do                                          {inverse model training}
      begin
        DesOutp[1] := u - lininv(r,yp);                   {desired outp. = optimised u - linear}
        Train;                                            {inverese}
       end;

{-----------------------------------NEURAL PREDICTIVE CONTROL----------------------------------}
    ynn := model(u,ypo)                                   {combination model prediction}
    NelderMead(cost,1);                                   {find optimal input}
    u := min[1];                                          { and implement}

    with modl^ do                                         {forward model training}
      begin
        DesOutp[1] := yp-linmod(uo,ypo);                  {desired output = diff. betw. actual and}
        Train;                                            {linear model}
      end;

(*=============================================================================================*)
```