

Portland State University

PDXScholar

Dissertations and Theses

Dissertations and Theses

1996

Solutions of linear equations and a class of nonlinear equations using recurrent neural networks

Karl Mathia

Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Let us know how access to this document benefits you.

Recommended Citation

Mathia, Karl, "Solutions of linear equations and a class of nonlinear equations using recurrent neural networks" (1996). *Dissertations and Theses*. Paper 1355.

<https://doi.org/10.15760/etd.1354>

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

**SOLUTIONS OF LINEAR EQUATIONS AND A CLASS OF
NONLINEAR EQUATIONS USING RECURRENT
NEURAL NETWORKS**

by

KARL MATHIA

**A dissertation submitted in partial fulfillment of the
requirements for the degree of**

**DOCTOR OF PHILOSOPHY
in
ELECTRICAL AND COMPUTER ENGINEERING**

**Portland State University
1996**

UMI Number: 9628862

UMI Microform 9628862
Copyright 1996, by UMI Company. All rights reserved.

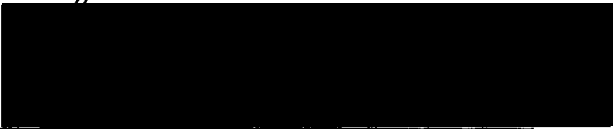
**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

DISSERTATION APPROVAL

The abstract and dissertation of Karl Mathia for the Doctor of Philosophy in Electrical and Computer Engineering were presented May 10, 1996, and accepted by the dissertation committee and the doctoral program.

COMMITTEE APPROVALS:



George G. Lendaris, Chair


Andrew M. Fraser



Y.C. Jenq


Gerardo Laffanjere


Richard E. Sacks


C. William Savery

DOCTORAL PROGRAM APPROVALS:


Rolf Schaumann, Director
Electrical Engineering Ph.D. Program

ACCPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by  on 7 June 1996

ABSTRACT

An abstract of the dissertation of Karl Mathia for the Doctor of Philosophy in Electrical and Computer Engineering presented May 10, 1996.

Title: Solutions of Linear Equations and a Class of Nonlinear Equations Using Recurrent Neural Networks

Artificial neural networks are computational paradigms which are inspired by biological neural networks (the human brain). Recurrent neural networks (RNNs) are characterized by neuron connections which include feedback paths. This dissertation uses the dynamics of RNN architectures for solving linear and certain nonlinear equations.

Neural network with linear dynamics (variants of the well-known Hopfield network) are used to solve systems of linear equations, where the network structure is adapted to match properties of the linear system in question. Nonlinear equations inturn are solved using the dynamics of nonlinear RNNs, which are based on feedforward multilayer perceptrons.

Neural networks are well-suited for implementation on special parallel hardware, due to their intrinsic parallelism. The RNNs developed here are implemented on a neural network processor (NNP) designed specifically for fast neural type processing, and are applied to the inverse kinematics problem in robotics, demonstrating their superior performance over alternative approaches.

To My Parents

ACKNOWLEDGMENTS

I wish to thank Dr. G.G. Lendaris, Dr. R.E. Sacks, and my entire dissertation committee (Dr. A.M. Fraser, Dr. G. Lafferriere, Dr. C.W. Savery, Dr. Y.C. Jenq) for their continuous advice and encouragement, Brad Colbert and Jeff Clark for their software support, Chadwick Cox for our fruitful discussions, Maria Ormaza for her motivation, and the management of Accurate Automation Corporation, in particular Robert Pap, for providing technical resources for this dissertation. This research was funded in part by the National Science Foundation (grant DMI-9321264) and the Office of Naval Research (contract N00014-91-C-0268).

Karl Mathia
Portland, Oregon
May 10, 1996

TABLE OF CONTENTS

	PAGE
ACKNOWLEDGEMENTS	iv
LIST OF TABLES.....	viii
LIST OF FIGURE	ix
CHAPTER	
I INTRODUCTION.....	1
Summary	1
Neural Network Concepts	3
Historical Notes	
Characterization of Artificial Neural Networks	
Mathematical Concepts	
Recurrent Neural Networks.....	9
Parallel Implementation and Inverse Kinematics	12
II SOLUTIONS OF LINEAR EQUATIONS	14
Problem Definition	15
Matrix Norms.....	16
Linear Hopfield Networks.....	18
The Ordinary Linear Hopfield Network	
The Generalized Linear Hopfield Network	
Systems of Linear Equations.....	21
Speed of Convergence	24
Linear Hopfield Networks and Gradient Descent	27
Examples	29
Dynamics of Linear Hopfield Networks	
Eigenvalues and Convergence Speed	

III	SOLUTIONS OF NONLINEAR EQUATIONS.	34
	Preliminaries	35
	Problem Definition	
	Feedforward Neural Networks	
	Assumptions	
	Minimizing a Vector Function.	39
	Minimizing a Scalar Function	41
	RNNs and Gradient Descent	
	Second-order Optimization	
	Newton's Method (2nd-Order)	
	Conjugate Gradient Algorithm	
	Scaled Conjugate Gradient Algorithm	
	Applicable Optimization Problems	
	Example	53
IV	FAMILIES OF FUNCTIONS	56
	Continuation Methods	56
	Concept	
	Solution Sets	
	Problem Definition	63
	Recurrent Neural Networks and Continuation Methods . . .	64
	Example	65
V	IMPLEMENTATION ON NEURAL NETWORK HARDWARE. 69	
	Parallel Processing.	70
	Neural Network Processor.	72
	Benchmarking	74
	Computational Speed and Scaling	
	Quantization Error	
	Recurrent Neural Networks on the NNP	81
VI	SOLUTIONS TO THE INVERSE KINEMATICS PROBLEM. . .	88
	The Inverse Kinematics Problem.	88
	Extendable Stiff Arm Manipulator	90
	Problem Definition	92
	Solution Approaches	93

Stability
Linear Recurrent Neural Network
Nonlinear Recurrent Neural Network

VII	CONCLUSIONS	101
	BIBLIOGRAPHY	105
	APPENDICES	
A	MATHEMATICS FOR CHAPTER 2	112
	Singular Value Decomposition	112
	Proof of Proposition 2.1.	115
	Proof of Proposition 2.2.	116
	Proof of Proposition 2.3.	120
	Proof of Theorem 2.4.	121
	Gradient of a Quadratic Vector Function	123
B	CALCULUS OF MATRIX-VALUED FUNCTIONS	125
C	THE EXAMPLE MLP IN CHAPTER 4	127
	Training and Testing	127
	Learned Weight Matrices.	127
D	EXTENDABLE STIFF ARM MANIPULATOR (ESAM).	129
	3-Dimensional Kinematic Model.	129
	2-Dimensional Kinematic and Dynamic Model	131
E	NNP IMPLEMENTATION OF A RNN	134
F	ROBOT APPROXIMATION IN CHAPTER 6	140
	Training and Testing	140
	Learned Weight Matrices.	141

LIST OF TABLES

TABLE		PAGE
I	Eigenvalues, condition numbers, and convergence speed.	33
II	Conjugate gradient algorithm.	48
III	Scaled conjugate gradient algorithm.	51
IV	RNN Performances for Error Functions $e(x)$, $E(x)$	54
V	Pseudo code of the continuation algorithm.	62
VI	Results for the example.	68
VII	Comparison of Newton's method and RNN.	84
VIII	Matrix inversion: LHN vs. CLAPACK.	87

LIST OF FIGURES

FIGURE	PAGE
1. Mathematical neuron model.	7
2. Neural network architectures: multilayer feedforward, recurrent. . .	8
3. Discrete-time recurrent neural network.	11
4. Linear Hopfield network: structure and neuron model.	18
5. Generalized linear Hopfield networks.	19
6. Eigenvalues of linear Hopfield networks.	26
7. Linear Hopfield networks and gradient descent.	28
8. State trajectories of LHN and GLHN.	31
9. Error surface and error trajectory of LHN and GLHN.	32
10. Multilayer perceptron with one hidden layer.	36
11. Recurrent neural network and Newton's method (first-order). . .	41
12. Recurrent neural network and gradient descent.	44
13. Recurrent neural network and Newton's method (second-order). .	47
14. Recurrent neural network and the SCG algorithm.	52
15. Error surface and definiteness in the example.	55
16. Recurrent neural network and continuation methods.	65
17. Continuation algorithm (left), solution trajectory (right).	67
18. 'Crossing' a solution (left) and a maximum (right).	67

19.	Block diagram of a single neural network processor.	73
20.	NNPs in a parallel multiprocessor environment.	74
21.	Computational complexity of MLP and a recurrent single layer. . .	78
22.	CBS benchmark for the neural network processor (NNP).	79
23.	CBS benchmark for the Intel Paragon.	79
24.	Measuring the NNP quantization-signal-to-noise-ratio.	80
25.	NNP quantization-signal-to-noise-ratio.	82
26.	NNP implementation of the first-order RNN.	83
27.	Speed of matrix inversion: Single NNP, SGI Onyx, SGI 340VGX. .	87
28.	Work and joint space of a 3-joint robot manipulator.	90
29.	Nonlinear inverse kinematics neurocontroller.	94
30.	Linear inverse kinematics neurocontrol.	97
31.	ESAM inverse kinematics: with GLHN (left), and MLP (right). . .	98
32.	Configuration of the Extendable Stiff Arm Manipulator.	131
33.	MLP approximation of the 2-joint ESAM forward kinematics. . .	141

CHAPTER 1

INTRODUCTION

Neural networks, or better artificial neural networks (ANNs), received increasing attention in physics and the engineering sciences over the last decade. The motivation for this interest is the learning capability of ANNs which allows, under suitable conditions, shifting the burden of performing difficult computational tasks from the engineer to the neural network, the ‘learning machine’, in areas such as control, signal processing and pattern recognition. The present work views ANNs from such an engineering perspective. But the concern here is not that of training neural networks. Instead, the particular problem addressed here is to solve the linear and nonlinear systems of equations represented by a class of *trained* neural networks. Solving such equations can be a challenging problem, even when a unique solution exists. An application example where the latter is not the case (i.e. the situation where the system is not invertible) is the classic inverse kinematics problem for redundant robot manipulators. The three main objectives for this research are therefore: 1) To develop recurrent neural networks (RNNs) for solving both linear equations and a broad class of nonlinear equations, and in addition, to apply them to certain optimization problems. 2) To demonstrate the RNN’s fast computational speed when implemented on parallel neural network hardware (NNH). 3) To apply the linear and a nonlinear RNNs to the tracking control of a robot manipulator as a simulation experiment.

1.1 Summary

Chapter 1 discusses the concept of artificial neural networks from an engineering perspective. Although the history of neural network research is briefly reviewed in the following, the biological plausibility of neural engineering models is not relevant for this work. Instead, ANNs are merely treated as a new computational paradigm

which emerged from neurophysiological brain models, and which can facilitate the accomplishment of certain engineering tasks. The properties and concepts of the ANN paradigm are characterized, providing a basis for a generic definition of ANNs. This definition includes the two fundamental classes of network architectures, namely feed-forward and feedback networks. Recurrent neural networks are defined as a special case of feedback networks and are discussed in more detail. This chapter concludes with a preview of implementations of ANNs on specialized parallel neural network hardware, and with a brief discussion of the inverse kinematics problem in robotics, the engineering application used to demonstrate the algorithms developed here.

Solution algorithms for *linear* equations using recurrent neural networks (RNNs) are developed in Chapter 2. Different properties of the system matrix (a linear operator) result in variations of the algorithm. The matrix properties considered are positive definiteness, invertibility, full row or column rank, and singular matrices with none of these properties (i.e. singularity). The algorithm is represented as a *Linear Hopfield Network* (LHN). It is also shown that certain variations in the algorithm result in an LHN architecture augmented with a feedforward structure, which is referred to as the Generalized Linear Hopfield Network (GLHN).

Solution algorithms for *nonlinear* equations are developed in Chapter 3. By extending the linear case above to the nonlinear case, the system matrix (a linear operator) is replaced by the nonlinear operator which defines the class of feedforward neural networks in [Hornik et al. 1989] and [Cybenko 1989]. Assuming that the inverse of the nonlinear ANN mapping f exists, nonlinear RNNs are developed for computing the inverse mapping f^{-1} , i.e. solving the equation $y = f(x)$ for x . Solution to certain optimization problems are also considered.

The linear and nonlinear RNNs in Chapter 2 and Chapter 3 are used for finding single solution *vectors* of functions with *fixed* parameters. In Chapter 4 these algorithms are extended for the solution of families of functions. Recurrent neural net-

works are applied for finding solution *trajectories* $x(t)$ for a family of nonlinear functions $y(t) = f(x(t))$. The algorithm is based on the algorithms in Chapter 3.

The benefits of the algorithms when implemented on parallel machines are discussed and demonstrated in Chapter 5. A Neural Network Processor (NNP), specialized for neural network applications, was used for experimentation. The NNP hardware and software is described, and the NNP performance is evaluated and RNNs developed in the previous chapters are implemented on the NNP. In Chapter 6 the RNNs are applied to the classic inverse kinematics problem in robotics. Here the control objective is to guide the end-effector of a robot manipulator along a prescribed trajectory $x(t)$ in the Cartesian work space. The problem is to determine the corresponding joint angles $\theta(t)$ as control inputs to the robot manipulator in order to accomplish the desired $x(t)$. The solution to this difficult problem is applied to both, non-redundant and redundant manipulators. The conclusions for this dissertation are presented in Chapter 7.

1.2 Neural Network Concepts

1.2.1 Historical Notes

The form and function of the human brain has fascinated researchers over centuries. One of the first contributions to modern neurophysiology was the pioneering work by Ramón y Cajál in 1911, who suggested neurons as basic elements of a complex brain structure [Haykin, 1994]. The neuron model introduced by McCulloch and Pitts (1943) also assumes neurons as the basic computational element of the brain. Their neuron model is still widely used, either in its original or various modified forms, as the basis for computational brain models. In particular, the McCulloch-Pitts neuron included the notion of synaptic connections of neurons as weighting factors, which is a key simplification for the representation of synapses in artificial neural networks. Over the decades our knowledge of the brain *structure* was refined, and

although we are far from a complete understanding of how the human brain functions, it is certain that its organization comprises distributed information processing in a large number of neurons and the extremely rich synaptic connection patterns. This enables the brain to perform complex information processing tasks, which have not yet been accomplished with conventional computers [Hertz et al. 1992].

Knowledge is acquired in the human brain via a *learning* process which occurs in two different ways. The brain of a newly born child shows only sparse connection patterns, but is developed rapidly over the first two years from birth. The connection pattern grows according to the child's perceived environment, with the most dramatic growth occurring during the first few months. This structural development may be viewed as "hard-wiring" the brain structure according to the experience of the individual child [Vester 1974]. The second type of learning employs the modification of synaptic connections between adjacent neurons. The first and most famous learning rule which describes the latter phenomenon was suggested by Hebb (1949). Hebbian learning postulates that the synapse between two simultaneously active neurons (correlated neuron activities) is strengthened. This learning process may be viewed as a "software design" according to the experience of the individual. According to the definition of 'knowledge' by Fischler and Firschein (1987), the stored information is then available for interpreting, predicting, and appropriately responding to the outside world.

Starting in the 1950's, engineers and computer scientists modified available neuron models and learning theories for their specific purposes¹. The architectures of these early ANNs can be separated according to two basic concepts: feedforward networks and feedback network architectures. Feedforward neural networks were applied e.g. by Uttley (1956*) for binary pattern classification, or by Gabor (1954*) for non-linear adaptive filtering. Rosenblatt (1958) invented the classic perceptron, a single neuron, and used it for pattern recognition. He also presented the perceptron conver-

1. Not all original contributions mentioned in this section were available to the author. These are marked by (*), and their descriptions are adopted from historical notes in [Haykin 1994] and [Hertz et al. 1991].

gence theorem, the first convergence proof for a weight adaptation process. Werbos (1974) described a reverse-mode learning algorithm in the context of general networks with neural networks as a special case. This learning method was rediscovered and popularized as the error back-propagation algorithm by Rumelhart, Hinton and Williams (1986). The second class of ANNs, *feedback neural networks*, were first investigated as associative memories. This line of research was established by Grossberg (1967) and by Taylor (1956*), who simulated functional activities of the nervous system with an electric circuit. The correlation matrix memory based on the outer product rule was independently investigated by Anderson, Kohonen, and Nakano in 1972. The brain-state-in-a-box model was proposed in 1977 [Anderson et al. 1977], laying the foundation for Hopfield's famous paper (1982). Hopfield established a class of dynamically stable feedback neural networks which today are called *Hopfield networks*.

1.2.2 Characterization of Artificial Neural Networks

Although the theories of artificial neural networks were motivated by their biological origins, many models and learning algorithms developed from an engineering perspective are not overly concerned about biological plausibility. Over the years, neural network research in the engineering and computer science community deviated (more or less) from its biologically inspired roots. But it is apparent that an explicit objective of neural network research has always been to find principles by which a large number of simple processing units ('local processing') can perform complex computations [Hryci 1992], and by which the knowledge to perform these computations is acquired by a learning process. The neurophysiological models *motivated* the design of artificial neural network paradigms, which usually incorporate the following characteristics:

- *Processing element (local processing)*. Variations of the McCulloch-Pitts neuron model are mostly used as basic ANN processing elements (PEs).

- *Layers of PEs (parallelism).* ANNs are organized in layers of parallel PEs, thus implementing parallel, distributed processing.
- *Distributed memory.* Information is stored in a (large) number of connection weights between layers of PEs.
- *Learning algorithm.* During the learning process, information can be stored either in the connection weights (via *weight adjustment*) or in the structure of the connections (via *variable structure learning*).

Variable structure learning is not well established yet, but is expected to gain increasing attention in ANN research [Baker and Farrell 1992]. Summarizing the above properties, the following definition of artificial neural networks is presented as an extended version of the definition in [Aleksander and Morton 1990].

Definition 1.1: An artificial neural network is a massively parallel processor that has the ability to store experimental knowledge and to make it available for use. It resembles the brain in two respects: 1) Knowledge is acquired through a learning process. 2) The connection weights between neurons, and/or the connection pattern among the neurons are adapted to store the learned knowledge. □

For this work, variable structure learning was added to the above definition cited above. Most learning algorithms do not apply variable structure learning. ANN architectures usually remain unchanged after the designer choice, which is based on experience and intuition. Only a few successful attempts have been made to model and implement the systematic growth of a human brain, see e.g. [Odri et al. 1993], whereas the inverse of this process, weight pruning, has been investigated by many researchers (for a survey of pruning algorithms, see [Reed 1993]). Definition 1.1 applies to feed-forward neural networks as well as feedback and recurrent neural networks.

1.2.3 Mathematical Concepts

This section reviews the relevant mathematical concepts of ANN paradigms for this work. A mathematical neuron model (processing element, or PE), is shown in Figure 1. The PE inputs x_i are multiplied with their corresponding connection weights w_i . In vector notation this refers to the inner product of input vector $x \in \mathbb{R}^p$ and weight vector $w \in \mathbb{R}^p$, resulting in the activation value a (Equation 1.1). The PE output y is given by the activation function (or transfer function) $g(a)$ in Equation 1.2.

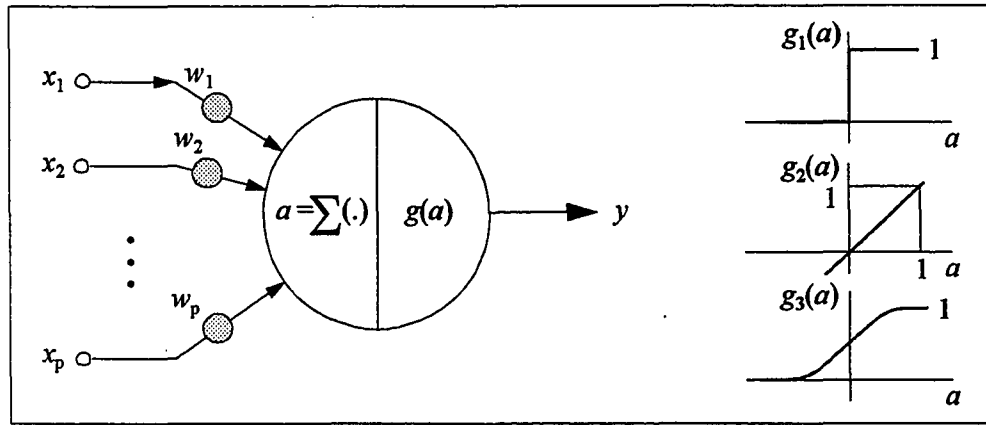


Figure 1. Mathematical neuron model.

$$a = w^T x, \quad (1.1)$$

$$y = g(a). \quad (1.2)$$

Examples of common activation functions are the step function, linear function and sigmoid function, labeled g_1 , g_2 , and g_3 , respectively, in Figure 1. The McCulloch-Pitts neuron originally employed a step function. The linear and sigmoid activation functions used in this work are defined as

$$g_2(a) = a, \quad (1.3)$$

$$g_3(a) = \frac{1}{1 + e^{-a}}. \quad (1.4)$$

The connection pattern between PEs of a neural network is called the *architecture* or *structure* of the ANN. The two ANN structures used for this work are shown in Figure 2, namely a multilayer feedforward network with one hidden layer and a recurrent network with full feedback connection. The Hopfield networks in Chapter 2 have a recurrent architecture, while the particular class of nonlinear functions in Chapter 3 are represented by multilayer perceptrons with a feedforward architecture. Processing elements are represented by shaded circles, input nodes by small circles. The purpose of the input nodes is merely to distribute the inputs signals to the hidden PEs.

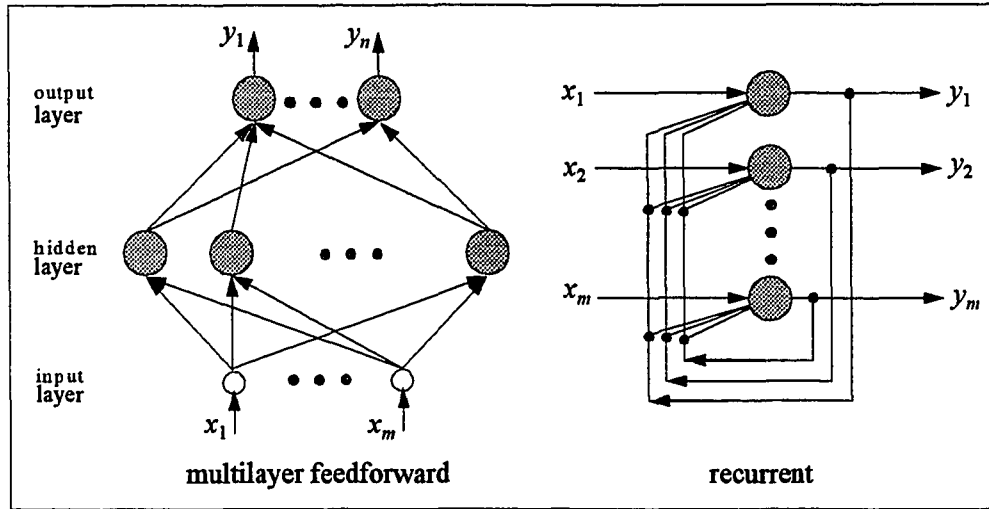


Figure 2. Neural network architectures: multilayer feedforward, recurrent.

The two architectures in Figure 2 provide the building blocks for more complex network architectures explored in the present work. The *multilayer perceptron* (MLP) is usually used as a function approximator. The universal approximator theorem [Hornik et al. 1989], [Cybenko 1989] proves the viability of feedforward ANNs with one hidden layer as a class of arbitrarily accurate universal approximators for a

broad class of functions [Hornik et al., 1989]. Of course, the theorem is only an existence proof, and it remains to design and train an appropriate network. Unsuccessful applications can then either be attributed to an inadequate learning process, an insufficient number of hidden PEs, or the lack of a deterministic relationship between input and desired output of the training data. Barron (1993) established a direct relation between the reduction of the approximation error and the number of hidden PEs. He showed that this reduction is independent of the dimension of the input space. The *curse of dimensionality* does not appear to apply to this class of MLPs, which is not true for series approximation methods such as Fourier series and splines. Although Hornik, et al., (1994) published similar results in the context of artificial neural networks, they “avoid contributing to the mystique associated with neural networks” and point out that it is more accurate to refer to function approximation by superposition of families of convex functions than referring to ‘neural networks’.

The recurrent ANN architecture in Figure 2 is the second building block used for this work. The discrete Hopfield network [Hopfield 1982] has McCulloch-Pitts neurons with binary activation functions. This network can be viewed as a nonlinear autoassociative memory or content-addressable memory, whose primary function is to store and retrieve binary patterns. Hopfield (1984) reports similar properties for the continuous Hopfield network with sigmoid transfer functions. Furthermore, the “energy decreasing” type of dynamics and the smoothness of the network outputs makes the continuous Hopfield network applicable to optimization problems [Tank and Hopfield 1986]. For the present work the *Linear* Hopfield Network (with linear transfer functions) is used.

1.3 Recurrent Neural Networks

The concept of feedback has been introduced to brain models e.g. by Grossberg (1967). In addition, Grossberg (1977) applied Lyapunov functions to the stability analysis of neural network models. Hopfield (1982) used an energy function to

describe the information storage in stable dynamical neural networks. The Hopfield network is now a popular feedback neural network model. Feedback neural networks are distinguished from feedforward neural networks by at least one feedback loop in the network structure [Haykin 1994, p. 20]. But the term ‘feedback’ is rarely used in the ANN literature, where the term ‘recurrent’ neural networks dominates. In the following, continuous-time feedback networks are used as a starting point for defining discrete-time *recurrent* neural networks.

Let a continuous-time dynamical neural network with input x and output y be given by a first-order vector differential equation of the form (the subscript ‘ c ’ denotes continuous time)

$$\frac{dy}{dt} = \dot{y} = f_c(x, y), \quad (1.5)$$

where f_c is *time-invariant* (not explicitly depending on time t). The discrete-time approximation of Equation 1.5 can be obtained by Euler’s forward difference [Drazin 1992, p. 29],

$$\frac{y_{k+1} - y_k}{h} = f_c(x_k, y_k). \quad (1.6)$$

The accuracy of this approximation increases with decreasing h , until identity with the original system is reached in the limit,

$$\dot{y} = \lim_{h \rightarrow 0} \frac{y_{k+1} - y_k}{h}. \quad (1.7)$$

Rearranging Equation 1.6 gives the recursive difference equation

$$y_{k+1} = y_k + h \cdot f_c(x_k, y_k), \quad (1.8)$$

$$= f_d(x_k, y_k). \quad (1.9)$$

The subscript ‘ d ’ in Equation 1.9 means *discrete*. This dissertation considers only discrete-time recurrent ANNs hereafter, and the subscript ‘ d ’ is omitted.

Definition 1.2: A recurrent artificial neural network is the discrete-time analog of the system in Equation 1.5, defined by a difference equation of the form

$$y_{k+1} = F(x_k, y_k), \quad (1.10)$$

where k denotes discrete time. The column vectors $x \in \mathfrak{R}^m$ and $y \in \mathfrak{R}^n$ are called the network *input* and *output*, respectively. The operator F satisfies the generic Definition 1.1 of artificial neural networks. \square

Definition 1.2 implies that the output of a recurrent network at time k_0 only depends on inputs and outputs at past time instants $k < k_0$. It also allows linear and nonlinear operators F . A generic block diagram representation is shown in Figure 3. Memory is implemented with the unit-delay matrix $I_n z^{-1}$, where I_n is an identity matrix of appropriate dimension, thus

$$y_k = [I_n z^{-1}] y_{k+1}. \quad (1.11)$$

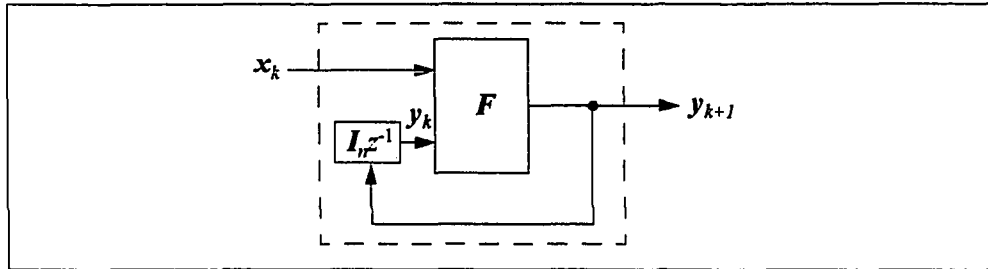


Figure 3. Discrete-time recurrent neural network.

The notion of *feedback* is more general than the notion of *recurrence*. A feedback system can have the form $y_{k+1} = F(x_k, y_k, y_{k+1})$, and thus y_{k+1} can be a function of itself. This is also called a *causal* discrete-time system [Oppenheim and Schaffer 1989, p. 20]. Recurrence can therefore be viewed as a special case of causal-

ity. If the defining equation of a causal ANN can be analytically solved for y_{k+1} , it can be rearranged as a recurrent network in the form in Definition 1.2. If this is not feasible, as is usually true for the nonlinear mapping of ANNs, a numerical solution is needed to compute the network output at each time step. This makes the use of causal ANNs computationally expensive and unattractive, which may be the reason why they do not (yet) appear in the literature.

1.4 Parallel Implementation and Inverse Kinematics

As a minimum requirement, neural networks must satisfy one of the following two conditions for being accepted as tools in the engineering sciences. First, ANNs must enable engineers to solve complex problems for which there are no other, or only unattractive, solutions (yet). Second, compared to conventional methods, the application of ANNs must provide a significant improvement in performance or cost. A key factor here could be that of sufficient computational speed, e.g. real-time capability for control or pattern recognition applications. Since ANN algorithms are inherently parallel in nature, their efficiency and computational speed can greatly benefit from implementations on dedicated parallel hardware. Currently available hardware platforms are reviewed in the following.

Assume that an ANN solution to a particular engineering problem has been designed and that it meets at least one of the above conditions. Then it remains to provide a cost-effective platform for its implementation. Reduced instruction set computers (RISC) are one possible digital approach, despite their usually serial architecture. The design of RISC processors is optimized for a small instruction set, which therefore can be executed at very high speed. Although this technology may be satisfactory for some ANN applications, for complex applications such as real-time image processing, the performance of RISC processors may be not sufficient [Hammerstrom 1992]. Very-large-scale integrated (VLSI) circuits are often suggested as a medium for the hardware implementations of ANNs, either analog or digital. VLSI technology can

provide the hardware complexity required for computational paradigms like ANNs [Mead 1989]. A drawback of this approach is the high research and development costs.

An attractive alternative regarding both performance and cost are neural network processors which consist of off-the-shelf hardware components and which are optimized for the intrinsic parallelism of neural networks. Such a neural network processor (NNP) was available for this work [Saeks et al. 1994]. The NNP is a multiple-instruction multiple-data (MIMD) processor and combines the advantages of a RISC-like instruction set and of parallel hardware (in a cost-effective way). Its architecture is designed particularly for the implementation of a variety of ANN algorithms and performs these computations at very high speed. The RNNs developed here were implemented and tested using the NNP.

The RNNs developed here are applied to the inverse kinematics problem, which arises in the control of robot manipulators. Here the usual control objective is to guide the end-effector along a desired trajectory $x(t)$ in Cartesian workspace, using the joint angles θ as control inputs. Given $x(t)$, the problem then is to determine the corresponding joint angle trajectory $\theta(t)$. The algorithms developed in this work are applied to the inverse kinematics control of different manipulator models, a non-redundant and a redundant one. The accuracy of simulation results obtained demonstrate the efficiency of the numerical methods.

CHAPTER 2

SOLUTIONS OF LINEAR EQUATIONS

Two similar artificial neural networks, the *linear Hopfield network* and the *generalized linear Hopfield network*, are presented in the following and used to solve systems of linear equations. Such systems can be solved using either analytical methods or iterative methods. Analytical methods follow a predesigned sequence of computations, e.g. the Gaussian elimination, therefore their computational cost is known *a priori*. In contrast, *iterative* solutions start with a ‘best-guess’ solution and, if successful, iteratively improve this approximation until it is sufficiently close to the exact solution. Here the computational cost, e.g. the number of iteration steps, is not known in advance. The idea of iterative methods goes back at least to Isaac Newton (1643-1727), but a computational platform for their efficient implementation did not exist until the invention of digital computers.

The Hopfield network was originally designed as a continuous-time discrete-state system [Hopfield 1982], but was later extended to various combinations of continuous-state, discrete-state, continuous-time and discrete-time systems. The *discrete* Hopfield network has McCulloch-Pitts neurons with binary activation functions, whereas the *continuous* Hopfield network employs sigmoidal activation functions [Hopfield 1984]. Wang and Li (1994) recently applied a continuous-time, continuous-state linear Hopfield network (LHN) to solve well-defined linear systems, and implemented the network as an analog circuit. The infinite dimensional case was solved by Sandberg (1994). In the present work the discrete-time *linear* Hopfield network (wherein the usual nonlinear activation function is replaced with a linear one) is presented and is used to solve systems of linear equations. The main result is that the generalized linear Hopfield network (GLHN), an LHN augmented with an additional feedforward layer, can compute the Moore-Penrose generalized inverse. As such, the GLHN solves arbitrary rank deficient linear equations. Both algorithms, LHN and

GLHN, are well suited for parallel machines, due to the parallel nature of neural networks.

In the following the problem for this chapter is defined, and various configurations of linear Hopfield networks are presented, depending on the properties of the linear system to be solved. As opposed to the states of networks with nonlinear ‘squashing’ functions, the states of LHNs are not guaranteed to lie in a compact set, and thus an alternative stability theory is presented to provide convergence to the solutions being sought. (Results of this section were in part obtained during a joint effort with Dr. G. Lendaris and Dr. R. Saeks [Lendaris, Mathia and Saeks 1995].)

2.1 Problem Definition

Systems of simultaneous linear equations of the form

$$y = Ax, \quad (2.1)$$

are considered, where $y \in \mathcal{R}^m$ is a given column vector, $A \in \mathcal{R}^{m \times n}$ is the system matrix, and $x \in \mathcal{R}^n$ is a vector of unknowns. The problem to be solved is twofold:

- Assuming A is positive definite ($A > 0$): find the unique solution x using an LHN.
- Assuming the rank of A is $r(A) < n$ and/or $r(A) < m$: using a GLHN network, find a solution x which is optimal in some sense (a unique solution does not exist).

The discrete-time LHN or the GLHN are the computational paradigms used to accomplish both tasks. The choice of network depends on the properties of A :

- A is square, symmetric and positive definite:

$$A \in \mathcal{R}^{n \times n}, A = A^T, x^T A x > 0 \text{ with } x \neq 0. \quad (2.2)$$

- A is square and invertible:

$$A \in \mathfrak{R}^{n \times n}, A^{-1} \text{ exists.} \quad (2.3)$$

- A is rectangular with full row or column rank:

$$A \in \mathfrak{R}^{m \times n}, r(A) = m, \text{ or } r(A) = n. \quad (2.4)$$

- A is row and column rank deficient:

$$A \in \mathfrak{R}^{m \times n}, r(A) \leq n \text{ and/or } r(A) \leq m. \quad (2.5)$$

2.2 Matrix Norms

Matrix norms needed for constructing LHNs are reviewed in the following.

Given two matrices, $A, B \in \mathfrak{R}^{m \times n}$, a matrix norm is a mapping $h: \mathfrak{R}^{m \times n} \rightarrow \mathfrak{R}$ with the following three properties (see, e.g., [Golub and van Loan 1989, p. 56]):

$$h(A) \geq 0, \text{ where } h(A) = 0 \Leftrightarrow A = 0, \quad (2.6)$$

$$h(A + B) \leq h(A) + h(B), \quad (2.7)$$

$$h(kA) = |k| \cdot h(A), k \in \mathfrak{R}, \quad (2.8)$$

The usual double bar notation is used here to designate a norm, $\|A\| = h(A)$. Common matrix norms are Hölder norms, which are induced by vector norms. Induced matrix norms can be viewed as the maximum ‘gain’ of a matrix A which, for a system $y = Ax$, is measured by the ratio of the norms of ‘input’ vector x and ‘output’ vector y , taken over all possible $\|x\|$:

$$\|A\| = \max_{\|x\|} \frac{\|Ax\|}{\|x\|}. \quad (2.9)$$

It is an important question for applications if $\|A\|$ can be computed efficiently on digital machines. It is not feasible to search over all input vectors with unity norm and to compare the norms of the corresponding outputs Equation (2.9). Algorithms for computing Hölder norms without such a search are available only for the 1, 2 and

∞ -norm [Daily 1991, p. 7]. For this reason, and for the special case in the present work, it is convenient to use the trace as an upper bound for the 2-norm of a square, symmetric, positive semi-definite matrix A , i.e. the spectral radius, $\rho(A) \leq \text{tr}(A)$. This is used here for the construction of linear Hopfield networks. The trace can be efficiently computed on digital machines. Compared to 1, 2 and ∞ -norms, its computation requires the least number of floating point operations. In particular, it does not require floating point *multiplications*.

For the construction of generalized linear Hopfield networks the singular value decomposition (SVD) is a means to prove the computation of the Moore-Penrose *pseudo-inverse* A^\dagger , which, in the real case, satisfies the four conditions [Rao and Mitra 1971, p. vii],

$$AA^\dagger A = A, A^\dagger AA^\dagger = A^\dagger, (AA^\dagger)^T = AA^\dagger, (A^\dagger A)^T = A^\dagger A. \quad (2.10)$$

The definition of the SVD is presented in Appendix A.1 and begins with the polar decomposition of the matrix A , $A = P \cdot M$, where P is the ‘phase’ and M is the ‘magnitude’ of A . The singular values are defined as the eigenvalues of M , that is $\sigma_i(A) \equiv \lambda_i(M) > 0$. The SVD is often presented in the form $A = U\Sigma V^*$, where, in the general case, A is an arbitrary $m \times n$ matrix, and Σ is a $m \times n$ diagonal matrix of the singular values σ_i . U and V are unitary matrices of appropriate dimensions. Using the SVD, the pseudo-inverse is $A^\dagger = V\Sigma^{-1}U^T$, see e.g. [Maciejowski 1989, p.79] and [Rao and Mitra 1971, p. 62]. Zero singular values are often removed from Σ in the representation in Equation (), together with the corresponding rows and columns of U and V .

2.3 Linear Hopfield Networks

One possible approach to solve the system $y = Ax$ numerically methods is to define the process $\Delta x_k = x_{k+1} - x_k = -Ax_k + y$, or

$$x_{k+1} = (I-A)x_k + y, \quad k = 0, 1, \dots \quad (2.11)$$

This iterative process is a Jacobi iteration, which is largely of theoretical interest, because the condition for convergence, i.e. $\|I - A\| < 1$ for some matrix norm, is not assured [Kreyszig 1989, p.1019]. This convergence problem does not apply to the discrete-time linear Hopfield networks presented in the following.

2.3.1 The Ordinary Linear Hopfield Network

The purpose of the linear Hopfield network (LHN) is to solve well-defined linear equations whose system matrix has the properties in Equation (2.2). The dynamics of LHNs are guaranteed to converge to the unique fixed point, i.e. the solution of the linear system in question. The discrete-time LHN is defined as follows, consistent with the definition of ANNs (Definition 1.1) and recurrent neural networks (Definition 1.2).

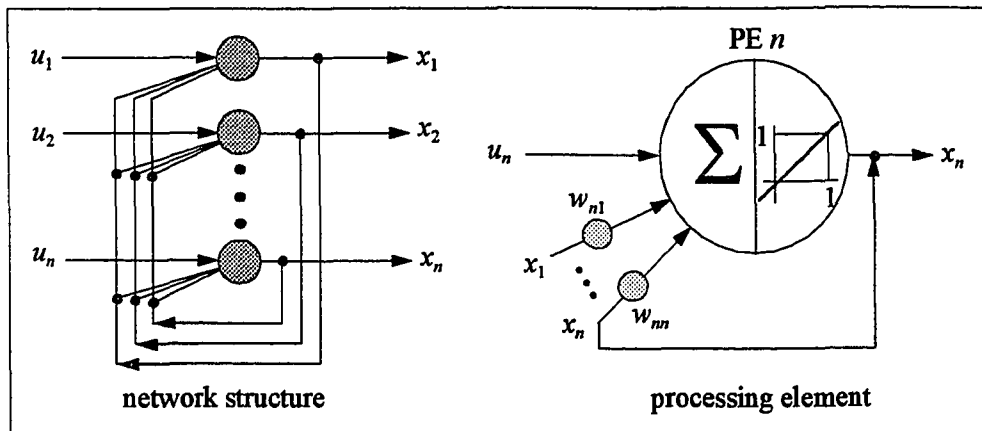


Figure 4. Linear Hopfield network: structure and neuron model.

Definition 2.1: The dynamics of the discrete-time linear Hopfield network are defined by the iterative process

$$x_{k+1} = Wx_k + u, \quad k = 0, 1, \dots, \quad (2.12)$$

where $W \in \mathbb{R}^{n \times n}$ is a square, symmetric, and positive definite weight matrix, $x \in \mathbb{R}^n$ is the vector of network states, and $u \in \mathbb{R}^n$ is a constant external input vector. \square

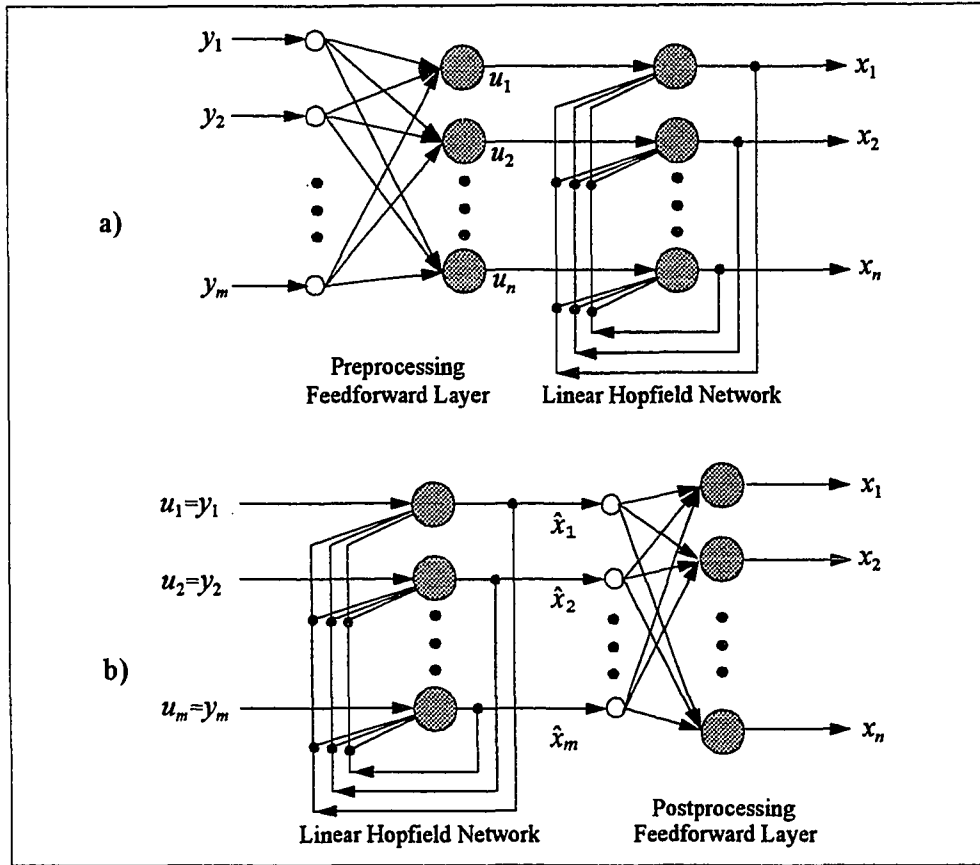


Figure 5. Generalized linear Hopfield networks.

The difference between the Hopfield networks and the LHN is the linear transfer function. The LHN architecture and a processing element (PE) are shown in Figure

4. Of course, W is nonsingular, because otherwise there would be a nonzero x such that $x^T W x = 0$, which violates the definition of positive definiteness. The LHN dynamics in Equation (2.12) are *synchronous*: the complete state vector x_k is updated before it is fed back for the computation of the next state x_{k+1} .

2.3.2 The Generalized Linear Hopfield Network

The purpose of the generalized linear Hopfield network (GLHN) is to solve well-defined linear equation with the property in Equation (2.3), and ill-defined linear equations whose system matrix have the properties in equations (2.4) or (2.5).

Definition 2.2: The generalized linear Hopfield network comprises an LHN and an additional preprocessing feedforward layer of linear elements. Its dynamics are defined by

$$u = W_{\text{FF}} \cdot y,$$

$$x_{k+1} = W \cdot x_k + u, \quad k = 0, 1, 2, \dots,$$

where $W \in \mathcal{R}^{n \times n}$, $W = W^T > 0$. The weight matrix of the feedforward layer, $W_{\text{FF}} \in \mathcal{R}^{n \times m}$, maps the constant external input vector $y \in \mathcal{R}^m$ to the LHN input $u \in \mathcal{R}^n$. \square

If a unique solution does not exist, the least squares solution is determined. This is equivalent to implicitly computing the Moore-Penrose pseudo-inverse and is accomplished by augmenting the LHN architecture with a preprocessing feedforward layer of linear nodes (Figure 5a). Postprocessing feedforward layers (Figure 5b) give the same results, as is discussed in Section 2.4. The two networks, LHN and GLHN, have potential advantages over existing algorithms when implemented on parallel

machines, due to the parallel nature of artificial neural networks. For the implementation on serial computer architectures other algorithms may be more efficient.

The following Proposition 2.1 implies stability and convergence of the LHN (and therefore GLHN). The proposition is a natural outgrowth of the theory of discrete-time systems and linear algebra (see e.g. [Kreyszig 1988, pp. 429, 1017], [Golub and Van Loan 1989, Lemma 7.3.2]). The proof is given in Appendix A.2.

Proposition 2.1: The linear Hopfield networks in Definition 2.1 and Definition 2.2 are asymptotically stable if $\|W\| < 1$ and they converge to a stable equilibrium from any initial condition x_0 . \square

2.4 Systems of Linear Equations

LHNs and GLHNs are presented in the following for solving linear equations $y = Ax$. The cases where A has the properties in equations (2.2) to (2.5) are considered.

A is square, symmetric, and positive definite. The design of LHNs for linear systems with the properties in Equation (2.2) is outlined in Proposition 2.2. The proof is given in Appendix A.3.

Proposition 2.2: Let A be square, symmetric, and positive definite. The linear Hopfield network with $W = I - \alpha A$ and $u = \alpha y$ converges to $x = A^{-1}y$ if $\alpha \in \mathbb{R}$ satisfies $0 < \alpha < 2/\text{tr}(A)$. \square

A is square and invertible. Proposition 2.2 is not applicable if the system matrix is invertible, i.e. square with maximum rank, but is not necessarily positive def-

inite A common technique to transform an indefinite linear system into a positive definite

$$A^T A x = A^T y. \quad (2.13)$$

This equation can be solved with a generalized linear Hopfield network with weight matrix $W = I - \alpha A^T A$ and preprocessing feedforward layer $W_{FF} = \alpha A^T$. Defining $\tilde{A}x = \tilde{y}$ with $\tilde{A} = A^T A$ and $\tilde{y} = A^T y$ gives an equation which can be solved using an LHN according to Proposition 2.2. The LHN will converge to

$$x = (A^T A)^{-1} A^T y = A^{-1} (A^T)^{-1} A^T y = A^{-1} y. \quad (2.14)$$

~~A is rectangular with full rank.~~ In the following it is shown that the GLHN also solves arbitrary equations by implicitly computing the Moore-Penrose pseudo-inverse.

For ill-defined systems, with $A \in \mathbb{R}^{m \times n}$, here the least squares solution can be computed, i.e. the problem is presented as a least squares optimization problem. The case where A has full row rank m or full column rank n is often called the full-rank least squares problem, which can be solved using the Moore-Penrose *pseudo-inverse* A^\dagger (see e.g. [Cichocki et al. 1992, p. 233], [Golub and van Loan 1989, p. 221], [Rao and Mitra 1971, p. 19]). If A has full *column* rank ($m > n$), the least squares solution is

$$x = (A^T A)^{-1} A^T y = A^\dagger y, \quad (2.15)$$

where A^\dagger is also called the *left inverse*. If A has full *row* rank ($m < n$), the solution is obtained using the *right inverse*,

$$x = A^T (A A^T)^{-1} y = A^\dagger y. \quad (2.16)$$

The following proposition implies how to construct a GLHN for computing the pseudo-inverse A^\dagger (the left inverse). The proof is given in Appendix A.4.

Proposition 2.3: Consider the linear equation $y = Ax$ with $A \in \mathfrak{R}^{m \times n}$. The full-rank least squares problem $x = A^\dagger y$ can be solved by

- $m > n$: GLHN 1 with $W = I - \alpha A^T A$, $W_{FF} = \alpha A^T$,
- $m < n$: GLHN 2 with $W = I - \alpha A A^T$, $W_{FF} = \alpha A^T$,

if the convergence rate $\alpha \in \mathfrak{R}$ satisfies $0 < \alpha < 2 / \text{tr}(A^T A)$. \square

The corresponding network architectures are shown in Figure 5. For problems with full column rank the GLHN has a preprocessing feedforward layer is used (Figure 5a), for problems full row rank a postprocessing feedforward layer is used (Figure 5b).

A is arbitrary. The generalized linear Hopfield network is an equation solver for systems with arbitrary A , including matrices with both deficient row and column rank. An (obvious) constraint is finite dimensionality of A . It is shown Theorem 2.4 that both GLHN architectures, i.e. with preprocessing and postprocessing feedforward layer, compute the same solution. The proof of Theorem 2.4 below. The proof is based on the proofs of Theorem 3.5.1 and Corollary 1 in [Rao and Mitra 1971, pp. 62-63] and is presented in Appendix A.5.

Theorem 2.4: With α as in Proposition 2.3, two following GLHNs solve the linear system $y = Ax$ with arbitrary $A \in \mathfrak{R}^{m \times n}$. Their weight matrices are:

$$W = I - \alpha A^T A, W_{FF} = \alpha A^T \text{ (preprocessing feedforward layer),}$$

$$W = I - \alpha A A^T, W_{FF} = \alpha A^T \text{ (postprocessing feedforward layer).}$$

Both networks compute the pseudo-inverse. i.e. $x = A^\dagger y$. \square

If A is invertible either GLHN will compute its inverse:

$$(A^T A)^{-1} A^T = A^{-1} (A^T)^{-1} A^T = A^{-1}, \quad (2.17)$$

$$A^T (A A^T)^{-1} = A^T (A^T)^{-1} A^{-1} = A^{-1}. \quad (2.18)$$

2.5 Speed of Convergence

This section provides an estimate for the convergence speed of linear Hopfield networks as a function of the system matrix A . The computational cost required by an iterative process for solving a numerical problem is not known *a priori* [Kreyszig 1988, pp. 1015], [Golub and van Loan 1989, pp. 508, 512], [Haykin 1994, p. 131]. In the present context this cost can be expressed by the number of iteration steps required, i.e. convergence speed. Appropriate measures to indicate the speed of convergence of iteration methods are investigated in the following. The discussion begins with a scalar system.

The scalar version of the linear system in Equation (2.1),

$$y = a \cdot x, \quad (2.19)$$

with $a, x, y \in \mathcal{R}$, is used to illustrate the concept. An LHN with a one, a scalar input linear PE, a scalar input $u = \alpha y$ and connection weight $w = 1 - \alpha a$, can solve this problem by

$$x_{k+1} = w \cdot x_k + u, \quad k = 0, 1, \dots \quad (2.20)$$

The convergence speed of this ‘network’ increases with decreasing $|w| = |1 - \alpha a| < 1$. For fast convergence a small $|w| \ll 1$ and consequently $|\alpha a| \approx 1$ is desired. It is clear that in the *unforced* multidimensional case (no external input) this rule translates to a fast convergence speed if the spectral radius of W is small in magnitude, i.e. $\rho(W) = \max \{ \lambda_i(W) \} \ll 1$. If ρ is considerably less than 1, the LHN states move rapidly towards the fixed point at each iteration step. It will be investi-

gated in a future project if and how this rule can be transitioned to the dynamics of *forced* systems (nonzero input).

With Theorem 2.4 the eigenvalues of GLHNs may be spread over $0 < \lambda_i(\alpha A^T A) < 2$, or $-1 < \lambda_i(W) < 1$. The *eigenvalue spread* [Messerschmitt 1982], or the *condition number* c , the of a square, symmetric, and positive definite matrix A ,

$$c(A) = \frac{\sigma_{\max}(\alpha A)}{\sigma_{\min}(\alpha A)} = \frac{\lambda_{\max}(\alpha A)}{\lambda_{\min}(\alpha A)} = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}. \quad (2.21)$$

Note that c is independent of α . A condition number $c \approx 1$ indicates that all eigenvalues are of similar magnitude.

It is hypothesized that the larger the condition number c of stable matrices, the slower is the convergence speed of the iterative process [Golub and Van Loan 1989]. Considering the foregoing discussion, it is, at first glance, surprising that a large condition number of the weight matrix W can indicate a *high* speed of convergence of linear Hopfield networks. Recall that an LHN requires that A is symmetric and positive definite. The construction of a GLHN involves the quadratic term $A^T A$ and would unnecessarily square and increase c and therefore reduce the convergence speed, since $\sigma^2(A) = \lambda(A^T A)$ [Green and Limbeer 1995, p. 28]:

$$c(A^T A) = (c(A))^2. \quad (2.22)$$

With the spectral shift theorem [Kreyszig 1989, p. 1034], the condition number of the GLHN is given by,

$$\frac{\lambda_{\max}(W)}{\lambda_{\min}(W)} = \frac{1 - \lambda_{\min}(\alpha A^T A)}{1 - \lambda_{\max}(\alpha A^T A)}. \quad (2.23)$$

The sequence of modifications imposed on the eigenvalues of A by Equation (2.23) during the construction of a GLHN is illustrated in Figure 6. The eigenvalues of the weight matrix W always lie within the unit circle, although eigenvalues within

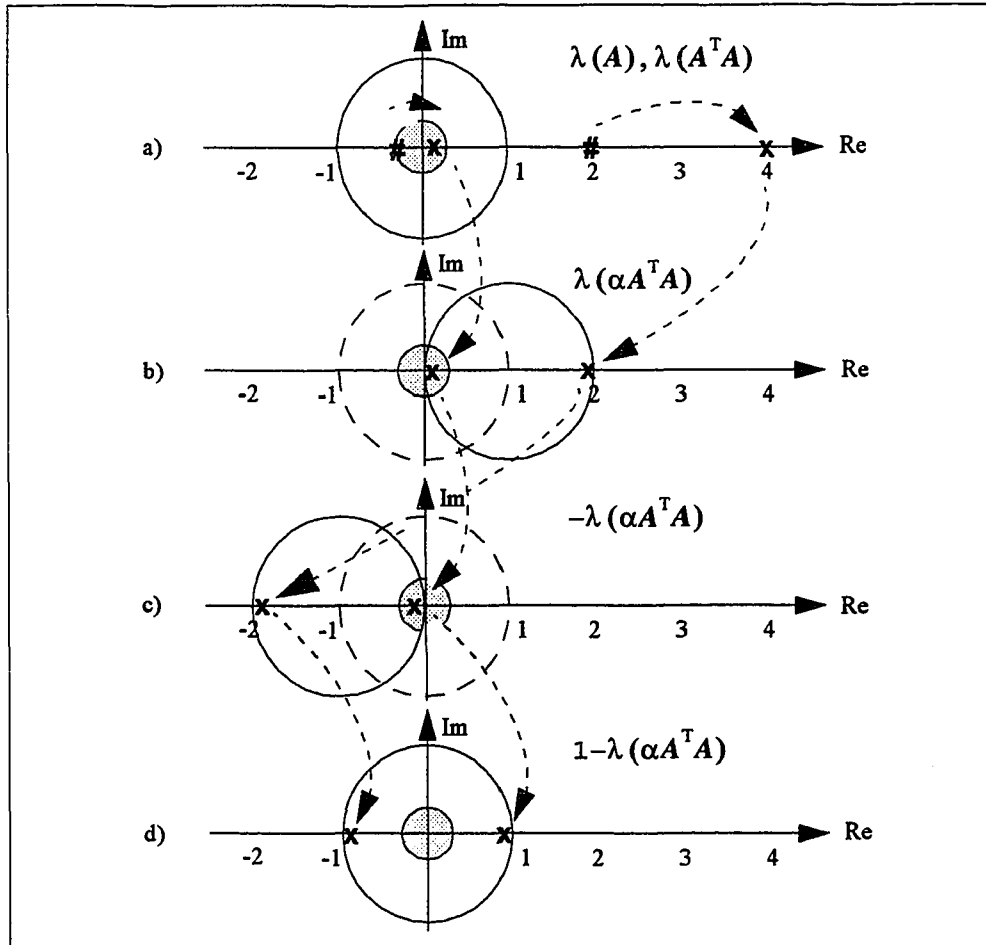


Figure 6. Eigenvalues of linear Hopfield networks.

some vicinity of the origin (shaded area) are preferred for faster convergence. Consider the real minimum and maximum eigenvalue of some $A \in \mathbb{R}^{n \times n}$, in Figure 6a marked by two #’s. Computing the squared term $A^T A$ also squares the eigenvalues, therefore increasing the difference in the eigenvalues’ magnitude. The squared values are shown in Figure 6a as two x’s. Multiplying $A^T A$ with the convergence rate α projects the eigenvalues into the unit circle centered at +1 (Figure 6b). Figure 6c shows the negative eigenvalues, before the spectral shift by +1 results in the stable

weight matrix W (Figure 6d). Both eigenvalues are far from the desired vicinity of the origin, and as a result the GLHN exhibits poor convergence speed. That is despite the ‘good’ condition number of W . Since both eigenvalues are similar in magnitude the GLHN dynamics affect the network states at each iteration step only slightly, resulting in a slow convergence speed. The corresponding numerical example is given in Section 2.7.2.

The above discussion demonstrates that the construction of linear Hopfield networks can improve the condition number of a linear system, which does not necessarily provide fast convergence. It may also be concluded that the use of generalized LHNs should be avoided for square and positive definite A , since the squared matrix moves the eigenvalues of W closer to the unit circle, thus decreasing convergence speed. The ordinary LHN is preferred for these cases. This principle, i.e. choosing an algorithm according to the properties of the system matrix, has also been considered in LAPACK, a state-of-the-art software package for linear algebra problems [Anderson et al. 1995]. The appropriate algorithms provided in LAPACK will be compared against the performance of hardware implementations of linear Hopfield networks in chapter 5.

2.6 Linear Hopfield Networks and Gradient Descent

The dynamics of linear Hopfield networks can also be represented as a gradient method which solves the linear system $y = Ax$. It is a common strategy in artificial neural networks applications to first define a quadratic error function E and then to design an ANN whose dynamics minimize E . Although this was not the underlying principle for the design of linear Hopfield networks, it is shown in the following that linear Hopfield networks also minimize appropriate error functions.

Linear Hopfield network. Let $A \in \mathbb{R}^{n \times n}$ be symmetric and positive definite, and let E be given by the quadratic function

$$E = \frac{1}{2} \mathbf{x}^\top \mathbf{A} \mathbf{x} - \mathbf{x}^\top \mathbf{y}, \quad (2.24)$$

with (see Appendix A.6)

$$\left(\frac{\partial E}{\partial \mathbf{x}} \right)^\top = \mathbf{A} \mathbf{x} - \mathbf{y}. \quad (2.25)$$

E is minimal for the solution vector, which can be obtained using the gradient method,

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha \cdot \left(\frac{\partial E}{\partial \mathbf{x}} \right)^\top = \mathbf{x}_k - \alpha \cdot (\mathbf{A} \mathbf{x} - \mathbf{y}) \quad (2.26)$$

$$= \mathbf{W} \cdot \mathbf{x}_k + \mathbf{u}, \quad (2.27)$$

which is identical to the definition of LHNs (Definition 2.1). Consequently, the LHN performs a gradient method to solve the linear system $\mathbf{y} = \mathbf{A} \mathbf{x}$. If the convergence rate parameter α satisfies the constraint in Proposition 2.2, convergence of the gradient method to the final solution \mathbf{x}_f is assured by sufficiently small ‘steps’ $\Delta \mathbf{x} = \mathbf{x}_{k+1} - \mathbf{x}_k$. This is illustrated in Figure 7 for a scalar x . Starting at (x_0, E_0) , Figure 7a shows an unstable gradient process, while the LHN trajectory in Figure 7b converges to \mathbf{x}_f (convergence of linear Hopfield networks is assured).

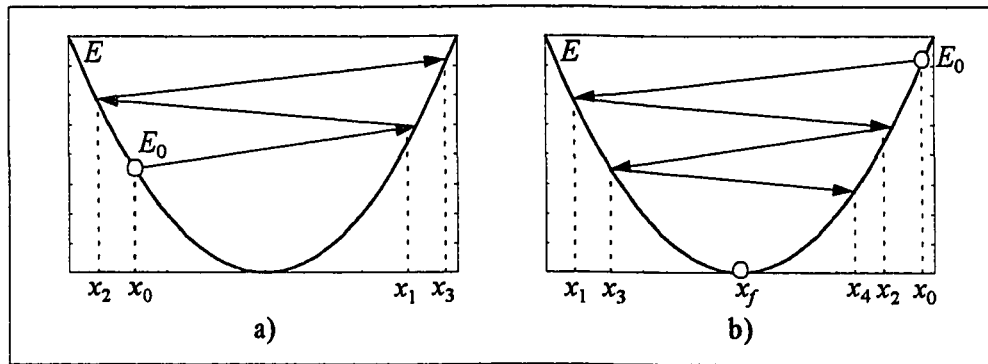


Figure 7. Linear Hopfield networks and gradient descent.

Generalized linear Hopfield network. The least squares problem solved by a GLHN is defined as follows: let $A \in \mathbb{R}^{m \times n}$ have full column rank n , and let E be given by the quadratic function

$$E = \frac{1}{2} \|r\|_2^2 = \frac{1}{2} \|Ax - y\|_2^2. \quad (2.28)$$

The solution vector x minimizes $\|r\|_2$ and therefore minimizes E . The gradient of E with respect to x is used to minimize E , as derived in Appendix A.6,

$$\left(\frac{\partial E}{\partial x} \right)^T = A^T Ax - A^T y. \quad (2.29)$$

E is minimal if the gradient is zero, resulting in the pseudo-inverse solution, $x = A^\dagger y$. The iteration process of the gradient method can be defined as

$$x_{k+1} = x_k + \alpha \cdot \left(\frac{\partial E}{\partial x} \right)^T = x_k - \alpha \cdot (A^T Ax - A^T y) \quad (2.30)$$

$$= W \cdot x_k + u, \quad (2.31)$$

which is the definition of GLHNs (Definition 2.2). Thus, the GLHN performs a gradient method for solving the least squares problem in Equation (2.28).

2.7 Examples

The examples below illustrate the foregoing discussion. The first example shows the iterative network dynamics and its relation with the gradient method. The second example shows the relation between eigenvalues, condition number and convergence speed.

2.7.1 Dynamics of Linear Hopfield Networks

The linear equation to be solved is given in Equation (2.32). The system has a unique solution, since the matrix A is square, symmetric, positive definite.

$$y = Ax, \text{ with } A = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}, y = \begin{bmatrix} 0.5 \\ 1 \end{bmatrix}, \quad (2.32)$$

$$x = A^{-1}y = \begin{bmatrix} -0.5 \\ 1.5 \end{bmatrix}. \quad (2.33)$$

The least square problem therefore does not apply and an appropriate LHN can solve the problem. But in order to compare their respective convergence speeds, of both networks, LHN and GLHN, are used to solve the problem.

Linear Hopfield network. For the LHN design, the convergence rate parameter α is chosen as

$$\alpha = \frac{1.9}{tr(A)} = 0.6333, \quad (2.34)$$

and the corresponding LHN dynamics are given by

$$x_{k+1} = W \cdot x_k + u = \begin{bmatrix} -0.2667 & -0.6333 \\ -0.6333 & 0.3667 \end{bmatrix} x_k + \begin{bmatrix} 0.3167 \\ 0.6333 \end{bmatrix}. \quad (2.35)$$

The eigenvalues of W , $\lambda_1(W) = -0.6581$ and $\lambda_2(W) = 0.7581$, are inside the unit circle and stable. The LHN trajectory in the x_1, x_2 -plane is shown in Figure 8. The trajectory starts at (0,0) and converges in 39 iteration steps to the solution (-0.5,1.5) of Equation (2.32).

The gradients of the error surface E are indicated by small arrows. The same trajectory together with the LHN error surface E is shown Figure 9 in three dimensions.

Generalized linear Hopfield network. For the GLHN design, the convergence rate parameter α chosen as

$$\alpha = \frac{1.9}{tr(A^T A)} = 0.2714, \quad (2.36)$$

and the corresponding GLHN dynamics are given by

$$x_{k+1} = W \cdot x_k = \begin{bmatrix} -0.3571 & -0.8143 \\ -0.8143 & 0.4571 \end{bmatrix} x_k + \begin{bmatrix} 0.5429 \\ 0.4071 \end{bmatrix}. \quad (2.37)$$

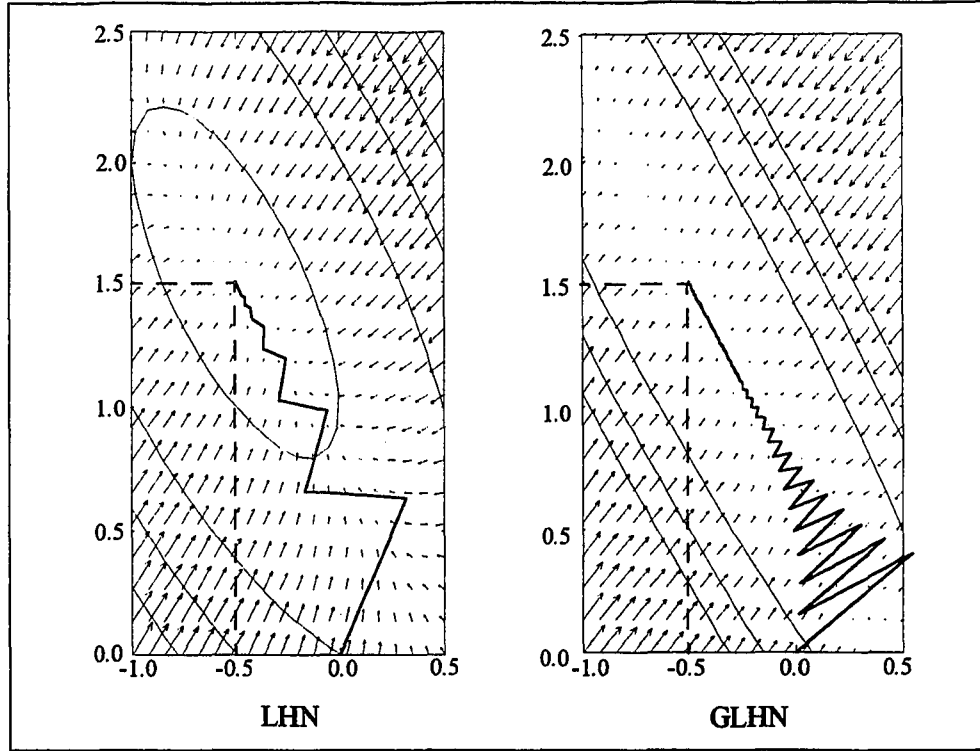


Figure 8. State trajectories of LHN and GLHN.

The eigenvalues of W are $\lambda_1(W) = -0.8604$, $\lambda_2(W) = 0.9604$, which are inside the unit circle and stable. The GLHN-trajectory in the x_1, x_2 -plane is shown in Figure 8. As does the LHN, the GLHN starts at $(0,0)$ and converges to the solution $(-0.5, 1.5)$ of Equation (2.32), but needs 213 iteration steps. The gradients of the error surface are indicated by small arrows. Figure 9 shows the same trajectory together with the GLHN error surface E in three dimensions.

Convergence speed. With Equation (2.21) and Equation (2.22) the condition number of A and $A^T A$ are

$$c(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)} = \frac{2.6180}{0.3820} = 6.8534. \quad (2.38)$$

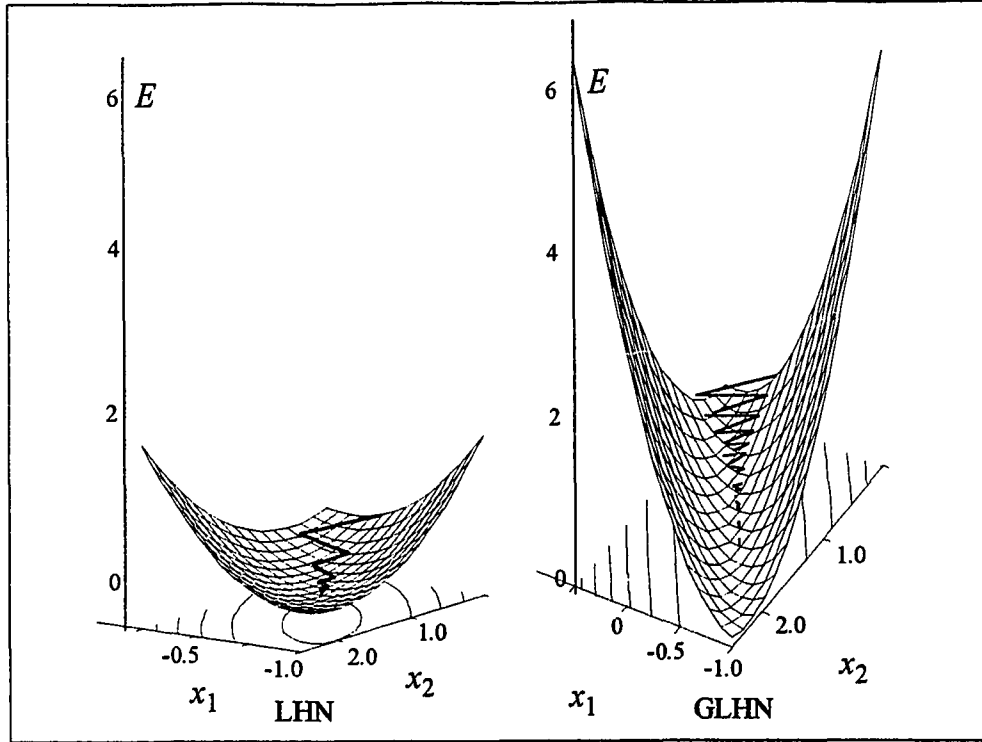


Figure 9. Error surface and error trajectory of LHN and GLHN.

$$c(A^T A) = \frac{\lambda_{\max}(A^T A)}{\lambda_{\min}(A^T A)} = \frac{6.8541}{0.1459} = 46.9781. \quad (2.39)$$

Figure 8 and Figure 9 show how the greater difference in eigenvalues ‘stretches’ the error surface horizontally and vertically. The error surface is a steeper and longer ‘valley’, in which the network states ‘bounce back and forth’ more often before reaching the minimum. The GLHN needed 213 iteration steps, whereas for the LHN 39 iterations were sufficient to meet the convergence criterion $x_{k+1} - x_k < 10^{-5}$.

2.7.2 Eigenvalues and Convergence Speed

This example illustrates the influence of the A_i ’s eigenvalues on condition number and convergence speed of linear Hopfield networks. Consider the two linear

equations $y = A_i x$, $i = 1, 2$, where the system matrices and the constant 'output' vector are

$$A_1 = \begin{bmatrix} -0.3 & 0.1 \\ 0.1 & 2 \end{bmatrix}, A_2 = \begin{bmatrix} -1.8 & 0.1 \\ 0.1 & 2 \end{bmatrix}, y = \begin{bmatrix} 1 \\ 1 \end{bmatrix}. \quad (2.40)$$

The iterative solutions of the two linear systems, system 1 (A_1) and system 2 (A_2), together with eigenvalues and condition numbers of the matrices involved in the specification of the employed GLHNs are given in Table I. The convergence rate parameter was set to $\alpha = 1.9 / (A^T A)$. The example for system 1 is sketched in Figure 6, Section 2.5. It is evident that the condition number of the poorly conditioned A_1 is improved by the construction of a corresponding GLHN (weight matrix W). It is also demonstrated that the condition number of the well conditioned A_2 is worsened. But yet the convergence speed is indicated by the condition of the A_i . The condition of W can be misleading (shaded row in Table I).

Table I. Eigenvalues, condition numbers, and convergence speed.

	System: $y = A_1 x$		System: $y = A_2 x$	
	$\lambda(\cdot)$	$c(\cdot)$	$\lambda(\cdot)$	$c(\cdot)$
A_i	[-0.304, 2.004]	6.586	[-1.803, 2.003]	1.111
$A_i^T A_i$	[0.093, 4.017]	43.374	[3.250, 4.011]	1.234
$\alpha A_i^T A_i$	[0.043, 1.857]	43.374	[0.850, 1.050]	1.234
$W = I - \alpha A_i^T A_i$	[0.957, -0.857]	1.117	[0.150, -0.050]	3.017
Initial vector	[0.0, 0.0]		[0.0, 0.0]	
Solution	[-3.115, 0.656]		[-0.526, 0.526]	
Iterations	219		7	

CHAPTER 3

SOLUTIONS OF NONLINEAR EQUATIONS

When analytical solutions to nonlinear equations are difficult or impossible to derive, iterative methods may be used. The recurrent neural networks (RNNs) developed in this chapter are one such method. The solution approach involves two steps. 1) A sufficiently accurate approximation of the nonlinear system is constructed using a feedforward neural network. Then 2) the nonlinear system's "equation" is solved by performing a numerical inversion process of the neural network. The numerical process is represented and implemented as a RNN. This solution approach is useful, for example, when a nonlinear system is given only in terms of sample points which serve as training data. The benefit of this methodology is a standardized set of algorithms for solving nonlinear systems, where the first step taken, i.e. parameterizing and training the network, is not of concern here. Numerous training methods have been presented in the literature.

The multilayer perceptron (MLP) is a convenient choice as the approximating neural network, because this feedforward neural network is a universal approximator [Hornik et al. 1989], [Cybenko 1988]. The universal approximator theorem implies that any continuous and nonconstant function can be approximated by an MLP to any degree of accuracy. (Of course, any other approximator with similar properties could equally be applied.) The RNNs developed here are based on both this class of feedforward neural networks and on well-established optimization methods of order n . Following common terminology, we call an optimization method (and therefore an associated RNN) of order n if it requires the n -th derivative for its iteration process, i.e. for the iterative approximation and minimization of the function in question. It is well known that incorporating higher-order information, i.e. up to the n -th derivative, provides a more accurate approximation and can dramatically improve convergence.

Unfortunately, higher-order terms also can make the numerical process unreasonably complex. Many optimization schemes therefore approximate not only the original function, but also its derivatives. This is demonstrated in the following. The efficiency of the proposed RNNs on specialized neural network hardware will be demonstrated in Chapter 5.

In the following sections, the problem for this chapter and the architecture of the MLP are defined. Assumptions are made and minimization techniques for vector and scalar functions are described. Based on these techniques, recurrent neural networks of order one and two are developed. An example illustrates the application of these RNNs. A discussion of their benefits and limitations concludes the chapter.

3.1 Preliminaries

3.1.1 Problem Definition

Consider the system of nonlinear equations, $f_o: \mathcal{R}^n \rightarrow \mathcal{R}^n$, with $y = f_o(x)$, where y and x are given and unknown n -dimensional column vectors, respectively. We assume that the ‘original’ system f_o is sufficiently approximated by a universal approximator f (a ‘multilayer perceptron’) of the structure in Equation 3.2. Thus we can write

$$y = f(x) . \quad (3.1)$$

The problem is to apply recurrent neural networks (RNNs, Definition 1.2) for solving the approximating system in Equation 3.1, i.e. to numerically invert f . Solutions of difference equations such as those implemented by RNNs are also called *fixed points* x_f , which are the analogue of equilibria of differential equations.

3.1.2 Feedforward Neural Networks

Universal Approximator. The class of feedforward neural networks used in this work is the well known *multilayer perceptron* (MLP), although the concept is applicable to any differentiable feedforward architectures. The MLP has been a widely used neural network since the backpropagation learning algorithm was invented [Werbos 1974] and popularized [Rumelhart et al. 1986].

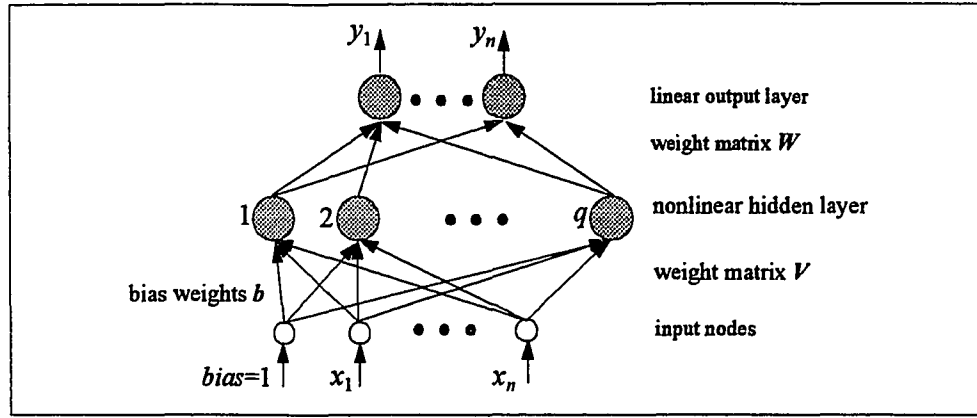


Figure 10. Multilayer perceptron with one hidden layer.

The particular MLP architecture defined in the universal approximator theorem [Hornik et al. 1989], [Cybenko1989] is used for design and evaluation of the RNNs. The network has n input nodes, one hidden layer with q sigmoidal processing elements (PEs), and an output layer with n linear PEs. The MLP is a nonlinear mapping $f: \mathcal{R}^n \rightarrow \mathcal{R}^n$ with input vector x and output vector y ,

$$f(x) = W \cdot \sigma(Vx + b) = W \cdot \sigma(a). \quad (3.2)$$

Figure 3.1 shows the MLP structure. The free parameters of the trained network, the weight matrices of hidden and output layers, $V \in \mathcal{R}^{q \times n}$ and $W \in \mathcal{R}^{n \times q}$, remain fixed for the present problem context. The bias vector b is applied to the hidden layer only, as is in the theorem. We define the activation vector of hidden PEs as

$a = Vx + b$. The (column) vector-valued function $\sigma(a) = [\sigma(a_1), \dots, \sigma(a_q)]$ represents the sigmoid transfer functions of PEs in the hidden layer, with $\sigma(a_i) = 1 / (1 + \exp(-a_i))$. The MLP achieves function approximation by the superposition of a sufficient number of ‘weighted’ basis functions [Haykin1994], [Hornik et al. 1989].

Jacobian and Hessian. The Jacobian matrix (matrix of first derivatives) and the Hessian (matrix of second derivatives) of the MLP in Equation 3.2 are needed for the derivation of the RNNs in the following sections. The Jacobian matrix is

$$\frac{\partial f(x)}{\partial x} = \frac{\partial f}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial a} \cdot \frac{\partial a}{\partial x} = W \cdot \frac{\partial \sigma(a)}{\partial a} \cdot V = W \cdot \Sigma'(a) \cdot V, \quad (3.3)$$

where $\Sigma'(a)$ is a diagonal matrix with elements $\sigma'(a_i) = \sigma(a_i) \cdot (1 - \sigma(a_i))$, the well known form of a sigmoid’s derivative [Haykin1994]. The symbol $\Sigma''(a)$ denotes a diagonal matrix of second derivatives $\sigma''(a_i)$, as defined below in Equation 3.10.

The Hessian is obtained by differentiating the Jacobian using the chain rule and Kronecker tensor products. Let I_q be an identity matrix of dimension q . With Theorem 4.5 in [Brewer 1978], the Hessian is

$$\frac{\partial f(x)}{\partial x^T \partial x} = W \cdot \frac{\partial \Sigma'(a)}{\partial x} \cdot V \quad (3.4)$$

$$= W \cdot \left(\frac{\partial a^T}{\partial x} \otimes I_q \right) \cdot \left(I_1 \otimes \frac{\partial \Sigma'(a)}{\partial a} \right) \cdot V \quad (3.5)$$

$$= W \cdot (V^T \otimes I_q) \cdot \left(\frac{\partial \Sigma'(a)}{\partial a} \right) \cdot V \quad (3.6)$$

$$= W \cdot \begin{bmatrix} V^T & 0 \\ & \ddots \\ 0 & V^T \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial \Sigma'(a)}{\partial a_1} \\ \dots \\ \frac{\partial \Sigma'(a)}{\partial a_q} \end{bmatrix} \cdot V \quad (3.7)$$

$$= W \cdot \begin{bmatrix} v_{11}\sigma''(a_1) & v_{21}\sigma''(a_2) & \dots & v_{q1}\sigma''(a_q) \\ v_{12}\sigma''(a_1) & v_{22}\sigma''(a_2) & \dots & v_{q2}\sigma''(a_q) \\ \dots & \dots & \dots & \dots \\ v_{1n}\sigma''(a_1) & v_{2n}\sigma''(a_2) & \dots & v_{qn}\sigma''(a_q) \end{bmatrix} \cdot V \quad (3.8)$$

$$= W \cdot \Sigma''(a) \cdot V. \quad (3.9)$$

The operator \otimes computes the Kronecker tensor product (Appendix B). The sigmoid's second derivative is obtained as

$$\sigma''(a_i) = \frac{d}{da_i}(\sigma'(a_i)) = \frac{d}{da_i}(\sigma(a_i) \cdot (1 - \sigma(a_i))) , \quad (3.10)$$

$$= \sigma'(a_i) \cdot (1 - \sigma(a_i)) - \sigma(a_i) \cdot \sigma'(a_i) , \quad (3.11)$$

$$= \sigma(a_i) - 3\sigma^2(a_i) + 2\sigma^3(a_i) . \quad (3.12)$$

The equivalent element-wise differentiation gives the Hessian's element in row i and column j as

$$\frac{\partial^2}{\partial x_i \partial x_j} f(x) = W \cdot (\Sigma''(a) \cdot v^i) \cdot v^j , \quad (3.13)$$

where v^i and v^j are the i -th and j -th column vectors in V , respectively. It is clear that the computation of the Hessian (second-order information) even for this relatively simple network architecture is very complex.

3.1.3 Assumptions

The application of RNNs for solving nonlinear systems proposed here requires the following assumptions regarding the universal approximator f in Equation 3.2. These assumptions provide the basis for the development of the recurrent neural networks in the remainder of this chapter:

- f is $\mathcal{R}^n \rightarrow \mathcal{R}^n$, thus locally convex. Thus suitable numerical methods will converge to the nearest fixed point x_f .
- The initial condition x_0 for the numerical method to be used is sufficiently close to x_f .
- f is continuous and nonconstant. This assumption is a direct requirement of the universal approximator theorem. If the original system is continuous and nonconstant, then so is its approximation.
- f is differentiable. i.e. its Jacobian exists.
- $\dim(f) < \dim(\sigma)$, or $n < q$. The number of processing elements in the MLP's hidden layer exceeds the number of input and output elements.

3.2 Minimizing a Vector Function

Newton's Methods (1st-order). Solution methods for vector functions $f(x)$ can be unreasonably complex for high-dimensional systems, and usually are limited to first-order methods. Newton's method for nonlinear systems is one of many iterative algorithms available for finding fixed points of nonlinear systems $y = f(x)$, provided $f: \mathcal{R}^n \rightarrow \mathcal{R}^n$ is continuously differentiable and the starting point x_0 is sufficiently close to the desired fixed point x_f . These conditions are satisfied by the assumptions in Section 3.1.3. Newton's method is usually applied to solve a vector-valued error function $e(x)$ for zero,

$$e(x) = f(x) - y = 0. \quad (3.14)$$

First-order optimization techniques like Newton's assume that a first-order Taylor series expansion is a sufficiently accurate approximation of f about a point x , thus we may write

$$f(x + \Delta x) = f(x) + f'(x) \Delta x, \quad (3.15)$$

where $f' = J$ is the Jacobian, which exists and is continuous according to our assumption in Section 3.1.3. The solution is numerically accomplished by the iteration process

$$x_{k+1} = x_k - J_k^{-1} (f_k - y) = x_k - J_k^{-1} e_k. \quad (3.16)$$

where $J_k = J(x_k)$ and $f_k = f(x_k)$. The fixed point is approached in incremental steps Δx in the direction of the negative local gradients. The corresponding RNN is introduced in the following, and is also implemented on a neural network processor in Chapter 5 as an example for ANN implementations on specialized hardware.

Recurrent Neural Network. Applying Newton's method to an MLP of the form in Equation 3.2 results in a class of RNNs which implement a numerical solution of the function represented by the MLP. The inverse in Equation 3.16 exists for all a , V , and W , because f is n -to- n , Σ'_k is nonsingular, V has full column rank, and W has full row rank. (Σ'_k is nonsingular because it is a diagonal matrix of nonzero sigmoid functions.) This is proven by contradiction: If V did not have full column rank, then there would exist a nonzero x such that $f(x) = 0$. But this contradicts the assumption that f is invertible. The proof for W with deficient row rank follows along the same line. Thus the Jacobian is nonsingular and its inverse exists and the process in Equation 3.16 provides a pattern for designing the class of recurrent neural networks.

Figure 11 shows a block diagram representation of Newton's method (top) and the corresponding RNN (bottom) for iteratively finding the solution x . The RNN comprises a generalized linear Hopfield network and an multilayer perceptron (MLP) as building blocks. The GLHN, and not just the LHN, is needed to compute the generalized inverse of J , since J is not necessarily symmetric and positive definite. The GLHN dynamics are, with feedback weight matrix W_{FB} and feedforward weight matrix W_{FF} ,

$$W_{FF} = \alpha [J(a)]^T, \quad W_{FB} = I - \alpha [J(a)]^T J(a), \quad (3.17)$$

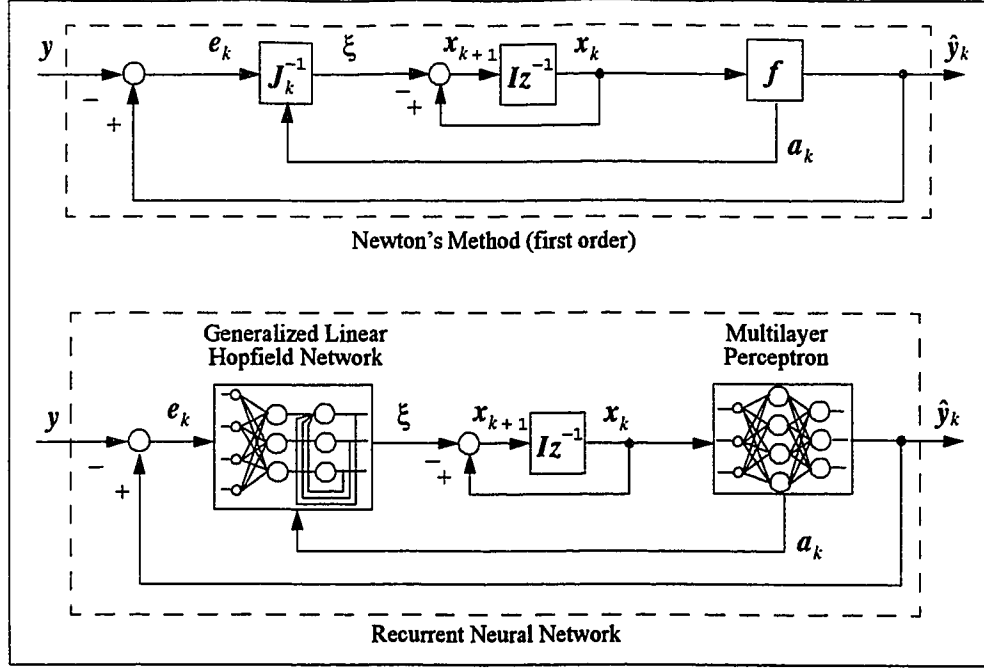


Figure 11. Recurrent neural network and Newton's method (first-order).

with $0 < \alpha < 2/\text{trace}(J^T J)$, and

$$\xi_{i+1} = W_{FB} \cdot \xi_i + W_{FF} \cdot e_k. \quad (3.18)$$

The notation in Equation 3.18 is different from the theory presented in Chapter 2 (Definition 2.1). Here the symbol ξ represents the GLHN state vector, and i is the iteration indicator for the GLHN (not RNN) dynamics. The RNN minimizes the error e_k between the desired y and MLP output \hat{y}_k . At every iteration step k the GLHN implicitly computes the inverse Jacobian. This requires an update of the GLHN weight matrices (via the Jacobian matrix) using the vector of activation values a_k .

3.3 Minimizing a Scalar Function

A common way to solve nonlinear and high-dimensional functions is to define and solve an optimization problem. One can define a scalar, multi-variate error func-

tion $E(\mathbf{x})$ and search for the optimal solution vector \mathbf{x}^* , such that $E(\mathbf{x}^*) = \min \{E(\mathbf{x})\}$. Many numerical optimization schemes are available for this purpose. We follow the same strategy and develop RNNs based on first and second-order optimization methods. The error function used is the usual squared error,

$$E(\mathbf{x}) = 1/2 \cdot \mathbf{e}^T(\mathbf{x}) \cdot \mathbf{e}(\mathbf{x}), \quad (3.19)$$

with $\mathbf{e}(\mathbf{x}) = \mathbf{f}(\mathbf{x}) - \mathbf{y}$. Minimizing this scalar error function solves the nonlinear equation represented by the network, i.e. results in the network input \mathbf{x} for a given desired (constant) output \mathbf{y} . The gradient of E is:

$$\frac{\partial E}{\partial \mathbf{x}}(\mathbf{x}) = \frac{1}{2} \cdot \left(\frac{\partial \mathbf{e}^T}{\partial \mathbf{x}}(\mathbf{x}) \cdot \mathbf{e}(\mathbf{x}) + \mathbf{e}^T(\mathbf{x}) \cdot \frac{\partial \mathbf{e}}{\partial \mathbf{x}}(\mathbf{x}) \right) \quad (3.20)$$

$$= \mathbf{e}^T(\mathbf{x}) \cdot \frac{\partial \mathbf{e}}{\partial \mathbf{x}}(\mathbf{x}) \quad (3.21)$$

$$= \mathbf{e}^T(\mathbf{x}) \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}), \quad (3.22)$$

(with the MLP gradient from Equation 3.3). Equation 3.21 relies on the transpose of scalar (the product of a row and a column vector) is again a scalar, so the product commutes.

With the above gradient, the Hessian of E is:

$$\frac{\partial^2 E}{\partial \mathbf{x}^T \partial \mathbf{x}}(\mathbf{x}) = \frac{\partial}{\partial \mathbf{x}^T} \{ \mathbf{e}^T(\mathbf{x}) \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}) \} \quad (3.23)$$

$$= \mathbf{e}^T(\mathbf{x}) \cdot \frac{\partial^2 \mathbf{f}}{\partial \mathbf{x}^T \partial \mathbf{x}}(\mathbf{x}) + \frac{\partial \mathbf{f}^T}{\partial \mathbf{x}}(\mathbf{x}) \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{x}^T}(\mathbf{x}) \quad (3.24)$$

$$= \mathbf{e}^T(\mathbf{x}) \cdot \mathbf{W} \cdot \Sigma''(\mathbf{a}) \cdot \mathbf{V} + \mathbf{V}^T \cdot \Sigma'(\mathbf{a}) \cdot \mathbf{W}^T \cdot \mathbf{W} \cdot \Sigma'(\mathbf{a}) \cdot \mathbf{V} \quad (3.25)$$

3.3.1 RNNs and Gradient Descent

Algorithm. First-order optimization methods are based on the assumption that a first-order approximation of E about some point \mathbf{x} is sufficiently accurate, thus

$E(\mathbf{x} + \Delta\mathbf{x}) = E(\mathbf{x}) + E'(\mathbf{x})\Delta\mathbf{x}$, where $E' = \partial E/\partial\mathbf{x}$ is the error gradient. The approximating function, not the original, is iteratively minimized by solving for

$$\frac{\partial}{\partial\Delta\mathbf{x}^T}\{E'(\mathbf{x})\Delta\mathbf{x}\} = E'(\mathbf{x}) = \mathbf{0}^T, \quad (3.26)$$

which is accomplished by applying the classical gradient descent method for the MLP, i.e. by approaching zero in incremental steps in the direction of the negative gradients, $\Delta\mathbf{x} = -E'(\mathbf{x})$. With $\Delta\mathbf{x} = \mathbf{x}_{k+1} - \mathbf{x}_k$,

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \cdot (E'(\mathbf{x}))^T \quad (3.27)$$

$$= \mathbf{x}_k - \eta \cdot [f'(\mathbf{x}_k)]^T \cdot \mathbf{e}(\mathbf{x}_k), \quad (3.28)$$

where $\mathbf{a}_k = V\mathbf{x}_k + \mathbf{b}$. The constant scalar η is the ‘step size’. The MLP mapping f and its derivative are given in Section 3.1.2 (see also [Mathia and Saeks 1995]).

A potential drawback of first-order methods is poor convergence, which mainly corresponds to a poor first-order approximation. Techniques have been suggested to include second-order information into the optimization process. For example, a popular first-order technique in the context of artificial neural networks is error backpropagation learning [Werbos 1974], a training algorithm for optimizing the connection weights of a feedforward ANN. ‘Backprop’ can be viewed as an unconstrained nonlinear optimization scheme which combines the classical gradient descent and backsubstitution methods. The usual backprop weight update rule is

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k + \Delta\mathbf{x}_{k-1} = \mathbf{x}_k + \eta E'(\mathbf{x}) + \mu(\mathbf{x}_k - \mathbf{x}_{k-1}), \quad (3.29)$$

where \mathbf{x} denotes a vector of connection weights and the scalar η the learning rate. The ‘momentum term’ $\Delta\mathbf{x}_{k-1}$, with the momentum constant μ , is a (not always successful) *ad hoc* attempt to include *second-order information* in order to increase convergence speed. Second-order methods will be discussed next. Also note that training algorithms like backprop search the *weight space* of a network, whereas we are concerned with searching the *input space*.

Recurrent Neural Network. Figure 11 shows the block diagram of the gradient descent method (top) and the corresponding RNN (bottom). The RNN comprises a multilayer perceptron and its transposed Jacobian in Equation 3.3 as building blocks and minimizes the error e_k between the desired y and MLP output \hat{y}_k at every iteration step k . The MLP Jacobian can be represented as a feedforward network with a structure similar to the MLP itself, but its computation requires the vector of activation values a_k . This is illustrated in the figure.

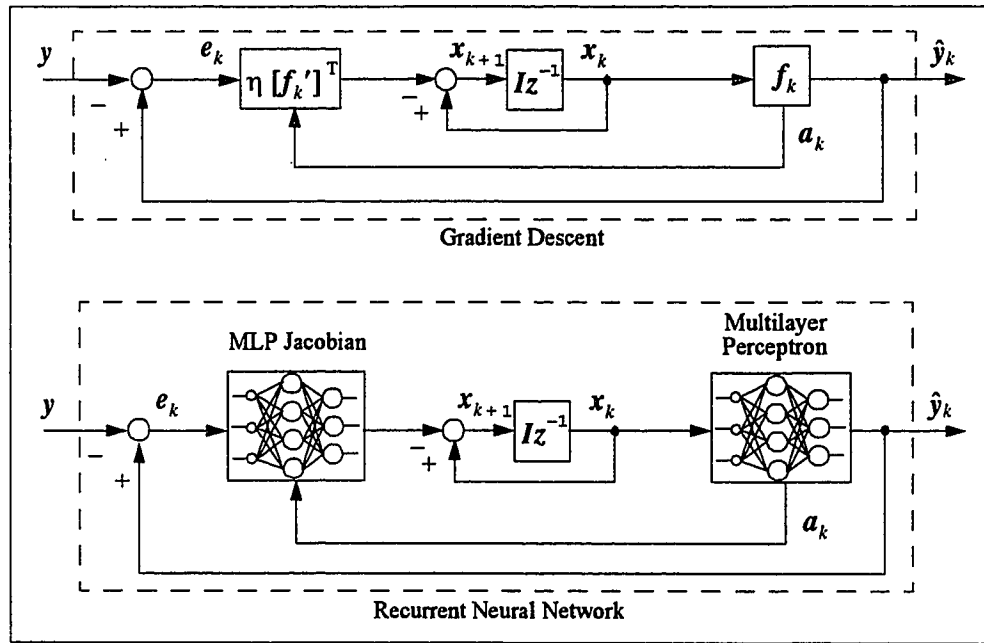


Figure 12. Recurrent neural network and gradient descent.

3.3.2 Second-order Optimization

Recurrent neural networks are designed based on second-order optimization techniques. The objective is an *efficient* algorithm, so second-order optimization techniques of decreasing computational complexity are reviewed. These are ‘decreasing’ in the sense that - depending on the algorithm - the Hessian of a scalar, multi-variate error function is 1) computed *and* inverted (Newton’s method), 2) *only* computed and

its inversion avoided (conjugate gradient algorithm), 3) *both* computation and inversion of the Hessian are avoided (scaled conjugate gradient algorithm). We focus on 3), but also perform the computations in 1) and 2) for comparison purposes (see Section 3.4).

Second-order optimization techniques assume that a second-order Taylor series expansion is a sufficiently accurate approximation of the error function E about a point \mathbf{x} , thus we may write

$$E(\mathbf{x} + \Delta\mathbf{x}) = E(\mathbf{x}) + \Delta E(\mathbf{x}) = E(\mathbf{x}) + E'(\mathbf{x})\Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T E''(\mathbf{x})\Delta\mathbf{x}. \quad (3.30)$$

This assumption is appropriate near extrema, where higher order terms vanish. The gradient E' and the Hessian E'' are given in Equation 3.22 and Equation 3.25 and are assumed continuous. The approximating function is minimized by solving

$$\frac{\partial}{\partial \Delta\mathbf{x}} \{ \Delta E(\mathbf{x}) \} = E'(\mathbf{x}) + \Delta\mathbf{x}^T E''(\mathbf{x}) = \mathbf{0}^T \quad (3.31)$$

for the optimal $\Delta\mathbf{x}^*$, which results in,

$$\Delta\mathbf{x}^* = -[E''(\mathbf{x})]^{-1} \cdot [E'(\mathbf{x})]^T. \quad (3.32)$$

The Hessian E'' must be *positive definite*, which is a stringent constraint as will be demonstrated below. Also note that the Hessian of continuous functions is symmetric, and that the gradient E' is a row vector [Dieudonne 1960]. For the remainder of this chapter we write $\mathbf{g} = E'$ and $\mathbf{H} = E''$.

3.3.3 Newton's Method (2nd-Order)

Algorithm. The key calculation for all second-order optimization techniques based on Newton's methods is Equation 3.32. Newton's method is defined by the iteration process

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{H}_k^{-1} \cdot \mathbf{g}_k^T. \quad (3.33)$$

where $H_k = H(a_k)$ and $g_k = g(a_k)$. One iteration step reduces the approximating function in Equation 3.30. Many iterations may be needed until the minimum of the approximating function is sufficiently close to the minimum of the original function. A drawback of Newton's method and its variants is the computation, storage, and inversion of the Hessian at each iteration step, which can be computationally expensive, in particular for high-dimensional systems. Quasi-Newton methods, such as the conjugate gradient algorithm, avoid these computations.

Recurrent Neural Network. Figure 11 shows the block diagram of Newton's method (top) and the corresponding RNN (bottom). The RNN comprises three networks as building blocks: the function approximating MLP, the transposed MLP Jacobian, and a generalized linear Hopfield network (GLHN) which computes the inverse Hessian. According to Newton's method, the RNN minimizes the error e_k between the desired y and MLP output \hat{y}_k at every iteration step k . As in the previous figure, the MLP Jacobian can be represented as a feedforward network with structure similar to the MLP itself. Both, the computation of the MLP Jacobian and the GLHN require the vector of activation values a_k .

3.3.4 Conjugate Gradient Algorithm

The conjugate gradient (CG) algorithm iteratively computes the inverse Hessian matrix of quadratic functions, while limiting the number of iteration steps to n by searching only along basis vectors (orthogonal axes) of the n -dimensional search space. The vector from starting point ξ_1 to the solution ξ^* is therefore some linear combination of n basis vectors p_i of length α_i (the vector ξ is the CG optimization variable, since the usual symbol x is reserved for the RNN state vector),

$$\xi^* - \xi_1 = \sum_{i=1}^n \alpha_i \cdot p_i. \quad (3.34)$$

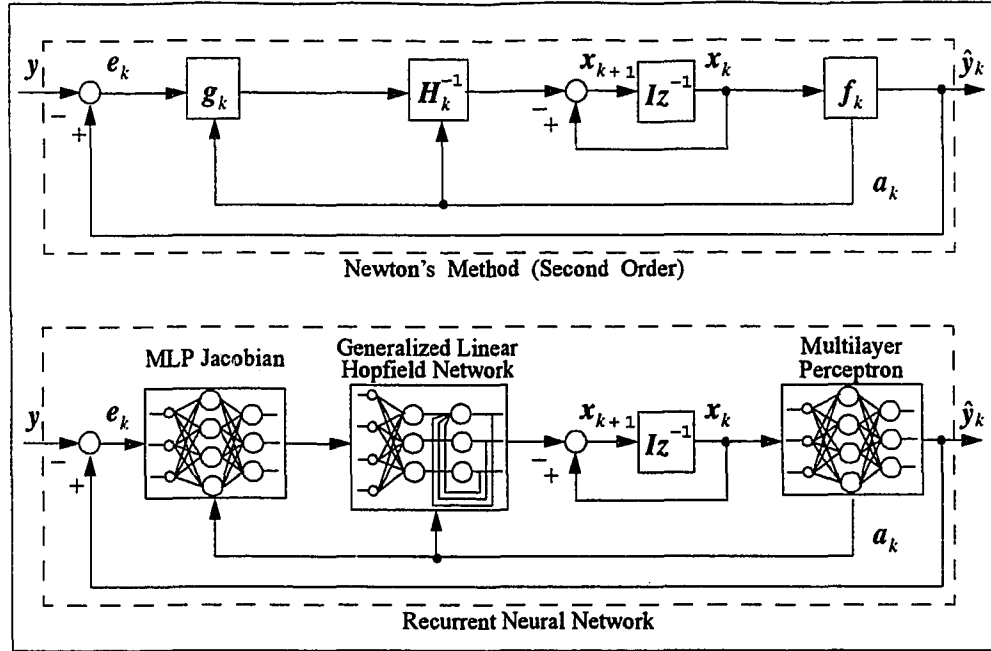


Figure 13. Recurrent neural network and Newton's method (second-order).

The question is: How can we obtain an appropriate set of basis vectors p_i and the associated step sizes α_i ? Fortunately, if the gradient and the Hessian are known at the starting point, these entities can be computed while the algorithm progresses, based on the orthogonality of each new gradient g_{k+1} to the previous search direction p_k [Hestenes 1980]. Which basis is being constructed depends on ξ_1 . The standard CG algorithm is shown in Table II.

Table II. Conjugate gradient algorithm.

Choose ξ_1 (column vector). Compute $H = E''(\xi_1)$, $g_1 = -E'(\xi_1)$.	
Set $p_1 = g_1$ (row vectors).	
LOOP: $i = 1..dim(\xi)$	
$\xi_{i+1} = \xi_i + \alpha_i p_i^T$, with $\alpha_i = p_i g_i^T / p_i H p_i^T$	(3.35.a)
$g_{i+1} = g_i - \alpha_i p_i H$	(3.36.b)
$p_{i+1} = g_{i+1} + \beta_i p_i$, with $\beta_i = p_i H g_{i+1}^T / p_i H p_i^T$	(3.37.c)
end	

One complete loop of the standard CG algorithm is equivalent to one Newton iteration and converges in n steps for *quadratic* functions (of course, the Hessian must be positive definite). The loop must be repeated for nonquadratic functions. Orthogonality of subsequent search directions is maintained with respect to two norms: one is the inner product $p_i g_i^T$, the second is the ‘weighted’ inner product $p_i H p_i^T$. If computation of the Hessian is to be avoided, different estimation techniques are available for α_i and β_i , usually at the cost of slower convergence. The step size α_i is the solution to $\min \{E(\xi + \alpha_i p)\}$ and can be found using a line search. For β_i well known estimates are available [Hestenes 1980], [Møller 1993].

3.3.5 Scaled Conjugate Gradient Algorithm

Algorithm. An elegant way to compute the product $H\xi$ of a Hessian and an arbitrary vector ξ has been independently rediscovered for neural networks in [Møller 1993], [Pearlmutter 1994] and [Werbos 1988]. It is applied here for the numerical inversion, where the *exact* values for α_i and β_i are used. The scaled conjugate gradient (SCG) algorithm circumvents the multiplication of Hessian and search direction, $H p_i$, and computes this product (vector) directly, without computing the Hessian. The idea is to differentiate the function in question (here the MLP f) with respect to a sca-

lar, which greatly facilitates the process. (In the following ξ denotes the SCG variable, since the usual symbol x is reserved for the RNN state vector.) The derivation begins with the first-order expansion of the error gradient,

$$g(\xi + \Delta\xi) = g(\xi) + \Delta\xi^T H. \quad (3.38)$$

Transposing Equation 3.38 and setting $\Delta\xi = r p^T$, with scalar r and (row) vector p , gives

$$r \cdot H p = g(\xi + r \cdot p^T) - g(\xi). \quad (3.39)$$

The desired product $H p$ is obtained by dividing Equation 3.39 by r and taking the limit,

$$p = \lim_{r \rightarrow 0} \frac{g(\xi + r p^T) - g(\xi)}{r} = \frac{\partial}{\partial r} [g(\xi + r p^T)]_{r=0} = R_p \{g(\xi)\}, \quad (3.40)$$

which coincides with the definition of a derivative. The definition of the differential operator $R_p \{g(\xi)\} = H p^T$ is due to [Pearlmutter 1994]. The usual rules for differential operators apply. Here the desired product is obtained by applying this operator to those equations which compute gradients. In the following we simplify the notation to $R\{\cdot\}$.

The SCG algorithm is applied to the numerical inversion of nonlinear functions represented by the MLP in Equation 3.2. The error function is the usual squared error in $E(x)$, and its minimization solves the nonlinear equation represented by the network, i.e. finds the network input x for a given desired output y . The application of Newton's method and the standard CG algorithm is straightforward if the Hessian of E is known. We apply the SCG algorithm for computing $H p$. With $g = E'$ and $f' = \partial f / \partial x$,

$$H p = R\{g(x)\} = R\{e^T(x) \cdot f'(x)\} \quad (3.41)$$

$$= R \{ e^T(x) \} \cdot f'(x) + e^T(x) \cdot R \{ f'(x) \} . \quad (3.42)$$

The two $R \{ . \}$ terms in Equation 3.42 are (with $a = Vx$, $a_p = V(x + rp^T)$, $c = Vp^T$):

$$R \{ e^T(x) \} = (W \cdot \Sigma'(a) \cdot V \cdot p^T)^T, \quad (3.43)$$

$$R \{ f'(x) \} = R \{ W \cdot \Sigma'(a) \cdot V \} = W \cdot R \{ \Sigma'(a) \} \cdot V \quad (3.44)$$

$$= \left(\left[\frac{\partial a_p^T V^T}{\partial r} \right]_{r=0} \otimes I_{\text{row}(\Sigma')} \right) \cdot \left(I_{\text{col}(r)} \otimes \left[\frac{\partial \Sigma'(a_p)}{\partial a_p} \right]_{r=0} \right) \quad (3.45)$$

$$= (c^T \otimes I_q) \cdot \left(1 \otimes \frac{\partial \Sigma'(a)}{\partial a} \right) = \begin{bmatrix} c^T & 0 \\ & \ddots \\ 0 & c^T \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial \Sigma'(a)}{\partial a_1} \\ \dots \\ \frac{\partial \Sigma'(a)}{\partial a_q} \end{bmatrix} \quad (3.46)$$

$$= \begin{bmatrix} c_1 \cdot \sigma''(a_1) & & 0 \\ & \ddots & \\ 0 & & c_q \cdot \sigma''(a_q) \end{bmatrix} = \Sigma_c''(a) . \quad (3.47)$$

The operator \oplus performs the Kronecker tensor product [Brewer 1978]. As before, I_q denotes an identity of dimension q . The operators $\text{row}(\cdot)$ and $\text{col}(\cdot)$ return the number of rows and columns of a matrix, respectively. Using the above derivations gives

$$Hp = [p^T V^T \Sigma'(a) W^T W \Sigma'(a) V] + [e^T(x) W \Sigma_c''(a) V], \quad (3.48)$$

which can be numerically optimized for implementation. This is a form of Equation 3.40 based on the specific structure of an MLP defined by Equation 3.2. We need only to substitute the weights and biases of the MLP (V, W, b) into Equation 3.48 and then into the SCG algorithm in Table II. The resulting algorithm is listed in Table III.

Table III. Scaled conjugate gradient algorithm.

Choose ξ_1 (column vector). Compute $g_1 = -E'(\xi_1)$. Set $p_1 = g_1$ (row vectors).	
LOOP: $i = 1..dim(\xi)$	
$Hp_i^T = [pV^T \Sigma'(a) W^T W \Sigma'(a) V] + [e^T(x) W \Sigma_c''(a) V]$	(3.49.a)
$\xi_{i+1} = \xi_i + \alpha_i p_i^T$, with $\alpha_i = p_i g_i^T / p_i H p_i^T$	(3.50.b)
$g_{i+1} = g_i - \alpha_i p_i H$	(3.51.c)
$p_{i+1} = g_{i+1} + \beta_i p_i$, with $\beta_i = p_i H g_{i+1}^T / p_i H p_i^T$	(3.52.d)
end	

Recurrent Neural Network. Figure 11 shows the block diagram of the SCG algorithm (top) and the corresponding RNN (bottom). The RNN comprises the function approximating MLP, the transposed MLP Jacobian, and the SCG block which computes the inverse Hessian. As a variant of Newton's method, the RNN minimizes the error e_k between the desired y and MLP output \hat{y}_k at every iteration step k . As in the previous figures, the MLP Jacobian is represented as a feedforward network with a structure similar to the original MLP. Both, the computation of the MLP Jacobian and the SCG block require the vector of activation values a_k .

3.3.6 Applicable Optimization Problems

A problem associated with the RNNs in Section 3.2 for minimizing vector-valued functions is when the Jacobian is close to a singularity. In chapter 4, continuation methods are used to detect and avoid all singularities. Major constraints of second-order optimization methods are discussed in the following.

Important assumptions must be made when using second order optimization methods. First, the nonlinear function is (at least locally) convex, input and output space have the same dimension and the function has a nonsingular Jacobian. Second, a good estimate of the desired fixed point is required for choosing an initial condition in the positive definite neighborhood of that point. Third, the optimization process must

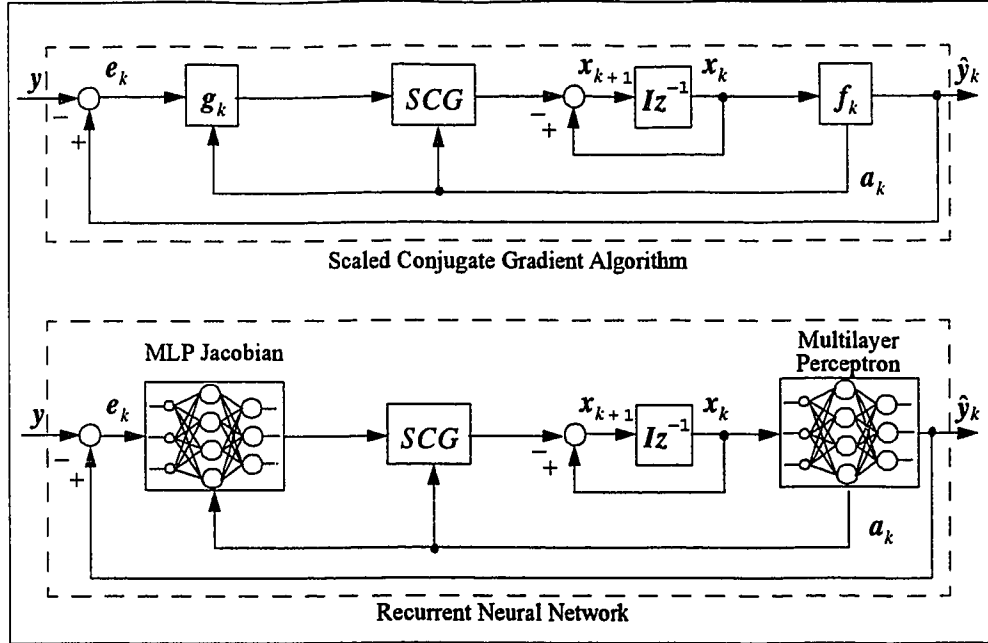


Figure 14. Recurrent neural network and the SCG algorithm.

not leave the positive definite portion of the error surface, otherwise the algorithm is not guaranteed to be stable. This is a stringent constraint, since in the usual case only a small portion of the error surface is positive definite, as is shown in Figure 15 for the example in the next section. If we were to randomly select an initial condition within this ‘mountainous’ area, there is no reason to expect that it will be in that positive definite region.

Techniques have been proposed to overcome some of these difficulties [Hestenes 1980], [Møller 1993], but any additional feature attached to the algorithms will cost computation time, whereas our goal is *efficient* problem solving for time-critical applications. We would rather impose constraints on the class of problems intended to be solved, and therefore assume *a priori* knowledge about the topology of the error function. This is reasonable near minima, where most functions behave like quadratics. Consider a scenario where the minimum is initially known (as in the example) and is used as the starting point. If the application context changes such that its

corresponding error surface moves away from that point a sufficiently small step, then the point is not the minimum any longer, but is still in the positive definite neighborhood of the new minimum. It can therefore be used as the new initial condition. If this process is repeated, the subsequent optimization processes ‘chase’ the moving minimum. An example of this class of problems is the inverse kinematics problem in robotics, where a sequence of joint angles is needed to guide the robot’s end effector along a desired Cartesian trajectory. The minimum is initially known - one simply measures the joint angles. The robot arm can then be moved in small steps, while an optimization process continuously finds the best joint angles, never leaving the positive definite neighborhood of the (moving) minimum.

3.4 Example

We assume a problem context in which the process to be solved is being represented by an MLP (Equation 3.2) which was trained using data from the process (see also [Lendaris and Mathia 1996]). We use the first and second order methods presented above to invert the nonlinear function which is represented by the neural network. In this example we use a two-variable system, and after training, our (two input, two hidden PE, two output) MLP has the following parameter values and given output y :

$$V = \begin{bmatrix} 1.0 & 0.5 \\ 0.6 & 1.5 \end{bmatrix}, W = \begin{bmatrix} 1.0 & 0.5 \\ 0.4 & 1.0 \end{bmatrix}, b = \begin{bmatrix} 0.3 \\ 0.5 \end{bmatrix}, y = \begin{bmatrix} 1.0440 \\ 0.8867 \end{bmatrix}. \quad (3.53)$$

Since this example was constructed for demonstration purposes, the desired fixed point $x_f = (1, -0.5)$ is already known and $x_0 = (0, -0.2)$ can be chosen as a convenient initial condition within the positive definite neighborhood of x_f , as required by the theory. We choose $\|e\|_\infty \leq 0.00001$ as the convergence criterion for the algorithms.

A recurrent neural network based on Newton’s method (first-order) was used to minimize a vector-valued error function for solving the MLP-approximated func-

tion. The usual (scalar) squared error function was minimized using RNNs based on the gradient descent method and all three second-order optimization techniques presented above. From a mathematical perspective all three second-order methods compute precisely the same trajectory, but the efficiency of their implementations varies. Newton's method (Equation 3.33) is the most complex, because it not only computes the Hessian of the error function to be minimized, but also inverts it at each iteration. The standard CG algorithm (Figure II) avoids the matrix inversion, but still needs the Hessian itself. The most efficient second-order technique of the three is the SCG algorithm which uses Equation 3.48 in the algorithm of Figure II, to avoid both, computation and inversion of the Hessian. As shown in the previous section, Equation 3.40 takes the form of Equation 3.48 for the MLP in Equation 3.2. The parameter values of Equation 3.53 were used in the experiment tabulated in Table IV. The table shows the convergence performance of the RNNs based on the optimization techniques above when applied to both the vector-valued error function $e(x)$ and scalar error function $E(x)$. The best convergence performance, most accurate solution, and least amount of floating point operations (flops) was achieved by Newton's method when applied to $e(x)$. This is shown in the first column of the table.

Table IV. RNN Performances for Error Functions $e(x)$, $E(x)$.

	$e(x)$	$E(x) = e^T(x) e(x)$			
	Newton	SCG	CG	Newton	Gradient
Iterations	4	7	7	7	974
Flops	825	1924	2141	2582	53620
Fixed Point	[1.0000, -0.5000]	[0.9999, -0.5000]	[0.9999, -0.5000]	[0.9999, -0.5000]	[0.9987, -0.4991]

The second-order RNNs (SCG, CG, Newton) can only be applied to $E(x)$, since $e(x)$ does not necessarily satisfy the positive definiteness condition. As is well known (and demonstrated here), the second-order techniques offer dramatic improvement in the number of iterations to converge over the first order (gradient descent) technique. It will be noticed there is similarly dramatic improvement in the amount of computation required to achieve the solution. Although this example represents a small-scale problem, we obtained similar results with larger networks. The $E(x)$ surface for our example problem is shown in Figure 15. The minimum (the desired solution) is shown as a black dot. The second figure ‘zooms in’ on the neighborhood of the minimum and shows the definiteness of the surface’s Hessian. Only a small fraction is positive definite (marked with ‘+’). All second order techniques require that the initial estimate (starting point) must be in this region (which was known for this example). In Chapter 5 it will be demonstrated how the RNNs can achieve very fast processing when implemented on special purpose neural network hardware.

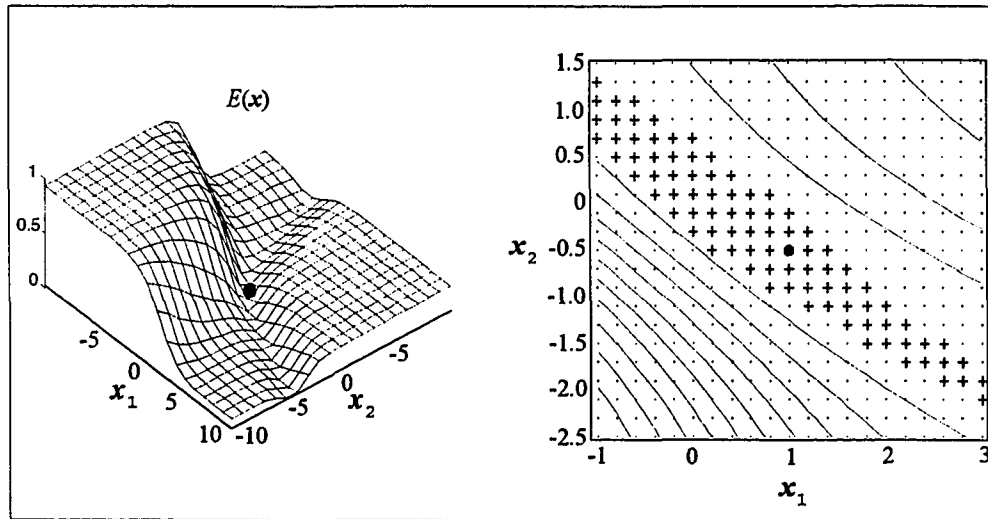


Figure 15. Error surface and definiteness in the example.

CHAPTER 4

FAMILIES OF FUNCTIONS

In the previous chapters, recurrent neural networks (RNNs) were designed for finding only a *single* solution x for a system of nonlinear equations. This (limited) solution approach is extended here to the design of RNNs which find *all* solutions to a given family of algebraic problems. The network design used is based on *continuation methods*, which convert the solution of a family of algebraic problems into a differential equation. The differential equation is constructed such that its trajectory goes through all solutions for the original problem if it starts at some previously known solution. An initial solution can be obtained using well-known classical methods. As throughout this work, the nonlinear system considered is a (trained) multilayer perceptron (MLP). The RNNs are constructed for finding all solutions to the MLP approximation of the given nonlinear problem. In the following sections, continuation methods are reviewed and the problem for this chapter is defined. Then RNNs are implemented based on a suitable continuation algorithm. An example illustrates the solution approach.

4.1 Continuation Methods

4.1.1 Concept

The concept behind all continuation methods is to convert the solution of a continuously parameterized family of algebraic problems into the solution of an appropriate differential equation. The points on the trajectory are the solutions to the given problem and can be obtained by numerical integration if an initial solution is known. Let r be the scalar, independent parameter which is mapped to a constant b

by the algebraic family of problems $P(r)$, and let $s(r)$ be the associated solutions. Thus, $s(r)$ can be viewed as an intermediate ‘state’ variable of the system,

$$b = P_r(s(r)). \quad (4.1)$$

If a solution $s(r_0)$ for some initial parameter r_0 can be found, the remaining solutions to the problems are obtained by “continuing” $s(r_0)$ via numerical integration of a differential equation

$$\frac{\partial s(r)}{\partial r} = F(s(r), r, \dots), \quad (4.2)$$

where the function $F(s(r), r, \dots)$ is constructed using the original problem $P_r(s(r))$.

The solutions to the original algebraic problem are the points on the solution trajectory to Equation 4.2. A potential problem (besides implementation issues) is the possibility of multiple disconnected trajectory ‘branches’. If multiple branches exist, a particular branch is determined by the choice of r_0 , and the system will ‘stay’ on this branch. Solutions on other branches would therefore remain undiscovered.

The advent of digital computers made numerical solution approaches like continuation methods practical for scientists and engineers. Accordingly, these algorithms were initially used in the 1970’s. For example, continuation methods have been applied to circuit analysis [Chao et al. 1977], sparse matrix inversion [Saeks 1979], eigenvalue problems [Green et al., 1980], and sensitivity analysis [Saeks et al. 1980].

4.1.2 Solution Sets

Algorithm. A special case of continuation methods arises if *not* all points on the trajectory in Equation 4.2 are solutions of the original algebraic parameterized family of problems, but all solutions are trajectory points. Here the problem reduces to finding the solution set whose elements s satisfy Equation 4.1. A continuation algorithm for this class of problems may be viewed as a ‘degenerate’ version of the usual

continuation methods outlined above. The particular algorithm used here is presented in [Chao et al. 1975]. The notation is changed for the present context. Without loss of generality let $b = 0$. A special case of the family of algebraic problems $P_r(s(r))$ is represented by

$$g(x) = 0, \quad (4.3)$$

where $g: \mathcal{R}^n \rightarrow \mathcal{R}^n$ is a continuously parameterized and differentiable function, and x denotes an n -dimensional vector of unknowns. The goal is to find the finite solution set whose elements x_j , $j = 1, 2, \dots$ satisfy Equation 4.3. Rather than solving this algebraic problem directly, a differential equation $dx/dt = F(x(t))$ is constructed using the n coordinate functions $g_i(x)$, $i = 1, \dots, n$. (The construction of $F(x)$ is presented in detail below.) The solution trajectory $x(t)$ goes through all solutions to the original problem. The starting point for such a differential equation in x -space is the differential equation in g -space,

$$\begin{aligned} \frac{\partial g_i(x(t))}{\partial t} &= -g_i(x(t)), \quad g_i(x(0)) = g_i^0 = 0, \quad i = 1, \dots, n-1 \\ \frac{\partial g_n(x(t))}{\partial t} &= \pm g_n(x(t)), \quad g_n(x(0)) = g_n^0 \end{aligned} \quad (4.4a,b)$$

where $x(t)$ is an appropriate intermediate 'state' variable, i.e. a mapping $x: [0, \infty) \rightarrow \mathcal{R}^n$. In the following, it is shown that the stable equilibria of the dynamical system in Equation 4.4 are solutions to the original algebraic problem in Equation 4.3. Therefore system *dynamics* are a means to solve an *algebraic* problem. Furthermore, the solution trajectory Equation 4.4 travels through *all* solutions (under assumptions stated below).

The idea is that, as long as the differential equations in Equation 4.4a stay on the state space curve $c: g_i(x) = 0, i = 1, \dots, n-1$ (where the subsystems intercept at zero), the dynamics of the subsystem in Equation 4.4b will find solutions x^* ,

where $g_n(x^*) = 0$. This can be shown using the solution to Equation 4.4 in g -space,

$$\begin{aligned} g_i(x(t)) &= g_i^0 \cdot e^{-t} = 0, \quad i = 1, \dots, n-1 \\ g_n(x(t)) &= g_n^0 \cdot e^{\pm t} \end{aligned} \quad (4.5a,b)$$

As can be seen from Equation 4.5a, the coordinate functions g_1, \dots, g_{n-1} will stay at zero if g_i^0 is zero. This requirement is satisfied if the initial condition $x(0)$ lies on c . With the minus sign in the exponent of Equation 4.5b, the subsystem g_n will converge to zero, i.e. $g_n(x(t)) = 0$ for $t \rightarrow \infty$. Therefore, the stable equilibria $x(t \rightarrow \infty)$ are also the solutions to the family of algebraic problems in Equation 4.3.

Maneuvering on the curve c . The flexibility provided by changing the sign in Equation 4.5b is essential. Switching between positive and negative exponents allows one to choose the direction in which the system travels on the curve c , i.e. to move towards or away from solutions and singularity points (extrema). Once a *solution* x^* has been reached, i.e. $g(x^*) = 0$, a point close to, but on the 'opposite' side, of the solution is chosen, where g_n changes sign and therefore moves away from x^* , continuing the search for further solutions along the trajectory $x(t)$. Similar steps are taken if a *singularity point* has been detected by the algorithm. In [Chao et al. 1975] it is shown that the directional derivative of g_n in the tangential direction of $x(t)$ changes sign only if the determinant of the Jacobian matrix $\partial g / \partial x$ changes sign. Thus, when detecting and crossing a singularity, the system must be forced to move away from that point by switching the sign in Equation 4.5b. Details about the implementation are presented in Section 4.3.

State trajectory. The minus sign in Equation 4.4a (which appears in the exponent of Equation 4.5a) is not needed in theory, where the g_i start at initial conditions

$g_i^0 = 0$ and remain zero until infinity. Unfortunately, instability due to quantization noise is a potential problem when Equation 4.4 is solved on digital computers using numerical integration. The negative exponent guarantees stability if one or more g_i have small but non-zero values by forcing the system back onto c if excursions away from c occur.

For deriving the numerical integration scheme used here (Euler's method), Equation 4.4 is rewritten in the form of Equation 4.2 by defining

$$G(x(t)) = \begin{bmatrix} -g_1(x(t)) \\ \vdots \\ -g_{n-1}(x(t)) \\ \pm g_n(x(t)) \end{bmatrix}, \quad G_0 = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ g_n^0 \end{bmatrix}, \quad (4.6)$$

so the new system is

$$\frac{\partial g(x(t))}{\partial t} = G(x(t)), \quad G(x(t_0)) = G_0. \quad (4.7)$$

Using the chain rule and $J = \partial g / \partial x$, the partial derivative $\partial g / \partial t$ can also be expressed as

$$\frac{\partial g}{\partial t} = \frac{\partial g}{\partial x} \cdot \frac{\partial x}{\partial t} = J \cdot \dot{x}. \quad (4.8)$$

This linearization results in the desired differential equation, whose solution is the desired trajectory in state space,

$$\dot{x} = J[x(t)]^{-1} \cdot G(x(t)) \quad (4.9)$$

$$= F(x(t)), \quad x(0) = x_0. \quad (4.10)$$

The solutions x^* to this system are obtained by *numerical integration*. For example, using Euler's method

$$x_{k+1} = x_k + \Delta x_k \quad (4.11)$$

$$= x_k + \eta \cdot J(x_k)^{-1} \cdot G(x_k) \quad (4.12)$$

$$= F(x_k), \quad x(0) = x_0. \quad (4.13)$$

where η is a suitable step size and k represents discrete time.

Initial condition, solutions, singularities. An *initial condition* x_0 must lie on the curve c and can be found by any classical search method. For the present work, the gradient descent technique was used to minimize the cost function

$$E(x) = [g_1, g_2, \dots, g_{n-1}]^T \cdot [g_1, g_2, \dots, g_{n-1}] = \sum_{i=1}^{n-1} [g_i(x)]^2, \quad (4.14)$$

which is zero for $x \in c$.

The software implementation of the above continuation algorithm must include an efficient technique for detecting solutions and singularities. The discrete-time system in Equation 4.13 approaches a *solution* x^* when g_n approaches zero, since solutions in x -space correspond to zeros in g -space. When g_n becomes sufficiently small, i.e. $|g_n| < \varepsilon^*$ for some suitable $\varepsilon^* > 0$, the system approaches a stable equilibrium and the Δx in Equation 4.11 approach zero. Then the x -trajectory can be continued using the last step Δx until g_n switches sign (i.e. the zero is found). The ‘blind walk’ is continued until the process leaves the ε -neighborhood of x^* . If the first point found outside the neighborhood is not a point on c , the stability properties of Equation 4.13 will bring the system state back to c . Alternatively, the gradient descent technique can be applied to find a new starting point on c . The solution x^* is then interpolated between the last and new trajectory point outside the ε -neighborhood. An example is shown in Figure 18 of the example below.

The *singularities* on the curve c correspond to minima, maxima, and saddle points of g_n , where the Jacobian matrix $J(x)$ in Equation 4.8 is singular ($\det(J(x)) = 0$). Along c , the determinant can be monitored, and if $|\det(J)| < \varepsilon_\infty$

for some suitable $\varepsilon_\infty > 0$, the singularity can be 'crossed' using the same technique as for the solution points. The pseudo code for the continuation algorithm is listed in Table V.

Table V. Pseudo code of the continuation algorithm.

```

Find initial condition  $x_0$  using some classical search technique.
Choose initial search direction, i.e. the plus or minus sign for
 $g_n$ .
Set  $k = 0$ .
LOOP (search finite state space).
    Compute Jacobian  $J(x_k)$ .
    Compute functions  $g(x_k)$ ,  $F(x_k)$ .
     $x_{k+1} \leftarrow x_k + \eta \cdot J(x_k)^{-1} \cdot F(x_k)$ 

    /* Sufficiently close to solution or singularity? */
    if ( $g_n < \varepsilon^*$ )
        Choose new point on c across solution.
        Interpolate solution  $x^*$ .
         $x_{k+1} \leftarrow \text{new point}$ 
    elseif ( $\det(J) < \varepsilon_\infty$ )
        Choose new point on c across singularity.
        Interpolate singularity  $x_\infty$ .
        Change sign of  $g_n(x)$ .
         $x_{k+1} \leftarrow \text{new point}$ 
    endif

     $k \leftarrow k + 1$ 
    if (outside state space) OR (trajectory cycle detected)
        Terminate LOOP.
    endif
end

if (trajectory is NOT a cycle)
    Change search direction (sign).
    Set  $x = x_0$ .
    Run LOOP.
endif

```

4.2 Problem Definition

Consider a multilayer perceptron (MLP), defined as the nonlinear mapping in Equation 3.2, $f: \mathcal{R}^n \rightarrow \mathcal{R}^n$,

$$f(x) = W \cdot \sigma(Vx + b) = W \cdot \sigma(a), \quad (4.15)$$

which is assumed to approximate some ‘original’ system of nonlinear equations. The parameters $V \in \mathcal{R}^{q \times n}$ and $W \in \mathcal{R}^{n \times q}$ are the weight matrices of hidden and output layer, and $b \in \mathcal{R}^q$ is the vector of bias weights to the hidden layer. The input vector to the MLP is $x \in \mathcal{R}^n$. The sigmoid transfer functions of hidden PEs are represented by the vector function $\sigma(a)$, evaluated at the activation values a . Further details about the MLP are given in Section 3.1.2.

The problem here is to apply recurrent neural networks (Definition 1.2) for finding all solutions which satisfy

$$y = W \cdot \sigma(Vx + b). \quad (4.16)$$

The recurrent neural networks (RNNs) are based on the MLP in question and the continuation algorithm discussed above. The networks implement the differential equation in Equation 4.10. Given a suitable initial condition, the network dynamics solve Equation 4.10, finding the trajectory $x(t)$ on which all desired solution to the MLP in Equation 4.16 reside. According to the universal approximator theorem [Hornik et al. 1989], the original system is assumed to be continuously differentiable, thus its MLP approximation f is too. It is also assumed that the solutions are continuously parameterized over a finite region. Furthermore, the RNN’s trajectory is assumed to have a single branch, i.e. the trajectory does not comprise more than one disconnected section, thus any trajectory point can be reached from the initial condition.

4.3 Recurrent Neural Networks and Continuation Methods

The continuation algorithm presented above for a multilayer perceptron mapping $y = f(x)$ (see Section 4.2) is implemented in the form of a recurrent neural network (RNN). The derivation begins with representing Equation 4.3 in the form of Equation 4.16,

$$g(x) = f(x) - y = 0. \quad (4.17)$$

where the elements of $g(x)$, the coordinate functions $g_i(x)$, are

$$g_i(x) = w_i \cdot \sigma(Vx + b) - y_i, \quad i = 1, \dots, n, \quad (4.18)$$

and the Jacobian matrix is (see also Equation 3.3)

$$\frac{\partial g(x)}{\partial x} = W \cdot \Sigma'(a) \cdot V. \quad (4.19)$$

The continuation algorithm in Equation 4.12, applied to the MLP relation in Equation 4.16, represented in the form of an RNN $x_{k+1} = F(x_k)$ is

$$x_{k+1} = x_k + \eta \cdot J(x_k)^{-1} \cdot G(x_k) \quad (4.20)$$

$$= x_k + \eta \cdot \left(\frac{\partial g(x)}{\partial x} \right)^{-1} \cdot \begin{bmatrix} -g_1(x) \\ \vdots \\ \pm g_n(x) \end{bmatrix} \quad (4.21)$$

$$= x_k + \eta \cdot (W \cdot \Sigma'(a) \cdot V)^{-1} \cdot \begin{bmatrix} -w_1 \cdot \sigma(Vx + b) \\ \vdots \\ \pm w_n \cdot \sigma(Vx + b) \end{bmatrix}, \quad (4.22)$$

where w_i is the i -th row of W .

A block diagram of the basic continuation algorithm, and the associated RNN are shown in Figure 16. As in the previous chapter, a generalized linear Hopfield (GLHN) network is used to compute the inverse Jacobian matrix. For the scope of the present work, an 'outside observer' decides if the state x_k is in the close vicinity of a

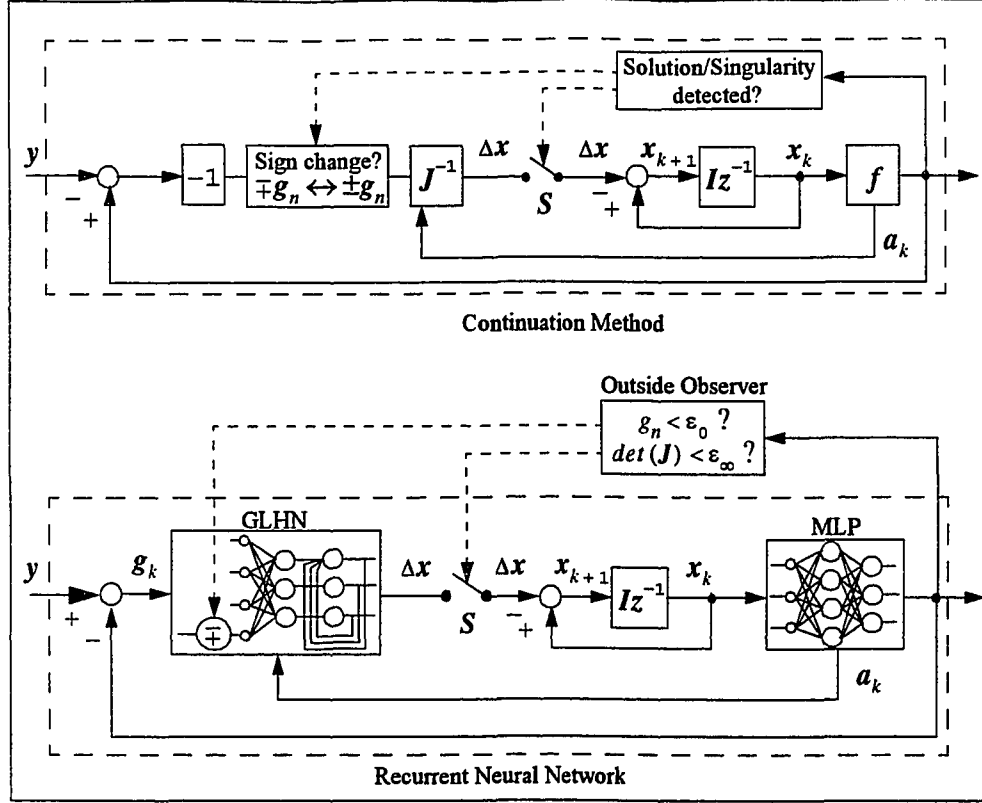


Figure 16. Recurrent neural network and continuation methods.

solution x_0 or a singularity x_∞ . The observer decides when to change the sign (at the input of the generalized linear Hopfield network) and on when to cross some critical point x_0 or x_∞ . A switch, S , is used to cross singularities. If S is open, Δx will not be updated by the GLHN until the observer decides that the state x has moved far enough from the critical point and closes S .

4.4 Example

The above discussion is illustrated using an MLP $f: \mathbb{R}^2 \rightarrow \mathbb{R}^2$ with twelve hidden PEs and weight matrices $V \in \mathbb{R}^{12 \times 2}$, $W \in \mathbb{R}^{2 \times 12}$, and $b \in \mathbb{R}^{12}$ (listed in Appendix C). The MLP outputs represent a quadratic ‘bowl’ and an upside-down ‘val-

ley', respectively. The desired solutions are the points where the zero levels of both coordinate functions g_1, g_2 intercept, where

$$g(x) = W \cdot \sigma(Vx + b) - y \quad (4.23)$$

The given output for the MLP equation was chosen as $y = [0.5, 0.3]^T$. The results are listed in Table VI. The example was implemented in Matlab 4.0. The gradient descent search for an initial condition x_0 on the solution trajectory $x(t)$ began at a first guess x_g and successfully found x_0 after 216 iterations. Given x_0 , the continuation algorithm produced x and g_2 -trajectories of 1002 points each (995 iterations + 7 interpolated solutions/singularities), and found two solutions, two minima, two maxima, and one saddle point. This is shown in Figure 17 (left), where solutions are marked with 'o' and singularities with 'x'. The right figure shows the x -trajectory only, together with some g_2 -contours about the zero level.

The 'crossing' of a solution point is shown in Figure 18 (left). The discrete points on the trajectory are marked with small circles. The g_2 -boundary for the vicinity of zeros was set to $\varepsilon^* = 0.0001$. When the algorithm reaches this boundary, it 'blindly' keeps moving in the direction determined last, until it leaves the solution vicinity. The solution point x^* associated with the zero found is then approximated by the midpoint between entry and exit point. (A more sophisticated interpolation method could be used to approximate the zero point with higher accuracy.) As is shown in the figure, the solution approximation obtained is fairly accurate. The last direction was determined using first and second order information, i.e.

$$x_{k+1} = x_k + \Delta x_k + \Delta x_{k-1}.$$

A similar technique is applied to approximate singular points x_∞ . This is illustrated in Figure 18 (right). The criterion for detecting the vicinity of a some x_∞ (i.e. the limit for $\det(J)$) was set to $\varepsilon_\infty = 0.07$. After the singularity has been approximated by the midpoint between the entry and exit points, the gradient descent technique was used to bring the RNN state x onto the curve c . The approximation

accuracy for singularities can be limited, because the step size in their vicinity increases. As can be seen from Figure 17, the zeros are accurately placed on the intersections of the zero levels of g_1 and g_2 , whereas the trajectory tends to deviate from the desired line about singularities. Although variable step size techniques can improve the results, this is not necessary for the present work, where the problem is to *avoid* singularities, not to *find* them.

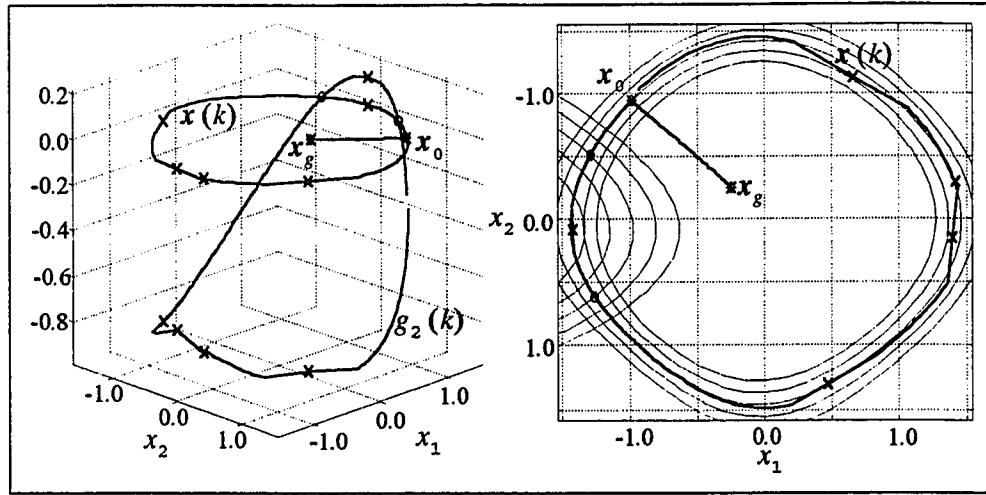


Figure 17. Continuation algorithm (left), solution trajectory (right).

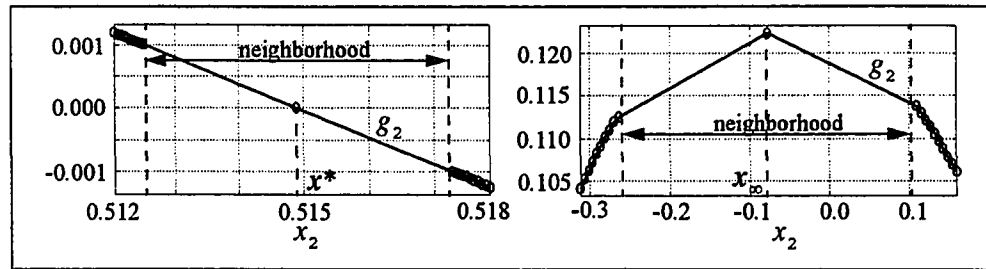


Figure 18. 'Crossing' a solution (left) and a maximum (right).

Table VI. Results for the example.

```

*** Continuation Method Example (MLP) ***

Initial guess xg for starting point x0:
xg = (-0.2500, 0.2500)
yg = (-0.5032,-0.2402)

*** Gradient descent ***
Initial condition x0 after 216 iterations:
x0 = (-0.9862,0.9443)
y0 = (-0.00009731,-0.2451)

*** Continuation Method ***
Solution      x(221) = (-1.2930, 0.5150)
Minimum       x(483) = (-1.4230,-0.0770)
Solution      x(699) = (-1.2600,-0.6130)
Maximum       x(928) = ( 0.4720,-1.2970)
Minimum       x(945) = ( 1.3920,-0.1420)
Saddle Point  x(946) = ( 1.4130, 0.2998)
Maximum       x(959) = ( 0.6620, 1.1340)

Cycle completed at x(995)=(-0.9352,1.0300)
Continuation method: 1002 trajectory points.

```

CHAPTER 5

IMPLEMENTATION ON NEURAL NETWORK HARDWARE

Although artificial neural networks are increasingly recognized in academic and engineering communities as powerful tools for complex problem solving tasks, their use in time-critical applications often demands high performance, high cost hardware systems. Many real-world applications of neural networks eventually will require specialized neural network hardware (NNH) to achieve adequate performance at reasonable cost. The recurrent neural networks from the previous chapters are implemented here on such specialized NNH. The second generation of the multiple instruction multiple data stream (MIMD) neural network processor (NNP¹) used is currently under development, and the work presented here is an integral part of the development process.

It is argued in the previous chapters that the recurrent neural networks (RNNs) developed in this work can outperform conventional algorithms when implemented on specialized parallel NNH. This is demonstrated here by implementing the RNNs on the MIMD NNP and comparing their performance against that of implementations on a high-performance parallel supercomputer (an Intel Paragon). Following similar ideas in the literature, three NNH benchmarks are defined for this purpose and applied to both NNP and Paragon. The benchmark results support the hypothesis that NNH can outperform even a supercomputer in the special case of neural processing tasks.

Worldwide design efforts for NNH are still in an early stage, so the 'best' technology (if it exists) has not yet emerged and researchers are experimenting with a variety of possible realizations. Analog, digital, and hybrid techniques are candidates for

1. NNP^R is a registered trademark of Accurate Automation Corporation.

the implementation of neural network chips, neural network processors, and complete neurocomputer solutions. The intrinsic parallelism of ANNs, i.e. local processing and the storage of knowledge in distributed memory, is the major concept of these hardware systems, since the implementation of ANNs on *dedicated parallel* hardware is an obvious choice. The two main avenues of NNH designs are

- *General purpose* digital hardware platforms with a certain degree of flexibility for the implementation of a variety of ANN paradigms and learning algorithms, and
- *Special purpose* neural network processors or chips (analog, digital, or hybrid), specialized for the efficient implementation of ANN architectures with high speed.

The MIMD neural network processor used here is a realization of special purpose NNH. Its relevant features are discussed below. The variety of available NNH makes the choice of the ‘best’ system a highly problem dependent task. A set of benchmarks was therefore proposed to facilitate a comparison of different NNH systems [Rogers. et al. 1992], [van Keulen et al. 1994]. Here variations of these benchmarks are applied to the NNP, whose performance is compared against that of a common Pentium based computer and that of a high performance Intel Paragon multi-processor supercomputer.

5.1 Parallel Processing

Due to the large number of technical issues to be considered, any attempt to characterize *parallel processing* must be fairly general. Parallel processing can be described as a “computing technique which emphasizes the exploitation of available concurrence in a computational process” [Hazra 1995], and therefore requires more than one active process at any instant of computation. It is characterized by the *combi-*

nation of parallel hardware and parallel software, which manifests itself at different levels of sophistication. Using a categorization with, say three, levels, it is apparent that parallel processing at all three levels directly corresponds to the sophistication of the parallel software used, which is crucial to efficiently exploit a parallel computer architecture: Parallel processing at the highest level is carried out by multiple programs. At the medium level it is limited to concurrent tasks within a single program. This requires the decomposition of the problem at hand into simultaneously executable tasks, which is not always possible. The lowest level can be characterized by concurrence of multiple instructions, or even concurrence within an instruction. It is clear that the mere availability of parallel hardware resources itself does not guarantee parallel processing. It is also clear that intrinsic parallelism of ANNs makes them ideal for task composition and parallel implementations.

In 1966 Flynn proposed a set of basic computer categories which are still in use today: single instruction stream, single data stream (SISD), single instruction stream, multiple data stream (SIMD), and multiple instruction stream, multiple data stream (MIMD) [Patterson and Hennessy 1993]. The terms ‘single’ and ‘multiple’ refer to the specified data format of the system. For example, if integer arithmetic is used, multiple integers are processed concurrently on SIMD and MIMD machines. (Of course, relative to single-bit operations even a SISD processor may be considered a parallel machine). Sometimes it is argued that an *instruction pipeline* implements parallelism on a SISD processor. Indeed, pipelined events occur in overlapped time periods, thus pipelining employs some intrinsic parallelism. Although this technology is often called *temporal parallelism*, it only optimizes the exploitation of a *single* processor. Another type of parallelism is characterized by the replication of physical devices (in space), thus the term *spatial parallelism* [Lin 1995], [Patterson and Hennessy 1993, p. 309]. In the context of artificial neural networks the latter is preferred, and the relevant categories for the present discussion are SIMD and MIMD machines. The characteristics of neural networks (Section 1.2.2) facilitate the decomposition of neural processing into subprocesses, due to the already intrinsic parallelism of ANNs.

This allows the design of *dedicated neural network hardware*, which is designed for solving a particular set of problems, taking full advantage of ANN characteristics. A dedicated system has usually fewer processors and is more efficient and cost effective than that of a comparable general purpose parallel machine. The MIMD neural network processor used for the present work is such a dedicated system.

5.2 Neural Network Processor

The concept behind the digital special purpose neural network processor is to provide efficient neural processing for a variety of feedforward and recurrent neural networks at reasonable cost [Sacks et al. 1995]. The NNP consists of common electronic components and its connection-based architecture and instruction set is optimized for certain neural processing tasks.

An ANN assembler program is stored in the NNP's program memory and executed by the program control unit. Processing elements (PEs) as defined in Equation 1.1 and Equation 1.2 are implemented, where the scalar multiplication of one PE input with one weight is called a *connection*. The computation of connections and transfer functions are the most expensive operations in neural network implementations [Rogers et al. 1992]. Since ANNs consist usually of a large number of PEs, and the number of weights grows exponentially with the number of neurons, NNH systems should be optimized for these operations. This is the main idea behind the NNP design (Figure 19): a multiply-accumulate unit and transfer function lookup tables are the basis of the NNP's optimization. The inner product of a neuron input x and weight value w is computed (the neuron activation a). According to the activation a the neuron output y is selected from the transfer function lookup table and stored in the buffer memory. After the buffer and neuron memory have been interchanged, it is available for all other NNPs and/or the user program running on the host computer. Instruction pipelining enables the completion of one instruction per clock cycle (the clock frequency is 35 MHz).

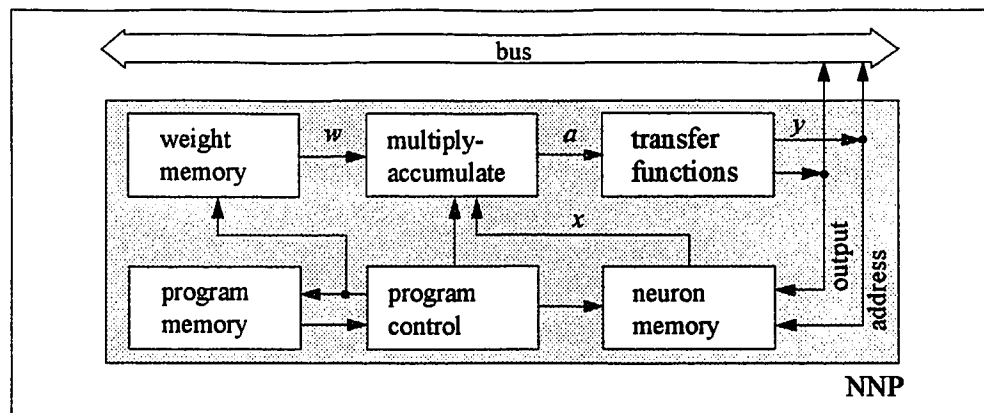


Figure 19. Block diagram of a single neural network processor.

The main features of a single NNP are:

- *Memory for 8K PEs and 32K weights.* A network size is limited by available memory. For example, a Hopfield network is limited to $n = \text{floor}(\sqrt{32768}) = 180$ PEs, where $\text{floor}(x)$ is the largest integer not exceeding real x .
- *Instruction set.* The NNP Assembler consists of nine instructions and resembles the characterization of ANNs.
- *16-bit fixed point arithmetic.* 16-bit fixed point arithmetic with twos-complement replaces time consuming floating operations.
- *Transfer function lookup tables.* Up to four different PE transfer functions are stored in 14-bit lookup tables (16768 values per table).
- *Dual neuron memories.* All PE outputs are updated and stored in a buffer memory, which becomes the neuron memory once all PEs have been processed. This technique prevents memory and bus contention.

The computational speed and available program and weight memory can be increased by using up to eight NNPs in a multiprocessor environment. The parallel NNPs communicate either via shared memory or shared variables, where in the latter case each processor is a complete configuration of CPU, memory, and control unit

(Figure 20). Each NNP is controlled by a separate program. The performance of the resulting MIMD NNP increases approximately *linearly* with the number of NNPs. Depending on the implemented ANN paradigm, the computational power of a multiprocessor MIMD NNP is that of $f/4$ single NNPs, where f is the average fan-in to each NNP. For the present work the maximum number was four, as depicted in (Figure 20).

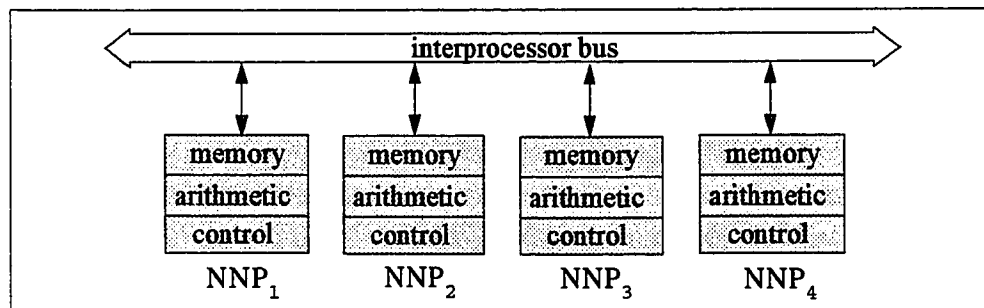


Figure 20. NNPs in a parallel multiprocessor environment.

5.3 Benchmarking

Benchmarking a computer, i.e. measuring its performance and comparing it against that of other machines, is a difficult task due to lack of well defined performance measures across a variety of platforms. Million instructions per second and million floating-point operations are examples of popular measures, but often fail to give a true picture of performance even when running programs on identical machines. The problem becomes even more complex across platforms with different architectures and instruction sets. The amount of work a computer can perform per given time period, i.e. execution time or *elapsed time*, has therefore been suggested as the most reliable measure from a user's perspective (assuming equivalent numerical precision on all machines) [Patterson and Hennessy 1993, p. 75]. In the context of neural networks the benchmarking problem has rarely been addressed. In the follow-

ing, existing benchmarks are reviewed, and two new benchmarks are proposed and applied to the NNP in a single and multiprocessor environment.

5.3.1 Computational Speed and Scaling

For neural networks, elapsed time should be related to the most expensive neural operations, i.e. connections and transfer functions. The number of transfer functions increases linearly with the network size and therefore becomes insignificant for larger networks, compared to the number of connection weights, which increases exponentially. *Connections-per-second* (CPS) have been proposed as one benchmark for the computational speed of ANN software simulators [Rogers et al. 1992]. Although the CPS benchmark is now rather popular, it does not reflect other important issues, like numerical precision and the scaling problem (performance as a function of the network size). These issues are considered here.

Since the numerical precision required for an application is problem dependent, the CPS benchmark is related here to the data format of connection weights and signals implemented on a NNH system. The computational cost of processing a neural network therefore depends on that data format [van Keulen et al. 1994]. The computational speed of ANN implementations also depends on the network size (number of PEs and connection weights). In theory, ANNs are parallel systems whose data throughput is independent of the network size, i.e. the number of neurons and weights. This has not (yet) been realized and the *scaling problem* indeed applies to ANN implementations as well. The following two benchmarks include the data formats of a connection weight and PE inputs, and the number of weights in the network. First the usual CPS measure is multiplied by the number of bytes used for both weight w and PE input x [Mathia et al. 1996].

Definition 5.1: *Connection-bytes-per-second (CBS)* are given by

$$CBS = \text{bytes}(w) \cdot \text{bytes}(x) \cdot CPS. \quad \square \tag{5.1}$$

The CBS benchmark allows different data formats of input x and weight w . The computational cost of a connection, i.e. of the bit-by-bit multiplication, is expressed by the product of *bytes* (x) and *bytes* (w). For example, the computation of a 16-bit connection on the NNP involves the multiplication of two 16-bit numbers, which requires the processing of four bytes. The computational cost of one CPS on the NNP is therefore equivalent to four CBS, $1 \text{ CPS} = 4 \text{ CBS}$. Unfortunately, manufacturers of NNH systems often do not provide details of their benchmarks. One CPS could relate to 1-bit connections (the least expensive operation), or it could mean 64-bit floating point operations (an expensive operation).

The scaling performance of NNH systems can be expressed by normalizing the CBS measure with respect to the number of weights in the network, N_w .

Definition 5.2: *Connection-bytes-per-second-per-weight (CBSW)* are given by

$$CBSW = \frac{CBS}{N_w}. \quad \square \quad (5.2)$$

Furthermore, the *theoretical* peak performance instead of the *realizable sustainable performance* (RSP) is often stated [Rogers et al. 1992]. Here the NNP's RSP has been measured, where only the on-board neural processing performance is considered (although the I/O between NNP and host computer is an important factor in applications). As is shown below, the NNP's RSP approaches the theoretical peak performance, which is unusual for computer systems and can be credited to the NNP's optimization for neural processing tasks. MLPs with the structure defined in Equation 3.2 have been implemented on the NNP and their computational speed has been measured using the CBS and CBSW benchmarks above. The results were compared to the implementations of the same networks on a Intel Paragon in the C-programming lan-

guage. The Paragon is a parallel supercomputer with up to 256 processing nodes, where each node consists of two Intel i860 processors. One i860 is used for interprocessor communication and memory management, the second i860 runs programs (for users or the operating system). The particular Paragon model used has 128 nodes.

When benchmarking software implementations of ANNs, the optimization capabilities of modern compilers must be considered. The code tested must assure that the intended operations are actually being performed. In order not to allow compiler shortcuts, random weights and random initial input vectors were used, and the (necessarily random) network outputs were returned and used as the inputs. This feedback assures unpredictable numbers and therefore a non-optimized program, thereby preventing undesired code optimization. If all layers of the feedforward network have the same size, only a single recurrent layer is equivalent from a computational perspective. This is illustrated in Figure 21 by ‘unfolding’ the recurrent layer in time: the computational complexity of processing one feedforward layer is equivalent to that of processing one recurrent iteration. The MLP bias nodes are replaced by external inputs to the recurrent single layer. All matrices and input and vectors were implemented as indexed arrays in dynamically allocated memory.

The dynamics of the single recurrent layer used for obtaining the CBS and CBSW benchmarks are given by (compare Equation 3.2)

$$x_{k+1} = \sigma(V \cdot x_k + b), \quad (5.3)$$

where V is a random n -by- n weight matrix, b is a random n -dimensional input vector, x_k is the n -dimensional network state at iteration k , and σ is a vector of sigmoid transfer functions. The n^2 connections and n (nonlinear) transfer function per iteration are the most expensive computations for a C-program (for larger networks the transfer functions become insignificant relative to connections). Networks with layers of 2 PEs to 512 PEs (if possible) were implemented, whose performances were

measured using the CBS benchmark by running the recurrent layer for a given number of iterations, measuring the elapsed time and applying Definition 5.1.

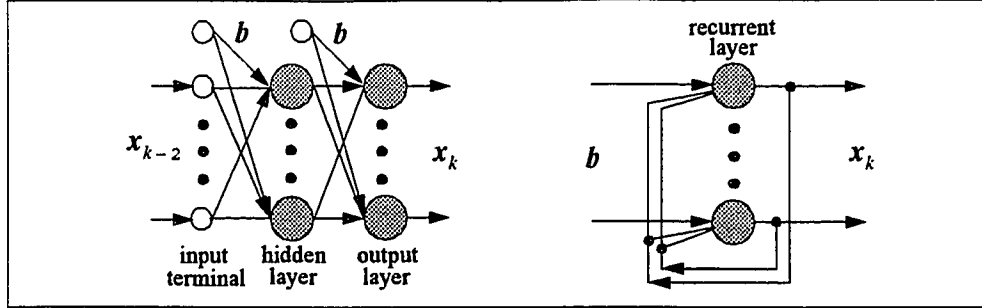


Figure 21. Computational complexity of MLP and a recurrent single layer.

The Intel i860 is a RISC processor whose instruction set does not distinguish between different data formats (e.g. integer, floating point). All computations use full 32-bit registers. Other user specified data formats are converted to and from 32 bits if needed. Consequently, for benchmarking the NNP one may refer to the same 16-bit integer data format.

The results are illustrated in Figure 22 for the NNP and in Figure 23 for the Intel Paragon. The measured CBS are shown as a function of the number of PEs and processors. The number of parallel NNPs was limited to four for this work, but up to 128 parallel nodes (two RISC i860 processors each) on the Intel Paragon were used for benchmarking. It is apparent that the maximum CBS computed by the NNP outperforms that of the Paragon by a factor of approximately four, which clearly can be credited to the NNP's specialization for neural processing tasks. The best performance for a given number of processing elements is shown in both figures as a line of interconnected black dots. This relates to the usual *load balancing* problem on parallel machines and suggests an *optimal number of PEs per processor* from a computational perspective. For the NNP this number was determined as 2, for the Paragon as 32. The Paragon's CBS performance declines if the problem context deviates from this opti-

mum, whereas NNP performance increases linearly (for the available number of NNPs), which meets the design specifications.

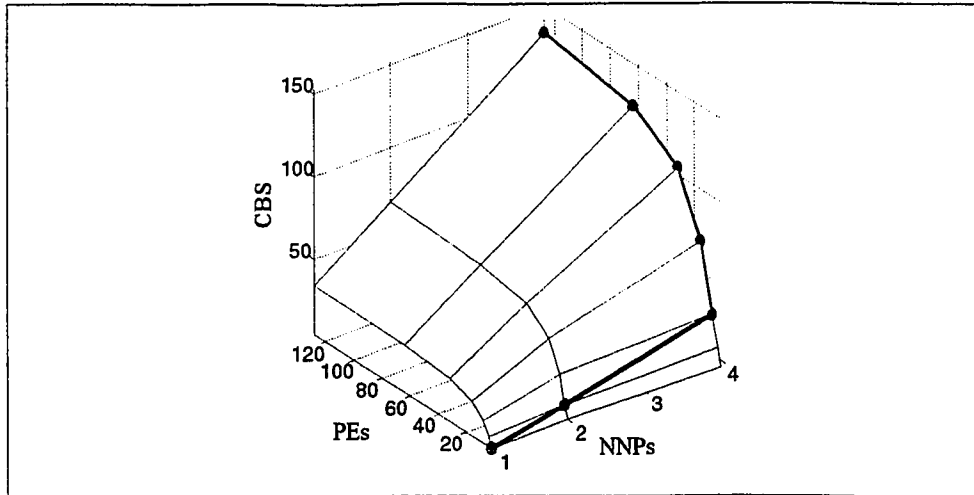


Figure 22. CBS benchmark for the neural network processor (NNP).

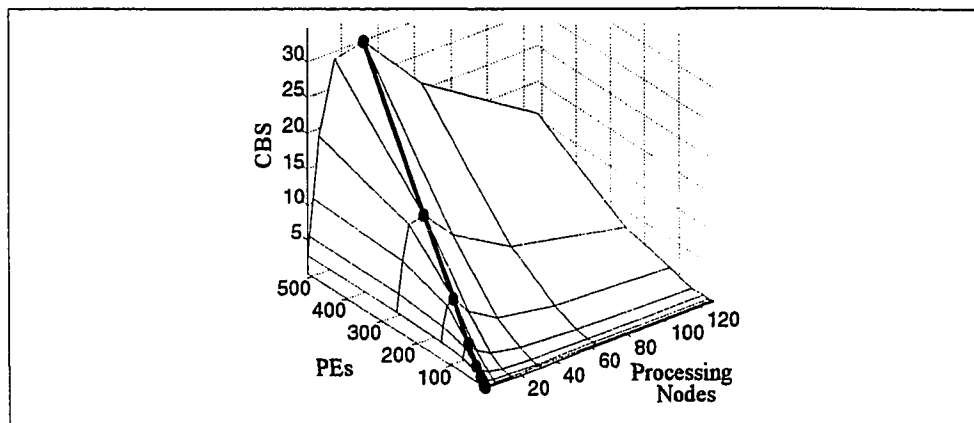


Figure 23. CBS benchmark for the Intel Paragon.

5.3.2 Quantization Error

The numerical precision of neural network hardware can be compromised for high computational speed by using fixed-point integer instead of floating point arithmetic, as was done in the case of the NNP, which employs a 16-bit fixed point arith-

metic. The quantization error due to this numerical limitation was experimentally investigated for this work. A single-input single-output version of the MLP in Equation 3.2 was implemented on the NNP and was viewed as a ‘black box’, which is exposed to quantization noise. The output of this system for a given input is compared to the output of an MLP with high numerical precision, here a software simulator (Figure 20). Here the *quantization-signal-to-noise-ratio* (QSNR) for different numbers of hidden PEs and weight ranges (in the NNP’s fixed point format) characterize the NNP’s loss in numerical precision compared to ‘ideal’ MLP implementation (also compare [van Keulen et al. 1994], [Withagen 1994], [Wray 1995], [Xie and Jabri 1992]).

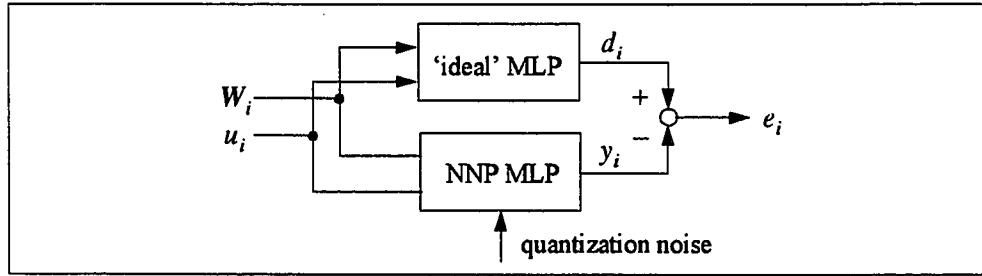


Figure 24. Measuring the NNP quantization-signal-to-noise-ratio.

Let $d_i, i = 1, \dots, N$ be the scalar output series of the ideal MLP (the ‘desired’ signal), and let $e_i = d_i - y_i$ be the error in the processor output y_i . The QSNR is defined in terms of the signal ‘power’ in the NNP output and output error [Papoulis 1991],

$$QSNR = 10 \cdot \log \frac{\sum y_i^2}{\sum e_i^2} \text{ [dB]}. \quad (5.4)$$

The ‘ideal’ MLP is represented by a software simulator using 32-bit floating point data. In 1000 sample measurements a weight matrix W and input u were ran-

domly selected and downloaded to both the ‘ideal’ MLP as well as to its NNP implementation. The associated output series d_i and y_i were measured. Numerical precision is lost twice: first when converting W and u from 32-bit floating point format to the 16-bit integer format on the NNP, and second through quantization noise when processing these integer numbers. The influence of the number of hidden PEs and the fixed point *weight range* on the numerical precision of NNP implementations were investigated. The number of hidden PEs was varied from 1, 2, 4,...,64. The weight range is the number of bits before the fixed point. The weight ranges 0, 1, 2, 3, and 4 bits were used for the experiments, i.e. the randomly selected connection weights were in the intervals $[-1.000, +0.999]$, $[-16.000, +15.999]$. Inputs were selected from $[-1,+1]$. As expected, an increasing weight range causes decreasing numerical precision, the same applies to the network size (number of hidden PEs). The results are summarized in Figure 25. The QSNR for the NNP (in decibel) is plotted against the number of hidden PEs of the single-input single-output MLP. It is apparent that, although the network size causes a drop from 66.23 dB for one hidden PE down to 43.38 dB for 64 PEs, the major precision impact is the numerical range of connection weights. Even a weight range of 4 (i.e. 4 bits on the left of the binary point) causes a drop of approximately 40 dB and makes an application of the NNP impossible in many cases. Although these numerical problems can be circumvented under favorable circumstances, medium size networks and applications which require smaller weight ranges are preferred for the NNP.

5.4 Recurrent Neural Networks on the NNP

Newton’s method as presented in Section 3.2 is implemented on the MIMD NNP as an example for an NNP application of the recurrent neural networks presented in the previous chapters, using the same networks and parameters as in the example presented in Section 3.4. The idea is to apply one NNP per neural network in parallel, here one for the multilayer perceptron (MLP) and one for the generalized linear

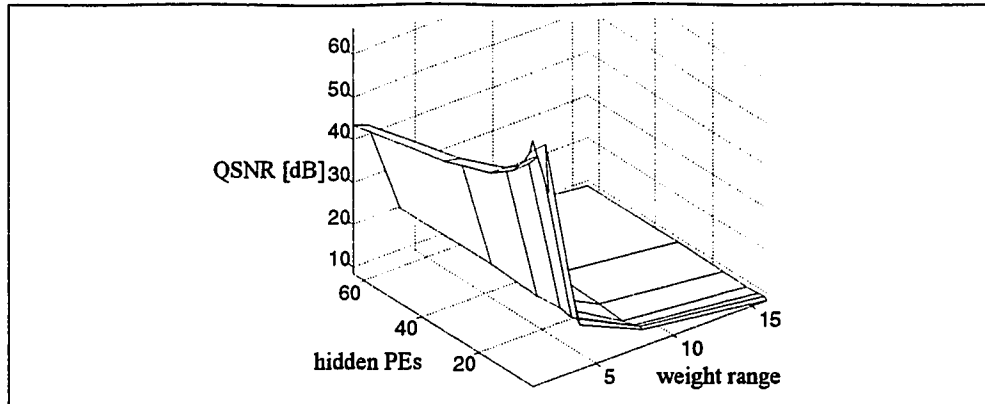


Figure 25. NNP quantization-signal-to-noise-ratio.

Hopfield network (GLHN). Also, the number of tasks assigned to the host computer, as well as the data I/O between the NNP and the host is to be minimized. The parallel operation of different ANN architectures on parallel modules of specialized neural network hardware appears to be the first implementation of its kind.

A block diagram is shown in Figure 26 (compare Newton's method in Figure 11). The complete recurrent neural network resides on the MIMD NNP, using two processors. The processors share common variables and buffer/neuron memory over the interprocessor bus. Here the shared variables are the output error e_k , Jacobian matrix J_k , and state vector x_k of the MLP. Another reason for using two NNPs is that the assignment of different subtasks to NNPs can require different fixed point formats ('neuron ranges' and 'weight ranges'), as well as different transfer functions on each processor. The number of available weight/neuron ranges and transfer functions per NNP are limited to four.

The program on the NNP host only monitors the NNP RNN output error e_k after every iteration k and restarts the NNP as long as it is above a certain convergence criterion ε . The minimum value for ε is limited by the NNP's fixed point arithmetic and also depends on the format chosen for the particular application. For the example $\varepsilon = 0.003$, which is slightly above the quantization error for e . Once

the RNN converges, the current RNN state vector x , i.e. the approximate NNP solution to Equation 3.2, is read off the bus.

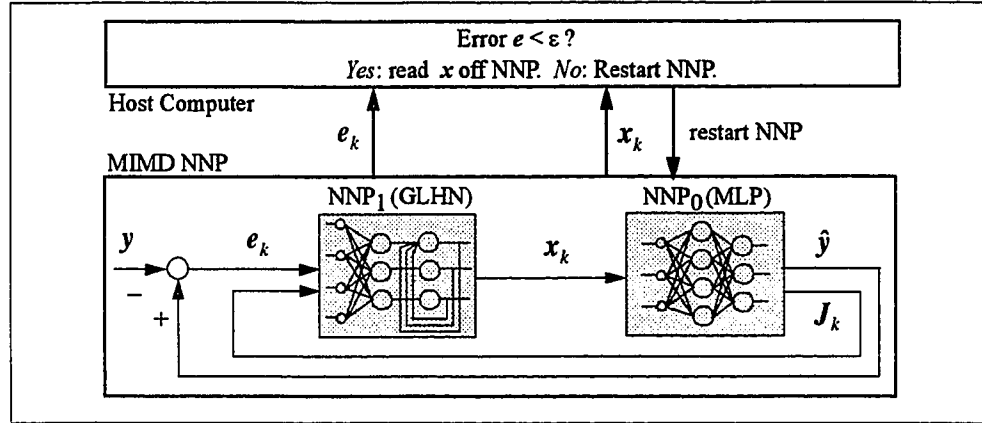


Figure 26. NNP implementation of the first-order RNN.

The neural network to be implemented determines the size of the assembler program for the NNP, and every processing element requires one multiply-accumulate (*mula*) instruction. Thus the resulting code can be repetitive and potentially large. Instead of coding ‘by hand’ it is convenient, and also more flexible, to write a code generator. In most cases this will be a straightforward task, due to the repetitiveness of NNP code. Such a code generator was written for the implementation of Newton’s method in C. The generated NNP program was then assembled to machine code, downloaded to the NNP and started by the host computer. The implementation of the RNN based Newton’s method (Section 3.2) in the example has only 2-by-2 weight matrices; the NNP assembler program is listed and commented in Appendix E. The program also updates the weight matrices of the generalized linear Hopfield network at every iteration. This is possible because the MLP Jacobian matrix varies only with the MLP’s sigmoid derivative (Equation 3.3). Both NNPs, NNP_0 and NNP_1 , can access the Jacobian as a shared variable and participate in its calculation. Conditional jumps are not (yet) available on the NNP. Thus a convergence criterion could not be

implemented and the GLHN dynamics are stopped after a specified number of iterations (here 21). The inverse Jacobian computed by the GLHN has therefore a limited precision.

Table VII. Comparison of Newton's method and RNN.

Newton's Method $ \varepsilon _1 < 0.00001$ after 3 iterations	RNN (Floating Point) $ \varepsilon _1 < 0.00001$ after 11 iterations	RNN (NNP) $ \varepsilon _1 < 0.004$ after 23 iterations
(0.0000, -0.2000)	(0.0000, -0.2000)	0.0000 -0.2000
(0.8950, -0.4589)	(1.0480, -0.5755)	0.6797 -0.2715
(0.9978, -0.4991)	(1.0830, -0.5564)	1.0391 -0.5527
(1.0000, -0.5000)	(1.0370, -0.5256)	1.2812 -0.6934
	(1.0170, -0.5116)	1.2578 -0.6699
	(1.0080, -0.5052)	1.2109 -0.6387
	(1.0030, -0.5023)	1.1562 -0.5996
	(1.0020, -0.5010)	1.1016 -0.5605
	(1.0010, -0.5005)	1.0547 -0.5293
	(1.0000, -0.5002)	1.0078 -0.4980
	(1.0000, -0.5001)	0.9688 -0.4590
	(1.0000, -0.5000)	0.9922 -0.4824
		1.0234 -0.5059
		1.0312 -0.5215
		1.0312 -0.5137
		1.0312 -0.5137
		...
		1.0312 -0.5137

The dynamics of the NNP implementation, together with those obtained by the equivalent Matlab 4.0 implementation, are shown in Table VII. The three columns show the x-trajectories computed by Newton's method (first order) and the associated RNN, both implemented in Matlab 4.0 (floating point). The results computed on the NNP (16-bit integer) are listed in the third column. The iteration process (RNN dynamics) was stopped when the $|\varepsilon|_1 < 0.00001$ for the floating point implementations, and $|\varepsilon|_1 < 0.004$ for the NNP version. It is apparent from the table that Newton's method (with an accurate matrix inversion) needs the least number of iterations (3). The RNN implementation in Matlab requires 11 iterations in order to achieve a result of the same precision, due to the limited number of GLHN iterations. The results computed by the NNP implementation is not only affected by a limited number

of GLHN iterations, but also the 16-bit arithmetic. This version of the RNN converged to an (approximate) result after 23 iterations.

The NNP has, for neural network implementations, computational advantages over general purpose machines. The NNP's fast neural processing has been demonstrated for this work in a series of experiments in which a single NNP was compared against two high performance multiprocessor machines, i.e. a Silicon Graphics Inc. (SGI) Onyx and a SGI 340VGX. The solution of well-defined linear equations has been chosen as a representative numerical problem for this comparison. Linear Hopfield networks were implemented on the NNP for iterative solutions, whereas LAPACK, a well-known software library of linear algebra functions [Anderson et al. 1995], was used on the SGI machines. The library was originally written in Fortran, but the C-version (CLAPACK) was used for the present work. The use of multiple processors increases the computational speed of such software, since one processor can be entirely assigned to the matrix inversion, whereas operating system tasks are being transferred to the remaining processors. The SGI Onyx employs two MIPS R4400 CPUs and two R4010 floating point coprocessors which are driven by a 200 MHz clock frequency. The 340VGX employs four MIPS R3000 CPUs and four R3010 floating point coprocessors at a clock frequency of 33 MHz. Both machines have 64 Mbytes main memory.

As pointed out above, a comparison of different machines as the NNP and the SGI computers is only meaningful under well defined conditions. Here it is assumed that all machines are equally suitable for the computational task in question, regardless of size, weight, power consumption, etc. It is in particular assumed that the NNP's numerical precision is sufficient ($|\varepsilon|_1 < 0.004$, see above). The iteration process of linear Hopfield networks (LHNs) and the CLAPACK function *sgesv_()* (the 32-bit floating point version) were compared using the solution of well-defined linear equations $Ax = y$. Since the convergence speed is mainly affected by the eigenvalues of

the LHN weight matrix (Section 2.5), LHNs with multiple eigenvalues λ were chosen as representative systems, i.e. with a diagonal system matrix,

$$\begin{bmatrix} \lambda & 0 \\ & \ddots \\ 0 & \lambda \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}, \quad n = 2, 4, \dots, 128. \quad (5.5)$$

The convergence speed of W -matrices with $\lambda_i = 0.1, \dots, 0.9$ were measured. The initial conditions of the LHNs were zero. Eigenvalues with smaller or larger absolute values achieve faster or slower converge, respectively. This is shown in Table VII for 128-dimensional matrices. Since the NNP does not provide conditional jumps, the host computer had to verify if the LHN on the NNP has converged yet (here every 10th iteration). The NNP convergence time (in increments of 10 iterations) is compared against the computation time needed by the CLAPACK function *sgesv_()* running on a SGI Onyx. The function *sgesv_()* solves linear systems $Ax = y$ using the inverse of A , which is computed using the factorization of A into its essential lower and upper triangular matrices ('LU' decomposition) [Kreyszig 1988]. For a given dimension n , this process requires a constant number of operations and is independent of the eigenvalues $\lambda(W)$ and $\lambda(A)$. As is shown in Table VII, up to eigenvalues of 0.7 the NNP iteration process outperforms the software package.

While the computation times for matrix inversion shown in Table VII are a function of the (multiple) eigenvalue of the weight matrix, Figure 27 shows the computation times as a function of the dimension of W (or A), measured with $\lambda(W) = 0.5$ for the three platforms available. The results clearly demonstrate the NNP's superior performance for this particular neural processing task, i.e. the inversion of the system in Equation 5.5. For a 2-dimensional system the computation times range from 0.018 milliseconds for a (SGI Onyx), to 42.180 milliseconds for a 12-dimensional matrix (SGI 340VGX). For all dimensions $n > 4$ the NNP outperforms

the computational speed of the SGI machines. The relatively long processing time for dimensions less than four is due to the 10-iteration increment and the communication overhead between the NNP and its host, and cannot be contributed to slow convergence of the LHN.

Table VIII. Matrix inversion: LHN vs. CLAPACK.

$\lambda(W)$	Iterations	NNP-time [s]	SGI-time [s]
0.1	2 (10)	0.0049	0.0110
0.2	3 (10)	0.0049	0.0110
0.3	3 (10)	0.0049	0.0110
0.4	5 (10)	0.0049	0.0110
0.5	6 (10)	0.0049	0.0110
0.6	9 (10)	0.0049	0.0110
0.7	13 (20)	0.0097	0.0110
0.8	22 (30)	0.0146	0.0110
0.9	53 (60)	0.0292	0.0110
0.95	122 (130)	0.0633	0.0110
0.99	779 (780)	0.3795	0.0110

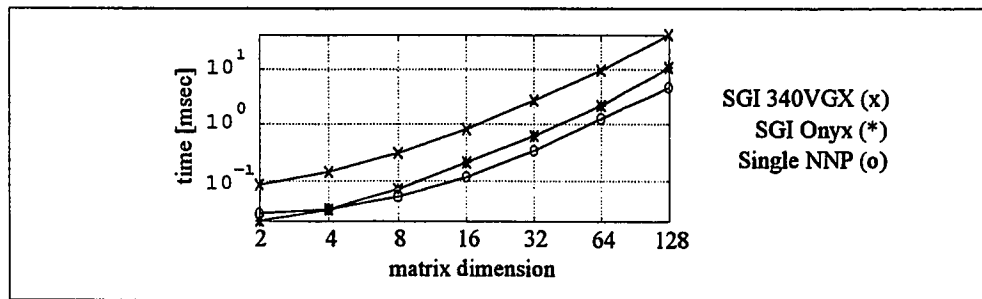


Figure 27. Speed of matrix inversion: Single NNP, SGI Onyx, SGI 340VGX.

CHAPTER 6

SOLUTIONS TO THE INVERSE KINEMATICS PROBLEM

The algorithms developed in the previous chapters are applied to the *inverse kinematics problem* in robotics. For robot manipulators, the control objective is usually to track a prescribed end-effector trajectory. Although this trajectory is typically specified in the manipulator's Cartesian *work space*, it is accomplished by executing a corresponding trajectory of control commands in the robot's *joint space*. These control commands are the joint angles applied to the joint motors. The inverse kinematics problem is to determine the joint angle trajectory, given the Cartesian end-effector trajectory. This is a difficult task, due to the nonlinearities and potential redundancy of the manipulator forward kinematics. Other problems regarding the trajectory tracking problem in robotics include path planning, manipulator dynamics and obstacle avoidance, to mention only a few [Craig 1989], but these are not considered here. In the following sections the inverse kinematics problem for the present work is defined, and the forward kinematics of a robot manipulator considered in this work are presented. Then recurrent neural networks from the previous chapters and their implementations are used as a basis for solutions to the inverse kinematics problem.

6.1 The Inverse Kinematics Problem

The end-effector position of robot manipulators is usually described in the Cartesian work space $X \subset \mathcal{R}^n$, but is controlled by adjusting a set of joint angles in the manipulator's joint space $\Theta \subset \mathcal{R}^m$. The two spaces are related by a nonlinear coordinate transformation, $f: \Theta \rightarrow X$,

$$x = f(\theta) . \tag{6.1}$$

The mapping f represents the manipulator's *forward kinematics*, which describes the geometry of robot arms by associating joint angles $\theta \in \Theta$ with Cartesian end-effector positions $x \in X$. The inverse kinematics (IK) problem is then to determine θ , given a desired position x , thus finding the *inverse kinematics*

$$\theta = f^{-1}(x). \quad (6.2)$$

The geometry of robot arm *motions* are described by associating joint angle *trajectories* $\theta(t)$ with Cartesian end-effector trajectories $x(t)$. The IK problem is then to find a corresponding $\theta(t)$ for a desired $x(t)$, i.e.

$$\theta(t) = f^{-1}(x(t)). \quad (6.3)$$

It is well-known that analytical solutions to the IK problem exist only for some specific kinematic structures. The usual lack of such solutions stems from redundancies and nonlinearities of the manipulator forward kinematics [Baker 1990]. Redundancy is given if, due to the mechanical manipulator design, more than one set of joint angles θ satisfies Equation 6.1 (a unique solution does not exist). This is the case if $m > n$, i.e. the joint angle space is of higher dimensionality than is the work space (but is also possible for $m = n$): the work space of robot manipulators is usually a subspace of the 3-dimensional Cartesian space ($n = 3$), while in many applications robots have more than three joints [Craig 1989], [McKerrow 1990]. Thus more degrees of freedom (joint angles θ_i) exist than constraints (desired Cartesian coordinates x_i), and Equation 6.1 would be underdetermined. A common strategy to overcome this difficulty is to choose a joint angle trajectory which, in some sense, is optimal among all possible solutions [Baker 1990], [Bestaoui 1991]. Another difficulty associated with the IK problem are the nonlinearities of manipulator forward kinematics. Thus numerical methods are often preferred, not only for kinematic structures without an analytic solution to the inverse kinematics problem [Bestaoui 1991].

The present work follows this strategy and applies recurrent neural networks from the previous chapters to the inverse kinematics problem.

6.2 Extendable Stiff Arm Manipulator

The relationship between the work space X and joint space Θ of a 3-joint robot manipulator ($m = n = 3$) is illustrated in Figure 28. The manipulator's control variables are the shoulder yaw angle θ_0 , shoulder pitch angle θ_1 , and the elbow pitch angle θ_2 . The length of link 1 and link 2 are denoted with l_1 and l_2 , respectively. This configuration will be used as an application example in the following sections. The robot manipulators used for the present work is the Extendable Stiff Arm Manipulator (ESAM), which is property of the NASA Marshall Space Flight Center in Huntsville, Alabama. The ESAM has a kinematic structure similar to the one illustrated in Figure 28.

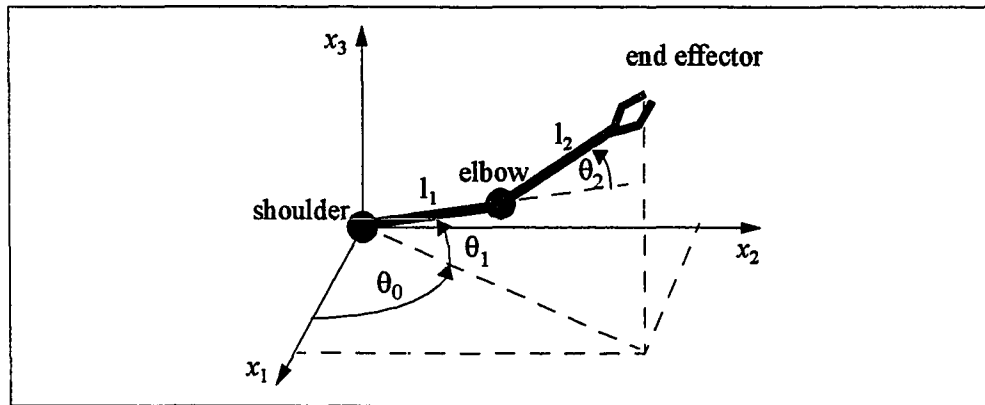


Figure 28. Work and joint space of a 3-joint robot manipulator.

Forward Kinematics. The ESAM configuration is similar to the one characterized in Figure 28. The ESAM work and joint space have the dimensionality $m = n = 3$. The manipulator has two links, two shoulder joints and one elbow joint. The control variables are shoulder yaw angle θ_0 , shoulder pitch angle θ_1 , and the

elbow pitch angle θ_2 . The ESAM configuration and the forward kinematics are shown in Appendix D (Figure 32).

Linearization. A common method for solving nonlinear problems is the linearization principle, which relies on sufficiently accurate first-order approximations of the local behavior of nonlinear systems [Sontag 1990]. The resulting linear system is a valid representation of the original nonlinear system for small perturbations from the linearization point. Many solutions to the inverse kinematics problem are based on this linearization approach.

The linear system is obtained via the time derivatives of Equation 6.1, yielding the velocity equation or *differential kinematics*

$$\frac{dx}{dt} = J(\theta) \frac{d\theta}{dt}, \quad (6.4)$$

where $J(\theta)$ is the manipulator Jacobian for a given set of joint angles θ :

$$J(\theta) = \begin{bmatrix} \frac{dx_1}{d\theta_1} & \frac{dx_1}{d\theta_2} & \cdots & \frac{dx_1}{d\theta_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dx_n}{d\theta_1} & \frac{dx_n}{d\theta_2} & \cdots & \frac{dx_n}{d\theta_m} \end{bmatrix}. \quad (6.5)$$

Common inverse kinematics approaches use the manipulator Jacobian in Equation 6.5: The forward kinematics are first linearized and then inverted in order to compute the next point on the joint angle trajectory. This computation must be repeated for each point along the desired Cartesian trajectory. One example is given below for the ESAM, whose dimensions of joint space and work space are $m = n = 3$. The ESAM's manipulator Jacobian is given in Appendix D.

For stability considerations a dynamical model of the manipulator is needed. Based on Newtonian mechanics, manipulator dynamics are usually described in joint angle space [Slotine and Li 1991],

$$H(\theta) \cdot \frac{d^2\theta}{dt^2} + C\left(\theta, \frac{d\theta}{dt}\right) \cdot \frac{d\theta}{dt} + G(\theta) = T, \quad (6.6)$$

where H is the m -dimensional, symmetric and positive definite manipulator inertia matrix, C is a $m \times m$ matrix, and G an m -dimensional vector of gravitational torques. The control input T consists of the torques applied to the manipulator joints. When combined, the second term in Equation 6.6 is the m -dimensional vector of centripetal and Coriolis torques. Details for the ESAM dynamics are given in Appendix D.2.

6.3 Problem Definition

The problem here is to apply the recurrent neural networks presented in the previous chapters to the inverse kinematics (IK) problem. The ESAM will serve as an application example. The RNN-based solutions to the IK problem will employ a linear approach and a nonlinear approach from chapters 2 and 3. A generalized linear Hopfield network (GLHN) with position feedback is used in the linear case. In the nonlinear case a multilayer perceptron (MLP) approximates the robot's forward kinematics. The (trained) MLP is the starting point for the construction of the RNNs, as was demonstrated in chapter 3.

The task of guiding the robot's end-effector position along a desired Cartesian trajectory $x_d(t)$ is formulated as a tracking problem. For a stability analysis and the design of a stabilizing controller, the system model is extended from a pure kinematics structure to a dynamical system by incorporating the manipulator dynamics. Let the nonlinear manipulator dynamics h and forward kinematics f be given by

$$\frac{d\theta}{dt} = h(\theta(t), T(t)), \quad (6.7)$$

$$x(t) = f(\theta(t)), \quad (6.8)$$

with joint angles θ , end-effector position x , and control input T (torques applied to the joints). The tracking problem is then to find a control law for the input $T(t)$ such

that, starting from any initial condition $\theta(0) \in \Theta$, the tracking error $x_d(t) - x(t)$ goes to zero for some t . It is assumed that sufficiently accurate position feedback is available. For the present work, the appropriate RNNs are to be designed and the stability of the control system must be guaranteed.

6.4 Solution Approaches

In the following three sections, the stability of the proposed control system is demonstrated, and two examples, a nonlinear and a linear RNN approach, are presented and demonstrated using computer simulations. Stability of the IK-controlled system can be provided using classical control theory (a recurrent neural network serves as the IK model): once the manipulator dynamics are stabilized, it remains to show that the IK control loop is stable if the IK modelling error stays within certain limits. This approach is facilitated by the above manipulator model with separate models for the dynamics and forward kinematics.

6.4.1 Stability

The stability problem is reduced to a standard controller design problem by representing the proposed neural IK control system in the common form $G/(I + G)^{-1}$ (with negative unity feedback and G as the subsystem in the forward path). The system is stable if and only if $\|G\| < 1$. Figure 29 illustrates how the IK control system can be represented in the desired form. The mathematical derivation is given below. The manipulator is represented by separate models for its dynamics and kinematic structure.

Figure 29a shows the overall neurocontrol system for the manipulator inverse kinematics, including the manipulator dynamics. As specified in the problem definition, the objective is to control the manipulator such that its end-effector trajectory in work space, x , tracks the desired trajectory x_d (all signals are functions of time).

Since the nonlinear control system operates *globally*, the command position $x_c = 2x_d - x$ is the input to the recurrent neural network (RNN), and not the usual position error. (This feedback law is easily verified: we have $x_d = x$ for perfect tracking, so $x_c = x_d$ is the command input to the RNN.) The RNN computes the (imperfect) inverse kinematics \tilde{f}^{-1} , returning the command joint angles θ_c . The tilde indicates a potential model mismatch, which is due to the modelling error of the MLP inside the RNN (see the example in Section 3.4). The position error is compensated via position feedback. The manipulator dynamics are assumed to be stabilized via some classical controller. The structure and dynamics of the stabilized manipulator dynamics (shaded in Figure 29) are of secondary interest for the present work and are represented simply by the nonlinear operator h .

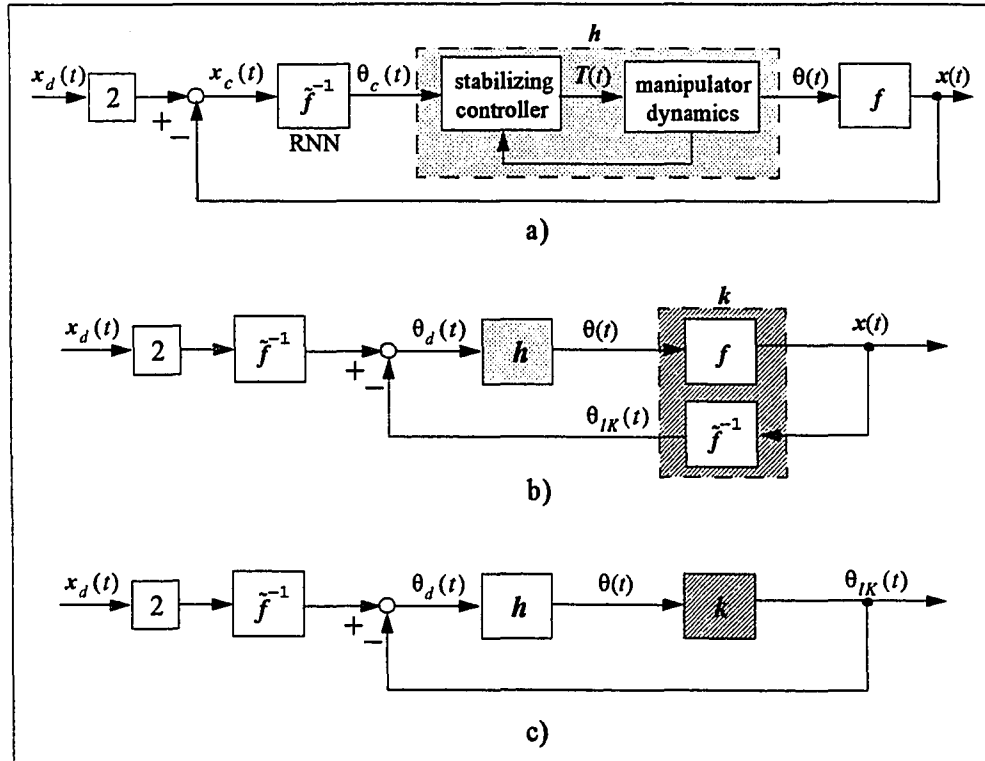


Figure 29. Nonlinear inverse kinematics neurocontroller.

The system in Figure 29b is a different representation of the system in Figure 29a. The manipulator forward kinematics are combined with the inverse kinematics model by shifting \tilde{f}^{-1} both to the feedback and to the input path. The output of the feedback portion is the joint angle vector θ_{IK} . Assuming a perfect inverse kinematics model, i.e. a perfectly trained RNN, this term would reduce to the identity, whereas a (more likely) imperfect model would deviate to some extent from the identity. This potential model mismatch is accounted for by the system k in Figure 29c, where the suitable control system structure for a stability analysis is achieved. The system output is now θ_{IK} , which does not affect the stability properties of the closed loop, since the forward and inverse kinematics are static maps.

Since h is stable, the transfer function of the loop in Figure 29c is $hk / (1 + hk)$, which is stable if $(1 + hk)^{-1}$ exists and is stable. The required bounds on k (and thus the model error) are derived in the following:

$$(1 + hk)^{-1} = [1 + hk + h - h]^{-1} \quad (6.9)$$

$$= [(1 + h) + h(k - 1)]^{-1} \quad (6.10)$$

$$= [(1 + h)(1 + (1 + h)^{-1}h(k - 1))]^{-1} \quad (6.11)$$

$$= [1 + (1 + h)^{-1}h(k - 1)]^{-1} (1 + h)^{-1}. \quad (6.12)$$

The term $(1 + h)^{-1}$ on the right hand side of Equation 6.12 represents the stabilized manipulator dynamics (see above). The second term is invertible and stable if and only if

$$\|(1 + h)^{-1}h(k - 1)\| < 1 \quad (6.13)$$

$$\|(1 + h)^{-1}h\| \cdot \|k - 1\| \leq 1. \quad (6.14)$$

The error bound for an imperfect neural network model of the manipulator forward kinematics can therefore be expressed as

$$\|k - 1\| < \frac{1}{\|(1 + h)^{-1}h\|}. \quad (6.15)$$

The above error bound also includes the linear case as a special case (where f is replaced by a linear operator, e.g. a matrix). The two neural control concepts below are designed and simulated based on this stability property. One uses a linear Hopfield network (chapter 2), the second uses a nonlinear MLP approximation (chapter 3). For the simulation experiments, the manipulator dynamics in Equation 6.6 are stabilized with a PID controller whose proportional, integral, and derivative gains were chosen as $k_p = 3000$, $k_i = 5000$, and $k_d = 75$. (There may exist better values.)

6.4.2 Linear Recurrent Neural Network

The linear neural control system for manipulator inverse kinematics is treated as a special case of the nonlinear concept discussed above. This is illustrated in Figure 30: the manipulator forward kinematics are linearized at a given operating point (joint angles). The inverse kinematics at the operating point are given by the (approximate) inverse Jacobian \tilde{J}^{-1} , which is computed by a Generalized Linear Hopfield Network (GLHN). (The subsequent integrator is assumed part of the GLHN and is not shown in the figure.) The Cartesian position error x_e is the input to the GLHN. This algorithm is based on a modern technique [Bestatoui 1991] which produces good results. The feedback loop is a contribution of this work [Mathia et al. 1994] which improves the algorithm significantly over the basic technique. A 2-dimensional kinematic and dynamic manipulator model of ESAM was used. The dimensions $m = n = 2$ are accomplished by fixing the shoulder yaw angle at zero ($\theta_0 = 0$). This is common practice for demonstration purposes and keeps the example simple [Miller et al. 1990], [Morgan and Özgüner 1985], [Seraji et al. 1986], [Slotine and Li 1991]. The planar ESAM kinematics and dynamics models are specified in Appendix D.2.

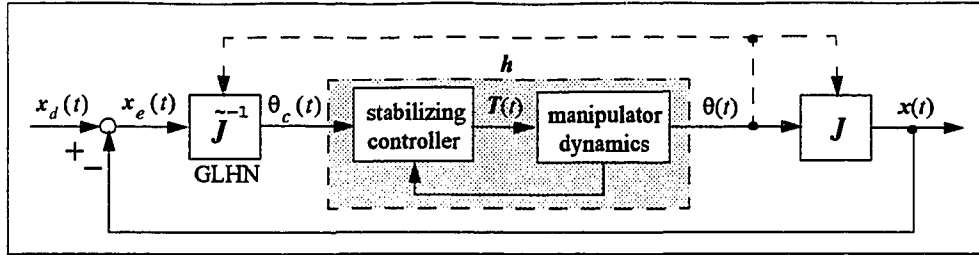


Figure 30. Linear inverse kinematics neurocontrol.

The GLHN computes the corresponding joint angle velocity for a given Cartesian velocity. The velocities are integrated to obtain the joint angles. Trajectory tracking is accomplished point by point: at a given time t_0 the joint angles $\theta(t_0)$ place the end-effector at the Cartesian position $x(t_0)$. Given the next desired Cartesian position $x_d(t_0 + 1)$, the position error is evaluated and applied to the GLHN, whose iteration process (plus subsequent integration) computes the associated joint angles $\theta(t_0 + 1)$. Thus the GLHN dynamics (see chapter 2) are given by

$$\Delta\theta_{k+1} = W \cdot \Delta\theta_k + u, \quad (6.16)$$

The subscript 'c' for the joint angle command θ_c is omitted in Equation 6.16. The weight matrix W and u are reevaluated for the iteration process at time t , using $\theta(t)$, according to

$$W = I - \alpha [J(\theta(t))]^T J(\theta(t)), \quad u = \alpha [J(\theta)]^T (x_d(t+1)). \quad (6.17)$$

The 'convergence rate' was set to $\alpha = 1.9$, as suggested in chapter 2. The GLHN converges to $\Delta\theta(t)$ and the new joint angles are

$$\theta_c(t+1) = \theta_c(t) + \Delta\theta_c(t), \quad (6.18)$$

A straight line from an initial end-effector position, $x_i = (1.307, 0.5348) [m]$, to a final end-effector position, $x_f = (0.419, 1.387) [m]$, was chosen as the desired Cartesian trajectory $x_d(t)$.

This corresponds to initial and final joint angles $\theta_i = (15^\circ, 35^\circ)$ and $\theta_f = (70^\circ, 15^\circ)$. The trajectory consisted of 301 points.

The resulting ESAM motion and the tracking error are illustrated in the left half of Figure 31. The straight line is the desired trajectory, starting at the lower right. The initial drop of the manipulator is caused by the gravitational force and manipulator inertias, which is not accounted for by the initial state of the control system. The neural controller sufficiently compensates for the tracking error at about the trajectory mid-point.

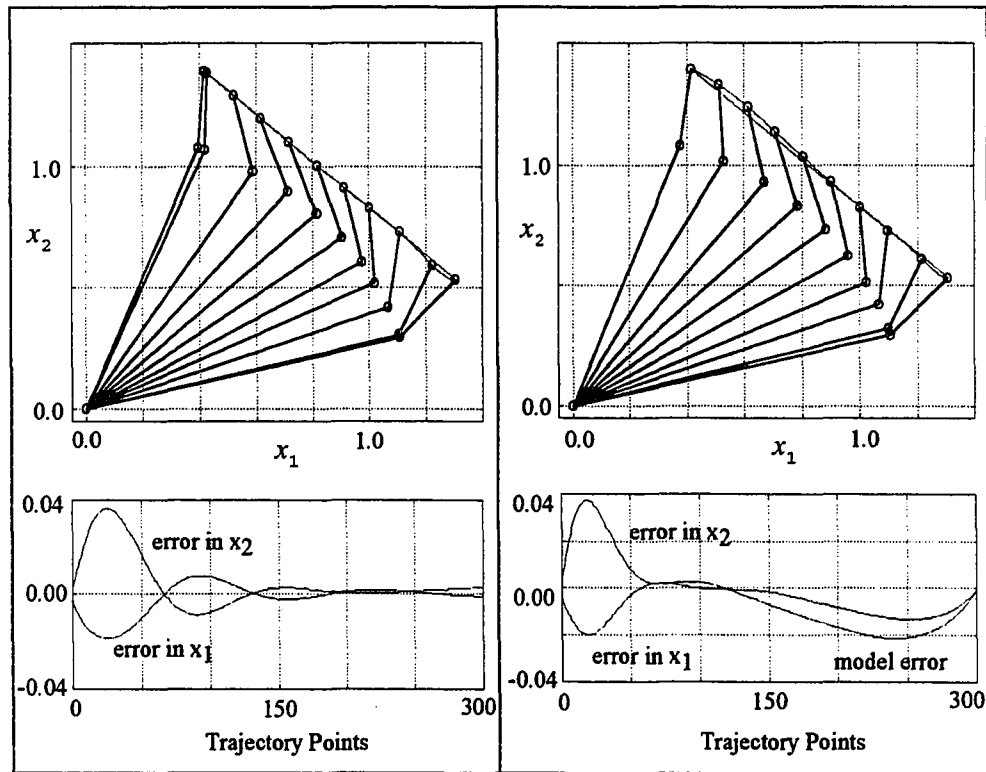


Figure 31. ESAM inverse kinematics: with GLHN (left), and MLP (right).

6.4.3 Nonlinear Recurrent Neural Network

The same trajectory tracking problem as above was simulated using an MLP-based nonlinear control system as shown in Figure 29a. The recurrent neural network

for computing the ESAM inverse kinematics was based on Newton's method (see Section 3.2). For this purpose, a multilayer perceptron (MLP) was trained to approximate the ESAM forward kinematics in the first quadrant of the work space. Details about the MLP architecture, training and approximation accuracy are reported in Appendix F.1. The same desired trajectory $x_d(t)$ as before.

Applying the initial ESAM joint angles $\theta(0)$ to the imperfect MLP model $\tilde{f}(\theta)$ results in the initial modeling error

$$e(0) = x(0) - \tilde{x}(0) = f[\theta(0)] - \tilde{f}[\theta(0)] . \quad (6.19)$$

This model error was eliminated in advance by 'moving' the modeled manipulator along a straight line from its initial Cartesian position to desired initial position $x_d(0)$. No dynamics had to be considered for this simple 'model matching' task, so the recurrent neural network in chapter 3 (Figure 11) could be used.

The resulting ESAM motion and tracking error are illustrated in the right half of Figure 31. The same straight line as above was used as the desired trajectory, starting at the lower right. Here the initial drop of the manipulator, caused by the gravitational force and manipulator inertias, is of similar magnitude as in the linear case. The tracking error is increasing where the MLP model is inaccurate (compare Figure 33 in Appendix F.1).

The main conclusions to be drawn from the above simulation experiments is that the nonlinear neurocontrol approach can substantially outperform the linear approach if the neural network model (here the MLP) of the system to be controlled is sufficiently accurate. As can be seen in Figure 31, the initial tracking error is practically identical for both concepts, but the nonlinear neurocontroller compensates faster and approximately asymptotically, whereas the linear approach exhibits the damped oscillation which are typical for linear control systems. This suggests that improved

neural network architectures training methods are essential for the success of neuro-control concepts.

Of course, only a very basic control system architecture without a controller in the outer loop for manipulator dynamics was used here. The MLP is a pure inverse kinematics controller. The tracking error observed in the experiment could be further reduced with a PI controller in the outer loop of Figure 29a. This (improved) version is not reported here since the stability proof presented above does not hold for this case.

CHAPTER 7

CONCLUSIONS

This dissertation had three main objectives. 1) To develop recurrent neural networks (RNNs) for solving both linear equations and a broad class of nonlinear equations, and in addition, apply them to certain optimization problems. 2) To demonstrate the RNN's fast computational speed when implemented on parallel neural network hardware (NNH). 3) To apply the linear and a nonlinear RNNs to the tracking control of a robot manipulator as a simulation experiment. All three objectives have been accomplished. The RNNs for the linear and nonlinear case were designed and implemented and tested in software. In addition, one linear and one nonlinear RNN was implemented on parallel NNH and fast neural processing has been demonstrated here with benchmark tests. The NNH benchmarks have been specifically defined for this work [Mathia et al. 1996]. The performance of the tracking neurocontrol system for a 2-joint robot manipulator has been demonstrated in simulation experiments.

The recurrent neural networks (RNNs) applied to the linear case are variants of the well-known Hopfield network with linear instead of the usual sigmoidal transfer functions. The RNNs applied to nonlinear equations are based on multilayer perceptrons and numerical methods. It has been shown that the solution approaches presented here are well suited even for time-critical applications if the RNNs are implemented on specialized neural network hardware. The system used for this work was a parallel MIMD neural network processor (NNP¹). The NNP's 16-bit integer arithmetic compromises numerical precision, but does not affect the class of applications considered here, e.g. the robotic inverse kinematics problem.

1. NNP^R is a registered trademark of Accurate Automation Corporation.

The Generalized Linear Hopfield Networks (GLHNs) presented in chapter 2 are linear variants of the Hopfield network and solve arbitrary systems of linear equations, i.e. well-defined systems, systems with full row or column rank, and systems, with both row and column rank deficiency [Mathia et al. 1994][Lendaris et al. 1995]. This is accomplished by enhancing a linear Hopfield network (LHN) with a feedforward layer of processing elements (PEs), which converts a non-definite system into a positive (semi)-definite system. If it exists, the GLHN dynamics implicitly compute the inverse system matrix. For rank deficient systems the GLHNs implicitly compute the Moore-Penrose pseudo-inverse and converge to the least squares solution. Stability and convergence proofs have also been presented. A critical issue for real-time applications can be the computation time needed for a matrix inversion. For GLHNs this relates to the convergence speed of their Jacobi-like iteration process, which mainly depends on the eigenvalues of the LHN weight matrix. In addition, the implementation of GLHNs on the NNP (hosted by a Unix workstation) has computation times equivalent to (and less than) those of common linear algebra software packages run on high-performance multiprocessor machines. This enables the use of such RNN implementations in engineering applications where speed is required. A novel feedback error correction process improves the usefulness of these RNN methods in applications such as inverse kinematics.

The RNNs used for solving systems of nonlinear equations in chapter 3 are based on the multilayer perceptron (MLP), a well-known feedforward neural network. The idea is first to approximate the nonlinear system in question with the MLP and then to solve the function represented by the MLP, instead of the original system [Mathia and Saeks 1995]. The algorithms implemented by the RNNs are classical methods such as Newton's methods and Cauchy's gradient descent as well as more recent ones like the conjugate gradient algorithm and the scaled conjugate gradient algorithm. The RNNs developed here are used for solving equations and optimization problems. A RNN based on Newton's method was implemented on the NNP to demonstrate the performance of these RNNs on NNH. Additionally, it is shown that

second order information and orthogonal search directions can substantially increase the efficiency of the RNNs. It is furthermore shown that these fast converging algorithms cannot be applied in general, since the required conditions are not always satisfied [Lendaris and Mathia 1996].

The RNNs presented in chapter 4 solve continuously parameterized *families* of algebraic problems, in contrast to the RNNs in chapters 2 and 3, which solve a single equation. The families of functions considered here have a finite set of solutions which depends on a variable, scalar parameter. The goal is to find *all* solutions, which is accomplished by using RNNs which are based on a continuation method. As is the concept behind all continuation methods, the families of algebraic problems in question are converted into the solution of an appropriate differential equation. The differential equation is solved via numerical integration performed by the RNNs.

An essential part of this work was to demonstrate that artificial neural networks, in particular the RNNs developed here, are well suited for real-time applications if they are implemented on neural network hardware (NNH). The NNH used here is a parallel MIMD neural network processor (NNP). In order to compare the NNP's computational speed against other hardware systems, two benchmarks were defined which reflect system performance with respect to neural-type processing tasks [Mathia et al. 1996]. Considering this very special class of computations, it is shown that the NNP indeed outperforms even the multiprocessor supercomputers used in the benchmark studies. The trade-off is the NNP's limited numerical precision, due to a 16-bit integer arithmetic. In fact, for the applications considered here the NNP's precision can be assumed to be sufficient, which has been demonstrated by implementations of GLHNs and nonlinear RNN based on Newton's method.

The application example for the RNN approaches developed here is the position control of a robot manipulator which is presented in chapter 6. The control problem is twofold. First, the manipulator's forward kinematics are inverted using a RNN in order to determine the required joint angle trajectory in order to track a desired end-effector trajectory (the inverse kinematics problem). Second, the

manipulator dynamics are stabilized and the position error is minimized with a suitable PI controller. The complexity of this control problem is reduced by the assumption that the manipulator kinematics and dynamics can be decoupled and that feedback of the end-effector position is available. This application of feedback to the inverse kinematics problem, which can be provided by some appropriate sensor, is a contribution of this work [Mathia and Saeks 1994][Mathia, Saeks, and Lendaris 1994]. The resulting linear and nonlinear neurocontrol approaches are applied to the control of a 2-joint robot manipulator in a simulation experiment. The comparison of both cases shows that the nonlinear approach in general exhibits better tracking and less oscillation than the linear control concept if a sufficiently accurate approximation of the manipulator's forward kinematics is available.

Considering the results presented here, as well as the commercial availability of neural network hardware for fast computation, one can conclude that the concept of using recurrent neural networks for solving systems of equations has great potential for a variety of real-time applications. Furthermore, the nonlinear solution approaches presented here can be applied to a variety of feedforward neural networks. On the other hand, the necessity of a sufficiently accurate approximation by an appropriate feedforward networks suggests that emphasis must be placed on developing an accurate ANN model before starting the solution process. Further research in developing good ANN models could improve the RNN solutions and extend the applicability to an even broader class of problems.

BIBLIOGRAPHY

- Accurate Automation Coropration (AAC) (1995), *AAC Neural Network MIMD Processor, Technical Data Sheet*.
- Aleksander I. and H. Morton (1990), *An Introduction to Neural Computing*, Chapman & Hall, London.
- Anderson E., Z. Bai, et al. (1995), *An LAPACK Users's Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA.
- Baker D.R. (1990), "Some Topological Problems in Robotics," *The Mathematical Intelligencer*, Vol. 12, No. 1, pp. 66-76.
- Baker W.L. and J.A. Farrell (1986), "An Introduction to Connectionist Learning Control Systems." In *Handbook of Intelligent Control* (D.A. White and D.E. Sofge, eds.), Vol. 1, Chapter 8, MIT Press, Cambridge, Massachusetts.
- Barron A.R. (1993), "Universal Approximation Bounds for Superposition of a Sigmoidal Function," *IEEE Transactions on Information Theory*, Vol. 39, pp. 930-945.
- Bestaoui Y. (1991), "An Unconstrained Optimization Approach to the Resolution of the Inverse Kinematic Problem of Redundant and Nonredundant Manipulators," *Robotics and Autonomous Systems*, Vol. 7, No. 1, pp. 37-45.
- Brewer J.W. (1978), "Kronecker Products and Matrix Calculus in System Theory," *IEEE Transactions on Circuits and Systems*, Vol. 25, No. 9, pp. 772-781.
- Bronstein I.N., K.A. Semendjajew (1981), *Taschenbuch der Mathematik*, Thun, Frankfurt/Main.
- Burden R.L., J.D. Faires (1988), *Numerical Analysis*, PWS-Kent, Boston, Massachusetts.
- Chao K.S., D.K. Liu, and C.T. Pan (1975), "A Systematic Search Method for Obtaining Multiple Solutions of Simultaneous Nonlinear Equations," *IEEE Transactions on Circuits and Systems*, Vol. 22, No. 9, pp. 748-753.
- Chao K.S. and R. Saeks (1977), "Continuation Methods in Circuit Analysis," *Proceedings of the IEEE*, Vol. 65, No. 8, pp. 1187-1194.
-

- Cichocki A. and R. Unbehauen (1993), *Neural Networks for Optimization and Signal Processing*, Wiley, New York, New York.
- Craig J.J. (1989), *Introduction to Robotics: Mechanics and Control*, Addison-Wesley, Reading, Massachusetts.
- Cybenko G. (1989), "Approximation by Superposition of a Sigmoidal Function," *Mathematics of Control, Signals, and Systems*, Vol. 2, pp. 303-314.
- Daily R.L. (1991), *Lecture Notes for the Workshop on H_∞ and μ Methods for Robust Control*, 1991 IEEE Conference on Decision and Control, Brighton, England.
- Dieudonné J. (1960), *Foundations of Modern Analysis*, Academic Press, New York.
- Drazin P.G. (1992), *Nonlinear Systems*, Cambridge University Press, Cambridge, England.
- Fischler M.A. and O. Firschein (1987), *Intelligence: The Eye, the Brain, and the Computer*, Addison-Wesley, Reading, Massachusetts.
- Gohberg I.C. and M.G. Krein (1969), *Introduction to the Theory of Linear Nonselfadjoint Operators*, American Mathematical Society, Providence, Rhode Island.
- Golub G.H. and C.F. Van Loan (1989), *Matrix Computations*, Johns Hopkins University Press, Baltimore, Massachusetts.
- Green M. and J.N. Limbeer (1995), *Linear Robust Control*, Prentice Hall, Englewood Cliffs, NJ.
- Green B., R. Saeks, and K.S. Chao (1980), "Continuation Algorithms for the Eigenvalue Problem," *Proceedings 1980 IEEE International Symposium on Circuits and Systems*, Vol. 3, pp. 11-16, Houston.
- Grossberg S. (1967), "Nonlinear difference-differential equations in prediction and learning theory," *Proceedings of the National Academy of Sciences of the U.S.A.*, Vol. 58, pp. 1329-1334.
- Grossberg S. (1977), "Pattern formation by the global limits of a nonlinear competitive interaction in n dimension," *Journal of Mathematical Biology*, Vol. 4, pp. 237-256.
- Hammerstrom D. (1990), "Electronic neural network implementation," Tutorial No. 5,

International Joint Conference on Neural networks, Baltimore, MD.

- Hazra T.K. (1995), "Parallel Computing," *IEEE Potentials*, August/September 1995, pp. 17-20.
- Hennesy J.L. and D.A. Patterson (1993), *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufman, San Mateo, California.
- Hestenes M.R. (1980), *Conjugate Direction Methods in Optimization*, Springer-Verlag, New York.
- Hopfield J.J. (1982), "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences of the U.S.A.*, Vol. 79, pp. 2554-2558.
- Hopfield J.J. (1984), "Neurons with graded response have collective computational properties like those of two-state neurons," *Proceedings of the National Academy of Sciences of the U.S.A.*, Vol. 81, pp. 3088-3092.
- Hornik K., M. Stinchcombe, H. White (1989), "Multilayer feedforward networks are universal approximators," *Neural Networks*, Vol. 2, pp. 359-366, 1989.
- Hornik K., M. Stinchcombe, H. White and P. Auer (1994), "Degree of approximation for feedforward networks approximating unknown mappings and their derivatives," *Neural Computation*, Vol. 6, pp. 1262-1275, 1994.
- Hrycej T. (1992), *Modular Learning in Neural Networks: A Modularized Approach to Neural Network Classification*, Wiley, New York, 1992.
- Jin L. (1995), "Parallel Processing," *IEEE Potentials*, December 1994/January 1995, pp. 17-20.
- Keulen E. van, S. Colak, H. Withagen, and H. Hegt (1994), "Neural Network Hardware Performance Criteria," *Proc. IEEE Int. Conf. Neural Networks*, Orlando/Florida, Vol. 3, pp. 1885-1888.
- Kreyszig E. (1988), *Advanced Engineering Mathematics*, Wiley, New York.
- Kuo C.Y. and Wang S.P. (1991), "Robust Position Control of Robotic Manipulators in Cartesian Coordinates," *IEEE Transactions on Robotics and Automation*, Vol. 7, No. 5, pp. 653-659.
- Lendaris G., K. Mathia and R. Saeks (1995), "Linear Hopfield Networks and Con-

- strained Optimization", (submitted to *IEEE Transactions on SMC*).
- Lendaris G. and K. Mathia (1996), "Efficient Numerical Inversion Using Feedforward Neural Networks", to appear in *Proc. World Congress on Neural Networks '96*, San Diego.
- Maciejowski J.M. (1989), *Multivariable Feedback Design*, Addison-Wesley, Reading, Massachusetts.
- Mathia K., R.E. Saeks (1994), "Inverse Kinematics via Linear Dynamic Networks", *Proc. of World Congress on Neural Networks '94*, San Diego.
- Mathia K., R.E. Saeks (1995), "Solving Nonlinear Equations Using Recurrent Neural Networks", *Proc. of World Congress on Neural Networks '95*, Washington D.C.
- Mathia K., R.E. Saeks and G.G. Lendaris (1994), "Linear Hopfield Networks, Inverse Kinematics and Constrained Optimization", *Proc. of IEEE Int'l Congress on Systems, Man and Cybernetics*.
- Mathia K., J. Clark, B. Colbert, and R. Saeks (1996), "Benchmarking an MIMD Neural Network Processor", to appear *Proceedings World Congress on Neural Networks '96*, San Diego.
- McCulloch W.S., W. Pitts (1988), "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, Vol. 5, pp. 115-133, 1943. Reprinted in *Neurocomputing, Foundations of Research* (J.A. Anderson and E. Rosenfeld, eds.), MIT Press, Cambridge, Massachusetts.
- McKerrow P.J. (1990), *Introduction to Robotics*, Addison-Wesley, Reading, Mass.
- Mead C. (1989), *Analog VLSI and Neural Systems*, Addison-Wesley, Reading, Mass.
- Messerschmitt D.G. (1987), "Echo Cancellation in Speech and Data Transmission," *IEEE Journal on Selected Areas in Communication*, Vol. 2, No. 2, pp. 283-297, March 1982. Reprinted in *Adaptive Signal Processing* (L.H. Sibul, ed.), IEEE Press, New York, New York.
- Miller T., R.S. Sutton, and P. Werbos (1990), *Neural Networks for Control*, MIT Press, Cambridge, Mass.
- Møller M.F. (1993), "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning," *Neural Networks*, Vol. 6, No. 4, pp. 525-533.
-

- Morgan R.G. and Ozguner U. (1985), "A Dezentralized Variable Structure Control," *IEEE Journal of Robotics and Automation*, Vol. 1, No. 1, pp. 57-65.
- Odri S.V., D.P. Petrovacki, and A. Krstonoic (1993), "Evolutional development of a multilevel neural networks," *Neural Networks*, Vol. 6, pp. 583-595.
- Oppenheim A.V. and R.W. Schaffer (1989), *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs.
- Papoulis A. (1991), *Probability, Random Variables, and Stochastic Processes*, McGraw Hill, New York.
- Pearlmutter B.A. (1994), "Fast Exact Multiplication by the Hessian," *Neural Computation*, Vol. 6, No. 1, pp. 147-160.
- Rao C.R. and S.K. Mitra (1971), *Generalized Inverse of Matrices and its Applications*, Wiley, New York, New York.
- Reed R. (1993), "Pruning Algorithms - A Survey," *IEEE Transactions on Neural Networks*, Vol. 4, No. 5, pp. 740-747.
- Rogers G.W., J.L. Solka, J.B. Ellis, and H. Szu (1992), "A Neurocomputing Benchmark For Digital Computers," *Proceedings SIMTEC and WNN '92*, Clear Lake, Texas, pp. 425-430, November 4-6.
- Rojas R. (1993), *Theorie der Neuronalen Netze*, Springer-Verlag, Berlin.
- Rosenblatt F. (1988), "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychological Review*, Vol. 65, pp. 386-408, 1958. Reprinted in *Neurocomputing, Foundations of Research* (J.A. Anderson and E. Rosenfeld, eds.), MIT Press, Cambridge, Mass.
- Rumelhart E., G.E. Hinton, and R.J. Williams (1986), "Learning Internal Representations by Error Propagation." In *Parallel Distributed Processing: Explorations in the Microstructures of Cognition* (E. Rumelhart and J.L. McClelland, eds.), Vol. 1, Chapter 8, MIT Press, Cambridge, Mass.
- Saeks R. (1979), "A Continuations Algorithm for Sparse Matrix Inversion," *Proceedings of the IEEE*, Vol. 67, No. 4, pp. 682-683.
- Saeks R. and K.S. Chao (1976), "Continuations Approach to Large-Change Sensitivity Analysis," *Electronic Circuits and Systems*, Vol. 1, No. 1, pp. 11-16.
-

- Saeks R., K. Priddy, K. Schnieder and S. Stowell (1994), "On the Design of an MIMD Neural Network Processor," *Proceedings World Congress on Neural Networks 1994*, San Diego, June 1994, Vol. 2, pp. 590-595.
- Sandberg I.W. (1994), "Iterative Solution of Linear Equations," *IEEE Transactions on Circuits and Systems*, Vol. 41, p. 676.
- Seraji H., M. Jamshidi, Y.T. Kim, and M. Shahinpoor (1986), "Linear Multivariable Control of Two-Link Robots," *Journal of Robotic Systems*, Vol. 3, No. 4, p. 349-365.
- Slotine J.J. and W. Li (1991), *Applied Nonlinear Control*, Prentice Hall, Englewood Cliffs, New Jersey.
- Sontag E.D. (1990), *Mathematical Control Theory*, Springer Verlag, New York, New York.
- Tank D.W. and J.J. Hopfield (1986), "Simple 'Neural' Optimization Networks: an A/D Converter, Signal Decision Circuit, and a Linear Programming Unit," *IEEE Transactions on Circuits and Systems*, Vol. 33, pp. 533-541.
- Keulen E. van, S. Colak, H. Withagen, and H. Hegt (1994), "Neural Networks Performance Criteria," *Proceedings IEEE Conference on Neural Networks*, Orlando, Vol. 3, pp. 1885-1888.
- Vester F. (1975), *Denken, Lernen, Vergessen*, Deutscher Taschenbuch Verlag, Stuttgart, Germany.
- Wang J. and H. Li (1994), "Solving Simultaneous Linear Equations Using Recurrent Neural Networks," *Information Sciences*, Vol. 76, No. 3 and 4, pp. 255-277.
- Werbos P.J. (1974), "Beyond regression: New Tools for Prediction and Analysis in the Behavioral Sciences," Ph.D. Thesis, Harvard University, Cambridge, Mass.
- Werbos P.J. (1988), "Backpropagation: Past and Future," *Proc. IEEE Int. Conf. Neural Networks*, Vol. 1, pp. 343-353, San Diego, California.
- Withagen H. (1994), "Reducing the Effect of Quantization by Weight Scaling," *Proceedings IEEE Conference on Neural Networks*, Orlando, Florida, Vol. 4, pp. 2128-2130.
- Wray J., and G.G. Green (1995), "Neural Networks, Approximation Theory, and Finite Precision Analysis," *Neural Networks*, Vol. 8, No. 1, pp. 31-37.
-

Xie Y., and M.A. Jabri (1992), "Analysis of the Effect of Quantization in Multilayer Neural Networks Using a Statistical Model," *IEEE Transactions on Neural Networks*, Vol. 3, No. 2, pp. 334-338.

APPENDIX A

MATHEMATICS FOR CHAPTER 2

A.1 Singular Value Decomposition

In the following the singular value decomposition (SVD) is derived. The underlying theory is well known and is usually presented for the infinite dimensional case. For the purpose of the present work, the discussion is limited to the finite dimensions. The derivation begins with the polar decomposition of the arbitrary (complex) n -by- n matrix A into its 'phase' P and 'magnitude' M ,

$$A = P \cdot M, \quad (\text{A.1})$$

where

$$P^* P = I, M = (A^* A)^{1/2}, \quad (\text{A.2})$$

i.e. P is unitary and M is square, symmetric, and positive definite. The operator $*$ is the transpose operator for complex matrices. The singular values of A , $\sigma_i(A)$, are defined as the eigenvalues of M . Furthermore, when using the p -norm,

$$\|A\|_p = \left(\sum_{j=1}^n [\sigma_j(A)]^p \right)^{1/p}, \quad (\text{A.3})$$

the trace can be shown to be an upper bound for the spectral radius of square, symmetric, and positive definite matrices (see below).

Considering the linear system $y = Ax$, let v_i , $i = 1, \dots, n$, be the eigenvectors of A , which form an orthonormal basis. Thus

$$v_i^* v_j = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}, \quad (\text{A.4})$$

and the vector x can be represented as the linear combination

$$\mathbf{x} = \sum_{j=1}^n c_j \mathbf{v}_j. \quad (\text{A.5})$$

The scalar c_j is equal to the inner product

$$\mathbf{v}_i^* \mathbf{x} = \mathbf{v}_i^* \cdot \sum_{j=1}^n c_j \mathbf{v}_j \quad (\text{A.6})$$

$$= \sum_{j=1}^n c_j \mathbf{v}_i^* \mathbf{v}_j \quad (\text{A.7})$$

$$= c_i \mathbf{v}_i^* \mathbf{v}_i \quad (\text{A.8})$$

$$= c_i. \quad (\text{A.9})$$

The vector \mathbf{x} in Equation A.5 can therefore be represented as

$$\mathbf{x} = \sum_{j=1}^n (\mathbf{v}_j^*, \mathbf{x}) \mathbf{v}_j = \sum_{j=1}^n \mathbf{v}_j (\mathbf{v}_j^*, \mathbf{x}), \quad (\text{A.10})$$

where (\cdot, \cdot) denotes the inner product.

Now consider some arbitrary matrix H . With Equation A.10 and the definition of eigenvalues λ_i , i.e. $H\mathbf{v}_i = \mathbf{v}_i \lambda_i(H)$, it is shown in the following that the product $H\mathbf{x}$ can be expressed in terms of the eigenvalues and eigenvectors of H :

$$H\mathbf{x} = H \cdot \sum_{j=1}^n \mathbf{v}_j \cdot (\mathbf{v}_j^*, \mathbf{x}) \quad (\text{A.11})$$

$$= \sum_{j=1}^n H\mathbf{v}_j \cdot (\mathbf{v}_j^*, \mathbf{x}) \quad (\text{A.12})$$

$$= \sum_{j=1}^n \mathbf{v}_j \cdot \lambda_j(H) \cdot (\mathbf{v}_j^*, \mathbf{x}). \quad (\text{A.13})$$

Applying the result in Equation A.13 to the magnitude M in Equation A.1, and by defining $\mathbf{u}_j = P\mathbf{v}_j$, we get

$$A\mathbf{x} = P \cdot (M\mathbf{x}) \quad (\text{A.14})$$

$$= P \cdot \left(\sum_{j=1}^n v_j \cdot \lambda_j(M) \cdot (v_j^*, x) \right) \quad (\text{A.15})$$

$$= \sum_{j=1}^n (P v_j) \cdot \lambda_j(M) \cdot (v_j^*, x) \quad (\text{A.16})$$

$$= \sum_{j=1}^n u_j \cdot \lambda_j(M) \cdot (v_j^*, x), \quad (\text{A.17})$$

where $\lambda_i(M)$ denotes the i -th eigenvalue of M . Now define the singular values of A as

$$\sigma_j(A) = \lambda_j(M) = \lambda_j(A^* A)^{1/2}, \quad (\text{A.18})$$

thus Equation A.17 becomes

$$Ax = \sum_{j=1}^n u_j \cdot \sigma_j(A) \cdot (v_j^*, x). \quad (\text{A.19})$$

Defining the matrices

$$U = [u_1 u_2 \dots u_n], \quad \Sigma = \begin{bmatrix} \sigma_1 & & 0 \\ & \sigma_2 & \\ & & \dots \\ 0 & & & \sigma_n \end{bmatrix}, \quad V = [v_1 v_2 \dots v_n], \quad (\text{A.20})$$

allows for the representation of the singular value decomposition in a form which is frequently used in the control literature [Maciejowski 1989], i.e. $A = U \Sigma V^*$:

$$V^* x = \begin{bmatrix} (v_1^*, x) \\ (v_2^*, x) \\ \dots \\ (v_n^*, x) \end{bmatrix} \quad (\text{A.21})$$

$$\Sigma V^* x = \begin{bmatrix} \sigma_1(v_1^*, x) \\ \sigma_2(v_2^*, x) \\ \dots \\ \sigma_n(v_n^*, x) \end{bmatrix} \quad (\text{A.22})$$

$$U \Sigma V^* x = \begin{bmatrix} u_1 & u_2 & \dots & u_n \end{bmatrix} \begin{bmatrix} \sigma_1(v_1^*, x) \\ \sigma_2(v_2^*, x) \\ \dots \\ \sigma_n(v_n^*, x) \end{bmatrix} = \sum_{j=1}^n u_j \cdot \sigma_j(A) \cdot (v_j^*, x). \quad (\text{A.23})$$

Therefore $Ax = U \Sigma V^* x$ and $A = U \Sigma V^*$. Zero singular values in Σ and the corresponding rows and columns in U and V , respectively, are usually removed from form the SVD [Maciejowski 1989].

For a squared matrix we have

$$A^* A = (V \Sigma U^*) (U \Sigma V^*) = V \Sigma^2 V^*, \quad (\text{A.24})$$

thus the trace of $A^* A$ is, with V unitary,

$$\text{tr}(A^* A) = \text{tr}(V^* \Sigma^2 V) = \text{tr}(\Sigma^2) = \sum_{j=1}^n \sigma_j^2(A) = \|A\|_2^2. \quad (\text{A.25})$$

A.2 Proof of Proposition 2.1

The dynamics of the LHN and GLHN at time k , starting with initial condition x_0 , are given by

$$x_{k+1} = W x_k + u = W(W x_{k-1} + u) + u = \dots \quad (\text{A.26})$$

$$= W^{k+1} x_0 + \sum_{i=0}^k W^i u. \quad (\text{A.27})$$

With $\|W\| < 1$ it follows that the sequence $\|W^k\| \leq \|W\|^k$ is exponentially decreasing and approaches zero, $\lim_{k \rightarrow \infty} \|W^k\| = \lim_{k \rightarrow \infty} \|W\|^k = 0$ (a Neumann series), according to the submultiplicative property $\|AB\| \leq \|A\| \cdot \|B\|$ (Green and Limbeer, 1995, p. 30). Consequently, the sequence $\{W^{k+1}\}$ also exponentially approaches zero and the final point is independent of the initial condition x_0 . Thus, the fixed point in the limit is

$$x = \sum_{k=0}^{\infty} W^k \cdot u, \quad (\text{A.28})$$

Furthermore, since the sequence $\{W^k\}$ exponentially approaches zero, the sum in Equation A.28 is bounded and converges to a constant matrix. ■

A.3 Proof of Proposition 2.2

The ‘scaling’ of the linear system in Equation 2.1 with a constant factor, $\alpha Ax = \alpha y$, does not alter the solution x . The LHN corresponding to the scaled equation,

$$x_{k+1} = Wx_k + u = (I - \alpha A)x_k + \alpha y, \quad (\text{A.29})$$

converges if $\|W\| < 1$ (Proposition 2.1). This property can be assured by choosing an appropriate α . Assuming that the spectral radius ρ of the square, symmetric, and positive semi-definite matrix A satisfies $0 < \alpha < 2/\rho(A)$, we need to show that $\text{tr}(A) \geq \rho(A)$ and $\|I - \alpha A\| < 1$ in order to prove Proposition 2.2.

The eigenvalues λ_i of an n -dimensional square, symmetric, positive definite matrix A are real and non-negative, as follows from $0 \leq x^T Ax = x^T \lambda x = \lambda \|x\|^2$ for the eigenvector x corresponding to λ . Therefore $\text{tr}(A) = \sum \lambda_i \geq \lambda_i$ and

$$\text{tr}(A) \geq \lambda_{\max} = \rho(A). \quad (\text{A.30})$$

The proof for $\|I - \alpha A\| < 1$, where $\|\cdot\|$ denotes some Holder norm, is more complex. The proof here is represented in three steps. First it follows from the inequality $0 < \alpha < 2/\rho(A)$ that $-1 < 1 - \alpha \cdot \rho(A) < 1$, therefore

$$-1 < 1 - \alpha \rho(A) \leq 1 - \alpha \lambda_i < 1 \quad (\text{A.31})$$

$$|1 - \alpha \lambda_i| < 1. \quad (\text{A.32})$$

Second, the diagonalization theorem [Kreyszig 1988] is used to represent A in the form $A = U^T D U$, where U is an orthogonal matrix ($U^T U = I$) and D is a diagonal matrix with the λ_i as diagonal elements. Then

$$\|I - \alpha D\| = \max_{1 \leq i \leq n} |1 - \alpha \lambda_i|, \quad (\text{A.33})$$

which is proven in Remark 1 below. Furthermore we have

$$\|I - \alpha A\| = \|I - \alpha U^T D U\| = \|U^T (I - \alpha D) U\| = \|I - \alpha D\|, \quad (\text{A.34})$$

which is proven in Remark 2 below.

Now, with the above arguments, Proposition 2.2 follows, because

$$\|I - \alpha A\| = \|I - \alpha D\| = \max_{1 \leq i \leq n} |1 - \alpha \lambda_i| < 1. \quad (\text{A.35})$$

Remark 1. Equation A.33 can be proved with two inequalities. First, the eigenvalues of the diagonal matrix $I - \alpha D$ are given by $1 - \alpha \lambda_i$. Let x_i be a corresponding eigenvalue. Then, with $\|(I - \alpha D) x_i\| = |I - \alpha D| \|x_i\|$ and the definition in Equation 2.9 (see chapter 2), it follows that

$$\|I - \alpha D\| \geq \frac{\|(I - \alpha D) x_i\|}{\|x_i\|} = |1 - \alpha \lambda_i|. \quad (\text{A.36})$$

(Let $m = \max_{1 \leq i \leq n} |1 - \alpha \lambda_i|$ for an easier notation). Second, if we also show that $\|I - \alpha D\| \leq m$, then the desired equation $\|I - \alpha D\| = m$ follows. For any vector x ,

$$\|(I - \alpha D)x\| = \left(\sum_{i=1}^n |1 - \alpha \lambda_i|^p |x|^p \right)^{1/p} \leq \left(\sum_{i=1}^n m^p |x|^p \right)^{1/p} \quad (\text{A.37})$$

$$= m \left(\sum_{i=1}^n |x|^p \right)^{1/p} = m \|x\|. \quad (\text{A.38})$$

Therefore

$$\|I - \alpha D\| = \max_{\|x\| \neq 0} \frac{\|(I - \alpha D)x\|}{\|x\|} \leq m. \quad (\text{A.39})$$

Remark 2. We need to prove that, if U is orthogonal and M is any square matrix, then $\|U^T M U\| = \|M\|$. With $U^T U = I$ it follows that $\|Ux\| = x^T U^T U x = x^T x = \|x\|$, i.e. orthogonal matrices preserve norms. Also, since here U is orthogonal *and* square, U^T is also orthogonal and we have $U U^T = I$. Then, for an arbitrary vector $x \neq 0$,

$$\frac{\|U^T M U x\|}{\|x\|} = \frac{\|M U x\|}{\|x\|} \leq \frac{\|M\| \|U x\|}{\|x\|} = \frac{\|M\| \|x\|}{\|x\|} = \|M\|, \quad (\text{A.40})$$

so $\|U^T M U\| \leq \|M\|$. Also,

$$\frac{\|M x\|}{\|x\|} = \frac{\|U^T U M U^T U x\|}{\|x\|} = \frac{\|U M U^T U x\|}{\|x\|} \leq \frac{\|U M U^T\| \|U x\|}{\|x\|} \quad (\text{A.41})$$

$$= \frac{\|U M U^T\| \|x\|}{\|x\|} = \|U M U^T\|, \quad (\text{A.42})$$

so $\|M\| \leq \|U M U^T\|$.

In order to prove convergence to $x = A^{-1}y$, first refer to Proposition 2.1, which showed that in the limit the LHN dynamics converges to

$$x = \sum_{k=0}^{\infty} (W^k) u. \quad (\text{A.43})$$

The sum in the limit is

$$S_N = \sum_{k=0}^N W^k = I + W + W^1 + \dots + W^N, \quad (\text{A.44})$$

which, multiplied with W , gives

$$WS_N = W + W^1 + \dots + W^{N+1}. \quad (\text{A.45})$$

Subtracting Equation A.45 from Equation A.44 yields S_N as

$$S_N - WS_N = I - W^{N+1}, \quad (\text{A.46})$$

$$S_N = (I - W)^{-1} (I - W^{N+1}). \quad (\text{A.47})$$

The power of W approaches zero as $N \rightarrow \infty$, since $\|W\| < 1$, yielding the Neumann series

$$S_N = \sum_{k=0}^{\infty} W^k = (I - W)^{-1}. \quad (\text{A.48})$$

The inverse exists because the $n \times n$ matrix $I - W$ is nonsingular, i.e. its determinant is nonzero. The determinant equals the product of the eigenvalues,

$$\det(I - W) = \prod_{i=1}^n \lambda_i(I - W) = \prod_{i=1}^n \lambda_i(I - (I - \alpha A)) = \alpha \cdot \prod_{i=1}^n \lambda_i(A). \quad (\text{A.49})$$

The determinant is nonzero because α is nonzero and all eigenvalues of A are nonzero. (An alternative proof is given in (Golub and van Loan 1989, Lemma 2.3.3)).

It is straightforward to show convergence of the LHN to the desired solution, using the above findings:

$$x = \sum_{k=0}^{\infty} (W^k) u = (I - W)^{-1} u \quad (\text{A.50})$$

$$= (I - (I - \alpha A))^{-1} \alpha y = (\alpha A)^{-1} \alpha y \quad (\text{A.51})$$

$$= A^{-1} y. \quad (\text{A.52})$$

This completes the proof. ■

A.4 Proof of Proposition 2.3

Since $A \in \mathfrak{R}^{m \times n}$, $m > n$, has full column rank n , the LHN weight matrices converge to the pseudo-inverse A^\dagger ,

$$A^\dagger = \sum_{k=0}^{\infty} (W^k) W_{FF} = (I - W)^{-1} W_{FF} = (I - (I - \alpha A^T A))^{-1} \alpha A^T \quad (\text{A.53})$$

$$= (A^T A)^{-1} A^T. \quad (\text{A.54})$$

Equation A.53 is a direct result of Equation A.48. The inverse $(A^T A)^{-1}$ exists because the quadratic term $A^T A \in \mathfrak{R}^{n \times n}$ is square, symmetric and positive and has maximum rank n , since A has full column rank. Thus the GLHN solves the full rank least squares problem and converges to

$$x = A^\dagger y. \quad (\text{A.55})$$

A similar argument holds for full row rank of A :

$$A^\dagger = W_{FF} \sum_{k=0}^{\infty} W^k = \alpha A^T (I - (I - \alpha A A^T))^{-1} \quad (\text{A.56})$$

$$= A^T (A A^T)^{-1}. \quad \blacksquare \quad (\text{A.57})$$

A.5 Proof of Theorem 2.4

If $A \in \mathbb{R}^{m \times n}$ is rank deficient the quadratic term $A^T A \in \mathbb{R}^{n \times n}$ has not maximum rank n . Thus the inverse $(A^T A)^{-1}$ does not exist and the proof from Appendix A.4 does not apply. The following proof makes use of the singular value decomposition (SVD). Let $S \in \mathbb{R}^{m \times n}$ and $S^\dagger \in \mathbb{R}^{n \times m}$ be

$$S = \begin{bmatrix} \Sigma & 0 \\ 0 & 0 \end{bmatrix}, \quad S^\dagger = \begin{bmatrix} \Sigma^{-1} & 0 \\ 0 & 0 \end{bmatrix}, \quad (\text{A.58})$$

where zero singular values are removed from Σ and placed on the diagonal of S (Section 2.2). It is shown in the following that the GLHN with a preprocessing feedforward layer converges to

$$A^\dagger = \sum_{k=0}^{\infty} (W^k) W_{FF} = VS^\dagger U^T. \quad (\text{A.59})$$

This result is then extended to GLHNs with a postprocessing feedforward layer.

GLHN with preprocessing feedforward layer. Recall from Proposition 2.1 that $\|W\| < 1$ assures convergence for the LHN. For the GLHN we have

$$\sum_{k=0}^{\infty} W^k = \sum_{k=0}^{\infty} (I - \alpha A^T A)^k, \quad (\text{A.60})$$

where

$$(I - \alpha A^T A)^k = (I - \alpha VS^T U^T U S V^T)^k \quad (\text{A.61})$$

$$= (VV^T)^k (I - \alpha VS^T S V^T)^k (VV^T)^k \quad (\text{A.62})$$

$$= V^k (V^T V - \alpha V^T V S^T S V^T V)^k (V^T)^k. \quad (\text{A.63})$$

Recall that S is diagonal and that U and V are unitary matrices with $V^T V = I$ and $U^T U = I$. Thus

$$(I - \alpha A^T A)^k = V(I - \alpha S^2)^k V^T. \quad (\text{A.64})$$

The sum in Equation A.59 then becomes

$$\sum_{k=0}^{\infty} (W^k) \cdot W_{FF} = \sum_{k=0}^{\infty} (I - \alpha A^T A)^k \cdot \alpha A^T \quad (\text{A.65})$$

$$= \sum_{k=0}^{\infty} V(I - \alpha S^2)^k V^T \cdot \alpha V S^T U^T \quad (\text{A.66})$$

$$= V \left\{ \sum_{k=0}^{\infty} (I - \alpha S^2)^k \alpha S \right\} U^T. \quad (\text{A.67})$$

$$= V D U^T. \quad (\text{A.68})$$

In order to satisfy Equation A.59 it remains to show that $D = S^\dagger$. Verify that D is diagonal, since S is diagonal. The diagonal elements d_{ii} of D may be determined as functions of the diagonal elements s_{ii} of S , one at a time:

- $s_{ii} = \sigma_i$: If s_{ii} is one of the r nonzero singular values, the corresponding d_{ii} is obtained via the geometric series,

$$d_{ii} = \sum_{k=0}^{\infty} (1 - \alpha \sigma_i^2)^k \alpha \sigma_i = \frac{\alpha \sigma_i}{1 - [1 - \alpha \sigma_i^2]} = \frac{1}{\sigma_i}. \quad (\text{A.69})$$

- $s_{ii} = 0$: If s_{ii} is zero, then the corresponding d_{ii} is also zero,

$$d_{ii} = 0. \quad (\text{A.70})$$

Thus $D = S^\dagger$ and the GLHN with preprocessing feedforward layer computes the pseudo-inverse according to Equation A.59. This completes the first part of Theorem 2.4.

GLHN with postprocessing feedforward layer. The above proof is extended here for GLHNs with feedback weight matrix $\in \mathbb{R}^{m \times m}$ and postprocessing feedforward layer $W_{\text{FF}} \in \mathbb{R}^{n \times m}$. It is shown that GLHNs with this structure also converge to the pseudo-inverse, i.e.

$$A^\dagger = W_{\text{FF}} \sum_{k=0}^{\infty} W^k. \quad (\text{A.71})$$

The equality $A^\dagger = [(A^T)^\dagger]^T$ is applied to the above results:

$$(A^T)^\dagger = \sum_{k=0}^{\infty} (I - \alpha A A^T)^k \cdot \alpha A, \quad (\text{A.72})$$

$$A^\dagger = [(A^T)^\dagger]^T = \left[\sum_{k=0}^{\infty} (I - \alpha A A^T)^k \cdot \alpha A \right]^T \quad (\text{A.73})$$

$$= \alpha A^T \left[\sum_{k=0}^{\infty} (I - \alpha A A^T)^k \right]^T \quad (\text{A.74})$$

$$= W_{\text{FF}} \sum_{k=0}^{\infty} W^k. \blacksquare \quad (\text{A.75})$$

A.6 Gradient of a Quadratic Vector Function

The quadratic error function in Section 2.6, Equation 2.28, is given by

$$E = \frac{1}{2} \|r\|_2^2 = \frac{1}{2} \|Ax - y\|_2^2 = \frac{1}{2} (Ax - y)^T \cdot (Ax - y) \quad (\text{A.76})$$

$$= \frac{1}{2} (x^T A^T Ax - x^T A^T y - y^T Ax + y^T y). \quad (\text{A.77})$$

With $B = B^T = A^T A$, the gradient of E with respect to x is

$$\frac{\partial E}{\partial x} = \frac{1}{2} \left(\frac{\partial}{\partial x} (x^T B x) - \frac{\partial}{\partial x} (x^T A^T y) - \frac{\partial}{\partial x} (y^T A x) + \frac{\partial}{\partial x} y^T y \right) \quad (\text{A.78})$$

$$= \frac{1}{2} \left(\frac{\partial \mathbf{x}^T}{\partial \mathbf{x}} \cdot \mathbf{B} \mathbf{x} + \mathbf{x}^T \mathbf{B} \cdot \frac{\partial \mathbf{x}}{\partial \mathbf{x}} - \frac{\partial \mathbf{x}^T}{\partial \mathbf{x}} \mathbf{A}^T \mathbf{y} - \mathbf{y}^T \mathbf{A} \frac{\partial \mathbf{x}}{\partial \mathbf{x}} \right) \quad (\text{A.79})$$

$$= \frac{1}{2} (\mathbf{x}^T \mathbf{B}^T + \mathbf{x}^T \mathbf{B} - \mathbf{y}^T \mathbf{A} - \mathbf{y}^T \mathbf{A}) \quad (\text{A.80})$$

$$= \frac{1}{2} (\mathbf{x}^T (\mathbf{B}^T + \mathbf{B}) - 2\mathbf{y}^T \mathbf{A}) \quad (\text{A.81})$$

$$= \mathbf{x}^T \mathbf{A}^T \mathbf{A} - \mathbf{y}^T \mathbf{A}. \quad (\text{A.82})$$

E is minimal if the gradient is zero, $\partial E / \partial \mathbf{x} = \mathbf{0}$. After transposing both sides we get

$$\mathbf{A}^T \mathbf{A} \mathbf{x} = \mathbf{A}^T \mathbf{y}. \quad (\text{A.83})$$

APPENDIX B

CALCULUS OF MATRIX-VALUED FUNCTIONS

The algebras related to Kronecker tensor products have several applications in systems theory. A review is presented in (Brewer 1978). For the present work the Kronecker tensor and the chain rule for matrix-valued functions are sufficient and are reviewed here.

Kronecker tensor product. The Kronecker tensor product of two matrices $A \in \mathfrak{R}^{m \times n}$ and $B \in \mathfrak{R}^{r \times s}$, is denoted $A \otimes B \in \mathfrak{R}^{mr \times ns}$ and is defined by

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1n}B \\ a_{21}B & & & \dots \\ \dots & & & \\ a_{m1}B & \dots & & a_{mn}B \end{bmatrix}. \quad (\text{B.1})$$

Vector operator vec . The vec operator provides a means to maintain conformability of objects in matrix calculus. Let $k \in \mathfrak{R}$, $x \in \mathfrak{R}^n$, and $A \in \mathfrak{R}^{m \times n}$. vec is a linear operator acting on vector spaces, and concatenates the columns of a matrix into a mn -by-1 column vector,

$$\text{vec}(A) = \text{vec} \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & & & \dots \\ \dots & & & \\ a_{m1} & \dots & & a_{mn} \end{bmatrix} = \text{vec} [A_1 \ A_2 \ \dots \ A_n] = \begin{bmatrix} A_1 \\ A_2 \\ \dots \\ A_n \end{bmatrix}. \quad (\text{B.2})$$

Special cases are $\text{vec}(k) = k$ and $\text{vec}(x) = x$.

Chain rule for matrix-valued functions. Consider the three matrices $A \in \mathfrak{R}^{m \times n}$, $B \in \mathfrak{R}^{r \times s}$, and $C \in \mathfrak{R}^{u \times v}$. Assume the dependencies

$A = f(B), B = f(C)$ between A, B, C , which are denoted by $A = A(B), B = B(C)$. Then the following chain rule applies for the differentiation of A with respect to C :

$$\frac{\partial A(B(C))}{\partial C} = \left(I_u \otimes \frac{\partial A}{\partial \text{vec}(B)} \right) \cdot \left(\frac{\partial [\text{vec}(B^T)]}{\partial C} \otimes I_n \right), \quad (\text{B.3})$$

$$= \left(\frac{\partial \text{vec}(B^T)}{\partial C} \otimes I_m \right) \cdot \left(I_v \otimes \frac{\partial A}{\partial \text{vec}(B)} \right), \quad (\text{B.4})$$

where I_n represents a square identity matrix of dimension n .

APPENDIX C

THE EXAMPLE MLP IN CHAPTER 4

The example multilayer perceptron (MLP) in Chapter 4 has 12 hidden PEs and approximates a nonlinear function with a mapping $f: \mathcal{R}^2 \rightarrow \mathcal{R}^2$,

$$y = f(x) = W \cdot \sigma(Vx + b). \quad (\text{C.1})$$

The weight matrices $V \in \mathcal{R}^{12 \times 2}$, $W \in \mathcal{R}^{2 \times 12}$, and $b \in \mathcal{R}^{12}$ were trained using the neural network simulator program NeuralWorks/Professional II. The learning procedure and the weight matrices are listed below.

C.1 Training and Testing

The training data were created using Equation 3.21, with $-1.5 < x < 1.5$ in 0.1 steps (961 I/O pairs). The test set was a subset of the train set with $-1.0 < x < 1.0$ (441 I/O pairs). I/O pairs from the train set were randomly selected and presented 50000 times, with an epoch size of 16 (weight updates every 16 presentations). The learning parameters were scheduled as follows:

Iterations	1..10000	10001..50000
Hidden Layer		
Learning Rate	0.3000	0.1500
Momentum Term	0.4000	0.2000
Output Layer		
Learning Rate	0.1500	0.0750
Momentum Term	0.4000	0.2000

C.2 Learned Weight Matrices

$V =$ [+1.9375 +0.1208
+0.5432 -0.9867
+1.3107 -0.0737
+0.7235 -0.8471
+0.0845 -1.3675

```

+0.0609 +0.6612
-0.3401 +0.7709
-0.2712 -1.6558
-1.4272 +0.3139
+0.0340 -0.1822
-0.2147 +2.1811
+0.8144 +0.9126];

```

```

b = [ +2.0192
        -0.9011
        -1.0765
        -0.2878
        -1.4023
        -0.4683
        -0.0315
        +1.9331
        -0.9451
        -1.6186
        +2.3766
        -1.6101 ];

```

```

W = [-0.9914 +1.1386 +1.0069 +0.2329 +0.8751 +0.2936 ...
        +1.2142 -0.7002 +0.8451 -0.4491 -0.4822 +0.9226      ;
        -0.4696 -0.2654 -0.4284 -0.2008 -0.4848 -0.6168 ...
        -0.5948 +0.6008 -0.1832 +0.2134 +1.0099 -0.6103      ];

```

APPENDIX D

EXTENDABLE STIFF ARM MANIPULATOR (ESAM)

D.1 3-Dimensional Kinematic Model

Forward Kinematics. The ESAM parameters are (refer to Figure 32 below)

$$l_0 = 0.586 \text{ m}, l_1 = 0.137 \text{ m}, l_2 = 1.146 \text{ m}, l_3 = 0.311 \text{ m}.$$

The offset parameters are

$$L = 0.198 \text{ m}, \Theta = 0.808 \text{ rad}, l_{\text{off}} = 0.143 \text{ m}.$$

The shoulder yaw joint coincides with the origin of the Cartesian work space (l_0 merely adds a bias to the model and is neglected.) The offset l_{off} from the axis of rotation x_3 is introduced by the shoulder joint.

Forward Kinematics. The ESAM forward kinematics are given by

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} l_1 \cos \theta_0 \cos \theta_1 + l_2 \cos \theta_0 \cos (\theta_1 + \theta_2) \\ l_1 \sin \theta_0 \cos \theta_1 + l_2 \sin \theta_0 \cos (\theta_1 + \theta_2) \\ l_1 \sin \theta_1 + l_2 \sin (\theta_1 + \theta_2) \end{bmatrix} + \begin{bmatrix} L \cos (\Psi (\theta_0)) \\ L \sin (\Psi (\theta_0)) \\ 0 \end{bmatrix}, \quad (\text{D.1})$$

The offset in x_1 and x_2 caused by the shoulder joint is a function of the shoulder yaw angle θ_0 :

$$L = \sqrt{L_1 + L_2}, \quad \Psi (\theta_0) = \theta_0 - \text{atan} (L_2 / L_1). \quad (\text{D.2})$$

Jacobian Matrix. All three joints of the ESAM are modelled. Thus the ESAM's joint space and work space are of equal dimension $m = n = 3$ (refer to Figure 28), and the manipulator Jacobian is therefore square. The effect of the offset of the shoulder yaw joint is modelled by a separate matrix:

$$J_{\text{ESAM}} (\theta) = J (\theta) + J_{\text{Offset}} (\Theta). \quad (\text{D.3})$$

The manipulator Jacobian, not regarding offsets, is

$$J(\theta) = \begin{bmatrix} \frac{dx_1}{d\theta_0} & \frac{dx_1}{d\theta_1} & \frac{dx_1}{d\theta_2} \\ \frac{dx_2}{d\theta_0} & \frac{dx_2}{d\theta_1} & \frac{dx_2}{d\theta_2} \\ \frac{dx_3}{d\theta_0} & \frac{dx_3}{d\theta_1} & \frac{dx_3}{d\theta_2} \end{bmatrix}, \quad (D.4)$$

with the time derivatives given by:

$$\frac{dx_1}{d\theta_0} = -\sin\theta_0 (l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2)), \quad (D.5)$$

$$\frac{dx_1}{d\theta_1} = -\cos\theta_0 (l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2)), \quad (D.6)$$

$$\frac{dx_1}{d\theta_2} = -l_2 \cos\theta_0 \sin(\theta_1 + \theta_2), \quad (D.7)$$

$$\frac{dx_2}{d\theta_0} = \cos\theta_0 (l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2)), \quad (D.8)$$

$$\frac{dx_2}{d\theta_1} = -\sin\theta_0 (l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2)), \quad (D.9)$$

$$\frac{dx_2}{d\theta_2} = -l_2 \sin\theta_0 \sin(\theta_1 + \theta_2), \quad (D.10)$$

$$\frac{dx_3}{d\theta_0} = 0, \quad (D.11)$$

$$\frac{dx_3}{d\theta_1} = l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2), \quad (D.12)$$

$$\frac{dx_3}{d\theta_2} = l_2 \cos(\theta_1 + \theta_2). \quad (D.13)$$

The offset introduced by the shoulder joints is represented by

$$J_{Offset}(\Theta) = \frac{\partial x}{\partial \Theta} = \begin{bmatrix} \frac{dx_1}{d\Theta} \\ \frac{dx_2}{d\Theta} \\ \frac{dx_3}{d\Theta} \end{bmatrix} = \begin{bmatrix} -L \cdot \sin \Theta \\ L \cdot \sin \Theta \\ 0 \end{bmatrix}. \quad (D.14)$$

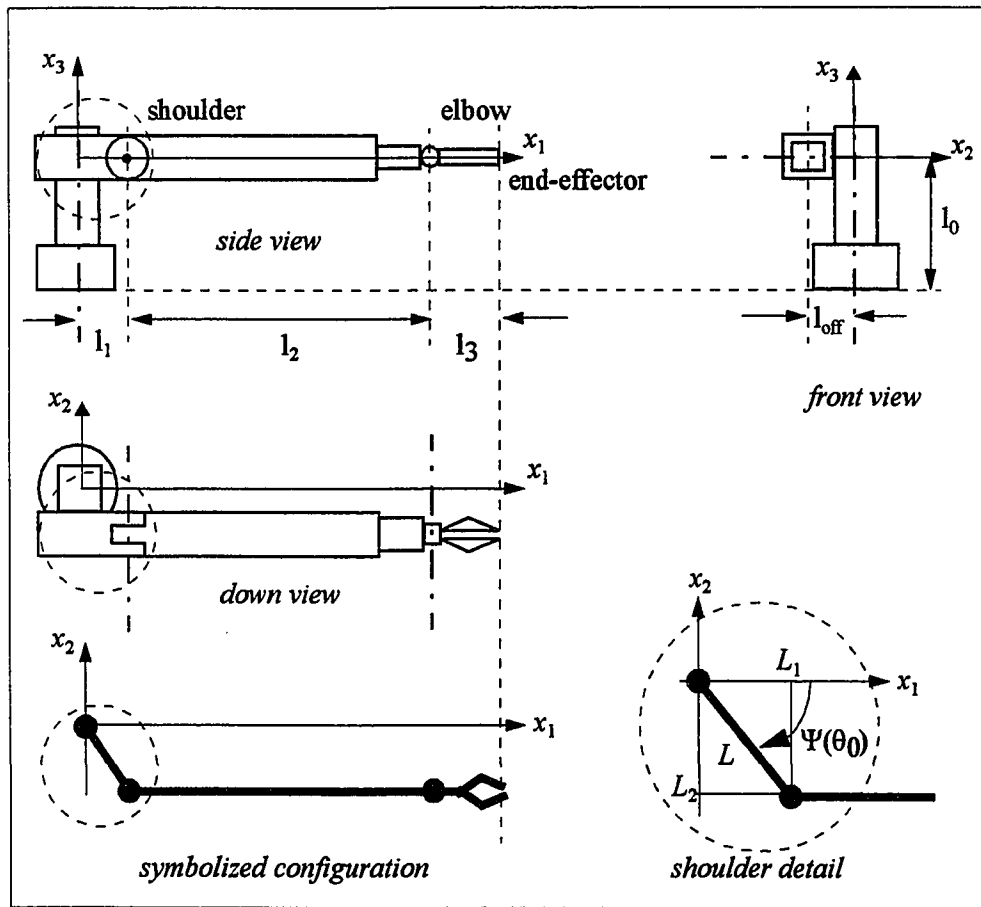


Figure 32. Configuration of the Extendable Stiff Arm Manipulator.

D.2 2-Dimensional Kinematic and Dynamic Model

A simplified 2-dimensional kinematic and dynamic ESAM models (in the vertical x_1/x_2 -plane) were used for the trajectory tracking examples in chapter 6.

Forward kinematics and Jacobian matrix. The commonly used 2-dimentional manipulator kinematics are

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) \\ l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2) \end{bmatrix}, \quad (\text{D.15})$$

the Jacobian matrix is given by

$$J(\theta) = \begin{bmatrix} -l_1 \sin(\theta_1) - l_2 \sin(\theta_1 + \theta_2) & -l_2 \sin(\theta_1 + \theta_2) \\ l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) & l_2 \cos(\theta_1 + \theta_2) \end{bmatrix}. \quad (\text{D.16})$$

As before, the size of the two links are denoted with the parameters l_1 and l_2 , the shoulder and elbow pitch angles are θ_1 and θ_2 .

Nonlinear dynamics. Based on Newtonian mechanics, manipulator dynamics are usually described in joint angle space [Slotine and Li 1991],

$$H(\theta) \cdot \frac{d^2\theta}{dt^2} + C\left(\theta, \frac{d\theta}{dt}\right) \cdot \frac{d\theta}{dt} + G(\theta) = T. \quad (\text{D.17})$$

where H is the m -dimensional, square, symmetric, and positive definite manipulator inertia matrix, C is a $m \times m$ matrix, and g an m -dimensional vector of gravitational torques. The control input T consists of the torques applied to the manipulator joints. The matrices in Equation D.17 are specified as follows [Slotine and Li 1991]:

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}, G = \begin{bmatrix} g_1 \\ g_2 \end{bmatrix} \text{ with} \quad (\text{D.18})$$

$$h_{11} = (m_1 + m_2) l_1^2 + m_2 l_2 (l_2 + 2 l_1 \cos(\theta_2)) + J_1 + J_2 \quad (\text{D.19})$$

$$h_{12} = m_2 l_2 (l_2 + 2 l_1 \cos(\theta_2)) + J_2 \quad (\text{D.20})$$

$$h_{12} = h_{21} \quad (\text{D.21})$$

$$h_{22} = m_2 l_2^2 + J_2 \quad (\text{D.22})$$

$$b_{11} = -m_2 l_1 l_2 \sin(\theta_2) \frac{d\theta_2}{dt} \quad (D.23)$$

$$b_{12} = -m_2 l_1 l_2 \sin(\theta_2) \left(\frac{d\theta_1}{dt} + \frac{d\theta_2}{dt} \right) \quad (D.24)$$

$$b_{21} = m_2 l_1 l_2 \sin(\theta_2) \frac{d\theta_1}{dt} \quad (D.25)$$

$$b_{22} = 0 \quad (D.26)$$

$$g_1 = g \left[(m_1 + m_2) l_1 \cos(\theta_1) + m_2 l_2 \cos(\theta_1 + \theta_2) \right] \quad (D.27)$$

$$g_2 = g \left[m_2 l_2 \cos(\theta_1 + \theta_2) \right]. \quad (D.28)$$

For the present work the masses of link 1 and 2 were chosen as $m_1 = 1 \text{ [kg]}$ and $m_2 = 0.8 \text{ [kg]}$. The inertias of link 1 and 2 were set to $J_1 = 1 \text{ [kg} \cdot \text{s}^2]$ and $J_2 = 0.5 \text{ [kg} \cdot \text{s}^2]$. A sampling period of 0.002 [s] was used for the numerical simulation. The well known gravitational constant is $g = 9.81 \text{ [(kg} \cdot \text{m)/s}^2]$.

APPENDIX E

NNP IMPLEMENTATION OF A RNN

The NNP program (in the NNP assembly language) of Newton's method for the multilayer perceptron is listed on the following pages. The example implemented is presented in Section 3.4 and Section 5.4, refer to those sections for details. The NNP program was created by a code generator written specifically for this purpose in the C-programming language, and was then converted into machine code by the NNP assembler. The program is documented in the following paragraphs, where comments in the program are repeated to identify specific program sections.

Definitions. Different types of assembly instructions are available: A *#directive* starts with a '#' and effects how the NNP code is assembled into machine code. A *label:* ends with a colon and is used to hold values such as constants, memory addresses, transfer function identifiers, and program addresses. An *instruction* is directly assembled into machine code. The program listed below begins with a series of definitions. Examples are:

```
#processors 2:           ; The program is for 2 NNPs.
mlp_range: #nr 1:       ; The MLP neuron range is 1.
#proc 0 .. end_proc:    ; The following code is performed by NNP 0.
```

Multilayer Perceptron. The assembly code for the multilayer perceptron (MLP) on NNP 0 follows the definitions ("*MLP feedforward sections*"). The MLP input *mlp_input* (i.e. RNN state vector \mathbf{x}) at iteration k is copied to the buffer memory so that it will be available to calculate the new \mathbf{x} . The following code processes the MLP's hidden layer ("*calculate hidden layer*") and the sigmoid derivatives. The first four lines of code compute the activation value a_1 and the sigmoid output of the first hidden PE and is repeated here for documentation purposes (here subscripts are indices of vector and matrix elements, not discrete time).

```

mull mlp_input+0,1.000.    ; multiply and load into accumulator:
    a1 ← V11x1,
mula mlp_input+1,0.400.    ; multiply and add to accumulator:
    a1 ← a1 + V12x2,
mula bias+0,0.300.         ; add bias accumulator: a1 ← a1 + b1,
lbtfa mlp_hidden+0,sig.     ; compute sigmoid for given activation: σ(a1),
lbtfa a_sig_d+0,dsig.       ; compute sigmoid derivative: σ'(a1).

```

These instructions are repeated for the second hidden PE and appear in similar versions throughout the program. The neuron and buffer memory are then interchanged (*inbm*) in order to make the PE outputs available for future NNP access. The output layer is processed in a similar manner as is the hidden layer.

Hopfield input and weight matrices. The calculation of the GLHN's weight matrices is based on the MLP's Jacobian J and its transpose ("*calculate jacobian J and its transpose J'*"). The product of matrices V and W with the sigmoid derivatives of hidden PEs has been partially precomputed on the host. In order to obtain the J it remains to multiply the precomputed scalar vw products with the associated sigmoid derivative. The MLP output and both Jacobian and its transpose are then copied to the buffer memory for future NNP access. The squared Jacobian, i.e. the product $J^T J$ is computed next ("*calculate J'*J*"). All the calculation discussed so far are performed on NNP 0. In parallel to those computations the GLHN input is being computed on NNP 1 ("*calculate the input to the GLHN*") and copied to buffer memory, while the squared Jacobian $J^T J$ is being copied to buffer ("*copy JTJ to buffer*") and its trace is being calculated by NNP 0 ("*calculate 1/trace*"). The negative and positive scale factors α and $-\alpha$ are needed for the calculation of the GLHN's feedforward and feedback weight matrices, W_{ff} and W_{fb} . The weights are written to buffer and thus available for further NNP access.

Generalized Linear Hopfield Network. The GLHN input vector *glhn_input* is multiplied by the weight matrix W_{ff} and written to the address of the input of the recurrent Hopfield network, *hop_input*. This input will be needed at every iteration of

the Hopfield network, and therefore copied to buffer memory for NNP access. The dynamics of the Hopfield network (i.e. iterations) are performed in a loop (*hop_loop*), which begins with loading the loop counter for 101 iterations (`llc 101`) and ends if the counter reaches zero (`djnz ... decrement and jump on non-zero`). Processing the GLHN's recurrent portion requires a different weight range than the MLP. The effective GLHN numerical range for the present example is 6, whereas the effective range for the MLP is 2. In order to maintain the GLHN range, a weight range equal to the neuron range (here 3) is required. This is accomplished by the directive `#wr glhn_range`. The GLHN output is copied to buffer memory and is subtracted from the old MLP input, which gives new MLP input. This requires the adjustment of MLP input to the GLHN neuron range ("*move mlp_input into GLHN neuron range*"), ("*calculate new mlp_input*"). The entire NNP program above is repeated by the host computer until the RNN output error is sufficiently small.

Listing: NNP assembler code of the RNN implementation.

```

#processors 2

; define ranges
mlp_range: #nr 1
glhn_range: #nr 3

; define the x-fer functions
#proc 0
sig: #tf mlp_range + 1, mlp_range, sigmoid, 1, 1, 0
dsig: #tf mlp_range +
1, mlp_range, sigmoid_d, 1, 1
linear: #tf mlp_range + 1, mlp_range, scalar, 1
inverse: #tf mlp_range + 1, glhn_range,
power, 1, -1
#end_proc

#proc 1
linear1: #tf glhn_range + 1, glhn_range, sca-
lar, 1
linear2: #tf glhn_range + glhn_range,
glhn_range, scalar, 1
linear3: #tf mlp_range + 1, glhn_range, scalar, 1
linear4: #tf glhn_range + 1, mlp_range, scalar, 1
#end_proc

#nr mlp_range
mlp_input: #nb 2 ; mlp input nodes
#bb 2
mlp_hidden: #nb 2 ; hidden layer nodes
#bb 2
mlp_output: #nb 2 ; output layer nodes
#bb 2
desired: #nb 2 ; desired output
#bb 2
a_sig_d: #nb 2 ; activ. derivative
#bb 2
jacobian: #nb 4 ; jacobian matrix
#bb 4
jacobianT: #nb 4 ; jacobian-transpose
#bb 4
JTJ: #nb 4 ; J'*J
#bb 4
bias: #nw 1.000000 ; bias node
#bw 1.000000

#nr glhn_range
mlp_input_temp: #nb 2 ; mlp input nodes
#bb 2
glhn_input: #nb 2 ; input to feedforward
#bb 2
hop_input: #nb 2 ; input to hopfield
#bb 2
glhn_output: #nb 2 ; output of hopfield
#bb 2
Wff_mtx: #nw 1.000000
#bw 1.000000
#nw 0.000000
#bw 0.000000
#nw 0.000000
#bw 0.000000
#nw 1.000000
#bw 1.000000
Wfb_mtx: #nw 1.000000
#bw 1.000000
#nw 0.000000
#bw 0.000000
#nw 0.000000
#bw 0.000000
#nw 1.000000
#bw 1.000000

one: #nw 1.000000 ; bias node
#bw 1.000000
trace: #nw 1.0 ; trace of jacobian
#bw 1.0
alpha: #nw 1.0 ; alpha value
#bw 1.0
nalpha: #nw 1.0 ; negative alpha
#bw 1.0

#wr 1 ; weight range for GLHN and MLP

#proc 0
;*****
; MLP feedforward sections

; copy mlp_input to buffer
mull mlp_input + 0, 1.0
lbtbf mlp_input + 0, linear
mull mlp_input + 1, 1.0
lbtbf mlp_input + 1, linear

; calculate hidden layer
mull mlp_input, 1.000000
mula mlp_input + 1, 0.400000
mula bias, 0.300000
lbtbf mlp_hidden + 0, sig
lbtbf a_sig_d + 0, dsig
mull mlp_input, 0.600000
mula mlp_input + 1, 1.500000
mula bias, 0.500000
lbtbf mlp_hidden + 1, sig
lbtbf a_sig_d + 1, dsig
#end_proc

; bring all writes forward for
; next calculation
inbm

#proc 0
; calculate output layer (no bias)
mull mlp_hidden, 1.000000
mula mlp_hidden + 1, 0.500000
mula bias, 0.0
lbtbf mlp_output + 0, linear
mull mlp_hidden, 0.400000
mula mlp_hidden + 1, 1.000000
mula bias, 0.0
lbtbf mlp_output + 1, linear
; calculate jacobian J and its transpose J'
; (weights are precomputed WV products)
mull a_sig_d, 1.000000
mula a_sig_d + 1, 0.300000
lbtbf jacobian + 0, linear
lbtbf jacobianT + 0, linear
mull a_sig_d, 0.400000
mula a_sig_d + 1, 0.750000
lbtbf jacobian + 1, linear
lbtbf jacobianT + 2, linear
mull a_sig_d, 0.400000
mula a_sig_d + 1, 0.600000
lbtbf jacobian + 2, linear
lbtbf jacobianT + 1, linear
mull a_sig_d, 0.160000
mula a_sig_d + 1, 1.500000
lbtbf jacobian + 3, linear
lbtbf jacobianT + 3, linear
#end_proc

; bring all writes forward for next calcula-
tion
inbm

```

Listing (cont'd): NNP assembler code of the RNN implementation.

```

#proc 0
; copy mlp_output to buffer
mull mlp_output + 0, 1.0
lbtbf mlp_output + 0, linear
mull mlp_output + 1, 1.0
lbtbf mlp_output + 1, linear
; copy jacobian to buffer
mull jacobian + 0, 1.0
lbtbf jacobian + 0, linear
mull jacobian + 1, 1.0
lbtbf jacobian + 1, linear
mull jacobian + 2, 1.0
lbtbf jacobian + 2, linear
mull jacobian + 3, 1.0
lbtbf jacobian + 3, linear
; copy J' (jacobianT) to buffer
mull jacobianT + 0, 1.0
lbtbf jacobianT + 0, linear
mull jacobianT + 1, 1.0
lbtbf jacobianT + 1, linear
mull jacobianT + 2, 1.0
lbtbf jacobianT + 2, linear
mull jacobianT + 3, 1.0
lbtbf jacobianT + 3, linear
; calculate J' * J (JTV)
mull jacobianT + 0, 0.0
mulap jacobian + 0
mula jacobianT + 1, 0.0
mulap jacobian + 2
lbtbf JTV + 0, linear
mull jacobianT + 0, 0.0
mulap jacobian + 1
mula jacobianT + 1, 0.0
mulap jacobian + 3
lbtbf JTV + 1, linear
mull jacobianT + 2, 0.0
mulap jacobian + 0
mula jacobianT + 3, 0.0
mulap jacobian + 2
lbtbf JTV + 2, linear
mull jacobianT + 2, 0.0
mulap jacobian + 1
mula jacobianT + 3, 0.0
mulap jacobian + 3
lbtbf JTV + 3, linear
#end_proc

; bring all writes forward for next calculation
inbm

#proc 1
; calculate the input to the GLHN
; (linear3 converts MLP neuron range
; to GLHN neuron range)
mull mlp_output + 0, 1.0
mula desired + 0, -1.0
lbtbf glhn_input + 0, linear3
mull mlp_output + 1, 1.0
mula desired + 1, -1.0
lbtbf glhn_input + 1, linear3
#end_proc

; bring all writes forward for next calculation
inbm

#proc 1
; copy glhn_input to buffer
mull glhn_input + 0, 1.0
lbtbf glhn_input + 0, linear1
mull glhn_input + 1, 1.0
lbtbf glhn_input + 1, linear1
#end_proc

#proc 0
; copy JTV to buffer
mull JTV + 0, 1.0
lbtbf JTV + 0, linear
mull JTV + 1, 1.0
lbtbf JTV + 1, linear
mull JTV + 2, 1.0
lbtbf JTV + 2, linear
mull JTV + 3, 1.0
lbtbf JTV + 3, linear

; calculate 1/trace(J'J)
mull JTV + 0, 1.0
mula JTV + 3, 1.0
lbtbf trace, inverse
#end_proc

; bring all writes forward for next calculation
inbm

#proc 1
; calculate (negative) alpha = 1.9/
trace(J'J)
mull trace, 1.9
lbtbf alpha, linear1
mull trace, -1.9
lbtbf nalpha, linear1
#end_proc

; bring all writes forward for next calculation
inbm

#proc 1
; calculate Wff = alpha*J'*J
mull jacobianT + 0, 0.0
mulap alpha
lbtbf Wff_mtx + 0, linear1
mull jacobianT + 1, 0.0
mulap alpha
lbtbf Wff_mtx + 1, linear1
mull jacobianT + 2, 0.0
mulap alpha
lbtbf Wff_mtx + 2, linear1
mull jacobianT + 3, 0.0
mulap alpha
lbtbf Wff_mtx + 3, linear1

; calculate Wfb = I-alpha*J'*J
mull one, 1.0
mula nalpha, 0.0
mulap JTV + 0
lbtbf Wfb_mtx + 0, linear1
mull nalpha, 0.0
mulap JTV + 1
lbtbf Wfb_mtx + 1, linear1
mull nalpha, 0.0
mulap JTV + 2
lbtbf Wfb_mtx + 2, linear1
mull one, 1.0
mula nalpha, 0.0
mulap JTV + 3
lbtbf Wfb_mtx + 3, linear1
#end_proc

; bring all writes forward for next calculation
inbm

```

Listing (cont'd): NNP assembler code of the RNN implementation.

```

#proc 1
; copy Wff_mtx to buffer
mull Wff_mtx + 0, 1.0
lbtbf Wff_mtx + 0, linear1
mull Wff_mtx + 1, 1.0
lbtbf Wff_mtx + 1, linear1
mull Wff_mtx + 2, 1.0
lbtbf Wff_mtx + 2, linear1
mull Wff_mtx + 3, 1.0
lbtbf Wff_mtx + 3, linear1

; copy Wfb_mtx to buffer
mull Wfb_mtx + 0, 1.0
lbtbf Wfb_mtx + 0, linear1
mull Wfb_mtx + 1, 1.0
lbtbf Wfb_mtx + 1, linear1
mull Wfb_mtx + 2, 1.0
lbtbf Wfb_mtx + 2, linear1
mull Wfb_mtx + 3, 1.0
lbtbf Wfb_mtx + 3, linear1

#end_proc

#proc 1
;*****
; GLHN: feedforward and hopfield sections

; GLHN: begin feedforward layer
mull glhn_input, 0.0
mulap Wff_mtx + 0
mula glhn_input + 1, 0.0
mulap Wff_mtx + 1
lbtbf hop_input + 0, linear2
mull glhn_input, 0.0
mulap Wff_mtx + 2
mula glhn_input + 1, 0.0
mulap Wff_mtx + 3
lbtbf hop_input + 1, linear2
; end feed forward

#end_proc

; bring all writes forward for next
; calculation
inbm

#proc 1
; copy hop_input to buffer
mull hop_input + 0, 1.0
lbtbf hop_input + 0, linear1
mull hop_input + 1, 1.0
lbtbf hop_input + 1, linear1

#end_proc

; GLHN: begin recurrent hopfield section
;
lbc 101; 101 iterations
hop_loop: nop

;wr glhn_range ; switch to hopfield range

#proc 1
mull Wfb_mtx + 0, 0.0
mulap glhn_output + 0
mula Wfb_mtx + 1, 0.0
mulap glhn_output + 1
mula hop_input + 0, 1.0
lbtbf glhn_output + 0, linear2
mull Wfb_mtx + 2, 0.0
mulap glhn_output + 0
mula Wfb_mtx + 3, 0.0
mulap glhn_output + 1
mula hop_input + 1, 1.0
lbtbf glhn_output + 1, linear2
#end_proc

;wr 1 ; reset range

; bring all writes forward for next calc.
inbm

djmp hop_loop
; GLHN: end recurrent hopfield section

#proc 1
; copy glhn_output to buffer
mull glhn_output + 0, 1.0
lbtbf glhn_output + 0, linear1
mull glhn_output + 1, 1.0
lbtbf glhn_output + 1, linear1

; move mlp_input into GLHN neuron range
mull mlp_input + 0, 1.0
lbtbf mlp_input_temp + 0, linear3
mull mlp_input + 1, 1.0
lbtbf mlp_input_temp + 1, linear3
#end_proc

; bring all writes forward for next calculation
inbm

#proc 1
; calculate new mlp_input
mull mlp_input_temp + 0, 1.0
mula glhn_output + 0, -1.0
lbtbf mlp_input + 0, linear4
mull mlp_input_temp + 1, 1.0
mula glhn_output + 1, -1.0
lbtbf mlp_input + 1, linear4
#end_proc

; bring all writes forward for next
; calculation
inbm
stop

```


APPENDIX F

ROBOT APPROXIMATION IN CHAPTER 6

The planar (2-joint) forward kinematics of the Extendable Stiff Arm Manipulator (ESAM) in Chapter 6 are approximated by a multilayer perceptron (MLP) with 25 hidden PEs. The MLP $f: \mathcal{R}^2 \rightarrow \mathcal{R}^2$ maps joint angles θ to Cartesian coordinates x ,

$$x = f(\theta) = W \cdot \sigma(V\theta + b). \quad (\text{F.1})$$

The weight matrices $V \in \mathcal{R}^{25 \times 2}$, $W \in \mathcal{R}^{2 \times 25}$, and $b \in \mathcal{R}^{25}$ were trained using the neural network simulator program NeuralWorks/Professional II. The learning procedure and the weight matrices are listed below. Although rather imprecise, the MLP approximation is sufficient for the present control context, since the compensation for model errors is to demonstrated. Furthermore, MLP *training* is beyond the scope of this work.

F.1 Training and Testing

The training data, i.e. I/O pairs {joint angles θ , desired Cartesian coordinate x }, were created using the 2-dimensional ESAM forward kinematics in Equation D.15. 2000 joint angles $\theta = [\theta_1, \theta_2]^T$ in the interval $5^\circ < \theta_i < 85^\circ$ were randomly chosen and the associated Cartesian coordinates x were determined. Another 2000 I/O pairs were created with $5^\circ < \theta_i < 40^\circ$ for a fair representation of all partitions of the Cartesian work space during training. The training data were presented to the MLP 100000 times with an epoch size of 10 (weight updates every 10 presentations). The learning rate and momentum term were changed according to the following schedule:

Iteration	..10000	..30000	..70000	..100000
Hidden Layer				
Learning Rate	0.30	0.15	0.0375	0.0023
Momentum Term	0.40	0.20	0.0500	0.0031
Output Layer				
Learning Rate	0.15	0.075	0.0188	0.0012
Momentum Term	0.40	0.20	0.0500	0.0031

The resulting approximation is illustrated in Figure 33 for the joint angle trajectory given by $15^\circ < \theta_1 < 80^\circ$ and $15^\circ < \theta_2 < 45^\circ$.

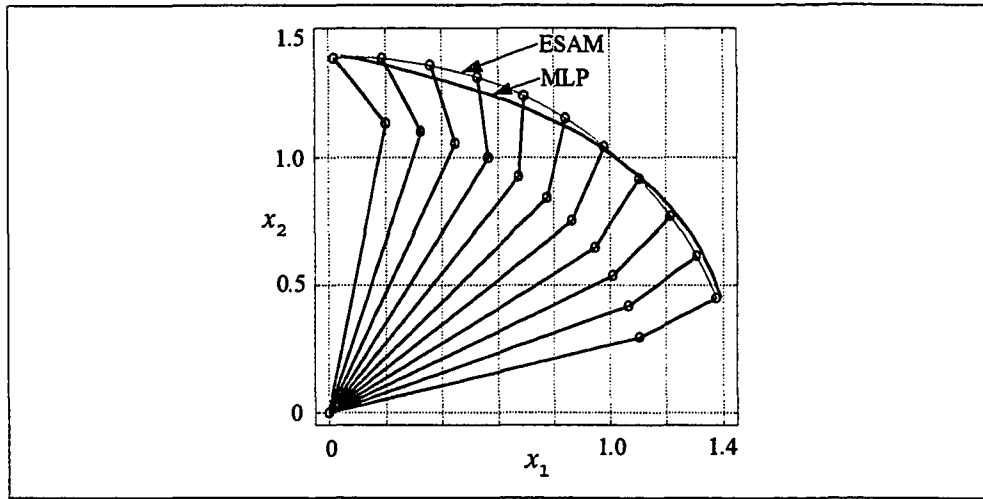


Figure 33. MLP approximation of the 2-joint ESAM forward kinematics.

F.2 Learned Weight Matrices

$V = \begin{bmatrix} -1.5676 & -0.6778 \\ +1.4860 & -0.0586 \\ -2.0895 & -1.3990 \\ -0.5787 & -1.1007 \\ -1.9486 & -1.1629 \\ -2.3048 & -0.8382 \\ -0.9271 & -1.1570 \\ -0.7143 & -1.1328 \\ -2.2981 & -1.5234 \\ -2.7479 & -1.1982 \\ -1.7470 & -0.5587 \\ -2.7971 & -1.8828 \end{bmatrix}$

```

-0.8860 -0.5191
-2.6173 -1.0272
-2.4979 -1.2455
-1.2464 -1.3208
-2.2684 -0.6751
-2.1203 -0.7404
-2.2462 +0.1165
-2.0104 -0.6271
-2.0243 -0.7987
-2.3626 -1.3522
-2.0970 -0.9980
-2.3240 -0.6423
-2.8726 -1.6892 ];

```

```

b = [ -1.2054 ; -0.2007 ; -3.3947 ; -0.5118 ; -3.0471
        -0.6521 ; -1.1205 ; +0.8022 ; -3.1129 ; -1.7851
        -1.9445 ; -2.5863 ; -0.4851 ; -1.2263 ; -2.7818
        -1.6286 ; -1.9738 ; -1.0340 ; +2.4479 ; -1.6926
        -0.7195 ; -2.7059 ; -3.0515 ; -1.6839 ; -2.3710 ];

```

```

W = [-0.0717 -0.9912 -0.2814 +0.6790 -0.5250...
        -0.4215 +0.4300 +1.0005 -0.2665 -0.1676...
        -0.2303 +0.0546 +0.4994 -0.2729 +0.0806...
        +0.0312 -0.0589 -0.5488 +1.5108 -0.0782...
        -0.4145 -0.2241 -0.2197 -0.3608 -0.1948 ;
        -0.2537 +1.5411 -0.6733 +0.3137 -0.5738...
        -0.6311 -0.0524 +0.4014 -0.6847 -0.5607...
        -0.2590 -0.6808 -0.0156 -0.5817 -0.5851...
        -0.1615 -0.4353 -0.3969 -0.0742 -0.3810...
        -0.4895 -0.5659 -0.5479 -0.4338 -0.6688 ];

```