In-the-loop training of a VLSI implementation of a smart sensor with low resolution programmable digital weights.

Jinming (Jim). Yang

University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
Yang, Jinming (Jim), "In-the-loop training of a VLSI implementation of a smart sensor with low resolution programmable digital weights." (2000). Electronic Theses and Dissertations. 2492.
https://scholar.uwindsor.ca/etd/2492

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48108-1346 USA
800-521-0800

UMI
In-The-Loop Training of a VLSI Implementation of a Smart Sensor with Low Resolution Programmable Digital Weights

by

Jinming (Jim) Yang
B.Sc., Tsinghua University, 1987
M.A.Sc., Tsinghua University, 1989

A Dissertation
Submitted to the College of Graduate Studies and Research through the Electrical and Computer Engineering Program in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy at the University of Windsor

Windsor, Ontario, Canada

May 2000

© 2000 Jinming (Jim) Yang

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-52447-7
ABSTRACT

A method for the in-the-loop training of neural networks with low resolution synaptic weights is developed in this thesis. This research was motivated by the need to train an intelligent sensor that had been designed and fabricated with low resolution weights in order to meet constraints that were imposed upon the designers. The training method developed in this thesis can be also be used as a fault tolerant training procedure for networks with high resolution weights that have been effectively reduced to low resolution weights due to malfunctioning circuits. The proposed training method is conceptually new and employs three distinct but interrelated parts that each required the development of new approaches.

In the first part, a model of each neuron activation function in the sensor is determined in terms of a small ideal neural subnetwork by using an in-the-loop system identification strategy that is based on a knowledge of the sensor's architecture.

In the second part, a complete neural network model for the sensor is developed that utilizes the known sensor architecture in conjunction with neural subnetworks in place of the individual sensor neurons. This model can be trained using continuous weights and is required as part of the strategy for training networks with highly quantized weights. The standard backpropagation training algorithm can not be used directly since no explicit analytical expression for the neuron activation function is available. A variation of the backpropagation algorithm has been derived in the thesis that uses input/output data from the subnetwork together with an approximated derivative
expression in order to reach convergence. The continuous weights determined using this algorithm can be readily quantized to their nearest allowable values so that the effects of weight quantization can be seen immediately.

In the third part of the method an algorithm for training networks with low resolution weights has been developed that measures the sensitivity of each weight with respect to the error function and then perturbs the weights with higher sensitivity in an iterative training/retraining procedure until the desired convergence is reached. The iterative procedure employs the neuron model derived in the first part together with the training method developed in the second part.

The application of the in-the-loop training procedure based on these three parts leads to a set of convergent synaptic weights that can be implemented exactly using low resolution digital multipliers.
DEDICATION

To my wife, my children and my parents

for all that they mean to me.
ACKNOWLEDGEMENTS

First, my special thanks go to my external examiner, Dr. S. Draghici, for his invaluable advice as well as for his offering relevant references in finalizing the thesis.

I would like to express my sincere thanks to my supervisor, Dr. W. C. Miller, for his advice, guidance and encouragement throughout the progress of this thesis.

My sincere appreciation also goes to my thesis committee: Dr. R. A. Frost, Dr. G. A. Jullien, Dr. M. Ahmadi, for their encouragement, advice and critical appreciation during the course of this research.

The advice and help from other faculty members in Electrical and Computer Engineering are also greatly appreciated.

My sincere appreciation also goes to the Natural Science and Engineering Research Council of Canada (NSERC) for its financial support during the academic years 1994-1999.

On a personal level, I am indebted to both my parents and my parents-in-law. They are an ongoing help to me and have made significant contributions to my research by supporting me throughout the work. Finally, I would like to extend my sincerest thanks and gratitude to my wife, Meng, for her moral support, help, encouragement and love.
TABLE OF CONTENTS

ABSTRACT .................................................................................................................. III
DEDICATION ............................................................................................................... V
ACKNOWLEDGEMENTS .......................................................................................... VI
TABLE OF CONTENTS ............................................................................................. VII
LIST OF FIGURES ..................................................................................................... XI
LIST OF TABLES ....................................................................................................... XV
NOMENCLATURE ...................................................................................................... XVI

CHAPTER 1: INTRODUCTION

1.1 AN OVERVIEW

1.2 ON IMPLEMENTATIONS OF NEURAL NETWORKS

1.3 THE SCOPE OF THE THESIS

CHAPTER 2: THE TRAINING OF PHYSICAL NEURAL NETWORKS

2.1 INTRODUCTION

2.2 A NEURAL NETWORK-BASED INTELLIGENT SENSOR

2.3 ON-CHIP, OFF-CHIP AND IN-THE-LOOP TRAINING

2.4 THE PROBLEMS OF TRAINING PHYSICAL NETWORKS

2.5 SOLVING THE PROBLEMS: THE METHODOLOGIES

2.6 SUMMARY

CHAPTER 3: NEURON MODELING FOR IN-THE-LOOP TRAINING

3.1 INTRODUCTION

3.2 MODELING NEURONS COMPRISED OF

DISTRIBUTED NONLINEAR RESISTORS

vii
4.3.1 THE STANDARD BP ALGORITHM
4.3.2 A BP-BASED ALGORITHM: NLBP
4.3.3 WEIGHT PERTURBATION METHODS
4.3.4 DIRECT SEARCH METHODS

4.4 THE APPROXIMATED SERIES-DERIVATIVE METHOD OF BP TRAINING

4.5 THEORETICAL ANALYSIS FOR THE APPROXIMATED
   DERIVATIVE METHOD
   4.5.1 THE ILLUSTRATIVE ANALYSIS
   4.5.2 THE ANALYTICAL EXPLANATION

4.6 TEST OF THE ALGORITHM
   4.6.1 PATTERN RECOGNITION WITH THE SMART SENSOR
   4.6.2 EXCLUSIVE-OR PROBLEM

4.7 SUMMARY

CHAPTER 5: TRAINING WITH LOW-RESOLUTION DIGITAL WEIGHTS

5.1 INTRODUCTION

5.2 ON USING DIGITAL WEIGHTS WITH LIMITED PRECISION AND RANGE

5.3 TRAINING ALGORITHMS USING LIMITED PRECISION WEIGHTS

5.4 ON DIGITAL WEIGHTS AND WEIGHT QUANTIZATION

5.5 A MATHEMATICAL ANALYSIS OF WEIGHT QUANTIZATION

5.6 THE TRAINING STRATEGIES FOR DEALING WITH DIGITAL WEIGHTS

5.7 A NEW ALGORITHM FOR OBTAINING
   OPTIMAL QUANTIZED WEIGHTS
   5.7.1 THE INTRODUCTION OF THE ALGORITHM
   5.7.2 THE WEIGHT SENSITIVITIES TO QUANTIZATION
   5.7.3 PERTURBING THE WEIGHTS OF HIGH SENSITIVITY
   5.7.4 THE TRAINING ALGORITHM AND ITS FLOW CHART
5.8 EXPERIMENTAL RESULTS

5.9 COMPARISON WITH SIMULATED ANNEALING METHOD

5.10 SUMMARY

CHAPTER 6: SUMMARY AND CONCLUSIONS

6.1 SUMMARY

6.2 CONCLUSIONS

6.3 FUTURE WORK

REFERENCES

APPENDIX: MATLAB SOURCE CODE

1: FUNCTION APPROXIMATION USING A SNN

2: THE AUTOCORRELATION TEST ON ERROR SIGNALS

3: PROGRAM FOR OBSERVING WEIGHT QUANTIZATION EFFECTS

4: PROGRAMS FOR TRAINING A LNN

5: SUBROUTINES

VITA AUCTORIS
LIST OF FIGURES

Figure 2.1: The Block Diagram of a \{4,3,2\} Feedforward Neural Network Built With the Hybrid Building Blocks. BB: Building Block; \(\xi_1, \ldots, \xi_4\): Input Units; \(V_1, \ldots, V_3\): Hidden Neuron Outputs; \(O_1, O_2\): Output Units.

Figure 2.2: The Schematic Diagram of the Multiplying DAC Synapse.

Figure 2.3: The Circuit Diagram of the Active Non-Linear Resistor.

Figure 2.4: The Transfer Characteristics of the Nonlinear Active Resistors.

Figure 2.5: A Block Diagram of the In-the-loop Training Mode.

Figure 2.6: Two Distinctive Performance Curves Using Weight Values Obtained By Training the Hardware and Software Networks Respectively (Adapted From Reference [53]).

Figure 2.7: The Comparisons of a Network Performance (Measured By Sum-Squared Errors) Between Using Continuous Weights and Quantized Weights.

Figure 2.8: The Training Patterns and Desired Outputs Used Throughout This Work.

Figure 3.1: A Family Curves Generated by the Equations (3.7) to (3.10).

Figure 3.2: Modeling a Physical Neuron Using a Small Neural Network (Subnet).

Figure 3.3: Modeling the Neuron Activation Function Using a Small Neural Network with 1 Input Node, 25 Hidden Neurons and 1 Output Node.

Figure 3.4: Identifying Activation Functions Connected in Series for Two Different Cases.
Figure 3.5: The Simulation Results When Modeling the Two Unknown Activation Functions Connected in Series by Two Subnets.

Figure 3.6: The Result of Modeling F1 Where F1 and F2 Are Connected in Series as in Figure 3.4.

Figure 3.7: The Complete Neural Network Model When All the Physical Neurons are Modeled by Subnets.

Figure 3.8: The Three Different Error Signals of the Identical MSE Values.

Figure 3.9: (a) A Feed-Forward Neural Network with One Hidden Layer and Two Layers of Nonlinear Neurons; (b) A Close-Up View of a Neuron Node with Inputs and Outputs.

Figure 3.10: The Autocorrelation Tests for Small Neural Networks with Hidden Neurons of (a) 10; (b) 25; (c) 30; (d) 45.

Figure 3.11: The Sum-Squared Error Ranges After 1800 Training Epoches.

Figure 3.12: The Block Diagrams of Transfer Characteristics of a Multiplication Stage Cascaded with a Neuron Circuit.

Figure 4.1: (a) The Complete Neural Network Model When All the Physical Neurons are Modeled by Subnets (SNN). (b) The Notations for Subnets.

Figure 4.2: The Diagram of a Two Layer Feed-Forward Neural Network, Showing the Notations for All the Units and Weights.

Figure 4.3: The Input-Output Characteristics of the Activation Function \( F(x) \) Realized by the Circuit Shown in Figure 2.3 and its Derivative \( F'(x) \).

Figure 4.4: The Illustration of the Two Distinctive Cases of Optimization Search Where the Method Works in (a) but Fails in (b).
Figure 4.5: An Illustration of the Utility of ADL Method in Getting-Out-Of a Local Minimum Trap. (a) The Difference of Gradient Descent Search and Derivative Approximation Search; (b) The Situation That Derivation May Get Trapped in a Local Minimum.

Figure 4.6: A Training Curve and Its Corresponding Learning Rate Curve for Training the Smart Sensor Using the Given Patterns.

Figure 4.7: The Four Training Curves of the Smart Sensor Network When Trained by the Patterns Given in Figure 2.8.

Figure 4.8: A Comparison of Training Curves of the Standard BP, a Weight Perturbation Method and Derivative Approximation Method.

Figure 4.9: The Diagrams of Neural Network Topologies That Can Be Used for XOR Classification Problem.

Figure 4.10: A Training Curve and the Corresponding Weight Locus of W2(2) Versus W1(2): The Case of a Relatively ‘Straight’ Locus.

Figure 4.11: A Training Curve and the Corresponding Weight Locus of W2(2) Versus W1(2): The Case of a Wavelike but still ‘Straight’ Locus.

Figure 4.12: A Training Curve and the Corresponding Weight Locus of W2(2) Versus W1(2): The Case of the Weight Searching Direction Changing Abruptly.

Figure 4.13: A Training Curve and the Corresponding Weight Locus of W2(2) Versus W1(2): The Case of a Relatively ‘Straight’ Locus but With a Chain-Like Part.

Figure 4.14: A Training Curve and the Corresponding Weight Locus of W2(2) Versus W1(2): The Case of a Twisted Locus at the Beginning Part.
Figure 4.15: A Training Curve and the Corresponding Weight Locus of W2(2) Versus W1(2): The Case of the Most Twisted Locus in the Simulations.

Figure 4.16: A Training Curve and the Corresponding Weight Locus of W2(2) Versus W1(2): The Case of a Quite 'Straight' Locus When Using a Weight Perturbation Algorithm.

Figure 5.1: (a) A Set of Weights; (b) The Histogram of the Set of Weights Shown in (a).

Figure 5.2: The Flow-Chart of the Proposed Algorithm for Dealing With Digital Weights.

Figure 5.3: A Training/Retraining Curve of the Proposed Algorithm Which Includes 20 Times of Training/Retraining Procedures.

Figure 5.4: A Close-Up View of the Training/Retraining Curve Shown in Figure 5.3.

Figure 5.5: The Sum-Squared Errors for Quantized Weights With Respect to the 20 Training/Retraining Procedures.

Figure 5.6: The Two Curves Obtained by Conducting the Same Training Task Using Different Initial Random Weights.
LIST OF TABLES

Table 4.1: The Key Numbers of Training Epochs for the Three Training Schemes.
Table 4.2: Input-Output Relationship of XOR.
Table 4.3: The Key Numbers of Training Epochs of the Three Training Schemes for XOR Classification Problem.
### NOMENCLATURE

- \(a, b\) : The random variables used for analyzing the proposed BP-based algorithm.
- \(B_n\) : The bit number of digital weights.
- \(C_{\text{ox}}\) : The oxide capacitance per unit area in MOS devices.
- \(d\) : The dimension of \(\hat{W}_N\).
- \(E, E_k, E_k^3\) : The output errors, cost function or energy function.
- \(e_k\) : The output errors used in the description of NLBP algorithm.
- \(e^{(h)}\) and \(e^{(o)}\) : The disturbance sources added to the hidden and output neurons.
- \(\Delta E, \Delta E_k, \Delta E_k^3\) : The incremental change in output errors, cost function or energy function.
- \(F\) : A function that is to be modeled by a small neural network.
- \(g\) : The symbol for activation function used throughout this work.
- \(H_0, H_1\) : The hypotheses of statistical tests presented in this work.
- \(h^{(k)}_i\) : The weighted summation of the signals from the \((k-1)\)-th layer (the previous layer) in a multilayer neural network.
- \(I_{\text{in}}\) : The input current of the neuron circuit.
- \(i,j,k,p,q\) : Integer indices.
- \(K_c\) : The coefficient used in deriving the desired properties of a subnet.
- \(K_n\) : The product of \(\mu_n\) and \(C_{\text{ox}}\).
- \(K_p\) : The product of \(\mu_p\) and \(C_{\text{ox}}\).
L_l,...,4: The channel length of a MOS device.

LC: The loop counter used in the flowchart of the algorithm for deal with quantized weights.

M: Mean value of the disturbance of e^{(h)} and e^{(o)}.

N_h: The number of hidden units.

N_i: The number of input units.

N_o: The number of output units.

N_t: The preset maximum number of trials of training.

N(m,P): Gaussian distribution function centered at m and with a variance P.

O_k: The k-th output.

P_k: The probability of a neuron that changes to the state of S_k=1.

P_N,P_w: The asymptotic covariance matrix of \( \hat{W}_N \).

p(x): The function that approximates the derivative of activation function.

R^N: The space of the N-dimensional real vectors.

r_s: The resolution of discretized weights.

\( \hat{r}_c \): The estimate of the covariance function of \( \varepsilon \).

SP_w: The vector set composed of weight vectors W, i.e., W \in SP_w.

T: The temperature used in the simulated annealing method.

V_{Bias_1}, V_{B1}: The first bias voltage for the neuron circuit in the smart sensor.

V_{Bias_2}, V_{B2}: The second bias voltage for the neuron circuit in the smart sensor.

V_k: The output from the k-th hidden neuron.

V_{out}: The output voltage of the neuron circuit used in this work.

V_{TN}: Threshold voltage of NMOS devices.
$V_{TP}$: Threshold voltage of PMOS devices.

$\overline{V}(\cdot), V_N(\cdot)$: The validity measures for model validation.

$w_{ij}$: A specific synaptic weight connected from i-th node to j-th node.

$w_{ij}^{(k)}$: The synaptic weight from i-th node to j-th node in k-th layer in a multiple layer neural network.

$\mathbf{w}$: A real vector composed of $(w_1, w_2, \ldots, w_m)$.

$w_{1..d}$: The channel width of a MOS device.

$\mathbf{W}, \hat{\mathbf{w}}$: A set of synaptic weights expressed by a vector in Euclidean space.

$\hat{\mathbf{W}}, \mathbf{w}^*$: A set of trained weights that are regarded as estimates from perspective of system identification.

$\Delta w_{ij}$: The weight change or the magnitude of weight perturbation for $w_{ij}$.

$\mathbf{X}$: A real vector composed of $(x_1, x_2, \ldots, x_m)$.

$x_i$: The normalized quantity used for estimating the covariance function $\tilde{f}_x$.

$\bar{x}_n$: The number of changes of sign in the error signal sequence.

$\mathbf{Z}^N$: The input-teacher data pairs for training a subnet.

$\beta$: The parameter that controls the shape of the function $p(x)$.

$\chi$: The ratio of the current error chance with respect to that of the previous time used in adaptive learning rate scheme.

$\chi^2(m)$: $\chi^2$ distribution function centered at $m$.

$\delta^a, \delta$: An intermediate variable used in the derivation process of BP algorithm.

$\varepsilon, \varepsilon(t)$: The prediction errors (also called error signal or residuals) used in system identification.
$\epsilon_1, \epsilon_2$: The small positive real numbers used in the adaptive learning rate as control parameters.

$\Phi$: The weight magnitude that is determined for weight quantization.

$\gamma$: A random variable used for determining the number of changes of sign in error signal.

$\eta$: The learning rate.

$\kappa$: The proportional coefficient of linear synaptic weights.

$\lambda^2$: The variance of white noise.

$\mu$: Integer index specifically used for the number of patterns.

$\mu_n$: The electron mobility parameter in NMOS devices.

$\mu_p$: The electron mobility parameter in PMOS devices.

$\hat{\theta}_N$: The N-dimensional vector of a set of estimated parameters.

$\rho(\cdot)$: A distance function that measures the distance between two given functions.

$\sigma(\cdot)$: The standard sigmoidal function.

$\tau$: The time constants used in statistical tests.

$\xi_k$: The input to the k-th input units.

$\psi$: The variable representing the weighted error information used in the analysis of the proposed BP-based algorithm.

$\Delta_0$: The quantization level of the least significant bit of digital weights.

$\Phi$: The set of quantized weights that can yield low output errors, which also represents the subspace of the weight space $\mathbb{SP}_w$. 
\( \Gamma: \) The maximum number of trials for the weight perturbation.

\( \Lambda: \) The variance matrix of white noise.
CHAPTER 1: INTRODUCTION

1.1 An Overview

Neural networks and intelligent systems in the wide sense are among the rapidly developing engineering fields today. Many systems utilizing smart sensors and intelligent systems function as part of a real-time engineering computing environment [1] in order to minimize risk and uncertainty.

Neural networks have been inspired by biological systems where large numbers of neurons, that individually function rather slowly and imperfectly in parallel [2], collectively perform tasks that even the current largest serial or distributed computers have not been able to match. Artificial neural networks are made of many relatively simple nonlinear elements or neurons connected to one another by paths whose gains or synaptic weights are adjusted as part of a training process. They differ from the standard Von Neumann computers in that they characteristically process information in a manner that is highly parallel rather than serial, and that they "learn" by adjusting interconnect path weights and neuron activation thresholds so as to minimize some error criteria.

In their current state, neural networks are especially useful in dealing with problems related to pattern recognition. Some existing neural network systems can efficiently and rapidly learn to separate enormously complex decision spaces [3]. The problems of coordinating many neural networks, each a specialist in dividing some
portion of the decision space, has also been solved [4]. Neural networks have been used to recognize handwriting characters, assembly line parts, signatures, diagnose engine and assembly line problems [5].

At the present time, neural network research has focused primarily in four areas: a theoretical analysis of their behavior; the development of efficient methods for training such systems; physical implementation using microelectronics; and the development of commercial applications for this class of computing systems. These studies have relied primarily on digital simulations of neural network models programmed on traditional computers. In order to realize the commercial potential of these massively parallel computing systems – a VLSI implementation of programmable neural networks must be developed.

1.2 On Implementations of Neural Networks

Neural networks have been shown to be a powerful computing paradigm for many important real-world applications. Thus, a great deal of effort has been expended searching for the best approach to implementing neural network computing models. Implementations of neural networks and related concepts can be done in both software and hardware. The three broad areas of neural network implementation are computer-based software, electronic hardware, and opto-electronic hardware [6].

In the software implementations based on using the current computers, one has to step sequentially through each interconnection to update the state of a neuron, a process that is painfully slow when the number of interconnections in a neural network is large.
Only with special-purpose hardware can one hope to exploit the massive parallelism inherent in neural network models.

Optical implementations of neural networks model neuron firing activity as a beam of light and use light modulators to realize synaptic weights [7]. Lenses and holograms are used to route neural signals (light) between neuron units. The basic neuron unit is typically composed of a photodetector, an electronic operational amplifier and a light emitting element. The use of electronic components in the neuron unit limits much of the speed potential promised by photonics.

Some electronic implementations of neural networks often encode neuron firing strength as either a voltage or as the frequency of voltage pulses [8, 9]. In analog circuitry, the synaptic weights are commonly modeled as either resistive or capacitive elements. A variety of proposals have been suggested for achieving variable gain resistive based synaptic weights in neural networks. Digitally programmable synaptic weights have also been used in conjunction with analog neurons by using digital to analog converters (D/A) together with digital memory and multiplier circuits [10-13].

The rapid development of VLSI technologies and algorithms in recent years has allowed for the implementation of analog, digital or mixed signal architecture for neural network. Some VLSI realizations of neural networks are especially sensitive to device parametrics and exhibit wide variations in operating characteristics [10, 14]. This is particularly true of CMOS VLSI implementations that map the activation function and synapses to the transconductance of p-load and n-load devices. The dynamic range in these devices is limited and device characteristics can drift as a function of time, temperature and repeated cycling of the charge stored on a floating gate. There also exists a mismatch
between our ability to fabricate massive numbers of synaptic elements on a VLSI chip and the relatively small number of wire bond connections that can be made to a VLSI chip. It is these wire bonds that can limit the bandwidth of information that can pass into a VLSI neural chip and this could become a critical design constraint that ultimately limits overall system performance [15-17]. Therefore, at present time, the scope of VLSI neural network applications will be basically determined by the development of VLSI technologies.

The special problems for VLSI implementations have been dealt with in [15]. The author pointed out that electronic implementations of neural networks are faced with several major constraints regarding: (1) Precision (accuracy); (2) Interconnectivity (fan-in and fan-out); (3) Area; (4) Power; (5) I/O bandwidth. A series of methodologies to conquer these problems have developed in [15, 18, 19, 20, 21, 22] which have formed a special category of VLSI implementation of neural networks called “VLSI friendly” design.

A VLSI-friendly neural network is defined as a network which can be conveniently implemented in VLSI hardware [130]. In this research, since the neural network with limited precision of weights is of particular interest, a VLSI-friendly neural network specifically refers to a neural network using exclusively integer weights in a narrow interval. A VLSI friendly constructive algorithm would be able to [20]: (1) Decide the appropriate architecture for the network; (2) Construct a solution network (both architecture and weights); (3) Do so using limited precision integer weights in a range chosen by the user; (4) Construct solutions with a limited fan-in chosen by the user. Theoretical foundations for limited fan-in problems are elucidated in [22]. While the fan-
in problems are normally considered together with digital weights of limited precision and range in designing a VLSI implementation of neural networks [22], the fan-in problems are out of the scope of this thesis and will not be discussed further here.

Physical realizations of large neural network computing systems require that a simple technique be found for implementing each of its principal components – neuron units and synaptic connections. There is some debate over the operational requirements for these components, but a number of characteristics are thought to be desirable. The programmability of these systems is enhanced when the synapses can assume a wide range of connection strength. Ideally, the synaptic connections should be continuously adjustable. It is also desirable that these synapses be dynamically programmable so that the neural system will be capable of adaptive behavior. Similarly, the neuron units should provide some mechanism for adjusting their activation threshold. Simulation results suggest that nonlinear response characteristics within the neuron suppress noise within the machine and help stabilize system behavior [9].

A neural network architecture can be implemented as an integrated circuit by using analog, digital, or mixed analog/digital structures. Generally speaking, the implementation in dedicated analog hardware offers an attractive alternative for reasons of speed, compactness and lack of limited resolution in computation that is present in digital hardware. The advantages of an analog implementation also include simple basic blocks and communication, which leads to small area and thus larger networks on a single chip. The reason for this is that multiplication can be realized based on Ohm’s law and summation on Kirchhoff’s current law. However, the storage of analog weight values for the synapse is difficult and also the noise immunity is worse than in digital
circuits. The neural network implementations using analog circuits have been studied extensively, e.g., [14, 17, 23-30]. A thorough review of analog hardware implementation can be found in [26]. On the other hand, the main advantages of digital circuit design techniques are high precision, ease of weight storage, ease of sequencing and multiplexing, ease of interfacing, ease of design and high reliability [31, 32]. However, the overall area of digital structures is larger because of the computation and communication methods employed in digital circuits. The asynchronous implementation, which is the case in most of the analog implementations, is closer to the biological world. Nevertheless, from the VLSI point of view the synchronous implementations, usually digital ones, are much easier to handle. Some examples of digital implementation of neural networks can be found in [33-36]. The mixed analog/digital implementations aim at capturing the advantages of both approaches and seeking a compromise between the limitations of the two distinct solutions. Many mixed-signal implementations have been proposed, e.g., [10-13, 31]. A typical example of a mixed signal approach is the use of a local digital memory for storing the values of the weights. Such digital weight values can use several bits and a D/A converter [1, 136, 140].

It has been found that in-the-loop training is necessary [33] due to the variations in the circuits used to construct the neurons and synaptic weights. These variations can arise from process variations that occur at the time of fabrication. Since a programmable neural network is desired the use of digital weights becomes essential. The discrete nature of the digital weights requires that any viable training procedure must be able to deal with weight quantization effects. This thesis deals with the in-the-loop training of neural networks based smart sensors with low-resolution programmable digital weights.
1.3 The Scope of the Thesis

This thesis develops an in-the-loop training method for a VLSI implementation of a feed-forward neural network with low-resolution programmable digital weights. Chapter 1 provides an introduction to neural networks and outlines the material found in the various chapters of the thesis. Chapter 2 addresses some issues on the training of the VLSI implementations of neural networks, such as the training modes, the problems encountered in training neural network hardware implementations. In the first section of Chapter 2, a VLSI neural network integrated circuit (IC), the smart sensor, is described in details as it will be used throughout in this work as a prototype of neural network implementation.

Chapters 3 to 5 are dedicated to both solving these problems and detailing the proposed in-the-loop training method. In Chapter 3, the procedure of modeling the neuron activation function during the in-the-loop training is presented. The related issues such as the principle of neuron modeling and model validation from system identification perspective are also considered. Additionally, discussions on some other perspectives, such as the comparison with the look-up table method and the treatment of the nonlinearities of synaptic weight multiplication, are also presented.

In Chapter 4, a modified Backpropagation (BP) algorithm is described which can be used as an alternative to the weight perturbation-based methods and can also take advantages of the BP algorithm. The explanations of the proposed algorithm from both an illustrative and an analytical perspectives are considered. Due to the fact that nonlinear systems are involved it is not possible to obtain a closed-form analytical expression for the convergence properties of the algorithm, our analyses have heavily
relied on experimental results. Besides the results of testing the algorithm on the smart sensor, the simulation results of solving XOR classification problem is given and the visualization of weight updating process is illustrated. Following this are discussions on related methods including the so-called feedforward training method and the nonlinear BP algorithm.

Chapter 5 is dedicated to dealing with the issues arising from weight quantization effects. The chapter starts with the review of issues on using limited precision weights and the state-of-the-art training algorithms for the purpose. The definitions of digital weights and weight quantization are then given. A mathematical analysis of weight quantization effects is presented which serves as a general procedure of analyzing the weight quantization effects from statistical viewpoint. The available training schemes for dealing with digital weights are reviewed and then the existent problems are analyzed. In order to solve the problems, we presented a new approach to dealing with weight quantization effects. The experimental results show the usability and efficiency of the approach. In the last part of the chapter, the nature of the proposed approach is analyzed by comparing to the simulated annealing method.

The summary and conclusions are presented in the last chapter. The last chapter also serves as the summary and highlight of the independent contributions of this work. Finally, future work is proposed which can be regarded as the potential extensions of this thesis.
CHAPTER 2: THE TRAINING OF PHYSICAL NEURAL NETWORKS

2.1 Introduction

Researchers in the VLSI research group at the University of Windsor have developed a number of implementations of smart sensors that are based on the use of a neural network with an integrated photosensitive array. The most recent designs [37, 38] feature a modular approach that employs programmable digital weights. The neuron activation function has been realized by using a number of distributed nonlinear resistor connected in parallel. The purpose of this research is to develop a robust method for the in-the-loop training of these smart sensors.

2.2 A Neural Network-Based Intelligent Sensor

In this section, we shall first review the mixed-signal implementation of a neural network intelligent sensor described in [10, 37] so as to ensure that the proposed in-the-loop training strategy is applicable to this type of architecture. The programmable smart sensor that is to be trained consists of a VLSI realization of a multi-layer feedforward neural network with an integrated photosensor array [10, 37, 39]. The smart sensor architecture is based on a hybrid or mixed-signal VLSI neural network implementation that features design simplicity, scalability, area efficiency, reduced interconnection problems and increased robustness [10]. These advantages have been shown in a number of papers [39-42].
The most significant feature of the sensor is its modular architecture consisting of a number of universal digital/analog building blocks. Each universal building block is comprised of a multiplying Digital-to-Analog converter (DAC) synapse, a portion of nonlinear distributed neuron and compact digital registers for programming and storing a

Figure 2.1: The Block Diagram of a (4,3,2) Feedforward Neural Network Built With the Hybrid Building Blocks. BB: Building Block; \( \xi_1, \ldots, \xi_4 \): Input Units; \( V_1, \ldots, V_3 \): Hidden Neuron Outputs; \( O_1, O_2 \): Output Units.
synaptic weight value. The modular architecture has been used to design an optically-coupled neural network for on-line process control and/or pattern classification applications. A \( \{N_i, N_h, N_o\} \) feed-forward neural network with \( N_i \) input units, \( N_h \) hidden neurons and \( N_o \) output units can be implemented simply by interconnecting regular \( (N_i \times N_h) \) and \( (N_h \times N_o) \) arrays of the building blocks. Figure 2.1 shows a typical \( \{4,3,2\} \) neural network built with \( 4 \times 3 + 3 \times 2 = 18 \) hybrid blocks.

The schematic diagram of the multiplying DAC synapse is shown in Figure 2.2. The multiplying DAC receives 5-bit digital signals from a weight register with bits \( D_0, \ldots, D_3 \) giving the magnitude of the input and bit \( D_4 \) determining the sign. The multiplying DAC synapse consists of a V-to-I converter at the input, a set of binary-weighted current mirrors and a sign-bit circuit that switches the direction of output current.
The neurons used in the smart sensor consist of a number of distributed nonlinear resistors connected in parallel. Each nonlinear resistor has been realized using a four-transistor active circuit implementation. A circuit with four MOS devices that approximate a sigmoidal-like nonlinear activation function has been used. The s-shaped function is constructed using 4 quadratic segments that are joined together by carefully selecting the bias voltages ($V_{Bias_1}$ and $V_{Bias_2}$) and device geometry of the four transistors. Due to the employment of distributed neurons in the smart sensor, the fabrication imperfections can be compensated to some extent [37]. The transistor channel widths are adjusted to reach 0 to 5V at extreme synaptic currents, i.e., ±100μA. The circuit diagram is given in Figure 2.3, and the simulation results for the V-I characteristics of a 4-MOS nonlinear neuron and a 2-MOS version of neuron circuit given in [43] are shown together.

![Circuit Diagram](image)

Figure 2.3: The Circuit Diagram of the Active Non-Linear Resistor.
in Figure 2.4 for the purpose of comparison. Compared to the neuron characteristics obtained in [43] (a curve in Figure 2.4), the circuit shown in Figure 2.3 provides a smoother transition between the segments (b curve in Figure 2.4). This has been achieved by slightly overlapping the conduction regions of M3 and M4 so that a differentiable characteristic is obtained. A differentiable neuron characteristic is desirable for in-the-loop training using popular gradient-based algorithms. The conduction regions of each of the 4 transistors are specified as follows:

\[(f_1)\ 0<V_{out}<V_{Bias1}-V_{TN}: \quad M4 \text{ and } M2 \text{ "ON"}, \text{ other devices "OFF";}\]

\[(f_2)\ V_{Bias1}-V_{TN}<V_{out}<V_{Bias2}-V_{TN}: \quad M4 \text{ "ON"}, \text{ other devices "OFF";}\]

\[(f_3)\ V_{Bias1}+|V_{TP}|<V_{out}<V_{Bias2}+|V_{TP}|: \quad M3 \text{ "ON"}, \text{ other devices "OFF";}\]

\[(f_4)\ V_{DD}>V_{out}>V_{Bias2}+|V_{TP}|: \quad M3 \text{ and } M1 \text{ "ON"}, \text{ other devices "OFF";}\]

where \(V_{TN}\) and \(V_{TP}\) are NMOS and PMOS threshold voltage respectively. We assume \(V_{Bias1}-V_{TN}<V_{Bias2}-V_{TN}\); at the same time \(V_{Bias2}-V_{TN}\) should be equal to or slightly greater

![Figure 2.4: The Transfer Characteristics of the Nonlinear Active Resistor.](image-url)
than $V_{Bias} + |V_{TP}|$ (overlapping condition).

The nonlinear property of each resistor may vary slightly due to fabrication process validation that may occur from one corner of the die to another. However, since each neuron is based on a number of nonlinear resistors distributed over the die area, the effects of process variations on a neuron's activation function is minimized by an averaging effect [37]. Variations in the neuron activation function that deviates from the assumed ideal training model can cause significant changes in the network outputs. Thus even with a robust neuron circuit realization the need for an in-the-loop training methodology exists.

Design constraints imposed on the allowable area of the smart optical sensor severely limited the number of bits and hence size of the programmable synaptic weights. The designers found [10] that it was necessary to create a custom layout for a multiplier with a resolution not exceeding 5 bits in order to satisfy the design criteria. Thus any in-the-loop training procedure developed for a smart sensor of this nature must be able to mitigate the effects of weight quantization on the training algorithm.

In this thesis, we are primarily concerned about the nonideal nature of the neuron activation function and the quantization effects imposed by low-resolution implementations. The present smart optical sensor has no on-chip learning/training capabilities and thus in-the-loop training must be relied on exclusively.

2.3 On-Chip, Off-Chip and In-The-Loop Training

In order to train an IC chip of a neural network there are three possibilities of interest: on-chip training, off-chip training and in-the-loop training [26].
On-chip training: The realization of an on-chip training and learning capability is desired by many researchers as a primary requirement when designing neural network ICs as in [44-48] [141]. Some researchers believe that a training and learning process must be carried out on-the-chip in order to exploit the network parallelism without creating an input-output bottleneck [46]. However, the implementation of on-chip learning can require a hardware structure of formidable complexity and the use of a comprehensive training algorithm [49, 50]. In addition, the errors and approximations inherent in analog circuitry, combined with synaptic weight quantization effects may prove to be very difficult to accommodate in the training algorithm [51].

Off-chip training: Given a specific training task, the training can be carried out on a separate computer based on a selected neural network model. Upon the completion of the training, downloading the obtained weights onto the VLSI implementations terminates the process of "off-the-chip" training. Off-chip learning maximizes the flexibility of training approaches and of the architectures that can be chosen for the network. The major problem associated with off-chip training is that there is no guarantee that the model on which the training is based fully reflects the characteristics of the physical VLSI implementation of the neural network being trained. In real-world applications off-chip training still suffers from the effects of idealized neural network models and need for generating all the input/output data prior to training [52].

In-the-loop training: Alternatively, the chip can be subject to a "chip-in-the-loop" learning mode. In this mode of training, the network itself generates the outputs for given inputs and synaptic weight values. A neural network model and training algorithm is still required to be resident on the associated host computer. This mode of learning
enables a more robust training as the characteristics and imperfections (nonidealities) of the physical neural network IC are included and accounted for in each weight adaptation step [53].

The in-the-loop training strategy offers a good trade-off between the hardware complexity problems involved in realizing an on-chip training capability and the constraints associated with off-chip learning described above. Generally speaking, a

![Diagram of the In-the-loop Training Mode](image)

**Figure 2.5: A Block Diagram of the In-the-loop Training Mode.**

training procedure is categorized into in-the-loop training mode if the training is carried out on a network model by the aid of computer software and the neural network chip can interact with the resident training software during the training process. As illustrated in Figure 2.5, the input data and the synaptic weights are controllable by the in-the-loop computer and the output data and output errors are observable. Note that all the synaptic
weights are controllable but not observable, this causes some difficulties in modeling the neural network implementation. The modeling for the neural network hardware is usually conducted via the in-the-loop data on the computer. When the training is done, the computer should be able to come up with a set of weights such that the neural network hardware being trained will produce the expected outputs by simply downloading the weights into the neural network hardware. The training strategies described in [24, 49, 53] belong to this category of training modes.

It should be pointed out here that in the course of designing the smart sensor, the emphases have been given to modularity, simplicity and scalability instead of on-chip learning capability, thus in-the-loop training strategy has been the original choice.

In this work, we have used two terms \textit{learning} and \textit{training} interchangeably. Learning means that an intelligent machine of neural network architecture can actively adapt the strengths of its weight connections to its environment by achieving some certain requirements; while training means that the weight connections of a neural network are \textit{passively} adjusted by human's supervision. Thus the two terms represent the same concepts from different perspectives [54].

2.4 The Problems of Training Physical Networks

One of the most interesting aspects of neural networks is their learning capability. A wide variety of learning algorithms have been developed, among which most of them are limited to the use in a software environment. The training is carried out by adaptively changing the interconnection strengths between neurons in a neural network. In this way, for example, a classifier can be built, not by programming the network, but by presenting it with a number of training examples and allowing the network to build up the
discriminating function automatically. With a proper weight adjustment algorithm, for example, the standard BP algorithm, numerous presentations of the training data can produce a network that gives the correct input-output relations for the training data. If the network has the proper architecture and if there are sufficient training data, the network may be able to generalize, i.e. it will also give correct outputs for input data it has never seen before [55].

From the implementation point of view the most important aspect is that many learning techniques require interconnection weights that are adjustable in small steps. Such an interconnection requires considerable circuitry and is difficult to build in a small die area in VLSI regardless of the selected circuitry [22, 56]. In a digital circuit, the die area for implementing a Digital-to-Analog conversion circuit block, that is normally needed in a hybrid implementation of neural networks, will increase exponentially with the resolution of the digital weights. Various approaches to building analog networks with a high resolution in the weights have been explored. However, they are not so successful as expected due to the existence of nonidealities inherent with analog circuitry. For example, it seems that during learning the BP algorithm requires a resolution of more than 8 bits in the digital weights in order to learn a problem large enough to be of practical interest [57]. Analog circuits with high-precision synaptic weights can be built [58] but the advantage of a smaller area compared with a digital circuit is lost. Therefore, analog circuits are of greatest interest where only moderate precision is required.

In the retrieving (recall) phase a neural network is very tolerant to low precision in the weights as well as in the neuron states. Typically, there are large numbers of inputs contributing to one result and statistical errors are reduced due to averaging.
effects. In one example, in a network with 60,000 weights, the resolution in the weights was reduced to 5 bits and the neuron states were quantized to just 3 levels throughout most of the network. Despite this reduction in resolution the performance of the network remained unchanged compared with the network that had the full precision 32 bits in the weights and the neuron states [57]. For training procedures using the BP algorithm, however, weights of high precision are required. Thus learning algorithms have to be wisely chosen and/or modified so that they are tolerant to imperfections/nonidealities of the analog hardware as well as the weights of low precision.

Some characteristics as well as some trade-offs and issues of analog hardware implementation of neural networks have been studied extensively in [26]. Parameters such as precision, chip area, power consumption, speed and noise susceptibility are also discussed in the paper. In the rest part of this section, we summarize the general problems encountered when training a hybrid VLSI implementation of a feedforward neural network like the smart sensor. The three major problems of interest in this work are discussed in the following.

(1) Analog Circuit Implementation: The first problem of training a hardware implementation of a neural network is due to the existence of nonidealities typical of analog hardware systems. Generally speaking, these nonidealities may include fabrication variations, offset voltages, DRAM cell leakage currents, noise induced by switches and clock feedthrough [25]. In VLSI implementation of neural networks, the circuit nonidealities may include, more specifically, nonlinear synapses, neurons that deviate from idealized activation functions, or errors and limitations in storing weights. The nonidealities, which may have a negligible
effect on digital circuits, can be particularly troublesome in the fabrication of analog circuits since the circuit function depends critically on the component parameters such as the channel length and width of transistors. For the case of the smart sensor, the nonidealities considered in this work are mainly related to the offset voltages found in the activation function circuit and the nonideal approximation to the sigmoid function using the circuit illustrated in Figure 2.3 as well as the discontinuities present in the approximation. The nonidealities may be related to the robustness of the neuron circuit design and/or process variations over the area of the neural network implementation.

When the neurons of a programmable neural network are realized by an analog CMOS implementation there can be a significant deviation from the

![Figure 2.6: Two Distinctive Performance Curves Using Weight Values Obtained By Training the Hardware and Software Networks Respectively (Adapted From Reference [53]).](image)

20
desired nonlinear activation function. Thus when a neural network is trained using an idealized representation for the neuron activation function the resultant set of weights are incorrect for use with the physical implementation of the network. A test of the nonidealities associated with analog neural network is given in [53]. The authors chose the system identification problem of a ballistic trajectory and the training curves are shown in Figure 2.6, in which the performance of the networks when the training is done only on the software network and resulting weights are implemented in both the hardware and software networks. In this experiment, the synaptic weight values were obtained by training for the ideal simulated network only. These same weight values were then used for both the simulation and the hardware. Since the parameters used in the simulation were determined from measurements of the average behavior of the network, this is equivalent to designing the network in simulation and then implementing it in hardware. The results clearly show that the performance of the hardware when it is trained in simulation is much poorer than that obtained by directly training the network itself. Although the simulation accurately accounts for the behavior of the network as a statistical ensemble, the hardware’s behavior depends on the detailed nature of each of its elements. For a particular input and a set of synaptic weights, if we calculate the expected output of each neuron in the network, and then compare these calculations with the experimentally observed values, we find only a vague correspondence.

The ability of the network hardware to compute at a level of accuracy greatly exceeding that of its components is not a result of the average response of a large
ensemble of components. It is, instead, a consequence of the adaptive process itself [51]. In fact, a neural network is trained to execute its particular analog computational tasks in the presence of its individual analog circuit nonidealities. In VLSI implementation of a neural network like the smart sensor, the degree of variation in the components is enough to render almost useless a precalculation of the synaptic weight values. However, by adaptively training in the presence of these same variations in the in-the-loop manner, their effects on the network can be reduced to a negligible level.

(2) Weight Quantization Effects: The second problem in the training of the smart sensor which is also applicable to all the neural network implementations with digital weights arises from weight quantization (or discretization) and limited weight dynamic range. The topics have been studied extensively both from simulation experiments as well as the theoretical and practical perspectives, e.g. [59—76, 137, 138]. The storage of weights has been one of the problems encountered in the design of neural network circuitry both in digital and in the analog implementations. Many different techniques have been employed for weight storage, such as digital memories (either on-chip or off-chip) and analog circuitry using capacitors and incorporated refreshing circuitry. The common question here is the amount of accuracy required to store the weights. In digital memories, this question manifests itself in a choice between wider word and lengths in memory registers that will result in trade-offs in size and, possibly, speed. The question that has to be addressed here is the minimum number of bits necessary for successful operation. From the viewpoint of VLSI IC design of a
digital weight-based implementation, the weight range and resolution are in fact one specification, i.e., the range is fixed when the resolution and the digital-to-analog conversion ratio are given and vice versa.

Although the required weight range and precision depend strongly on the problem difficulty [61, 62], it is generally believed that five-bit weight registers are normally sufficient for the feedforward computations, but 12-bit resolution is necessary for learning [60]. In order to learn a problem that is large enough and

Figure 2.7: The Comparisons of a Network Performance (Measured By Sum-Squared Errors) Between Using Continuous Weights and Quantized Weights.

of practical interest using the BP algorithm the number of bits in the digital weight should be at least eight, which represents a lower bound for digital weight
resolution from [57]. More updated results show that even 8-bit weights are sufficient for the retrieving (execution phase), but 14-16 bits are needed if backpropagation is used for learning [65, 66, 67]. The same catastrophic failure of learning (i.e., learning is no more possible) at 12-14 bits has been reported in [68, 69, 70] for normal computations; but the authors have come with a convincing solution which alleviates this problem: probabilistic rounding. By using their method, 6-8 bits may be sufficient for most problems.

Since a unified conclusion cannot be drawn for the number of bits of digital weights, in the implementation of the smart sensor, five-bit weight registers are employed as a result of design simplicity and area efficiency constraints. When a

![Figure 2.8: The Training Patterns and Desired Outputs Used Throughout This Work.](image)

VLSI neural network chip is trained in-the-loop, one expects that downloading the quantized weights obtained from training the software-based neural network
model into the chip will produce the identical results (or the best possible results it can) as observed in the simulations.

One of our experiments shows that using the quantized weights obtained by discretizing the resultant continuous weights could generate totally unacceptable output errors compared to the results obtained from the corresponding continuous weights. Thus, one cannot simply quantize the continuous weights by rounding the weight values to their nearest integers. In Figure 2.7, the comparisons of a network performance (measured by sum-squared errors) between using continuous weights and quantized weights are given. The figure illustrates the simulation results where the sum-squared errors (SSE) evaluated from the quantized weights could be very different from those evaluated using the corresponding continuous weights. In the experiment, a neural network that has the same structure with the prototype smart sensor in a software environment is trained 20 times on software by using the patterns shown in Figure 2.8. The patterns have the number of input units is 64 (8×8 photosensor array), while the number of the desired output is 4. Throughout this work, we will use these patterns which are the same patterns as appeared in [10]. The training processes are repeatedly performed for twenty times with each individual training process starting from a different and random initial set of continuous weights. Then we obtain twenty different sets of continuous weight solutions all of which will meet some preset training criterion (SSE=0.02). Followed is the weight quantization procedure that will be applied to all the continuous weight solutions. It can be seen from Figure 2.7 that the error differences in the trials No. 14 and No. 19 are
acceptable while the error difference of the trial No. 13 is significantly large and thus unacceptable.

Henceforth, when a neural network IC chip is fabricated, one of our objectives of the in-the-loop training becomes how to determine a set of weights that meet the given training criterion and meanwhile, are most robust to the weight quantization effects. Some new algorithms have been designed for constructing neural network hardware with low-resolution digital weights as in [71-78], which shall be discussed in more details in Chapter 5.

(3) Training Algorithms: The third problem of training a VLSI implementation of a neural network is the inefficiency of the current training algorithms. It is known that most of commonly-used training and/or learning algorithms are based on the assumptions that the neurons' characteristics can be modeled by some explicit analytical expressions, that the weight resolution and range are not limited and that the calculations are conducted such that round-off and truncation errors are not significant. However, these assumptions are not applicable to the training of the hardware implementations of neural networks like the smart sensor, as already stated earlier in this section. Therefore, the most popular training scheme, the standard BP algorithm, is usually not suitable for the purpose of training a VLSI IC realization of neural network.

Generally speaking, the main drawbacks of BP algorithm are well known; it gets trapped at local minima, it converges relatively slowly, it exhibits scaling problems, and it cannot be easily implemented using VLSI technologies [79]. The first three points are inherent in the standard BP algorithm. More particularly
for the last point, we notice that the standard BP algorithm assumes that both the nonlinear neuron activation function and its derivative are known analytically. When an actual physical realization of a neural network is to be trained in-the-loop, neither of these two functions are known.

2.5 Solving the Problems: The Methodologies

The primary objective of this work is to solve the problems of training a VLSI implementation of neural network, such as the smart sensor introduced in the first section of this chapter. In section 2.3, it was shown that the most popular and widely-used training algorithm, the standard BP, cannot directly be employed for the training of the smart sensor. As to other available training methods, one basically has two categories of theoretically feasible training methods. One of them is an exhaustive search method (such as a genetic algorithm which shall be mentioned in Chapter 5) which is not discussed further in this context since it is extremely computationally expensive and thus suffers from very slow convergence rates. The other category of training methods is the weight perturbation-based methods. This category of methods is well suited to the training of analog VLSI implementations of neural networks for the reason that no assumptions about the synapse and neuron circuit's characteristics and error gradients are measured rather than calculated using any analytical mathematical equations [46]. In addition, weight perturbation-based methods are easier to be implemented in hardware than the BP method.

The weight perturbation-based methods are combinatorial in nature and have the time complexity of $O(N^4)$ where $N$ is the number of synaptic weights in the network [80]. Even an improved algorithm from the basic weight perturbation method described in [80]
is of the time complexity of $O(N^3)$ where the improvement in time complexity is obtained at the expense of a more complex architecture. Thus, the training using a weight perturbation method involves a large amount of computations and convergence is not guaranteed due to the existence of local minimums.

The other problem associated with weight perturbation-based methods is that they still require a gradient calculation. This might present problems especially in case of very large neural network architecture (with a few thousands of weight connections), where the change in the error due to the weight perturbation is very small [45]. Obviously, the perturbation size determines the accuracy of the estimated gradient, which in turn determines the efficiency of the weight perturbation method. More specifically, for a large neural network, and especially when the training process approaches its last stage, the output error would become very small. At this point, in order to further reduce the output error, the perturbation size has to be very small too. This means the perturbation size in the last stage of training should be much smaller than, or at least comparable to, the output error. For instance, in [80], the convergence criterion was 0.001, while the perturbation size was chosen as 0.00001 to ensure the output error was reduced below the training criterion. This requirement on the perturbation size is obviously not met when digital weights with low resolution are employed since the smallest possible change of a weight value is the size of weight resolution.

In fact, weight quantization effects complicate any gradient-based calculation. Therefore, previously available in-the-loop training schemes [24, 49, 53] for physical neural network VLSI implementation did not deal with the problems associated with weight quantization when low resolution programmable digital weights were employed.
due to area constraints. Hence, there exists a need to develop an alternative training methodology for neural network based devices using programmable low-resolution weights.

The method developed in this thesis to train neural networks with programmable low-resolution weights has three distinct parts. The first part of the method consists of constructing a model for the VLSI implementation of the neural network being trained in the in-the-loop training environment. The model is a mapping from the physical neural network to the in-the-loop computer that includes: the actual structure of the network, a set of "quantizable" weight variables that emulate the programmable digital weights, and the various neuron activation functions with actual transfer characteristics. In constructing the model, the only information that is not readily available from a priori knowledge of the network is the transfer characteristics of the nonlinear neuron activation function. In this work, the neuron activation function has been modeled with an embedded small feedforward neural network whose structure has been determined using measured in-the-loop data from the actual network. In this manner, the required training can be carried out on a constructed model that is equivalent to the original physical neural network. The modeling procedures and the related issues are elaborated in Chapter 3. The resultant model allows the use of "continuous weights" as opposed to highly quantizable digital weights and thus a weight perturbation method of training is practicable.

In the second part of the method, it is necessary to train the model constructed in part one. Since analytical expressions for the neuron activation function are not available, as the neurons have been modeled with an ensemble of small feedforward
neural networks, the standard BP algorithm cannot be used directly as it requires expressions for activation function and its derivative. A weight perturbation method can be employed to train the model but the method still involves intensive computations and thus suffers from slow convergence due to the inherent nature of combinatorial calculations as shown in the results developed in Chapter 4.

In part one a model with continuous weights and in which the neurons were represented by small feedforward neural networks was developed. Part two considers the training of the model developed in part one. Since analytical functions are not available for the neuron activation function or its derivative a standard BP algorithm cannot be employed directly. In this research, a modified BP algorithm is derived where the activation function is characterized by input-output values generated by a small feedforward neural network and the required analytical expression for the derivative used is based on the derivative of an ideal sigmoidal function. The algorithm and related issues are developed in Chapter 4.

Part three of the research develops a training method that is applicable for highly quantized weights. Up to this stage, all the training calculations have been carried out using continuous synaptic weights. To convert the obtained continuous weights into quantized weights, one can round off all the continuous weights to their nearest allowed quantized values. However, the quantized weights obtained from this quantization operation could significantly deviate from the original continuous weight values due to low resolution digital weights and thus the output error could be very large. To deal with this problem, a new approach is described in Chapter 5, that measures the sensitivity of each weight with respect to the error function and then perturbs the weights with higher
sensitivity in an iterative training/retraining procedure. The new algorithm leads to a set of quantized weights that can be implemented using the available digital multipliers so as to obtain the correct output values.

2.6 Summary

This chapter starts with a detailed description of the mixed-signal implementation of a neural network intelligent sensor developed in the VLSI research group at the University of Windsor so as to ensure that the proposed in-the-loop training strategy in this work is applicable to this type of architecture. Then the three typical training modes (on-chip training, in-the-loop training and off-chip training) are presented and compared. Consequently, the advantages and disadvantages of the three training modes are analyzed. The problems when training a physical implementation of neural networks are summarized, which can be regarded as the target problems that this thesis is intent to solve. The last part of this chapter addresses the methodologies that we employ in this thesis to solve the stated problems. This part also clarifies the role of this work in the context of neural network research.
CHAPTER 3: NEURON MODELING FOR IN-THE-LOOP TRAINING

3.1 Introduction

A method for the in-the-loop training of physical VLSI neural networks that have low-resolution programmable digital weights is developed in the thesis. Normally weight perturbation methods would be used for in-the-loop training when the actual nonlinear activation functions of the various neurons are not known analytically. However, in the case when low resolution programmable digital weights must be used, weight perturbation methods will usually not converge due to quantization constraints on the manner in which the weight can be perturbed and hence another approach is necessary.

It is necessary to model the various neurons’ actual nonlinearities since learning algorithms are strongly related to the sigmoid nonlinearity used [81]. To do this, it is possible to model the various neurons’ actual nonlinear activation functions in terms of an ideal small neural network using measured in-the-loop data. The in-the-loop training method proposed in this thesis utilizes neural network models of the activation functions. This chapter deals with how the actual nonlinear activation functions of the neurons in a VLSI implementation can be modeled in terms of small ideal neural networks. In the following section, a mathematical derivation procedure for establishing mathematical models for the characteristics of the neurons in the smart sensor is described. Again, the same procedure is also applicable for finding out mathematical models for neurons implemented with different types of analog circuitry.
3.2 Modeling Neurons Comprised of Distributed Nonlinear Resistors

The neuron in the VLSI implementation of the smart sensors considered in this thesis are based on a number of distributed actual nonlinear resistors connected in parallel so as to synthesize a sigmoidal-like neuron activation function. This novel structure for a neuron implementation is described in [10, 37, 38]. The properties of the distributed nonlinear resistor based neuron are summarized here in a manner that will aid one in determining the best method for the in-the-loop training oriented modeling procedure.

The circuit for implementing an I-to-V activation function shown in Figure 2.3 is in fact a modified version of circuit diagram from [43]. In the modified circuit, the activation function is realized by replacing the resistor \( R \) in the original circuit with two extra MOS transistors. The activation function is then approximated with four quadratic segments.

The curves of the neuron's input-output characteristics obtained by running the HSpice™ simulation has been illustrated in Figure 2.4. From the figure, one can perceive intuitively that the modified circuit gives a better approximation to the ideal models of activation function (hyperbolic tangent function or sigmoid function).

In order to employ the training algorithms where the availability of the explicit analytical expressions for the neuron's characteristics is a necessity, it is desirable for us to find a set of explicit analytical equations for the four quadratic functions, which have been denoted by \( f_1, f_2, f_3 \) and \( f_4 \) in Figure 2.4. The definitions of the boundaries and regions have been described in details in last chapter.

It can be shown for the circuit that the devices in the circuit operate in saturation regions. Based on MOS transistor current-voltage characteristics [82], output voltage in
each region can be obtained as a function of input current by solving quadratic node
equations. The final formulae [38] are given in the equations (3.1) to (3.4) and the
notations used are the same as given in [38]. Assuming \( \frac{W_2}{L_2} = \frac{W_4}{L_4} \) and \( \frac{W_1}{L_1} = \frac{W_3}{L_3} \), then
the equations are

\[
f_1: \quad V_{\text{out}} = \frac{V_{BI} + V_{B2}}{2} - V_{TN} - \sqrt{\frac{1}{K_n} \cdot \frac{L_2}{W_2} \cdot |I_{in}| - \left( \frac{V_{B2} - V_{BI}}{2} \right)^2}
\]

(3.1)

\[
f_2: \quad V_{\text{out}} = V_{B2} - V_{TN} - \sqrt{\frac{2}{K_n} \cdot \frac{L_2}{W_2} \cdot |I_{in}|}
\]

(3.2)

\[
f_3: \quad V_{\text{out}} = V_{BI} + |V_{TP}| + \sqrt{\frac{2}{K_p} \cdot \frac{L_3}{W_3} \cdot I_{in}}
\]

(3.3)

\[
f_4: \quad V_{\text{out}} = \frac{V_{BI} + V_{B2}}{2} + |V_{TP}| + \sqrt{\frac{1}{K_p} \cdot \frac{L_1}{W_1} \cdot I_{in} - \left( \frac{V_{B2} - V_{BI}}{2} \right)^2}
\]

(3.4)

where \( K_n = \mu_n C_{ox} \) and \( K_p = \mu_p C_{ox} \). Note that \( \mu_n \) and \( \mu_p \) are the electron mobility parameters in NMOS and PMOS devices, respectively. The \( C_{ox} \) is the oxide capacitance per unit area in MOS transistors. The conditions that ensure the conjunction points to be smooth are:

\[
\frac{W_2}{L_2} = \frac{W_4}{L_4} \quad \text{and} \quad \frac{W_1}{L_1} = \frac{W_3}{L_3}
\]

(3.5)

To make the part composed of segments \( f_3 \) and \( f_4 \) and the part composed of \( f_1 \) and
\( f_2 \) symmetric with respect to the origin, the following condition has to be met:

\[
\frac{1}{K_n} \cdot \frac{L_4}{W_4} = \frac{1}{K_p} \cdot \frac{L_3}{W_3} = K
\]

(3.6)
where a new parameter \( K \) has been introduced. From the equations (3.5) and (3.6), we simplify the equations (3.1) to (3.4) to obtain:

\[
\begin{align*}
\text{f}_1: \quad V_{\text{out}} &= \frac{V_{B1} + V_{B2}}{2} - V_{TN} - \sqrt{K \cdot |I_{\text{in}}| - \left( \frac{V_{B2} - V_{B1}}{2} \right)^2} \\
\text{f}_2: \quad V_{\text{out}} &= V_{B2} - V_{TN} - \sqrt{2K \cdot |I_{\text{in}}|} \\
\text{f}_3: \quad V_{\text{out}} &= V_{B1} + |V_{TP}| + \sqrt{2K \cdot I_{\text{in}}} \quad (3.9) \\
\text{f}_4: \quad V_{\text{out}} &= \frac{V_{B1} + V_{B2}}{2} + |V_{TP}| + \sqrt{K \cdot I_{\text{in}} - \left( \frac{V_{B2} - V_{B1}}{2} \right)^2} \\ 
\end{align*}
\]

From the equations (3.7) to (3.10) we can see that the shape of the combined function can be controlled solely by the parameter \( K \) when \( V_{B1}, V_{B2}, V_{TN} \) and \( V_{TP} \) are fixed values. The parameter \( K \) depends on the dimensions of the MOS transistors. Figure 3.1 gives a set of curves obtained by applying the equations (3.7) to (3.10). It can
be seen that the parameter $K$ controls the shape of the s-shape activation function, and a small $K$ can 'stretch' the shape of the function.

Up to this stage, we have derived a set of analytic expressions for the activation function of the neuron circuit shown in Figure 2.3. Based on these analytic expressions, \textit{i.e.}, equations (3.5) to (3.10), one is allowed to use the standard BP algorithm to conduct the training for the smart sensor, since the derivative function of the activation function can be found analytically in segments. Corresponding to equations (3.7) to (3.10), the derivative function can be expressed segmentally in equations (3.11) to (3.14) as follows (we have used the font \textit{Arial} $f_{1-4}$ to denote the segments for derivative function):

$$f_1: \quad \frac{dV_{out}}{dI_{in}} = 0.5 \cdot K \cdot \sqrt{K \cdot |I_{in}| - \left(\frac{V_{B1} - V_{B2}}{2}\right)^2}$$  \hspace{1cm} (3.11)

$$f_2: \quad \frac{dV_{out}}{dI_{in}} = \frac{K}{\sqrt{2K \cdot |I_{in}|}}$$  \hspace{1cm} (3.12)

$$f_3: \quad \frac{dV_{out}}{dI_{in}} = \frac{K}{\sqrt{2K \cdot |I_{in}|}}$$  \hspace{1cm} (3.13)

$$f_4: \quad \frac{dV_{out}}{dI_{in}} = 0.5 \cdot K \cdot \sqrt{K \cdot |I_{in}| - \left(\frac{V_{B1} - V_{B2}}{2}\right)^2}$$  \hspace{1cm} (3.14)

Note that the four segments expressed in (3.11) to (3.14), which represent the derivative function, are symmetric about $V_{out}$-axis. It should be also pointed out that there exist discontinuity points in between the segments $f_1$ and $f_2$, and in between the segments $f_3$ and $f_4$. To handle the derivative evaluation for the discontinuity points, one can choose to take the average values as the derivative values for the discontinuity points.

From an idealized theoretical perspective, equations (3.5) to (3.10) in conjunction with equations (3.11) to (3.14) model an individual nonlinear resistor's behavior. In the
actual implementation, however, there are parameter variations that can lead to discontinuities at the boundaries of the operating regions associated with the four MOS devices. In addition, the actual activation function that is created by connecting distributed resistors of this type in parallel cannot be expressed analytically in terms of the parameters and functions appearing in equations (3.5) to (3.10). Thus, in practice a useful method for modeling actual VLSI neurons must utilize in-the-loop input-output measured data. The neuron modeling method developed in this thesis is described in the following sections.

3.3 Modeling Physical Neurons Using Small Neural Networks

We are faced with the problem of modeling an unknown but single-valued nonlinear activation function using measured data obtained during an in-the-loop training procedure. There are a number of methods available for modeling different classes of nonlinear systems [83]. One possible method might be the use of Volterra-integral series approximation. However, a model based on this type of approach would be very computationally intensive and the resulting series would be difficult to use as part of neural network training scheme. Since we are interested in training a neural network using a conventional method like BP method, there would be a practical advantage to modeling the nonlinear neuron activation functions in terms of a weighted series of sigmoidal functions [83], or in other words a small feedforward ideal neural networks.

Generally speaking, a real continuous function can be approximated by linear combinations of bounded sigmoidal functions [84]. More precisely, finite combinations of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild
conditions are imposed on the univariate function [85]. Thus a multilayer feedforward neural network with an ideal sigmoidal activation function is capable of learning an input-output mapping from a set of experimentally measured data.

This mechanism is formally elucidated in the following theorem [86]:

**Theorem 3.1:** Let $\sigma(x)$ be a nonconstant, bounded and monotone increasing continuous function. And let $K$ be a compact subset (bounded closed subset) of $\mathbb{R}^n$ and $f(x_1, \ldots, x_n)$ be a real valued continuous function on $K$. Then for an arbitrary $\varepsilon > 0$, there exist an integer $N$ and real constants $c_i$ $(i = 1, \ldots, N)$, $w_{ij}$ $(i = 1, \ldots, N, j = 1, \ldots, n)$ such that

$$\tilde{f}(x_1, \ldots, x_n) = \sum_{i=1}^{N} c_i \sigma \left( \sum_{j=1}^{n} w_{ij} x_j - \theta_j \right).$$  \hspace{1cm} (3.15)

satisfies

$$\max_{x \in K} |f(x_1, \ldots, x_n) - \tilde{f}(x_1, \ldots, x_n)| < \varepsilon$$  \hspace{1cm} (3.16)

where $X = (x_1, \ldots, x_n)$.

In other words, for an arbitrary $\varepsilon > 0$, there exists a three-layer network whose output functions for the hidden layer are $\sigma(x)$, whose output functions for input and output layers are linear and which has an input-output function $\tilde{f}(x_1, \ldots, x_n)$ such that the equation (3.16) holds [86]. A similar conclusion can be found in [87]. This paper rigorously establishes that standard feedforward networks with as few as one hidden layer using arbitrary squashing functions are capable of approximating any continuous function from one finite dimensional space to another to any desired degree of accuracy, provided that sufficiently many hidden units are available. In this sense, multilayer feedforward networks are a class of universal approximators.
The idea of the proposed in-the-loop training method is based on modeling the actual physical neuron using a small feedforward neural network, which we call subnet or SNN in the following descriptions. This identification procedure is valid for neural networks with one or two hidden layers of neurons as it utilizes the effect of varying the synaptic weights so as to generate input-output values for the activation function of a specific neuron in the network. The resultant experimental data that characterizes the activation function is used to train the subnet to model the actual physical implementation of all the neurons in the network. These small neural networks are then embedded in a larger neural network (LNN) that models the given complete implementation of neural network as shown in Figure 3.2. The input-output characteristics of a physical neuron can be measured by setting the synaptic weights to be unity and applying a ramp voltage

![Diagram of a small neural network](image)

*Figure 3.2: Modeling a Physical Neuron Using a Small Neural Network (Subnet).*
to the input. In this manner, the task of training an implementation of a neural network will become training the LNN model with ideal activation functions. Note that at this stage the LNN model does not include any analog circuit nonidealities. Or in other words, the actual characteristics have been mapped into the trained weights of the subnets.

In the next section, before we describe the neural modeling method in details, we will investigate some important issues of the use of neural networks as function approximators.

3.4 Neural Networks and Function Approximation

Several neural network architectures can be used in learning a mapping between a set of input data and a set of output data. This process can be considered as a specific method of function approximation. The problem of learning a mapping between an input and output space is essentially equivalent to the problem of synthesizing an associative memory that retrieves the appropriate output when presented with the original input data and generalizes when presented with new input data [88]. It is also equivalent to the problem of estimating the system that transforms inputs into outputs given a set of examples of input-output data pairs. A classical framework for this problem is approximation theory. Related methods are system identification techniques, used when it is possible to choose the input set, and system estimation techniques, used when the input-output pairs are given.

3.4.1 The Formulation of Approximation Problem

Approximation theory deals with the problem of approximating a function $F(W,X)$ which has a fixed number of parameters $W$ ($X$ and $W$ are real vectors $X = (x_1, x_2, \ldots)$.
\(x_1, \ldots, x_n\) and \(W=(w_1, w_2, \ldots, w_m)\). For a particular choice of \(F\), the problem is then to find the set of parameters \(W\) that provides the best possible approximation to a given function \(f(X)\). This is the learning step. Needless to say, it is very important to choose an approximation function \(F\) that is as compatible as possible approximating the function \(f(X)\). There would be little point in trying to learn an approximation if the chosen approximation function \(F(W, X)\) could give only a very poor representation of \(f(X)\), even with optimal parameter values.

To measure the quality of the approximation, one needs to introduce a distance function \(\rho(x)\) to determine the distance \(\rho[F(W, X), f(X)]\) of an approximation \(F(W, X)\) from \(f(X)\). The distance is usually a norm, for instance the standard \(L_2\) norm. The approximation problem can be then stated formally [83]:

*If \(f(X)\) is a continuous function defined on set \(X\), and \(F(W, X)\) is an approximating function that depends on continuously on \(W\) and \(X\), the approximation problem is to determine the parameters \(W^*\) in \(S_{P_w}\) such that*

\[
\rho[F(W^*, X), f(X)] < \rho[F(W, X), f(X)]
\]

*(3.17)*

*for all \(W\) in the set \(S_{P_w}\).*

If we substitute the approximating function \(F(W, X)\) with a small multilayer feedforward neural network, the complexity of the approximation is closely related to the problem of scaling and to the concept of order. In terms of the small neural network, it is closely related to the number of hidden layers and the number of hidden neurons.

One should also be concerned about the problem of which algorithm to use for finding the optimal values of the parameters \(W\) for a given choice of \(F\), which may also represent a small neural network. A standard technique in approximation is to perform
gradient descent in parameter space $\mathcal{S} P_w$ trying to minimize $p[F(W,X),f(X)]$. The standard BP algorithm, for instance, is standard gradient descent on a specific $F$ of the form

$$F(W,X) = \sigma \left( \sum_n w_n \sigma \left( \sum_i v_i \sigma \left( \ldots \sigma \left( \sum_j f_{j,i} x_j \right) \ldots \right) \right) \right)$$ (3.18)

From the point of view of learning as continuous approximation we can draw an equivalence with a standard approximation problem, surface reconstruction from sparse data points. In this analogy, learning simply means collecting the examples, i.e., the input coordinates $x_i, y_i$ and the corresponding output values at those locations (the height of the surface) $d_i$. This builds a look-up table. Generalization means estimating $d$ in locations $x, y$ where there are no examples, i.e., no data. This requires interpolating or, more generally, approximating the surface between the data points. Interpolation is the limit of approximation when there is no noise in the data. This example, given for a surface, i.e., the graph in $\mathbb{R}^2 \times \mathbb{R}$, can be immediately extended to mappings from $\mathbb{R}^n$ to $\mathbb{R}^m$ (and graphs in $\mathbb{R}^n \times \mathbb{R}^m$). In this sense learning is a problem of hypersurface reconstruction.

### 3.4.2 Approximation Properties of Multilayer Feedforward NN

Although there is no possibility of building a reasonable multilayer neural network to exactly represent a function, we still have the possibility to look for a multilayer neural network that can approximate to a given function with some tolerable errors. Examples with good approximation results for special problems can be found [89-92]. It has been proved that multilayer feedforward neural networks with fixed nonlinear
node functions and linear combinations of inputs to nodes can approximate many input-
output mappings with very small error [93].

In particular, four increasingly complex classes of functions are considered: (a) Boolean mappings \((\{0,1\}^N \rightarrow \{0,1\})\), (b) classification of real vectors \((\mathbb{R}^N \rightarrow \{0,1\})\), (c) continuous mappings from real values to real values \((\mathbb{R} \rightarrow \mathbb{R})\), and (d) continuous mappings from real vectors to real values \((\mathbb{R}^N \rightarrow \mathbb{R})\). In each one of these cases it is very important to have a representation that is powerful enough to represent the mapping to be learned. Below are some results adapted from [94] that describe the representational powers of neural networks for the classes of the functions previously mentioned.

(a) \(\{0,1\}^N \rightarrow \{0,1\}\). It is a classical result that Boolean mappings can be represented exactly by a network of linear threshold units with one hidden layer. One way to build such a network is to have \(2^N\) hidden units, one for each possible input state. The weights can be set so that each input produces an output of 1 at a unique hidden unit. Then the output node's value will be determined completely by the weight from the hidden unit to the output value.

(b) \(\mathbb{R}^N \rightarrow \{0,1\}\). It is well known that classification of arbitrary unions of non-convex polyhedral regions in \(\mathbb{R}^N\) can be performed with networks having two hidden layers of threshold elements. Nodes in the first hidden layer separate the input space with hyperplanes. Note in the second layer can be used to compute intersections (convex polyhedra) of the halfspaces defined by the hyperplanes of the first layer nodes. The weights at the output layer compute arbitrary unions of these regions.
(c) $\mathbb{R} \rightarrow \mathbb{R}$. For the case of networks with the standard sigmoid nonlinearities, which is defined as

$$f(x) = \frac{1}{1 + e^{-x}} \quad (3.19)$$

We have the following theorem [94]:

Theorem 3.2: Networks with one layer of hidden units can uniformly approximate (to within any $\varepsilon > 0$) any real continuous function on a finite real interval.

(d) $\mathbb{R}^N \rightarrow \mathbb{R}$. For the sigmoid nonlinearities, we have the following theorem [94]:

Theorem 3.3: Networks with two layers of hidden units can uniformly approximate (to within any $\varepsilon > 0$) any real continuous function of multiple real inputs.

To approximate the input-output characteristics of the curves such as the one shown in Figure 2.4, we know from Theorem 3.2 that a small neural network with one layer of hidden units can always approximate the desired activation function.

3.5 The Modeling Procedure

A procedure for identifying the activation functions of hidden neurons of a feedforward neural network, that is valid for neural networks with two or less hidden layers of neurons, has been developed in [95]. It utilizes the effect of varying the synaptic weights or varying the input values with fixed weights so as to generate input-output sample pairs measured values for the activation function of a specific neuron in the network. The resultant experimental data that characterizes the activation function is used to train a small neural network to model the actual physical implementation of all
the neurons in the network. These small neural networks are embedded in a larger neural network (LNN) that models the complete intelligent programmable sensor as shown in Figure 3.2.

The procedure to model the neuron activation function comprises two distinct phases: (i) Experimentally determining the input-output values associated with the activation function over the range of interest; and (ii) Modeling the activation function input-output data in terms of a small neural network.

3.5.1 Modeling Activation Function for an NN With One Hidden Layer

First consider measuring the neuron activation functions for a neural network with one hidden layer. To obtain the input-output characteristics of one neuron, we need to isolate a specific neuron in the network. This can be done with reference to Figure 3.2 by setting one of the weights in \( W^{(1)} \) (the weight connected from the input node to the neuron being modeled) and one in \( W^{(2)} \) (the weight connected from the neuron being modeled to the output node) to be unity, while setting all others to zero. A ramp voltage would then be applied to the said input node and the resultant output from the neuron is measured. Note that when setting the input weight in \( W^{(1)} \) to be unity, the range of the ramp voltage should be \( [-W_{\text{max}} \xi_{\text{max}}, +W_{\text{max}} \xi_{\text{max}}] \), where \( W_{\text{max}} \) and \( \xi_{\text{max}} \) denote the absolute values of the maximum weight value and input value, respectively. The activation function is thus characterized by a number of input-output pairs of sampled data over the range of interest. A neural network with one hidden layer is then trained using this data to model, or reproduce the neuron’s activation function.

An alternative method for obtaining the input-output characteristics of a neuron is to vary the weight in \( W^{(1)} \) (the weight connected from the input node to the neuron being
modeled) while setting the input value fixed. This method is valid if the digital weights with limited number of bits provide sufficient number of levels. It should be remarked here that we may need to sweep the input values as well as weight values if the D/A stage involves nonlinearities that are not negligible, since the nonlinearities associated with the D/A stage can be observed from the output when varying the weights. This will be discussed later in this chapter.

![Graph of Modeled vs Measured Data](image)

Figure 3.3: Modeling the Neuron Activation Function Using a Small Neural Network with 1 Input Node, 25 Hidden Neurons and 1 Output Node.

An experimental result is shown in Figure 3.3. The input-output pairs of sampled data that characterize the activation function are used to train a neural network with one input node, one hidden layer with 25 neurons (the problem of how to choose the number of hidden neurons will be studied later in this chapter), and one output layer with one node. The required synaptic weights are determined using the standard BP algorithm. The training procedure takes from 300 to 1000 iterations depending upon the initial
values of the weights and the number of input-output data pairs. The training converges with a mean-squared-error value of 0.002 for a function that ranges between 0 and 5. The number of input-output measured data has been chosen to be 81.

3.5.2 Modeling Activation Function for an NN With Two Hidden Layers

This method for measuring activation functions for a neural network with one hidden layer can be extended to measuring the neuron activation functions for a neural network with two hidden layers. Once again we need to be able to isolate a specific neuron in one of the two hidden layers the network. This can be done with reference to Figure 3.4 by setting the input node value and the incident weight connected to a neuron in the first layer, with activation function $F_1$, such that the neuron's output is driven to its maximum saturated value. In Figure 3.4, identifying activation functions connected in series is shown for two different cases: (a) two physical neurons are connected in series with a synaptic weight in between; (b) the output signals of $F_1$ can be observed by adding the reverse function of $F_2$ to the output of $F_2$. This saturated output can then be used to form a test input for the neuron in the second hidden layer, with activation function $F_2$, whose activation function is to be measured. The weight between the two selected hidden neurons is varied so as to produce a ramp input for the neuron in the second hidden layer. Note the number of the generated input data of the ramp input is limited by the number of bits in the digital weights. This is one of the limitations of this method. In this manner the resultant output from the neuron can be measured and used to determine a number of input-output pairs of sampled data over the range of interest.
Once the nonlinear activation function, \( F_2 \), for neuron in the second hidden layer is known an inverse function, \( F_2^{-1} \), can be found. The fact that the product of the activation function and its inverse is equal to unity is now exploited, \( F_2 F_2^{-1} = 1 \), so as to allow us to observe the output of the cascaded neuron in the first hidden layer. The identification of the neuron function, \( F_1 \), proceeds in the manner already described for the single hidden layer case.

Figure 3.4: Identifying Activation Functions Connected in Series for Two Different Cases.
Figure 3.5: The Simulation Results When Modeling the Two Unknown Activation Functions Connected in Series by Two Subnets.

An experiment has also been conducted for the case that two neurons with unknown activation functions are connected in series. For the structure shown in Figure 3.4(a), the input to $F_1$ and the synaptic weight $W^{(2)}$ are controllable; and the output from $F_2$ is observable. Suppose we are given two unknown activation functions connected in series $F_1$, $F_2$ and the relationship $F_2(F_1(x))$ as shown in Figure 3.5. Using the method described above, we first model the function $F_2^{-1}$, then pipeline this stage to the output of $F_2$. The input-output characteristics of $F_1$ can then be measured and modeled. The result of modeling $F_1$ is shown in Figure 3.6.

Once the subnets are known for each of the physical neurons in the sensor, the overall model for the complete neural network can be constructed as illustrated in Figure 3.7. The complete neural network model can be constructed when all the physical
Figure 3.6: The Result of Modeling F1 Where F1 and F2 Are Connected in Series as in Figure 3.4.

neurons are modeled by subnets and they are in turn embedded in the larger neural network. At this point, all the weight connections associated with subnets (solid lines in the figure) have been determined and will be fixed in the course of the training of the complete neural network. It is important to note here that all the neurons in subnets are now ones with ideal characteristics, i.e., sigmoidal functions. This means that we have "transferred" the nonidealities associated with the analog neurons in our original circuit to the weight values of subnets. Henceforth, the training of the complete neural network is to determine the weight strengths that are expressed with arrow lines in Figure 3.7. Chapter 4 will deal with the training issues of the complete neural network.

It should be remarked here that the complete neural network model shown in Figure 3.7 will have the identical input-output characteristic with its hardware
counterpart, although their structures are not the same. One can expect that the training of the complete neural network will not be an easy task. The standard BP algorithm cannot be employed in a straightforward manner, since the analytic expressions for the subnets are not available. However, the significance of the described neuron modeling procedures is that we have mapped the nonidealities associated with analog neurons to the database of subnets' weight values so that the software environment with the database will reflect the nonidealities and can be used for the purpose of in-the-loop training.

![Diagram of neural network](image)

**Figure 3.7: The Complete Neural Network Model When All the Physical Neurons Are Modeled by Subnets.**
3.6 The Desirable Properties of a Subnet Model

In previous sections we have described the mechanism and the procedure for modeling the activation functions for the training of the smart sensor. The idea of the whole method is to model each of the neurons in a physical neural network realization in terms of a small neural network (subnet) based on using in-the-loop data. Then a neural network model of the complete physical realization is formed with embedded subnets representing individual neurons as shown in Figure 3.7. Once the complete model for the neural network implementation is obtained, all the operations concerning the training can be done on this model. Henceforth, the procedure of modeling the physical neurons provides the basis of the proposed in-the-loop training scheme. Thus it is necessary to find out the desirable properties of a subnet model before we proceed any further.

It has been pointed out in the last section that the standard BP algorithm can be employed to train a chosen subnet for modeling a physical neuron. The completion of the training implies that the error signal has settled within a preset criterion, e.g., the mean squared error (MSE) criterion. Here we use error signal to refer to the error outputs between the subnet and the physical neuron being modeled, which show themselves as a sequence of error values with the sequence number being the number of the input-output data pairs. We will also use prediction error to refer to the error signal interchangeably in this context since the latter concept is a commonly-used terminology in system identification.

It should be remarked here that the MSE criterion does not provide enough information on the properties of an error signal. For instance, the three curves sketched in Figure 3.8 representing three distinct error signals may have the identical MSE values,
while the corresponding models could make big differences. Evidently, if the model of a physical neuron produces an error signal like the one shown in Figure 3.8(a) or Figure 3.8(b), the complete neural network model may have larger modeling errors due to the possible error accumulation when signals propagate forward through the layers of the neural network. If we consider the error signal as a random process, we may derive the properties that a desirable subnet model should possess.

Suppose we are given a feedforward neural network that has one hidden layer. It has two layers of neurons located in the hidden layer and output layer, respectively, refer to Figure 3.9(a). A close-up view of a neuron node topology with inputs and outputs is shown in Figure 3.9(b). If the activation function is approximated using a subnet, the

![Figure 3.8: The Three Different Error Signals of the Identical MSE Values.](image)
approximation errors can be assumed to be from a disturbance source: $e$, as shown in Figure 3.9(b). Most of the notations for the use of the following derivation are given in Figure 3.9 (a) and (b).

Now the quantity $h_i^{(1)}$ can be found using the equation (3.20) as follows,

$$h_i^{(1)} = \sum_{j=1}^{N_i} w_{ij}^{(1)} \xi_j, \quad (i=1,\ldots,N_h)$$

(3.20)

where we denotes the number of hidden neurons as $N_h$ and the number of input units as $N_i$. And the $w_{ij}^{(1)}$ represents the synaptic weight from input unit $j$ to hidden neuron $i$. Similarly, the $w_{ij}^{(2)}$ represents the synaptic weight from the $j$-th hidden neuron to the $i$-th output neuron.

![Diagram](image)

**Figure 3.9**: (a) A Feed-Forward Neural Network with One Hidden Layer and Two Layers of Nonlinear Neurons; (b) A Close-Up View of a Neuron Node with Inputs and Outputs.

54
When there is no disturbance, i.e., $e=0$, we can find $V_i$ by

$$V_i = f(h_i^{(1)}) \quad (i=1,\ldots,N_h)$$ \hspace{1cm} (3.21)

Following the same procedure, we have

$$h_i^{(2)} = \sum_{j=1}^{N_h} w_{ij}^{(2)} V_j \quad (i=1,\ldots,N_o)$$ \hspace{1cm} (3.22)

$$O_i = f(h_i^{(2)}) \quad (i=1,\ldots,N_o)$$ \hspace{1cm} (3.23)

If we substitute the activation function with a trained subnet using the procedure described earlier, there will exist approximation errors. The errors can be considered as a disturbance source added to the original activation function, as illustrated in Figure 3.9 (b). We use $e^{(h)}$ and $e^{(o)}$ to denote the disturbance sources added to the hidden layer and output layer, respectively. The variables with superscript "*" represent the quantities when the disturbance sources are considered. Then we can find the output when using the approximated version of activation function as follows:

$$h_i^{(2)*} = \sum_{j=1}^{N_h} w_{ij}^{(2)} V_j = \sum_{j=1}^{N_h} w_{ij}^{(2)} (V_j + e^{(h)_j}) = h_i^{(2)} + \sum_{j=1}^{N_h} w_{ij}^{(2)} e^{(h)_j} \quad (i=1,\ldots,N_o)$$ \hspace{1cm} (3.24)

$$O_i^* = f(h_i^{(2)*}) + e^{(o)_i} = f\left(h_i^{(2)} + \sum_{j=1}^{N_h} w_{ij}^{(2)} e^{(h)_j}\right) + e^{(o)_i} \quad (i=1,\ldots,N_o)$$ \hspace{1cm} (3.25)

The output errors caused by the disturbances can be determined by using the Taylor expansion in the following manner:

$$\Delta O_i = O_i^* - O_i = f\left(h_i^{(2)} + \sum_{j=1}^{N_h} w_{ij}^{(2)} e^{(h)_j}\right) + e^{(o)_i} - f\left(h_i^{(2)}\right)$$

$$= e^{(o)_i} + f'(h_i^{(2)}) \cdot \sum_{j=1}^{N_h} w_{ij}^{(2)} e^{(h)_j} + \Omega(\xi)$$ \hspace{1cm} (3.26)
where $\Omega(\xi)$ represents the high-order errors of the Taylor expansion in (3.26). If we omit the high-order error part, the variance of $\Delta O_i$ can thus determined by

\[
\text{var}(\Delta O_i) = \text{var}(e_i^{(o)}) + \text{var}\left(f'(h_i^{(2)}) \cdot \sum_{j=1}^{N_h} w_{ij}^{(2)} e_j^{(h)} \right) \\
= \text{var}(e_i^{(o)}) + \sum_{j=1}^{N_h} \left(f'(h_i^{(2)}) w_{ij}^{(2)} \right) \cdot \text{var}(e_j^{(h)})
\]

(3.27)

We further assume that the disturbances have the same variance $\lambda^2$, i.e.,

\[
\text{var}(e^{(h)}) = \text{var}(e^{(o)}) = \lambda^2
\]

then we have

\[
\text{var}(\Delta O_i) = \lambda^2 \cdot \left(1 + \sum_{j=1}^{N_h} \left(f'(h_i^{(2)}) w_{ij}^{(2)} \right) \right)
\]

(3.28)

Similarly, we also assume that the disturbances have the same mean value $M$, i.e.,

\[
\text{E}(e^{(h)}) = \text{E}(e^{(o)}) = M
\]

the mean value of $\Delta O_i$ can thus determined by

\[
\text{E}(\Delta O_i) = M \cdot \left(1 + \sum_{j=1}^{N_h} \left(f'(h_i^{(2)}) w_{ij}^{(2)} \right) \right)
\]

(3.29)

We denote the term in the parentheses in the equations (3.28) and (3.29) by a parameter $K_c$, i.e.,

\[
K_c = 1 + \sum_{j=1}^{N_h} \left(f'(h_i^{(2)}) w_{ij}^{(2)} \right)
\]

(3.30)

Now we have derived the statistical relationships between the output of a feedforward neural network and the disturbances added to its hidden and output neurons.

From the equations (3.28) and (3.29), we can draw the conclusions that the mean value and variance of the output in a feedforward network are both dependent on the mean value and variance of the disturbance signals added on the neurons. Furthermore, the two key statistical gauges tend to be magnified since the term included in the parentheses in
(3.28) and (3.29), \( K_c \), is greater than one. It should be noted that when \( K_c \gg 1 \), the output errors will be magnified significantly.

In particular, in order to minimize the output errors incurred from subnet models, we can define a desirable subnet model as the one which have the following properties: i) the error signal tends to a white noise process; ii) the error signal has a minimum variance (this is determined by the MSE value in this case); iii) the error signal has a symmetric distribution. Thus the error signal shown in Figure 3.8(c) is supposed to correspond to a better model. For now, we are ready to conduct statistical tests to determine if a chosen subnet can meet the required properties i) and iii).

3.7 Statistical Tests on Model Validation

Before we substitute an actual neuron in a VLSI implementation of neural network with a subnet model, one must concern the following two questions: a) whether a chosen subnet is competent for modeling a certain type of physical neuron? and b) whether the chosen subnet is appropriate for modeling the physical neuron. Or the question can be also asked if there exists an optimal neural network architecture that can be used as the subnet for the neuron modeling and how to find it.

From the viewpoint of system identification, the above two questions can be posed as: a) Is a subnet model large enough to characterize the physical neuron? b) Is a subnet model too complex, or does the model meet the parsimony conditions? It is necessary to point out that an oversized subnet model (overparametrized) can lead to unnecessarily complicated computations for finding the subnet's weights as well as for using the model. And an undersized subnet model (underparametrized) may be very
inaccurate. This section aims at answering the first question. The second question will be answered in the next section.

It should be noted here that the role of the statistical tests should not be exaggerated. In stead, the use of common sense still plays an important role in model validation and determination. Any statistical tests should not replace the study of plots of measured data and model output based on common sense [96]. More importantly, the statistical tests described in the following are only necessary conditions for the model validation in this context.

3.7.1 A Statistical Test on White Noise Property of An Error Signal

The first statistical test described in this subsection is based on the procedure in [96]. The test is to determine whether the prediction error tends to a zero mean white noise process or not. We use the notation of \( \varepsilon(t, \hat{\theta}_N) \) or \( \varepsilon(t) \) to denote the prediction errors (residuals), evaluated at the parameter estimate \( \hat{\theta}_N \), which in this work represents the trained weights of the subnet. For the model assumption (in statistical terms, the null hypothesis \( H_0 \)) that \( \varepsilon(t) \) is zero mean white noise an autocorrelation test is described as follows. Note that the statistical properties of the tests will be analyzed under the null hypothesis that the model assumption \( H_0 \) is actually satisfied.

First construct estimates of the covariance function as:

\[
\hat{r}_\tau = \frac{1}{N} \sum_{t=1}^{N-\tau} \varepsilon(t + \tau)\varepsilon(t) \quad (\tau \geq 0)
\]  

(3.31)

Since \( \varepsilon(t) \) is zero mean white noise (the assumption) then its covariance function is zero except at \( \tau = 0 \) (Lemma B.2 at pp.548 in [96]):
\[
\hat{r}_e(\tau) \to 0 \quad \tau \neq 0 \\
\hat{r}_e(0) \to \lambda^2 = E \varepsilon^2(t) \quad \text{as} \ N \to \infty
\]  

(3.32)

where \( \lambda^2 \) represents the variance of white noise and \( E \) represents the expectation. A normalized test quantity is considered:

\[
x_\tau = \frac{\hat{r}_e(\tau)}{\hat{r}_e(0)}
\]  

(3.33)

According to (3.32) one can expect \( x_\tau \) to be small for \( \tau \neq 0 \) and \( N \) to be large provided \( \varepsilon(t) \) is white noise. To illustrate what 'small' means one defines:

\[
r = \frac{1}{N} \sum_{t=1}^{N} \begin{pmatrix} 
\varepsilon(t-1) \\
\vdots \\
\varepsilon(t-m)
\end{pmatrix} = \begin{pmatrix} 
\hat{r}(1) \\
\vdots \\
\hat{r}(m)
\end{pmatrix}
\]  

(3.34)

It can be proved (Lemma B.3 at pp.550 in [96]) that \( r \) is asymptotically Gaussian distributed:

\[
\sqrt{N} \, r \xrightarrow{\text{dist}} N(0,P)
\]  

(3.35)

where the covariance matrix is \( P = \lim_{N \to \infty} EN \cdot r \cdot r^T \). Since the element of \( P \), \( P_{i,j} \) \((i,j=1, \ldots, m)\) can be evaluated as:

\[
P_{i,j} = \lim_{N \to \infty} \frac{1}{N} \sum_{t=1}^{N} \sum_{s=1}^{N} \varepsilon(t-i)\varepsilon(t)\varepsilon(s)\varepsilon(s-j)
\]

\[
= \lim_{N \to \infty} \left\{ \frac{1}{N} \sum_{t=1}^{N} \sum_{s=1}^{N} \varepsilon(t-i)\varepsilon(t)\varepsilon(s)\varepsilon(s-j) + \frac{1}{N} \sum_{t=1}^{N} \varepsilon(t-i)\varepsilon^2(t)\varepsilon(t-j) \right\}
\]

\[
= \lim_{N \to \infty} \left\{ 0 + 0 + \frac{1}{N} \sum_{t=1}^{N} \lambda^2 \varepsilon(t-i)\varepsilon(t-j) \right\} = \lambda^4 \delta_{i,j}
\]  

(3.36)

Henceforth,

\[
P = \lambda^4 I
\]  

(3.37)
The result of equation (3.35) implies that
\[ N r^T P^{-1} r = N r^T r / r^2 (0) \xrightarrow{\text{dist}} \chi^2 (m) \]  \hspace{1cm} (3.38)

Then for \( \tau > 0 \), it follows from equations (3.32) and (3.34) that (cf. Lemma B.4 in [54])
\[ \sqrt{N} x_\tau = \sqrt{N} \frac{\hat{r}_\tau (\tau)}{\hat{r}_\tau (0)} \xrightarrow{\text{dist}} N (0, 1) \]  \hspace{1cm} \text{for } \tau > 0 \hspace{1cm} (3.39)

From (3.39) we have (for large \( N \)) \( x_\tau \sim N (0, 1/N) \) and hence
\[ P (|x_\tau| \leq 1.96 / \sqrt{N}) = 0.95. \] The null hypothesis (that \( r_\tau (\tau) = 0 \)) can therefore be accepted if \( |x_\tau| \leq 1.96 / \sqrt{N} \) (with an unknown risk) and otherwise rejected (with a risk of 0.05). In the following we will conduct autocorrelation tests for some subnets with different number of hidden neurons.

From Theorem 3.2 we know that a subnet with one layer of hidden units can uniformly approximate any real continuous function on a finite real interval. Now we need to find out the optimal (in not a strict sense) number of hidden units that should be used in the subnet. To apply the autocorrelation test described above to the error signals between the outputs of a trained subnet and the neuron being modeled, one needs to know the errors evaluated not only at the positions of the input-output example pairs but also at the positions of other than the sampled points. It is no doubt that the number of sampled points will be limited and it is not possible to evaluate the errors at other than the sample points for a 'black-box'. Fortunately, we have the a priori knowledge about the physical neuron to be modeled.

To measure the errors evaluated at other than the sampled points between the output of an actual physical activation function and that of a subnet that models the activation function, we have to employ the explicit analytic expressions of the activation
function to be modeled. To do this, we exploit the equations (3.7-3.10) introduced in the last section. Thus here our analysis is based on modeling the activation function expressed by the equations (3.7-3.10). The input-output pairs of sampled data are obtained by sampling the curve with the input range [-2, 2] and the step size 0.05. The sampled data (81 pairs) are used to train a neural network with one input node, one hidden layer with 5-45 neurons (we will change the number of hidden neurons while observing the effects of changing the number), and one output layer with one node. The required synaptic weights of the subnets are determined using the standard BP algorithm. All the subnets are trained 2000 iterations.

We first model the activation function expressed by the equations (3.7—3.10) using the subnets with 10, 25, 30 and 45 hidden neurons, respectively. Then we conduct the autocorrelation tests for each of the subnets. For each of the trained subnets, 300 data points \( \{ \varepsilon(t) \} \) are formed in the following manner. First the 300 random numbers in the range of [-2,2] are generated. Then inputting each of these numbers to the trained subnets as well as equations (3.7—3.10), we subtract their outputs to obtain each element of \( \{ \varepsilon(t) \} \). Note the statistical tests consist of the hypothesis \( H_0: \{ \varepsilon(t) \} \) is white noise.

The normalized covariance functions of errors between the modeling subnet and the curve being modeled versus \( \tau \) and a 95% confidence interval \( x_\tau \) are given in Figure 3.10(a)-(d). Since \( x_\tau \sim N(0,1/N) \) the dotted lines in the figure are drawn at \( x=\pm 1.96/\sqrt{N} \). It can be seen that the subnet with 10 hidden neurons is not acceptable since its normalized covariance functions basically do not settle within the lines, while the subnets with 25, 30 and 45 hidden neurons are acceptable in this sense [97].
3.7.2 A Statistical Test on the Symmetry Property of an Error Signal

The second statistical test is based on the assumption that the error signal has a symmetric distribution (hypothesis $H_1$). This assumption is also one of the desirable properties described earlier in this section. Let $\bar{x}_N$ be the number of changes of sign in the sequence $\varepsilon(1), \varepsilon(2), \ldots, \varepsilon(N)$. One would intuitively expect that (under the null hypotheses $H_0$ and $H_1$) the residual sequence will, on average, change sign every second.

![Graphs showing autocorrelation tests for small neural networks with different hidden neuron counts.](image)

Figure 3.10: The Autocorrelation Tests for Small Neural Networks with Hidden Neurons of (a) 10; (b) 25; (c) 30; (d) 45.
time step. Hence we expect that $\bar{x}_N = N/2$. However, a more precise statistical analysis is required to see how well this expectation holds. For that purpose we introduce the random variables $\{\gamma_i\}_{i=1}^{N-1}$ by
\[
\gamma_i = \begin{cases} 
1 & \text{if } \varepsilon(i)\varepsilon(i+1) < 0 \\
0 & \text{if } \varepsilon(i)\varepsilon(i+1) > 0 
\end{cases}
\] (3.40)

Then
\[
\bar{x}_N = \sum_{i=1}^{N-1} \gamma_i
\] (3.41)

Observe that a) $\gamma_i = 0$ with probability 0.5 and $\gamma_i = 1$ with probability 0.5; and b) $\gamma_i$ and $\gamma_j$ are identically distributed and independent variables. Hence one can apply the central limit theorem to $\bar{x}_N$. Thus $\bar{x}_N$ is asymptotically Gaussian distributed, i.e.,
\[
\bar{x}_N \xrightarrow{\text{dist}} N(m, P)
\] (3.42)

with
\[
m = E\bar{x}_N = (N-1) E\gamma_i = (N-1)/2 = N/2
\]
\[
P = E(\bar{x}_N - m)^2 = E\bar{x}_N^2 - m^2 = E\sum_{i=1}^{N-1} \sum_{j=1}^{N-1} \gamma_i\gamma_j - (N-1)^2/4
\]
\[
= [(N-1)^2 - (N-1)]E(\gamma_i)^2 + (N-1)E(\gamma_i)^2 - (N-1)^2/4
\]
\[
= (N-1)E(\delta_i^2) - (N-1)/4 = N/4
\]

Therefore we have (for large $N$)
\[
\frac{\bar{x}_N - N/2}{\sqrt{N/2}} \xrightarrow{\text{dist}} N(0,1)
\] (3.43)

Hence a 95 percent confidence interval for $\bar{x}_N$ is given by
\[ \left| \bar{x}_N - \frac{N}{2} \right| \leq 1.96 \sqrt{\frac{N}{2}} \]  \hspace{1cm} (3.44)

Equivalently stated, the inequalities

\[ \frac{N}{2} - \frac{1.96}{2} \sqrt{N} \leq \bar{x}_N \leq \frac{N}{2} + \frac{1.96}{2} \sqrt{N} \]  \hspace{1cm} (3.45)

hold with 95 percent probability.

Now consider the numerical value of the test quantity \( \bar{x}_N \). For the error signal that has been used in Figure 3.10(b), the number of changes of sign is 155. Since the interval defined by (3.45) is (132, 166) it can be seen that the number of changed signs falls well inside this interval. Hence combining this test with the previous one should accept that \( \varepsilon(t) \) is a white noise process.

3.8 Model Structure Determination and Parsimony Principle

In system identification, both the model validation and the determination of model structure are important aspects. In this section we will review the **parsimony principle** and then employ the simulation method to determine whether a subnet suffices to model a given physical neuron.

The parsimony principle says, that when out of two or more competing models which all explain the data well, the model with the smallest number of independent parameters should be chosen [96]. To answer the question that if a trained subnet meets the principle, we need to study the relationship between the network size and its generalization performance [98].

The following derivation is based on [99]. From the system identification perspective, a proper training criterion is the validity measure:
\[ \nabla(w) = \lim_{N \to \infty} E(\nabla_N(w, Z^N)) \quad (3.46) \]

where \( \nabla_N(w, Z^N) \) is the prediction-error criterion used for the parameter estimation, \( Z^N = \{ [x_1, t_1], [x_2, t_2], \ldots, [x_N, t_N] \} \) denotes the training input-teacher (input - desired output) pairs, and \( w \) denotes the synaptic weights in a subnet. For generalization, the network is evaluated by its average performance, or the expectation of \( \nabla(\hat{w}_N) \), where \( \hat{w}_N \) represents the trained weight. Note that the expectation is taken over the entire sample space so it statistically represents the generalized space. A natural idea would be to evaluate how the subnet model performs "on the average" by computing

\[ E(\nabla(\hat{w}_N)) = J(M_k) \quad (3.47) \]

where \( M_k \) represents the subnet model structure. Now the function \( \nabla(w) \) and the criterion \( J \) are not known to the user, since they involve the expectation with respect to the true data. Hence \( \nabla(w) \) and \( J \) must be replaced by some approximation or estimate.

Let \( w^* \) be the minimum of \( \nabla(w) \) and expand this function around \( w^* \):

\[ \nabla(\hat{w}_N) = \nabla(w^*) + \frac{1}{2}(\hat{w}_N - w^*)^T \nabla'(\xi_N) (\hat{w}_N - w^*) \quad (3.48) \]

Similarly, since \( \nabla_N'(\hat{w}_N, Z^N) = 0 \),

\[ \nabla_N(\hat{w}_N, Z^N) = \nabla_N(w^*, Z^N) - \frac{1}{2}(\hat{w}_N - w^*)^T \nabla_N'(\xi_N, Z^N)(\hat{w}_N - w^*) \quad (3.49) \]

Take expectation of these two expressions and use the following asymptotic relationships

\[ E \frac{1}{2}(\hat{w}_N - w^*)^T \nabla'(\xi_N)(\hat{w}_N - w^*) \]
\[ = \frac{1}{2} E \text{tr} \left[ \nabla'(\xi_N)(\hat{w}_N - w^*)(\hat{w}_N - w^*)^T \right] \approx \frac{1}{2} \text{tr} \nabla'(w^*) p_N \quad (3.50) \]

65
where $P_N = (1/N)P_w$ is the asymptotic covariance matrix of $\hat{w}_N$. Note that $P_N$ decays as $1/N$.

Also,

$$E \frac{1}{2} (\hat{w}_N - w^*)^T \nabla^2 N(\xi_N, Z^N)(\hat{w}_N - w^*) = \frac{1}{2} \text{tr} \nabla^2(w^*)P_N$$

and

$$E \nabla N(w^*, Z^N) = \nabla(w^*)$$

This gives, from (3.48) and (3.49), respectively,

$$E \nabla(\hat{w}_N) = \nabla(w^*) + \frac{1}{2} \text{tr} \nabla^2(w^*)P_N$$

(3.53)

$$E \nabla N(\hat{w}_N, Z^N) = \nabla(w^*) - \frac{1}{2} \text{tr} \nabla^2(w^*)P_N$$

(3.54)

We can thus find $J(M_k)$ appeared in (3.47) by

$$E \nabla(\hat{w}_N) \equiv J(M_k)$$

$$J(M_k) = E \nabla(\hat{w}_N) = E \nabla N(\hat{w}_N, Z^N) + \text{tr} \nabla^2(w^*) P_N$$

$$= \nabla N(\hat{w}_N, Z^N) + \frac{1}{N} \text{tr} \{ \nabla^2(\hat{w}_N) P_w \}$$

(3.55)

where $P_N = (1/N)P_w$ is the asymptotic covariance matrix of $\hat{w}_N$. Note in the last approximation we replaced the expectation of $\nabla N(\hat{w}_N, Z^N)$ with the only observation we have of it, and used $P_N = (1/N)P_w$.

If a model function $\phi(w, x(n))$ closely approximates the desired output $t(n)$, then it is plausible to consider $t(n)$ as a random process

$$t(n) = \phi(t, x(n)) + \nu(n)$$

(3.56)
where $\phi(w, x(n))$ denotes its mean value and $v(n)$ assumed to be white noise with variance $\Lambda$. Then the training criterion $V_N(w, Z^M)$ becomes

$$ V_N(w, Z^M) = \frac{1}{N} \sum_{i=1}^{N} |y(n) - \phi(w, x(n))|^2 $$

(3.57)

Then we have the equation (so-called Akaike Information Criterion) [100]:

$$ E \nabla(\hat{w}_N) = \Lambda \left( 1 + \frac{d}{N} \right) $$

(3.58)

where $d$ represents the dimension of $\hat{w}_N$. The equation (3.58) shows that the expected prediction error variance increases with a relative amount of $d/N$. Thus, there is a penalty in using networks with unnecessarily many weights.

The experiment performed in this section is by using the measured data of a neuron of the intelligent sensor to train a subnet that characterizes the input-output

![SSE error range after 1800 training epochs](image)

Figure 3.11: The Sum-Squared Error Ranges After 1800 Training Epochs.
relationship of the neuron. We vary the number of the hidden neurons and observe the sum-squared error range after 1800 training epochs, when the output errors basically settle within some error range. The result of this experiment, which is given in Figure 3.11, indicates that for modeling a neuron of the intelligent sensor the subnet having 25 hidden neurons or above will suffice. In Figure 3.11, the number of hidden neurons has been varied from 5 to 41. This conclusion basically matches the results of the autocorrelation tests described earlier.

3.9 Discussions: Some Perspectives of Neural Modeling

The input-output mapping of an activation function can also be implemented in a silicon die by using a look-up table (LUT) when the mathematical expression for the activation function is not known analytically. In the first part of this section, the advantages and disadvantages of LUT method are analyzed. As well, the LUT method and the proposed neural modeling method are compared. The second part of this section discusses the potential application of the proposed neural modeling method to solving the nonlinearity problem existent in digital weight multiplication implementation.

3.9.1 Comparison to LUT Method

The lookup table (LUT) method is one of the most efficient methods for supporting floating-point arithmetic operations [98, 101, 102]. The LUT method can be used in on-chip training, off-chip training or in-the-loop training mode. The data of the table are normally prestored in ROM/RAM of an on-chip memory, or in a database of an off-chip memory, depending on which training mode is used. When the nonlinear activation function of an implementation of neural network cannot be explicitly modeled
by some mathematical expressions, they can be implemented by a memory fetch from a LUT.

LUT method belongs to the classical digital solution to the sigmoid nonlinearity. It becomes clear that the look-up table approach falls short from the aim of having a "good approximation" (performance) in a "small area" on the final chip (price) [26]. When the table is utilized to realize an input-output relationship of some certain activation function for on-chip learning, the problem will be the precision that the activation function is implemented. The requirement of chip area for realizing the LUT memory will limit the size of a LUT. The reality is that there is usually not enough chip area available for extra digital memory when on-chip learning capability is required to realize. When the LUTs are stored in a database for performing in-the-loop or off-chip training and are used as floating-point representations for the physical neurons on a chip, the sizes of LUTs are still crucial. The simulation study of [98] indicates that the performance with a 1024-entry lookup table is very close to what is achievable with the floating-point arithmetic method. But it is also clear that a smaller-size (256-entry) lookup table will result in degraded performance. If the variations of the characteristics of the neurons on a chip are significant and each of the neurons is in need to be represented by a LUT, the memory space that is used for storing these LUTs could be very large and the computation speed cannot be expected to be satisfactory.

To save the memory space for storing the LUTs, one can exploit a method called LUT assignment [103]. When the output of a nonlinear activation function is computed through a LUT, the LUT is usually shared by many neurons since the data of lookup tables are stored and recalled easily in a local memory. The LUT assignment could be a
difficult task. And it could produce bottleneck effects if the assignment is not done properly. The calculation of the output of an activation function using the LUT method is, by its nature, a process of interpolation. The precision of the calculation depends on the size of the lookup table as well as the interpolation method. The bottleneck effects combined with the interpolation procedure will result in slow computational speed.

The method of neuron modeling offers better performance from the viewpoint of interpolation. Moreover, the process of training a subnet using a set of input-output data is a process of data compression [98]. Hence, the memory space for storing the trained weights of a subnet will be much smaller than that for storing the LUT data when the same precision criterion of neuron computation is given. In other words, the neuron modeling method offers better performance than the LUT method when using the same size of memory. For instance, in the experiment we conducted in [95], we have used a subnet with 25 hidden neurons, which requires 50 weight values to be stored. The error between the measured and modeled nonlinear activation function was found to be 0.04% of the dynamic range, which looks enough for the neuron modeling purpose. It should be noted that the errors in the measured points from the neuron modeling method would not be zero while in a LUT method the error will be zero. In this perspective, a LUT method performs better. While the disparity of the MSE errors obtained from the two methods is not obvious and the conclusion cannot be made simply by observing the simulation results and using common sense, the further investigation on the errors will be a future work.

Finally, the neuron modeling method can be only used for the in-the-loop training or off-chip training. Besides these two applications, the LUT method can be applied as
an alternative scheme for implementing on-chip learning. This would be the advantage of LUT method over the neuron modeling method.

3.9.2 Nonlinearities in Weight Multiplication

Many of neural network chips fabricated in standard NMOS or CMOS technologies, such as those described in [104-106], employ some type of multiplier circuits to function as the synapses. However, a multiplier with a reasonable operating range, and with some deviation from the linear model can be quite commonplace. As the linearity of synapses is assumed in standard BP algorithm, the nonlinearities existent in weight multiplication circuits of a neural network chip are always concerns in the following training process.

One of the advantages of the in-the-loop training scheme is the flexibility of the model that is generated by mapping the hardware of neural network implementation (neural network IC chip) into the software environment installed “in-the-loop”. One can flexibly manipulate the operations on the model so as to train the neural network IC chip effectively. In the following, we describe an idea that can easily lend itself to handling the nonlinearities of the multiplication circuits. More specifically, by using the neuron modeling procedure we have described in this chapter, we are allowed to generate a model that characterize the input-output relationship of weight multiplier so that a nonlinearity-free multiplier can be assumed in the ensuing training process.

When there exists nonlinearity in a weight multiplier followed by a nonideal activation function, the transfer characteristics from an input and a synaptic weight to the output can be illustrated in Figure 3.12 (a), where nonlinear multiplication stage cascaded
Figure 3.12: The Block Diagrams of Transfer Characteristics of a Multiplication Stage Cascaded with a Neuron Circuit.

with a nonideal neuron. Note that the block diagram represents the situation when all the other weights are set to be zero. The transfer characteristics can be analytically written as

\[ V = F[f_{\text{nonlinear}}(\xi, w)] \]  \hspace{1cm} (3.59)

where \( F \) and \( f_{\text{nonlinear}} \) represent the input-output relationships of the neuron and synaptic weight, respectively. And we denote the input, the synaptic weight and the output with \( \xi, w \) and \( V \) respectively. When the multiplier is linear, the transfer characteristics can be expressed as

\[ V = F[f_{\text{linear}}(\xi, w)] = F[\kappa \xi w] \]  \hspace{1cm} (3.60)
where \( f_{\text{linear}} \) represents the linear relationship of the synaptic weight, and \( \kappa \) represents the proportional coefficient.

In the neuron modeling procedure we can see that any input and/or any weight value can be controlled arbitrarily. If we can vary the two quantities in a manner such that the ramp voltage to the neuron will sweep evenly from the negative maximum to positive maximum, then using our neuron modeling procedure we can obtain an activation function that models both the nonideal neuron as well as the nonlinearities existing in the multiplier. This is shown in Figure 12 (b) where the nonlinearity of weight multiplication can be transferred to the nonideal neuron if linear multiplication is desirable. An alternative method is to model the nonlinear multiplier component independently. This can be done by first modeling the nonideal neuron’s activation function, then use the inverse form of the activation function to isolate the nonlinear weight multiplier component. This idea has been given in the subsection 3.4.2 of this chapter.

The benefit of doing so is that the multiplier can thus be regarded as a linear component. Then in the ensuing in-the-loop training process, we are not concerned about the nonlinearities of multiplier stage. In other words, here we are modeling the relationship \( F[f_{\text{nonlinear}}(\ast)] \), while in previous cases, we were only looking for modeling the relationship of \( F(\ast) \).

To model both the nonlinear multiplier circuit and the nonideal neuron in a neural network IC chip, we will need a lot more detailed considerations beyond what have introduced above. They remain to be our future work.
3.10 Summary

This chapter deals with how the actual nonlinear activation functions of the neurons in a VLSI implementation can be modeled in terms of small ideal neural networks and related issues.

In section 3.2, the circuitry for implementing distributed neurons in the smart sensor is reviewed and analyzed. It has been shown that a set of analytical expressions for the activation function of the neuron can be derived based on MOS transistor current-voltage characteristics. In section 3.3, the theoretical basis for modeling physical neurons using small neural networks is presented. Since learning a mapping between a set of input data and a set of output data using a neural network architecture can be considered as a specific method of function approximation, section 3.4 gives approximation properties using a feedforward neural network from the perspective of approximation theory. The modeling procedure is described in details in section 3.5, which includes two different cases: modeling an activation function for a neural network with one hidden layer and two hidden layers. Two experimental results for the two individual cases can be found in the section. In section 3.6, the desirable properties of a small neural network are analyzed using a statistical mechanism. A model validation procedure based on statistical tests is described in section 3.7 which serves as a generic procedure to determine the validity of a small neural network model when modeling a specific activation function of a neuron. Section 3.8 deals with model structure determination and parsimony principle which lends itself to determining the efficiency of a specific neural network model. Some other related issues on neural modeling including comparison to a
look-up table method and ideas on solving nonlinearities of weight multiplication implementation are discussed in section 3.9.
CHAPTER 4: THE APPROXIMATED SERIES-DERIVATIVE METHOD

4.1 Introduction

We have introduced a new method for dealing with the physical and nonideal activation functions where the input-output characteristics cannot be modeled by explicit analytic equations. The method is to indirectly obtain measured values that represent the transfer relationship, or nonlinear activation function, of each of the actual analog neurons in a fully connected physical network that has a layer of input nodes, one or two hidden neuron layers, and a layer of output nodes. One can model each of the physical neurons by means of a small neural network or a subnet, that can be embedded in the larger neural network (LNN) or complete neural network model representing the complete physical VLSI realization as illustrated in Figure 3.7.

Once the larger neural network model is established, we are ready to proceed to the training phase so as to come up with a set of weights that can minimize the output errors. In this chapter, we will first review the currently available training algorithms for training a complete neural network in the in-the-loop manner. The drawbacks of these currently existing algorithms will be analyzed. A new method for the in-the-loop training of an LNN, the approximated series-derivative method, which is a Backpropagation-based method, will be proposed. We will then show the feasibility of the proposed method from perspectives of both theoretical analysis and simulation results.
To minimize the output errors of a neural network chip with digital weights like the smart sensor, we need to eventually come up with a set of discretized weights. If the training has been done using floating point representations, the resultant set of weights should be, sooner or later, customized into digital weights (quantized weights). In this chapter, we are only concerned about the weight values of floating point representations. The issues of dealing with quantized weights will be addressed in next chapter.

4.2 The Formulation of Complete Neural Network

In this section, we formulate the larger neural network where the nonlinear activation function has been modeled by a small feedforward neural network. Figure 4.1 gives the diagram of a two-layer feedforward neural network in which all the neurons are replaced with the embedded small neural networks. The notations that will be used in the following derivation are shown in Figure 4.1. Note that, without loss of generality, we have assumed that all the neurons have the identical transfer characteristics and hence only one set of weights for the subnets is used.

We first assume the ideal activation function for the subnet has the form of

\[ g(x) = \frac{1}{1 + e^{-2\beta x}} \]  \hspace{1cm} (4.1)

where \( \beta \) is a coefficient that controls the saturation property of the sigmoid function.

Then the transfer characteristics of the subnet can be expressed by equation (4.2) as:

\[ F(x) = \sum_{i=1}^{N} \omega_{i} g(\omega_{i} x) = \sum_{i=1}^{N} \omega_{i} \frac{1}{1 + e^{-2\beta x}} \]  \hspace{1cm} (4.2)
Figure 4.1: (a) The Complete Neural Network Model When All the Physical Neurons Are Modeled by Subnets (SNN). (b) The Notations for Subnets.

where $N$ denotes the number of hidden neurons in a subnet. The two layers of weights of the subnet are represented by $\omega$ and $\overline{\omega}$ with applicable indices.

For a specific set of input values (an input pattern), the output of the complete neural network can be explicitly expressed by the following equation:

$$O_i = F\left(\sum_j w_{ij}^{(2)} F\left(\sum_k w_{jk}^{(1)} \xi_k\right)\right)$$  \hspace{1cm} (4.3)

The training of the subnets has been described in the last Chapter. Thus the weights of the subnets are regarded as fixed and known parameters in this Chapter. The
training of the larger neural network is to determine the synaptic weights that are represented by \( W^{(1)} \) and \( W^{(2)} \), as shown in Figure 4.1.

4.3 The Algorithms for Training the Complete Neural Network

Generally speaking, for the training of a given LNN, one basically has three categories of training methods: BP-based algorithms, weight perturbation-based algorithms and direct search methods. It is necessary to investigate the existing algorithms that can be used for the training of the larger neural network. The advantages and disadvantages of using these algorithms for the training of the LNN are also discussed in this section.

4.3.1 The Standard BP Algorithm

We will describe the standard BP algorithm in more details in this section. Here

![Diagram of a Two Layer Feed-Forward Neural Network](image)

Figure 4.2: The Diagram of a Two Layer Feed-Forward Neural Network, Showing the Notations for All the Units and Weights.
we consider a two-layer network such as that illustrated by Figure 4.2 [107], on which the nomenclature of the related derivations is based. Note that the network is basically the one used in the last chapter. Specifically, output units are denoted by \( O_i \), hidden units by \( V_j \), and input terminals by \( \xi_k \). There are connections \( w_{jk}^{(1)} \) from the inputs to the hidden units, and \( w_{ij}^{(2)} \) from the hidden units to the output units. Note that the index \( i \) always refers to an output unit, \( j \) to a hidden one, and \( k \) to an input terminal. We label different patterns by a superscript \( \mu \), so input \( k \) is set to \( \xi_k^{\mu} \) when pattern \( \mu \) is being presented. The inputs can be binary (0/1, or \( \pm 1 \)) or continuous-valued. We use \( N \) for the number of input units and \( p \) for the number of input patterns (\( \mu = 1, 2, \ldots, p \)). In the following, we will begin with the formulation of the standard BP algorithm.

Given pattern \( \mu \), hidden unit \( j \) receives a net input

\[
h_j^{\mu} = \sum_k w_{jk}^{(1)} \xi_k^{\mu}
\]

and produces output

\[
V_j^{\mu} = g(h_j^{\mu}) = g \left( \sum_k w_{jk}^{(1)} \xi_k^{\mu} \right)
\]

Output unit \( i \) thus receives

\[
h_i^{\mu} = \sum_j w_{ij}^{(2)} V_j^{\mu} = \sum_j w_{ij}^{(2)} g \left( \sum_k w_{jk}^{(1)} \xi_k^{\mu} \right)
\]

and produces for the final output

\[
O_i^{\mu} = g(h_i^{\mu}) = g \left( \sum_j w_{ij}^{(2)} V_j^{\mu} \right) = g \left( \sum_j w_{ij}^{(2)} g \left( \sum_k w_{jk}^{(1)} \xi_k^{\mu} \right) \right)
\]

The error measure or cost function is usually defined.
\[ E(\tilde{w}) = \frac{1}{2} \sum_{\mu} \left[ \epsilon^\mu - O^\mu \right]^2 \]  

(4.8)

now becomes

\[ E(\tilde{w}) = \frac{1}{2} \sum_{\mu} \left[ \epsilon^\mu - \mathcal{g} \left( \sum_i w_i^{(2)} g \left( \sum_j w_j^{(1)} \xi^\mu_k \right) \right) \right]^2 \]  

(4.9)

This is clearly a continuous differentiable function of every weight, so we can use a gradient descent algorithm to change the weights.

For the hidden-to-output connections the gradient descent rule gives:

\[ \Delta w_{ij}^{(2)} = -\eta \frac{\partial E}{\partial w_{ij}^{(2)}} = \eta \sum_{\mu} \left[ \epsilon^\mu - O^\mu \right] g'(h^\mu_j) V^\mu_j = \eta \sum_{\mu} \delta^\mu_i V^\mu_j \]  

(4.10)

where we have defined

\[ \delta^\mu_i = g'(h^\mu_i) \left[ \epsilon^\mu_i - O^\mu_i \right] \]  

(4.11)

For the input-to-hidden connections \( \Delta w_{jk}^{(1)} \) we must differentiate with respect to the \( w_{jk}^{(1)} \)'s, which are more deeply embedded in (4.9). Using the chain rule, we obtain

\[ \Delta w_{jk}^{(1)} = -\eta \frac{\partial E}{\partial w_{jk}^{(1)}} = -\eta \sum_{\mu} \frac{\partial E}{\partial V^\mu_j} \frac{\partial V^\mu_j}{\partial w_{jk}^{(1)}} \]  

\[ = \eta \sum_{\mu} \left[ \epsilon^\mu - O^\mu \right] g'(h^\mu_j) w_{ij}^{(2)} g'(h^\mu_j) \xi^\mu_k \]  

\[ = \eta \sum_{\mu} \delta^\mu_i w_{ij}^{(2)} g'(h^\mu_j) \xi^\mu_k \]  

\[ = \eta \sum_{\mu} \delta^\mu_i \xi^\mu_k \]  

(4.12)

with

\[ \delta^\mu_j = g'(h^\mu_j) \sum_i w_i^{(2)} \delta_i^\mu \]  

(4.13)
Note that (4.10) and (4.12) has the same form, but with a different definition of the $\delta$'s.

In general, with an arbitrary number of layers, the BP update rule always has the form

$$\Delta w_{pq}^{(m)} = \eta \sum_{\text{patterns}} \delta_{\text{output}} \cdot V_{\text{input}}$$  \hspace{1cm} (4.14)

where output and input refer to the two ends $p$ and $q$ of the connections concerned, and $V$ stands for the appropriate input-end activation from a hidden unit or a real input. The meaning of $\delta$ depends on the layer concerned; for the last layer of connections it is given by (4.11), while for all other layers it is given by an equation similar to (4.13). It is quite straightforward to derive this generalized multilayer result as in (4.14) simply by further application of the chain rule.

The standard BP algorithm assumes that both the nonlinear neuron activation function and its derivative are known analytically. When an actual physical realization of a neural network whose neuron activation functions are modeled in terms of subnets is to be trained in-the-loop, neither of these two functions are known analytically. Although the transfer characteristics of the subnet can be expressed analytically in terms of a weighted summation of the ideal sigmoidal function as in equation (4.2), the intensive computation still limits the practical use of the standard BP algorithm for the training of an LNN. Moreover, for the case that the target LNN has different subnets, it is practically impossible to directly employ the standard BP algorithm by using the analytical equations like equation (4.2).

4.3.2 A BP-Based Algorithm: NLBP

One of the variants of BP algorithms that is called nonlinear BP (NLBP) method [108] is of interest in this context. The NLBP method can be thought of as a "nonlinear gradient descent" type of algorithm. In the standard BP algorithm one calculates the
errors on the hidden units from the errors on the output neurons by means of a linear expression. The NLBP algorithm has the advantage that the backward propagation of the errors goes through the same nonlinear units as the forward propagation activities. Using this method it is no longer necessary to calculate the derivative of the activation function for each neuron as it is in the standard BP algorithm. Whereas the derivatives are trivial to calculate in a simulation, they appear to be of major concern when implementing the algorithm in hardware. A brief description of the NLBP algorithm is given in the following. We have basically used the same notations as those in the last subsection of this chapter. For convenience, we have employed the index notations the same as in [108].

If the error measure for a feedforward neural network is given by equation (4.8) the derivatives of the error measure are

\[ e_k = \begin{cases} 
\zeta_k - V_k & \text{for the output units} \\
0 & \text{otherwise}
\end{cases} \]  

(4.15)

where as before we use \( \zeta_k \) to denote the target values for the output units and \( V_k \) is the actual output (we have used \( O_k \) to denote the actual output in the output layer). For an output unit in a feedforward network we thus find \( \delta_i = V_k'(\zeta_k - V_k) \). One of the ideas of the nonlinear BP is to force that interpretation on all the units, defining "effective targets" \( y \) such that

\[ \delta_i = y_i - V_i \]  

(4.16)

For small learning rate \( \eta \) the equation (4.11) can then be interpreted as a first-order Taylor expansion
\[ \Delta w_{ij} = \eta \delta_i V_j = \eta V_i \left( \sum_k \delta_k w_{ki} + e_i \right) V_j \]

\[ \equiv \left[ g \left( h_i + \eta \left( \sum_k \delta_k w_{ki} + e_i \right) \right) - g(h_i) \right] V_j \]  

(4.17)

where \( g(h_i) = V_i \). The first term in the brackets is just the output of a unit with the normal input and the back-propagated error added - the effective target:

\[ y_i = g \left( h_i + \frac{\eta}{\alpha_i} \left( \sum_k \delta_k w_{ki} + e_i \right) \right) \]  

(4.18)

Using (4.18) and (4.16) one can see that for a small \( \eta \) the weight change rule will be essentially the same as in standard BP algorithm. For consistency one may rewrite (4.17) as

\[ \Delta w_{ij} = \eta \delta_i V_j = \alpha_i \left[ g \left( h_i + \frac{\eta}{\alpha_i} \left( \sum_k \delta_k w_{ki} + e_i \right) \right) - g(h_i) \right] V_j \]  

(4.19)

where the \( \alpha_i \)'s are arbitrary parameters similar to the learning rates \( \eta_i \). The \( \delta_k \) can be rewritten as:

\[ \delta_k = \frac{\alpha_k}{\eta_k} (y_k - V_k) \]  

(4.20)

From (4.18) and (4.20) one gets

\[ y_i = g \left( h_i + \frac{\eta}{\alpha_i} \left( \sum_k \frac{\eta}{\alpha_i} (y_k - V_k) w_{ki} + e_i \right) \right) \]  

(4.21)

All the values of \( y_i \) are calculated by (4.21). Finally from (4.19) the weight update rule can be expressed as

\[ \Delta w_{ij} = \alpha_i \sum_{\mu} (y_i^\mu - V_i^\mu) V_j^\mu \]  

(4.22)

The fact that NLBP algorithm does not rely on the calculation of derivatives of activation function makes significant difference from the standard BP algorithm, although the NLBP algorithm has been derived by simple mathematical manipulation and is obviously a variant of the conventional BP algorithm. The basic idea of this method is to find a variable that is an "effective" or "moving" target for each hidden neuron in the network.

Using the procedure from which the NLBP method is derived, we find that the algorithm is very computationally intensive. Moreover, equation (4.17) has been obtained by using a first-order Taylor expansion with the assumption that the learning rate $\eta$ is small. Thus there will exist problems in choosing the magnitude of parameter $\eta$ and consequently the convergence cannot be guaranteed.

4.3.3 Weight Perturbation Methods

Normally, the training of the larger neural network, with the embedded subnet models of specific neurons, requires the use of a weight perturbation method [109] since the mathematical expressions for the modeled neuron's activation functions cannot be used directly in a back-propagation process, but are expressed in terms of subnets. The weight perturbation-based algorithms are well suited to analog VLSI implementations for the reason that they do not make assumptions about the synapse and neuron circuits characteristics and error gradients are measured rather than calculated [46]. Thus this algorithm is extremely tolerant of any nonidealities in analog circuits and an extremely compact circuit may be used from the hardware implementation viewpoint. The most general form of weight perturbation-based algorithms is the so-called serial weight perturbation in which weights are updated according to:
\[ \delta w_{ij} = -\eta \frac{\Delta E}{\Delta w_{ij}} = -\eta \frac{E_{\text{pert}} - E_{\text{nom}}}{\Delta w_{ij}} \]  

(4.23)

where \( \eta \) is the learning rate, \( \Delta w_{ij} \) is the perturbation magnitude, \( E_{\text{pert}} \) is the perturbed error and \( E_{\text{nom}} \) the nominal error. Based on the idea of serial weight perturbation, many variations have been devised in recent years, for instance, Madaline rule III [110], summed weight neuron perturbation [80], parallel stochastic weight perturbation [111] and the CHRP algorithm [112].

For the in-the-loop training of the complete physical VLSI implementation the error between the larger neural network model output and the desired output is easy to be obtained. Thus, the effect of weight perturbation in the model can be seen immediately. However, changing the synaptic weights in a combinatorial manner, as is done in the weight perturbation method, can take a long time to determine the set of weights that minimizes the error. In addition, as have stated in Chapter 2, it is very hard to determine a proper size for perturbing the weights, while the perturbation size is critical to the efficiency as well as the validity of the method [110]. Needless to say, if the perturbation size is too large, the derivative information found by weight perturbation is not accurate enough for the weight updating procedure. While the weight perturbation method is contingent on the derivative information of the error output with respect to each synaptic weight, the low accuracy of derivative calculation will degrade the performance of the whole method. If the perturbation size is too small, the ensuing computation will involve floating point operations with high precision which will in turn slow down the convergence speed. For the case that the activation function includes discontinuities, there exist possibilities that the weight perturbation is done within some discontinuous
segments. In this case the weight perturbation method breaks down if using an undersized perturbation.

4.3.4 Direct Search Methods

The feedforward (FF) training method presented in [113] belongs to direct search methods and is based on the so-called univariant search parallel to axes [114].

The FF training method is a very simple training law suitable for on-chip learning. Its merit is conceptual and implementational simplicity. Its signals do not propagate in both directions as in the BP algorithm but only in forward direction. It does not calculate or estimate any gradient. All it has to do is to detect an increase or decrease in the output error due to a change in a synaptic weight. Therefore, the method works for various types of activation function, a feature that makes it particularly effective in the case of unmodeled activation functions. It was claimed in [113] that this method is usually faster than the standard BP algorithm. The fact that such a simple training law works very well might indicate the munificence of high dimensionality [115].

The updating procedure of a particular weight \( w \) consists of three phases, namely, the perturbation, the forward steps, and the backward steps phase. They are briefly explained in the following. (a) The perturbation phase: The training also starts from a set of random initial weights. The direction of a search for one particular weight is determined by perturbation. If the total error reduces when the specified weight increases by a fraction of the step size \( s \), the direction of search is positive, otherwise it is negative. (b) The forward steps phase: If a forward step succeeds in reducing the total error it is considered to be successful. The forward steps continue as long as they are successful. This procedure stops if either the maximum allowable number is reached or an
unsuccessful step is encountered. The total error $E_{\text{last}}$, corresponding to the last successful step, is used as reference. (c) The backward steps phase: This phase starts when the forward steps fail to reduce the total error. A weight will be changed in a direction opposite to the direction of the forward steps. The size of the first step is $s/2$, and it is decreased progressively by a factor of two for every following step. The backward steps continue as long as they do not succeed in reducing the total error with respect to $E_{\text{last}}$. This procedure stops if either the maximum allowable number is reached or a decrease in the total error with respect to $E_{\text{last}}$ is encountered.

Since the FF method is based on univariant search parallel to axes, it may fail to converge if narrow ridges are present [114]. This might not be a problem since it does not search for the minimum of the total error along the direction of search, but it searches for a point that is near the minimum. The other problem associated with the FF method is that all the calculations are done in series, which will be very computationally intensive and time consuming.

4.4 The Approximated Series-Derivative Method of BP Training

While the training methods introduced in the last section are possibly usable for the training of a given larger neural network model, there exist problems from perspectives of training computation and convergence. Thus there is a need to develop an alternative training method that is specifically suitable for the training of an LNN.

The conventional BP algorithm is probably the best known of the training algorithms for neural networks [116]. While the standard BP algorithm cannot be directly applied to the training of a physical implementation of a neural network, there is a need to derive a modified version of the BP algorithm so that one can still take
advantage of the algorithm. The basic idea of the proposed BP-based algorithm for the in-the-loop training has been presented in our work [117].

The conventional BP training algorithm consists of two phases. In the first phase, called the forward phase, an input pattern is presented to the network, then propagated forward and output values are computed. These values are compared with the desired output values and in the second phase (backward phase) the weights on the connections are modified according to the difference between the desired and computed value as well as the derivative values of the concerned activation functions. Specifically, from Equations (4.10) and (4.12), we can see that the realization of the BP algorithm is closely contingent on the evaluation of the derivative of activation function. It should be noted here that the evaluation of derivative values of activation functions is only related to the process of weight modification. In other words, the derivative evaluation is important to the process of training, but not useful after the completion of the training.

If ideal neurons (e.g., with ideal sigmoidal characteristics) are used, the evaluation of derivative is an easy task because the derivative expression can be explicitly found. However, if nonideal neurons (which are referred to as having no explicit mathematical expression) are used, the derivative evaluation has been proved to be a rather difficult task [118]. Indeed, the objective of weight perturbation and node perturbation is to find the derivative information for output errors. Now the question arises: are we allowed to use approximated derivative values in (4.10) and (4.12) while we assume that the derivative approximation can be obtained from some known explicit equation? One thing we should keep in mind is that we are dealing with neural network, a structure which
would have not unique solutions meeting the cost function tolerance, therefore finding
the exact weight update direction seems not as critical as it appears.

It is desirable to find an analytically known function with a controllable parameter
so that a good approximation to the derivative of the physical nonlinear activation
function can be obtained by adjusting the parameter. One of the functions that can meet
the requirements is given by

![Graphs showing input-output characteristics and derivative](image)

Figure 4.3: The Input-Output Characteristics of the Activation Function \(F(x)\) Realized by
the Circuit Shown in Figure 2.3 and its Derivative \(F'(x)\).
\[ p(x) = \frac{2\beta e^{-2\beta x}}{(1 + e^{-2\beta x})^2} \]  \hspace{1cm} (4.24)

Figure 4.3 shows the input-output characteristics of the activation function \( F(x) \) realized by the circuit shown in Figure 2.3 and its derivative \( F'(x) \). The function \( p(x) \), which has an explicit analytical expression, is made to approximate the function of \( F'(x) \), although this approximation cannot be a good one in the sense that the deviation is quite large compared to the magnitude of \( F'(x) \). We will show later in this chapter that the algorithm can still work even though the approximating function could deviate from the actual derivation function significantly at some points. Note that the integration of \( p(x) \) will become a standard sigmoidal function as expressed in the following

\[ \int p(x) \, dx = \int \frac{2\beta e^{-2\beta x}}{(1 + e^{-2\beta x})^2} \, dx = \frac{1}{1 + e^{-2\beta x}} \]  \hspace{1cm} (4.25)

It should be remarked here that changing the parameter \( \beta \) can shift the curve of \( p(x) \) and can make the \( p(x) \) better fit \( F'(x) \). More details about changing \( \beta \) are trivial and will be not given in this context.

From the general form the update rule of BP algorithm shown in (4.14), we notice that out of the two terms inside the summation sign only \( \delta_{\text{output}} \) is related to the derivative of activation function. The term \( V_{\text{input}} \) is related to the activation function itself and can be easily evaluated in a complete neural network model. More specifically, if observing the equations (4.10) and (4.11), we can find that the utility of the approximated derivative information will alter the direction of \( \Delta w_{ij}^{(2)} \) slightly in Euclidean space. Note that the derivative information is only used in the error backward propagation and has nothing to do with signal forward propagation. In other words, using approximated derivative
information might make the training process lengthy but will not affect the accuracy of the trained results. Most importantly, by using approximated derivative information, one can take the advantage of the standard BP algorithm and a weight perturbation-based method can be avoided.

For the proposed training method to converge to a minimum in fewer steps it is also necessary to employ an adaptive learning rate (ADL) strategy [110]. Adaptive learning increases the incremental changes in the weights when the error is rapidly decreasing and decreases the incremental changes in the weights when the error is slowly decreasing or increasing. In particular, the learning rate is adapted using the following rule:

\[
\eta(t + 1) = \begin{cases} 
\eta(t) \cdot (1 - \varepsilon_1) & \text{if } \Delta E(t + 1)/\Delta E(t) > \chi \\
\eta(t) \cdot (1 + \varepsilon_2) & \text{if otherwise}
\end{cases}
\]  

(4.26)

where \( \varepsilon_1 \) and \( \varepsilon_2 \) are small positive constants, and \( \chi \) a preset error ratio. It will be shown in the following that the ADL method can minimize the possibility that the training converges to a local minimum.

4.5 Theoretical Analysis for the Approximated Derivative Method

The described approximated series-derivative training algorithm aims at avoiding a combinatorial method and taking advantage of the BP algorithm. By using the actual measured output error and the approximated derivative information, the standard BP algorithm can be employed. One must be concerned whether the algorithm can still converge to the desired solution. This section is dedicated to giving theoretical analyses on the convergence of the proposed BP-based training algorithm.
4.5.1 The Illustrative Analysis

Obviously, using approximated derivative information means the search for error function minimization is not in the steepest gradient direction, but in a direction deviated from the steepest direction. The deviation angle between the two directions depends on the goodness of the derivative approximation. However, as long as the deviation angle is less than 90°, the increment of weight updating will result in a reduction of the error function since the steepest gradient direction is perpendicular to the equipotential surface. Refer to Figure 4.4, it can be seen that the method works if the search is within A region, as shown in Figure 4.4(a), while the step of search fails to reduce the error function if the angle becomes obtuse as shown in Figure 4.4(b). In Figure 4.4, the thick solid line represents an equi-potential hypersurface. This angle tolerance greatly reduces the need for high derivative approximation accuracy.

Figure 4.4: The Two Distinctive Cases of Optimization Search Where the Method Works in (a) but Fails in (b).
It should be remarked here that the utility of the proposed algorithm is enhanced as we are dealing with a neural network; a structure that does not have a unique solution.

![Diagram of ADL Method in Getting-Out-Of a Local Minimum Trap](image)

Figure 4.5: An Illustration of the Utility of ADL Method in Getting-Out-Of a Local Minimum Trap. (a) The Difference of Gradient Descent Search and Derivative Approximation Search; (b) The Situation That Derivation May Get Trapped in a Local Minimum.

for cost function minimization. Therefore even several steps of weight updating fail to reduce the error function output, there still exists a high probability that another optimal solution will be reached.

In our proposed BP-based training method, the incorporated adaptive learning rate (ADL) method plays an important role in the process of steering the search direction to the final solution point. This is shown in Figure 4.5 (a) and (b). In the case that the training becomes stuck in some local minima due to using derivative approximation as illustrated in Figure 4.5(b), the ADL method can help the training get out from trapping in local minima. This is because the ADL will decrease the learning rate and therefore the step of weight changing will be also decreased when cost function tends to get larger.
Note here we don’t consider the case that a trial gets stuck in local minima or on a very flat plateau at the very beginning stage since even the standard BP faces the same problem. The avoidance method is to give up on that trial when no material progress is made after some number of training iterations.

4.5.2 The Analytical Explanation

From the equations (4.11) and (4.13) we notice that the quantity $\delta$ always has the form

$$\delta = g'(h) \cdot \psi$$  \hspace{1cm} (4.27)

where $g'(h)$ represents the derivative function and $\psi$ denotes error information. Specifically, in (4.11) $\psi = [e_i - O_i^\mu]$, and in (4.13) $\psi = \sum_i w_{ij}^{(2)} \delta_i^\mu$ which reflects weighted error information. By combining (4.11) and (4.14) we obtain

$$\Delta w_{pq} = \sum_\mu g'(h_p^\mu) \cdot (\eta \psi V)^\mu_{pq}$$  \hspace{1cm} (4.28)

Note that in (4.28) we have omitted the subscripts and superscripts since the purpose of the following derivations is to investigate the generalized properties of weight update rule. We rewrite (4.28) as

$$w_t = \sum_{\mu=1}^p a_t^\mu f(b_t^\mu)$$  \hspace{1cm} (4.29)

where we regard the incremental weight change $\Delta w$ as a vector with elements $\{\Delta w_t\}$ ($t=1,2,\ldots,p \times q$), and also we assumed that $a_t^\mu = (\eta \psi V)^\mu_{pq}$, $b_t^\mu = h_p^\mu$, and $f = g'$. Note that we have substituted $\Delta w_t$ with $w_t$ for simplicity.
Now suppose that we want to use another function \( f_a \) to approximate the derivative function \( f \), then similarly the approximated incremental weight change can be written as

\[
\mathbf{w}^a_i = \sum_{\mu=1}^{p} a_i^\mu f_a(b_\mu) \quad (4.30)
\]

To investigate how the use of the approximated function \( f^a \) affects the direction of the incremental weight change in Euclidean space, we make use of the following sufficient condition that ensures the approximated incremental weight change to reduce the error function:

\[
\|\mathbf{\tilde{w}} - \mathbf{\tilde{w}}^a\| < \|\mathbf{\tilde{w}}\| \quad (4.31)
\]

where we use \( \mathbf{\tilde{w}} \) and \( \mathbf{\tilde{w}}^a \) to denote the vector of the incremental weight changes, and \( \|\mathbf{\cdot}\| \) represents the distance norm. Refer to Figure 4.4, the condition is quite straightforward if we let the tip of \( \mathbf{\tilde{w}}^a \) fall in the circle with the center at the tip of \( \mathbf{\tilde{w}} \) and the radius of \( \|\mathbf{\tilde{w}}\| \).

Without loss of generality, we consider the case of \( \mu = 1 \) (\( p = 1 \)) and \( t = 1.2 \) which means from (4.29) and (4.30)

\[
\begin{align*}
\mathbf{\tilde{w}} &= \begin{bmatrix} a_1f(b_1) \\ a_2f(b_2) \end{bmatrix} \\
\mathbf{\tilde{w}}^a &= \begin{bmatrix} a_1^1f_a(b_1) \\ a_2^1f_a(b_1) \end{bmatrix}
\end{align*} \quad (4.32)
\]

For simplicity, we substitute \( a_1^1, a_2^1, b_1^1 \) and \( b_2^1 \) with \( a_1, a_2, b_1 \) and \( b_2 \), respectively. Then (4.32) can be rewritten as

\[
\begin{align*}
\mathbf{\tilde{w}} &= \begin{bmatrix} a_1f(b_1) \\ a_2f(b_2) \end{bmatrix} \\
\mathbf{\tilde{w}}^a &= \begin{bmatrix} a_1f_a(b_1) \\ a_2f_a(b_2) \end{bmatrix}
\end{align*} \quad (4.33)
\]

From (4.31) the condition can be expressed as

\[
a_1^2(f(b_1) - f_a(b_1))^2 + a_2^2(f(b_2) - f_a(b_2))^2 < a_1^2f^2(b_1) + a_2^2f^2(b_2) \quad (4.34)
\]
It follows that
\[ |f(b_1) - f_a(b_1)| < f(b_1) \]
\[ |f(b_2) - f_a(b_2)| < f(b_2) \]  \hspace{1cm} (4.35)

Note that we have assumed that \( f(\cdot) \) and \( f_a(\cdot) \) are both evaluated as positive values. In reality, this assumption makes sense since the derivative of an activation function should be always positive. Now if \( f > f_a \), the inequalities in (4.24) are always true; but if \( f_a > f \), then we have
\[ f < f_a < 2f \]  \hspace{1cm} (4.36)

The inequality (4.36) means that for (4.31) to be held, \( f_a \) could be chosen as large as two times of \( f \). This means that the approximated function can work for the algorithm if the difference between them is within 50% of the magnitude of the approximated function. Although this kind of approximation makes the convergence very hard in reality, it can be concluded that an approximated function is easy to find.

4.6 Test of the Algorithm

The theoretical analysis described above is an auxiliary explanation rather than a rigid proof in mathematics. Therefore, it is reasonable to rely heavily on extensive and stringent simulation experiments to determine the performance of our algorithm. In this section, we will use two sets of training patterns to do the simulation of training.

4.6.1 Pattern Recognition With the Smart Sensor

The first test described in this subsection can also be found in our previous work [117]. The neural network-based smart sensor, which we have described in details in chapter 2, has been used to test the proposed BP-based training algorithm. In the first simulation, the network is set to have 64 input nodes, 8 hidden neurons and 4 output
neurons. The actual input-output characteristics of the neuron's activation function have been measured. In Figure 4.3(a) the input-output characteristics (F(x)) of neurons in the smart sensor by measuring a neuron on the chip has been given.

![Graphs showing training and learning rate](image)

Figure 4.6: A Training Curve and Its Corresponding Learning Rate Curve for Training the Smart Sensor Using the Given Patterns.

The function which is used to approximate the derivative of F(x) is chosen as the one expressed in equation (4.24) with the parameter $\beta = 1.5e+4$ as is shown in Figure 4.3(b). The training patterns represent typical inputs that would occur when the smart sensor chip is being used as a programmable intelligent sensor [15], which is also given in Figure 2.8. The sum-squared error goal is chosen as 0.02. The other parameters are
chosen as follows: \( \eta(0) = 0.1, \varepsilon_1 = 0.3, \varepsilon_2 = 0.04 \) and \( \chi = 1.04 \). We have tested our algorithm quite a number of times using the same set of input patterns.

A typical training curve and its corresponding learn rate curve are given in Figure 4.6. The two curves represent the typical training process when conducting training for the smart sensor using the given patterns. It can be found that the reducing rate of the sum-squared errors always changes when learning proceeds. The rate changes coincide with the learning rate adaptations. We can see from the figure that the training curve finally converges to below the training criterion line.

Note that the training curve, which represents the typical training process of this training scheme, does not look smooth but looks “bumpy”. One should realize that this is

![Graph showing training curves for 253 epochs](image)

Figure 4.7: The Four Training Curves of the Smart Sensor Network When Trained by the Patterns Given in Figure 2.8.
the result of the derivative approximation. More specifically, the deviation of an optimization search direction determined by the approximated derivative information from the steepest gradient descent direction may incessantly alter, therefore, the reductions in its output error resulting from different incremental weight modifications will be changing accordingly. It also happens that the output error increases after a weight modification. However, our experiments indicate that in most of times, the output error will eventually reduce to below the desired value.

To see what happens if the training starts with different initial weights, we have performed the same training task four times, out of which each time the training started from a different set of random weights. In Figure 4.7, the four training curves are drawn in an identical figure, with each curve representing a training process starting from the four different sets of initial weights.

Theoretically, the derivative approximation method is possibly trapped in local minima, although we have not encountered the situation in our trials (the number is still limited) since the odds are really small. In Figure 4.7, four training curves out of the more than 100 trails have been randomly selected to illustrate the training process. From the curves in Figure 4.7, we notice that the number of training epochs a training takes before the sum-squared error (SSE) is reduced below the tolerance level (0.02) varies considerably and is closely related to the starting initial weights.

The same training task has been done using the standard weight perturbation method and the standard BP with ideal neurons for the purpose of comparison. In Figure 4.8, three training curves based on standard BP, derivative approximation and weight perturbation are plotted in the same figure. The figure does not give how many training
epochs the weight perturbation-based training algorithm takes before it achieves its cost function criterion (SSE:0.02). The average number of training epochs using the standard weight perturbation method is usually around 3300 for the current training task based on the results of 20 training tests, while the average number of training iterations using the proposed algorithm is around 320 based on 100 training tests, and the number for the standard BP is around 120, again based on the results of 100 training tests. Note all the training tests have been done starting from different random initial weight values.

Note that for the training curve of BP algorithm, we have assumed an ideal representation (sigmoidal function) for the activation functions, which is not the case for a physical realization of the feedforward neural network. The purpose of the performance comparison between the BP algorithm and the derivation approximation

![Graph](image)

Figure 4.8: A Comparison of Training Curves of the Standard BP, a Weight Perturbation Method and Derivative Approximation Method.
method is to give a general idea that how many more iterations have been increased due to the derivation approximation. The experimental results indicate that the proposed algorithm is superior to a weight perturbation-based training algorithm when applied to the in-the-loop training of actual physical realizations where the neuron activation functions can only be approximated using measured data.

The key numbers of training epochs for the three distinctive training schemes have been given in Table 4.1. We have used maximum, minimum, average number and standard deviation to gauge the number of training iterations for each method.

<table>
<thead>
<tr>
<th>Training method</th>
<th>Number of Tests</th>
<th>Maximum number</th>
<th>Minimum number</th>
<th>Average number</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP</td>
<td>100</td>
<td>205</td>
<td>87</td>
<td>123</td>
<td>21</td>
</tr>
<tr>
<td>DA</td>
<td>100</td>
<td>487</td>
<td>229</td>
<td>319</td>
<td>49</td>
</tr>
<tr>
<td>WP</td>
<td>20</td>
<td>4612</td>
<td>1892</td>
<td>3310</td>
<td>545</td>
</tr>
</tbody>
</table>

BP = Backpropagation; DA = Derivative Approximation; WP = Weight Perturbation

### 4.6.2 Exclusive-Or Problem

The Exclusive-Or (XOR) problem has been employed as a training/learning problem in many neural network publications. The XOR problem represents a classical benchmark pattern classification problem. It provides an example for the use of hidden units in multilayer neural networks. The input/output relation is specified in Table 4.2.

We note that this classification problem involves 4 input patterns and 2 output patterns.
Table 4.2: Input-Output Relation of XOR

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The network that is capable of working for XOR classification problem should have two input units, one output unit and one or more hidden units. A network without hidden units cannot classify the patterns, but a network with one hidden unit and with connections between the input and output units can classify the patterns [98], which is illustrated in Figure 4.9(a). For multilayer feedforward neural networks, the minimal structure that can work for the problem should have at least 2 hidden neurons as shown in Figure 4.9(b). In this experiment we have chosen a 2-layer feedforward network with 3 hidden neurons to perform the training tasks. The network topology is shown in Figure 4.9(c). The reason for choosing this topology is that the training process using the topology of Figure 4.9(b) takes too many training epochs before a reasonable training criterion is achieved. Also, our objective of conducting this experiment is to give the comparison of convergence speed among the BP algorithm, the derivative approximation method and weight perturbation method. To run the simulation for BP algorithm, we have assumed the ideal sigmoidal transfer characteristics for the hidden neurons of the structure of Figure 4.9(c). To run the simulation for the derivative approximation and weight perturbation methods, we have assumed that the hidden neurons have the input/output relationship as shown in Figure 4.3(a), which has been measured from the smart sensor. The second objective of conducting this experiment is to visualize the
process of the searching for minimizing the output errors when weight modification proceeds. In the simulations of this subsection, the sum-squared error goal is set to be 0.004. The other parameters are the same as those in last subsection. Besides the training curves, we will also plot the loci of one weight value ($W^{(2)}_1$) with respect to the other ($W^{(2)}_2$) in the second layer when training proceeds. We will call the loci as weight loci in the following. Note that there is no special reason that we choose the two specific weights to form the locus. The fact is that we would have similar loci if choosing other two synaptic weights to form a locus.

![Diagrams](image)

Figure 4.9: The Diagrams of Neural Network Topologies That Can Be Used for XOR Classification Problem.

The following seven training curves (Figure 4.10 – Figure 4.16) have been generated by training the network shown in Figure 4.9(c) with XOR problem and different random initial weights. The three training curves and their weight loci shown in Figure 4.10, Figure 4.11 and Figure 4.12 are obtained by using standard BP algorithm, while the training curves and corresponding loci shown in Figures. 4.13 to 4.15 are
obtained by using the derivative approximation method. In Figure 4.16, the result of using a weight perturbation method is given. Note that the results shown in Figures 4.10 to 4.12 represent three typical simulation results of the weight loci for BP algorithm.

And the results shown in Figures 4.13 - 4.15 also cover the three typical situations when running the derivative approximation algorithm. We have given only one training result for the weight perturbation simulation since we found that the training curves and weight loci obtained from different simulations are basically similar. In all the figures, we have used 'Starting' and 'Solution' to denote the starting point and solution point (in Euclidean space) of a weight locus.

In Figure 4.10, the locus is relatively 'straight', and the corresponding training curve appears somewhat 'smooth'. In Figure 4.11, we can see that somewhere in the locus looks 'wavelike' but its trend is still straight. Interestingly, its training curve is still relatively smooth. The reason for this may be related to the insensitivity of some weights in some training stages. In Figure 12, the whole trend of the locus is in no sense straight. Rather, the direction has wavered abruptly. In this case, the correspondent training curve appears to be 'bumpy'. This shows the existence of some 'bumpy' ridges on the optimization search surface.

The training curves and weight loci obtained from running the derivative approximation method are seemingly different from those of Figure 4.10 to Figure 4.12. In Figure 4.13, the whole locus has a chain-like part but is relatively 'straight'. Thus the training is still relatively faster and the curve is smooth if comparing the training curves shown in Figure 4.14 and Figure 4.15. The weight locus in Figure 4.14 looks quite twisted at the beginning part and wavy at the end part. They coincide the two bumpy
parts in the training curve. In Figure 4.15, the weight locus is the most twisted one in the simulations here. Correspondingly, the training has taken much more iterations than have the other two training process in Figure 4.13 and Figure 4.14. And the training curve looks quite 'bumpy' and is the least desirable type. It can be found that the training epochs in Figures 13-15 are more than those in Figures. 10-12. It should be recognized that this is the price we paid for not using the accurate derivative information but approximated one.

We have given only one figure for the training by weight perturbation method since they basically repeat the training curves and loci described above. In Figure 4.16, the locus looks quite 'straight' as is often the case in weight perturbation computation. But the correspondent training curve still looks very bumpy. The reason for this is that the derivatives can never be evaluated with an appropriate accuracy everywhere due to the perturbation size problem although theoretically the weight searching in a weight perturbation method is in the direction of steepest gradient descent. As can be expected, the training has taken much more iterations of computation comparing the other two methods.

From Figures 4.10 – 4.16, we can also find that the training epochs are closely related to their corresponding weight loci. When using the derivative approximation method, the weight loci apparently become more ‘irregular’, but the corresponding training epochs are not affected as much as it appears. This fact corroborates that using approximated derivatives is feasible and do not degrade the training performance as much as it does to the weight loci.
Table 4.3: The Key Numbers of Training Epochs of the Three Training Schemes for XOR Classification Problem.

\[ \eta(0)=0.1; \varepsilon_1=0.3; \varepsilon_2=0.04; \gamma=1.04; \text{SSE}=0.004; \]

<table>
<thead>
<tr>
<th>Training method</th>
<th>Number of Tests</th>
<th>Maximum number</th>
<th>Minimum number</th>
<th>Average number</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP</td>
<td>40</td>
<td>119</td>
<td>61</td>
<td>81</td>
<td>12</td>
</tr>
<tr>
<td>DA</td>
<td>40</td>
<td>231</td>
<td>78</td>
<td>152</td>
<td>26</td>
</tr>
<tr>
<td>WP</td>
<td>30</td>
<td>519</td>
<td>218</td>
<td>367</td>
<td>73</td>
</tr>
</tbody>
</table>

BP = Backpropagation; DA = Derivative Approximation; WP = Weight Perturbation

The key numbers of the training epochs for the three distinctive training schemes are given in Table 4.3. Generally, we can draw a conclusion that the derivative approximation method performs better than weight perturbation method. The standard BP algorithm is not applicable to the training of a physical realization of the feedforward neural network although it performs much better than the other two algorithms since it assumes ideal representations for the activation functions. From the data of BP algorithm and derivative approximation method, we can see the effects of using approximated derivative information.
Figure 4.10: A Training Curve and the Corresponding Weight Locus of $W_2(2)$ Versus $W_1(2)$: The Case of a Relatively 'Straight' Locus.
Figure 4.11: A Training Curve and the Corresponding Weight Locus of W2(2) Versus W1(2): The Case of a Wavelike but still 'Straight' Locus.
Figure 4.12: A Training Curve and the Corresponding Weight Locus of W2(2) Versus W1(2): The Case of the Weight Searching Direction Changing Abruptly.
Figure 4.13: A Training Curve and the Corresponding Weight Locus of $W_2(2)$ Versus $W_1(2)$: The Case of a Relatively 'Straight' Locus but With a Chain-Like Part.
Figure 4.14: A Training Curve and the Corresponding Weight Locus of $W_2^{(2)}$ Versus $W_1^{(2)}$: The Case of a Twisted Locus at the Beginning Part.
Figure 4.15: A Training Curve and the Corresponding Weight Locus of $W_2^{(2)}$ Versus $W_1^{(2)}$: The Case of the Most Twisted Locus in the Simulations.
Figure 4.16: A Training Curve and the Corresponding Weight Locus of $W_2^{(2)}$ Versus $W_1^{(2)}$.
The Case of a Quite 'Straight' Locus When Using a Weight Perturbation Algorithm.
4.7 Summary

This chapter deals with the training algorithms for the larger neural network (LNN) that has been constructed using the neural modeling procedure described in the last chapter. At the beginning of this chapter we have discussed the usability of the BP-based algorithms, weight perturbation-based algorithms and direct search methods which represent the three important categories of training algorithms.

The desirable training algorithms of interest in this chapter are those that can determine a set of weights for a feedforward neural network without using the analytical expression of the derivative function of activation function. It has been pointed out that the standard BP algorithm cannot be directly used for the training of an LNN, while weight perturbation-based algorithms suffers from convergence and computation problems. The NLBP method introduced in this chapter is a variant of the standard BP algorithm. The other category of training methods that can avoid the necessity for calculating the derivative of activation function is direct search methods and/or random search methods [119]. Theoretically, the genetic algorithms [120, 121] can be employed to conduct training tasks without using the derivative information of activation function. The issues on using genetic algorithms for neural network training will be addressed in next chapter. The training algorithm proposed in this chapter, the approximated series-derivative algorithm, belongs to the category of the BP-based algorithms. Compared to the NLBP method, the proposed method involves much less computation in each iteration of computation, but the NLBP method would take fewer training epochs. The comparison between these two training schemes should be done in our future work.
CHAPTER 5: TRAINING WITH LOW-RESOLUTION DIGITAL WEIGHTS

5.1 Introduction

In a programmable digital weight-based neural network implementation, the weight values can be only employed in discrete sizes. Thus the use of digital weights in a neural network implementation imposes new issues that are not present in simulation environments where most of the well-studied training methods apply particularly to continuous weight values. The digital weights in a neural network implementation are in fact bounded weights with limited resolution, the limited calculation precision and limited dynamic range. Most reported studies on training algorithms of artificial neural networks are carried out based on using floating point representations for the weight values. This provides a large range of values with a high degree of resolution, which are not realizable in a silicon implementation. A VLSI implementation of a neural network will normally not work properly when simply using the digitized version of the continuous weight solution due to the weight quantization effects.

In this chapter, the issues on using digital/integer weights with limited precision and range are first discussed. The lower bounds for limited precision weights networks are also investigated. The currently existing training algorithms for training neural networks with limited precision weights are then reviewed which serve as the context of the research presented in Chapter 5. The concepts of digital weights and weight quantization are then reviewed and formulated. A mathematical analysis of quantization
using a statistical model is presented, which is based on the derivation of [64]. The focus of this chapter deals with a training algorithm for VLSI neural networks with digital weights using the in-the-loop training strategy. It should be pointed out here that this algorithm is not an alternative one to the training algorithm that we proposed in last chapter, but it deals with the training problems when using digital weights.

5.2 On Using Digital Weights with Limited Precision and Range

The cost of the hardware implementation is critically sensitive to factors like the precision used for weights, the total number of bits and the maximum fan-in used in the network. Using weights with limited precision and range opens the road for efficient VLSI implementations because: i) a limited range for the weights can be translated into reduced storage requirements and ii) integer computation can be implemented in a more efficient way than floating point one [77]. In the papers [77] and [78], some aspects of the computational power of neural networks using integer weights in a very restricted range are analyzed. It shows that if the weights are restricted in a drastic way (both range and precision), the existence of a solution is not to be taken for granted anymore.

An important and interesting question is: how many bits of information do we need for a given problem or, alternatively, how complex a problem can we solve with a network of given complexity? From [129], we have the following existential proposition:

Proposition 1 [129]: Using integer weights in the range \([-p, p]\), one can correctly classify any set of patterns for which the minimum distance between two patterns of opposite classes is \(d_{\text{min}} \geq 1/p\).

Proposition 1 says that for a given minimum distance \(d_{\text{min}}\), there exists a neural network using weights in the given range that solves the problem. However, this result does not
say anything about the complexity or architecture of the solution network nor does it say anything about the training process itself.

The lower bound for the minimum distance between two patterns of opposite classes is modified in [78]. The new results on the capabilities of integer weights neural network in classification problems presented in [78] indicate that a neural network using integer weights in the range [-p, p] (where p is a small integer value) can classify correctly any set of patterns included in a hypercube of unit side length centered around the origin of \( \mathbb{R}^n \), \( n \geq 2 \), for which the minimum Euclidean distance between two patterns of opposite classes is \( d_{mn} \geq \frac{\sqrt{n-1}}{2p} \).

The following proposition 2 answers the question of the complexity of a solution network.

**Proposition 2** [129] [21] [134]: Let us consider a set of \( m \) patterns (in general position) from two classes in the hypersphere of radius \( D \leq 1 \) centered in origin of \( \mathbb{R}^n \). Let us consider \( d_{mn} = 1/p \) the minimum distance between two patterns belonging to different classes. Then, the number of bits necessary for the separation of the patterns (in general positions using the weights in the set \([-p, -p+1, ..., 0, 1, ..., p]\) is lower bounded by

\[
N_h = \lceil m \cdot n \cdot \log(2 \cdot p \cdot D) \rceil
\]

Propositions 1 and 2 offer the theoretical base for designing the network in accordance with the necessities of a given problem. Note that the number of bits calculated by Proposition 2 refers to bits in the informational entropy sense [21] and these bits do not correspond to the bits used to store the weights because the neural codification is less
than optimal. One should consider an implementation factor [130] to modify the formula in Proposition 2. Using the lower bound on the number of bits calculated as above and the number of bits used to store one weight (which is \( \log(2p+1) \)) for integer weights in the range \([-p, p]\), one can calculate a lower bound on the necessary number of weights using \( w \geq \frac{N_h}{\log(2/d_{mn} + 1)} \). The details of proof can be found in [130].

A network thus designed will offer the guarantees that i) there exists a weight state with weights in the given range that solves the problem and ii) the architecture is not undersized for the given problem.

Using the above equations, it can be shown that for the classification problem described in Chapter 2, the smart sensor is capable of classifying the patterns given in Figure 2.8. More specifically, with 5 bits of digital weights, the number of bits necessary for the separation of the patterns is lower bounded by \( N_h = 2 \times 64 \times \log(2 \times 15 \times \sqrt{14}) = 604 \). The first layer of the smart sensor has the number of bits: \( 8 \times 64 \times 5 = 2560 \).

The results presented in [130] also give updated worst-case lower bounds for the number of weights used by the network. The theoretical approach is used to calculate the necessary weight range, a lower bound for the number of bits necessary to solve the problem in the worst.

5.3 Training Algorithms Using Limited Precision Weights

Various issues relating the precision of the weights to various cost parameters are discussed in several hardware implementation review papers [17] [26] [142] [143] [144] and [145]. The efforts for using reduced precision weights can be divided into two major
trends [92]. The first trend was to modify existing algorithms, adapting them to a reduced precision. Successful examples of such algorithms include the continuous-discrete learning method [71] in which the errors obtained from the discrete network through the continuous network are back-propagated. This technique works well with 5-7 bits of weight. Another technique in the same category reduces the number of weights in a sequence of training sessions [146]. This technique approximates the sigmoid with a linear function and can achieve convergence with 4-7 bits of weight in general and as low as 2 bits. Another continuous discrete learning method is presented in [147]. In [147], local minima are avoided using two additional heuristics: a) all values are restricted to a given interval in the early stages of learning and b) the number of training epochs is increased for those training signals that cause errors larger than the average. The second trend is represented by novel techniques oriented towards low precision. Probabilistic rounding algorithms [70, 132] use a minimal weight update. When a proposed update $\Delta w_{ij}$ is smaller than this minimal update, the algorithms use the minimal one with a probability proportional to the magnitude of the proposed update $\Delta w_{ij}$. The second category groups together algorithms using dynamic rescaling of the weights and adapting the gain of activation function. The gain can be adapted using the average value of the incoming connections to a neuron [69] [76] or some form of gradient descent [148] [131]. Another approach to obtaining solutions with reduced weight precision is to treat the quantization error like white noise and use a dynamical quantization step calculated from the error. This particular approach has been used with a cascade-correlation architecture [149]. Model-free algorithms use a fixed architecture but no specific classical learning
paradigm. Examples include the whole approach of weight and node perturbation or stochastic learning [80] [144] [150] [151].

A particular interesting category of theoretical algorithms consists of the algorithms using limited precision integer weights. This type of algorithm is interesting because it is more efficient to implement in hardware integer weights and integer computation circuitry both in terms of chip space (VLSI area) and cost [92]. If the integer numbers are powers of two, multiplications and divisions reduce to shifts, thus becoming even more attractive for hardware implementation. [64], [133], [103] and [131] restrict the weight values to powers of two.

A version of the VLSI-friendly Constraint Based Decomposition (VCBD) training algorithm is presented in [20]. The algorithm is able to produce and then train a network using limited precision integer weights and units with limited fan-in. The algorithm does not require an a priori design of the network architecture and has a guaranteed convergence. An adapted version of the VCBD algorithms that is ready to train using limited precision weights is presented in [135].

A VLSI neural network classifier based on using integer-valued weights is described in [19] which aims at designing a neural network hardware implementation that exploits the characteristics of network built by the VCBD algorithm. The paper [19] presents the design of a CMOS neuron that fully exploits the characteristics of solutions produced by VCBD. The limited integer weight range enables efficient use of current-mode techniques for addition and multiplication operations within the neuron.
5.4 On Digital Weights and Weight Quantization

In this context, we use the terminology “weight quantization” and/or “quantize” to refer to the procedure of rounding off the continuous values to the nearest available sizes so that the floating point weight values can be realized using discrete values. When given a set of continuous weight values, one should determine the dynamic weight range first, and then find the resolution based on the given number of bits in the digital weight registers so as to quantize the continuous representations into quantized weights. More specifically, the procedure that maps a given set of continuous weights into a set of corresponding discretized weights includes the following steps:

(1) Determine the dynamic range according to the minimum and the maximum of the given set of continuous weights as well as the mean magnitude of the continuous weights. Based on the conclusion of [62], the dynamic range for the continuous weights is determined as \( (w_{\text{max}} - w_{\text{min}}) \), if \( (w_{\text{max}} - w_{\text{min}})/w_{\text{mean}} < 7 \). Note that we have used \( w_{\text{min}} \), \( w_{\text{max}} \) and \( w_{\text{mean}} \) to denote the minimum, the maximum and the mean value of the weights, respectively. However, if \( (w_{\text{max}} - w_{\text{min}})/w_{\text{mean}} \geq 7 \), the

![Figure 5.1: (a) A Set of Weights; (b) The Histogram of the Set of Weights Shown in (a).](image)
knowledge of weight distribution (e.g. the histogram of weight values) may be employed, and it may occur that some of weights with pronouncedly large magnitudes are cropped by using a hard limiting mechanism. This procedure is illustrated in Figure 5.1. A set of weights is shown in Figure 5.1(a) and the histogram is given in Figure 5.1(b). One may choose the weight range as [-1,1]. In this case two weights are to be cropped by the dot line shown in the figures. Our experiments indicates that in order not to pronouncedly degrade the performance of network we may be allowed to crop up to 10% of the large weight values if the magnitudes of them are particularly larger than the rest of the weights. In Figure 5.1, for instance, we may choose to crop the two weights with values more than 1 so that the two weight values become 1. Then by using a shifting operation we obtain a range [-\(\phi\), \(\phi\)] that is ready for the weight quantization step.

(2) Calculate the resolution using the obtained dynamic range and the available bit number of digital weight registers. If the weight range is [-\(\phi\), \(\phi\)] and the bit number is \(B_n\), then the resolution \(r_i\) can be determined by

\[
r_i = 2\phi/(2^{B_i} - 1)
\]  \hspace{1cm} (5.1)

For example, for digital weight register with five bits, we get 31 levels.

(3) Determine the quantized weights by rounding off each of the weight values to its nearest available integers. In other words, if we imagine we have \((2^{B_i} - 1)\) bins, then we place each of the continuous weight into the nearest available bin. A quantized weight \(w_{ij}^a\) can be expressed by

\[
w_{ijQ} = \text{round}(w_{ij}/r_i)
\]  \hspace{1cm} (5.2)
Note that $w_{ij}$ and $r_i$ are floating numbers while $w_{(p,q)}$ is an integer and we use "round" to represent the operation of rounding off the nearest integers.

In terms of sum-squared error, the weight quantization effects have been illustrated in Figure 2.7. The phenomenon is especially evident when the digital weight resolution is very low due to some design/fabrication constraints since the shifting between the continuous value and the quantized one could be significantly large. In VLSI implementation of digital weights, adding one more bit to a digital weight register means approximately doubling the die area for the D/A conversion circuitry [38]. As far as the current VLSI technologies are concerned, the die area constraints will inevitably limit the number of bits of digital weight registers. This is one of the main reasons that five-bit digital weight registers have been chosen in the smart sensor that has been described in Chapter 2.

The VLSI implementations of neural networks will restrict the range and resolution of the weight values regardless of the underlying specific approaches. To make a silicon implementation of a neural network function properly with quantized weights from the continuous values, one has to use digital weights with adequate precision and range to achieve some given training criterion [62]. The effects of weight precision and range on the performance of neural network classifiers have been studied by several research groups, e.g. in [60-64]. Although no widely recognized conclusions are available, it is generally believed that the required precision and range depend strongly on the problem difficulty, the network complexity, and the relationship between them [61]. This leads to different conclusions on the required number of bits and range for the digital weights due to the utility of different network structures and classification
tasks. For instance, the authors of reference [62] concluded that the digital weights need a range of four to seven times the mean magnitude of the continuous weights and a resolution of nine to eleven bits in order to obtain the same performance obtained with continuous weights. Whereas in [61], it was believed that a precision of five bits and a range equal to four times the mean weight magnitude produced the same performance as continuous weights. Since these conclusions are contingent on the classification tasks, one is usually not constrained by the precision and range requirements when designing a silicon implementation of neural network for a general purpose. Instead, circuit simplicity and area efficiency may be more dominant in a VLSI circuitry design. Indeed, once a VLSI implementation of a neural network with digital weights of some certain number of bits (in general, the available range of digital weights is determined by the number of bits and the D/A conversion ratio) is designed and fabricated, how to train the network to achieve an optimal performance (minimizing the error function) is an important aspect of digital weight precision and range problem. Note that to achieve an optimal performance does not mean to achieve some given training criterion, which might neither be available beforehand nor realizable in practice due to the limited resolution of digital weights.

5.5 A Mathematical Analysis of Weight Quantization

Weight imprecision due to weight quantization effects and component variation have sometimes been modeled as noisy weight changes [137] [139] using statistical methods. In this section, we apply a statistical model to weight quantization procedure. The following derivation is based on the work in [64] and [122]. Note that some
corrections for the derivation in [64] were given in [122], which we have employed in the following derivation.

Given a two-layer feedforward neural network as shown in Figure 4.2, we assume that all the inputs to the network are continuously valued and uniformly distributed between zero and one, then using the same notation as in Figure 4.1, we have

$$h_i^{(1)} = \sum_{k=1}^{K_i} w_{ik}^{(1)} \xi_k$$  \hspace{1cm} (5.3)

where $K_i$ represents the number of inputs. This equation is a linear transformation where the $w_{ik}^{(1)}$ are the elements of a matrix of transforming the vector $\xi_k$ into the vector $h_i^{(1)}$.

Here if we use $f$ to represent the activation function, then

$$V_i = f(h_i^{(1)})$$  \hspace{1cm} (5.4)

In this section, we assume the sigmoidal function $f(u) = \frac{1}{1 + e^{-u}}$ in the following derivation. The above equations (5.3) and (5.4) hold for an ideal network. When there is quantization, however, the following modifications have to be made

$$h_i^{(1)} = \sum_{k=1}^{K_i} (w_{ik}^{(1)} + \Delta w_{ik}^{(1)}) \cdot \xi_k = \sum_{k=1}^{K_i} w_{ik}^{(1)} \cdot \xi_k + \Delta h_i^{(1)}$$  \hspace{1cm} (5.5)

where $\Delta w_{ik}^{(1)}$ is the quantization error (noise) of the weights and

$$\Delta h_i^{(1)} = \sum_{k=1}^{K_i} \Delta w_{ik}^{(1)} \cdot \xi_k$$  \hspace{1cm} (5.6)

If the weights are quantized with a reasonably large number of bits and $\Delta_0$ is the quantization level of the least significant bit, then it is reasonable to make the standard approximation of a rectangular probability distribution function for the quantization error.

Thus the quantization error has a mean of zero and a variance
\[ \sigma^2_{\alpha_0} = \frac{\Delta_0^2}{12} \]  

(5.7)

As in [64], if assuming that inputs are uniformly distributed between zero and one, then the expectation of the inputs is 1/2. Hence, the variance of the inputs can be easily found by

\[ \sigma^2_{v_n} = \frac{1}{\sigma} \int (x - 1/2) \cdot p(x) \cdot dx = \frac{1}{\sigma} \int (x^2 - x + 1/4) \cdot dx = \frac{1}{12} \]  

(5.8)

If we further assume that the weights are uniformly distributed between \( W_{\text{max}} \) and \(-W_{\text{max}}\), then their expectation is zero and their variance is

\[ \sigma^2_{w_{\text{in}}} = \frac{W_{\text{max}}^2}{3} = \frac{(\Delta_0 \cdot 2^N)^2}{12} \]  

(5.9)

since \( W_{\text{max}} = \Delta_0 2^{N-1} \). Note that we have used \( N \) to denote the number of bits.

Since the products of the weight and the input are independent, the variance of \( h_i^{(l)} \) can be found by

\[ \sigma^2_{h_i^{(l)}} = \operatorname{var} \left[ \sum_{k=1}^{K_i} w_{ik}^{(l)} \cdot \xi_k \right] = \sum_{k=1}^{K_i} \operatorname{var}[w_{ik}^{(l)} \cdot \xi_k] \]  

(5.10)

Notice that \( w_{ik}^{(l)} \) and \( \xi_k \) are independent, then we have

\[ \operatorname{var}[w_{ik}^{(l)} \cdot \xi_k] = \mathbb{E}[w_{ik}^{(l)} \cdot \xi_k]^2 - \mathbb{E}^2[w_{ik}^{(l)} \cdot \xi_k] = \mathbb{E}[w_{ik}^{(l)}]^2 \cdot \mathbb{E}^2[\xi_k] = \operatorname{var}[w_{ik}^{(l)}] \cdot \mathbb{E}^2[\xi_k] \]  

(5.11)

Using the results of (5.8) and (5.9), we obtain

\[ \operatorname{var}[w_{ik}^{(l)} \cdot \xi_k] = \frac{(\Delta_0 \cdot 2^N)^2}{12} \cdot \left( \frac{1}{12} + \frac{1}{4} \right) = \frac{(\Delta_0 \cdot 2^N)^2}{36} \]  

(5.12)

Therefore, the variance of \( h_i^{(l)} \) is
\[
\sigma_{h_{[i]}}^2 = \frac{(A_0 \cdot 2^N)^2}{36} \cdot K_1
\]  

(5.13)

The variance of \( \Delta h_{[i]}^{(1)} \) can also be found as

\[
\sigma_{\Delta h_{[i]}}^2 = \frac{K_1 \cdot \Delta_0^2}{36}
\]

(5.14)

If we let \( A = \sigma_{h_{[i]}}^2 \) for simplicity of notation, then \( h_{[i]}^{(1)} \) will be in the interval \((-A, A)\) for over two-thirds of the values. Now that we know the standard deviation of the signal and the noise and their means just before the neuron, we can estimate the same parameters after neuron. We know that

\[
P_{V_i} = p_{h_{[i]}^{(1)}} \cdot \frac{dh}{dV}
\]

(5.15)

where \( p_{V_i} \) and \( p_{h_{[i]}^{(1)}} \) are the probability distributions of variables \( V \) and \( h \). The derivative of (5.15) can be found by

\[
V = \frac{1}{1 + e^{-h}} \Rightarrow h = \ln\left(\frac{1}{1 - V}\right) \Rightarrow \frac{dh}{dV} = \frac{1}{V(1 - V)}
\]

(5.16)

Hence, approximating that \( h_{[i]}^{(1)} \) is uniformly distributed over the interval \((-A, A)\), we have

\[
P_{V_i} = \frac{1}{2A} \cdot \frac{1}{V(1 - V)}
\]

(5.17)

Then the mean for \( V \) is found by

\[
E[V_i] = \int_{-A}^{A} (V \cdot p_{V_i}) dV = \int_{-f(A)}^{f(A)} \left(\frac{V}{V(1 - V)} \cdot \frac{1}{2A}\right) dV = \frac{1}{2}
\]

(5.18)

Using the same approach gives

\[
\sigma_{V_i}^2 = \frac{1}{2A} \int_{-f(A)}^{f(A)} \left(\left(V - \frac{1}{2}\right)^2 \cdot \frac{1}{V(1 - V)}\right) dV = \frac{1}{2A} \left(\frac{A}{2} - \tanh\left(\frac{A}{2}\right)\right)
\]

(5.19)
One can find the signal at the input of the second layer of neurons using the procedure outlined above

\[
\sigma_{h_i^{(2)}}^2 = \text{var} \left[ \sum_{m=1}^{K_2} w_{im}^{(2)} h_i^{(2)} \right] = \sum_{m=1}^{K_2} \text{var} \left[ w_{im}^{(2)} h_i^{(2)} \right] = \sum_{m=1}^{K_2} \text{var} \left[ w_{im}^{(2)} \right] \cdot \left( \text{var} [h_i^{(2)}] + E^2 [h_i^{(2)}] \right)\]

(5.20)

Plugging the results of (5.18) and (5.19) in (5.20) and assuming that the weights in the first layer have the same statistical properties as in the second layer, we obtain

\[
\sigma_{h_i^{(1)}}^2 = K_2 \cdot \left( \frac{\Delta_0 \cdot 2^N}{12} \right)^2 \left( \frac{1}{2A} \frac{A}{2} - \tanh \left( \frac{A}{2} \right) + \frac{1}{4} \right)
\]

(5.21)

\[
= \frac{(\Delta_0 \cdot 2^N)^2}{12} \cdot \frac{K_2}{2A} \left[ A - \tanh \left( \frac{A}{2} \right) \right]
\]

The above derivation can be regarded as procedures of deriving statistical properties for all the variables in a network. It's worth pointing out that from the equations (5.13) and (5.21) we can find the relations between the variance of inputs to neurons in the first and second layer and the quantization resolution (\(\Delta_0\)), the number bits (N) and the number of neurons (\(K_1\) and \(K_2\)), respectively. And the variances increase linearly with number of neurons in the layer of interest, quadratically with the quantization level and exponentially with the number of bits employed in weight quantization. This is why the outputs of a feedforward multilayer neural network could be very sensitive to the weight quantization manipulations. For instance, in the smart sensor we employ digital weights with N=5, \(\Delta_0=0.1\) and \(K_1=16\), the variance is 

\[
\sigma_{h_i^{(1)}}^2 = \frac{\left( 0.1 \times 2^5 \right)^2}{36} \cdot 16 \approx 4.55,
\]

and the standard deviation is \(\sigma_{h_i^{(1)}} = 2.13\). Comparing the ranges of inputs (0~1) and weights (-1.5~1.5), the standard deviation has been very large.
Therefore, when using quantized weights in a network, what we will be concerned is no longer the problem of degradation in the network, but the problem of whether the entire network works or not.

5.6 The Training Strategies for Dealing with Digital Weights

The VCBD algorithm introduced earlier in this Chapter basically establishes the base for constructing neural networks when given some certain classification problems. The training using VCBD can be done when a network is constructed. For training an implementation of neural network like the smart sensor, that has been given or fabricated, a training algorithm shall be able to find a set of quantized weights that can minimize the output errors anyway regardless of the capability of the given network or complexity of the given classification problem. In this regard, there are basically three training strategies to obtain a set of proper quantized weight values for a network with digital weights:

(1) First train the network with continuous weights, and then quantize (or discretize) the resultant continuous weights into the digitized version of the solution;

(2) Train the network directly using quantized weights in each iteration of weight updating computation;

(3) Using the genetic algorithm [120] to conduct a direct random search for a set of digital weights.

The learning method introduced in [103] belongs to the training strategy (1). In this method, it starts with continuous initial weights using the standard BP method. Then a weight normalization procedure is applied to ensure that the whole shifting dynamics is used and to maximize the match between continuous and discrete weights. Our
experiment indicates that this method can hardly work well when the number of bits of digital weights is low (say, 5 bits) since the shifting between the quantized weights and continuous weights could be significantly large with respect to weight dynamic range. As a result, the resulting continuous weights, even yielding a very low error function (e.g. MSE criterion) value, could produce a considerably large error when using the quantized weights.

The second training strategy can be found in [63]. Since the quantization of the weights introduces many spurious local minima, not corresponding to any meaningful solution, a modification was made in [63] to ensure the training to avoid remaining trapped in these local minima. The modification included using adapted learning rates [107] and resetting the learning rate to its upper limit when the learning rate becomes very small on a series of successive trails while failing to reduce the total error. Again this method cannot guarantee the convergence to a meaningful solution, especially when the number of bits associated with digital weights is low.

The third method, the genetic algorithm, belongs to random search method in optimization [120]. This method is feasible only for small size neural networks with low number of bits digital weights. The reason for this is that when the network size and/or digital weight length are large, the variables involved in computation would be so many that it will result in a prohibitively time-consuming training process. Therefore, this method is still limited to the training of a small sized neural network.

It can be seen that the above-mentioned three training strategies are not appropriate for the training of a neural network implementation with digital weights of low resolution, although they are effective for the situations as described in their
references. In the following, we will describe a new method for obtaining quantized weights with low resolution that will be the optimal set of weights for a given network and a training task.

5.7 A New Algorithm for Obtaining Optimal Quantized Weights

The idea of this algorithm and some experimental results have been presented in our work [123]. In the following, more details about this algorithm are given.

5.7.1 The Introduction of the Algorithm

Before we proceed to the formulation of our algorithm, we need to clarify some points on the existence of the optimal quantized weight solution. In this context, the term weight solution or solution is referred to as a set of weight values which can minimize the cost function. They may be considered as a vector or a matrix. For a given neural network architecture, a set of training patterns and a given training criterion, the solutions (sets of weights) are not unique due to the fact that a neural network architecture always includes the nature of redundancy. As shown in Figure 2.7, different sets of weights have different quantization effects on the performance of a network for a given classification task. Moreover, a continuous solution of weight values with low error outputs doesn't necessarily yield low error outputs when being quantized into its corresponding discrete solution. Therefore, there exists a subspace $\Phi$ in the weight space $SP_w$, i.e., $\Phi \subseteq SP_w$. And the space $SP_w$ is composed of all the continuous weight vectors that can yield error outputs below some certain preset criterion. A weight vector in the subspace $\Phi$ will yield a low error output (minimal by some criterion) when it is quantized into the discrete version of the continuous weight solution. Our objective here is to find one (or some)
weight vector(s) that can result in a minimal output error when being quantized into discrete representations for the weights.

We could have two alternative classes of approaches as described in the following to obtaining an optimal set of quantized weights that can result in the minimal error output. They are: (a) Finding a set of continuous weights that can best match the corresponding quantized version of weights while the continuous weights have been found to be optimal; and (b) Finding a set of continuous weights such that the error function is insensitive to small variations of the weight values. The size of these small variations should be around the magnitude of weight shifting due to the weight quantization procedure. The fundamental difficulty with the first class is that it is usually impractical to check all possible solutions for suitability for reasons of computational cost [119]. In the meanwhile, the class of approaches will heavily rely on the outcome of solution checks. The approach that we will describe in this section can be categorized into the second class of the approaches, where the method for realizing the idea is to estimate the sensitivity of all the synaptic weights to the variations of the quantization sizes. Then the weight values with high sensitivity will be perturbed using some rules and a retraining procedure is followed beginning with the newly obtained weights. The retraining procedure is repeated until an optimal quantized weight solution is found. We call the method weight perturbation based on quantization sensitivity.

In the following subsection, we will give the definition for weight sensitivity used in this work and address some related issues before we proceed any further.

5.7.2 The Weight Sensitivities to Quantization

We first define the error function change $\Delta E_{ij}$ as:
\[ \Delta E_{ij} = E(w_{ij} = w_{(ij)Q}) - E(w_{ij} = w_{(ij)\text{final}}) \] (5.22)

where \( E \) represents the error function, \( w_{(ij)\text{final}} \) denotes the final value of the weight upon the completion of a regular training phase using continuous representations, and \( w_{(ij)Q} \) denotes the quantized weight of \( w_{(ij)\text{final}} \). Note that if considering the synaptic weights as a matrix of \( W = \{w_{ij}\} \), then \( \Delta E \) can be also considered as a matrix of \( \Delta E = \{\Delta E_{ij}\} \). The subscriptions \( i \) and \( j \) represent an individual weight from node \( i \) to \( j \). Then the sensitivity \( S_{ij} \) for the weight \( w_{ij} \) is defined as:

\[
S_{ij} = \begin{cases} \\
\frac{\Delta E_{ij}}{|w_{(ij)Q} - w_{(ij)\text{final}}|} & \text{if } |w_{(ij)Q} - w_{(ij)\text{final}}| > \varepsilon \\
\frac{\Delta E_{ij}}{\varepsilon} & \text{if } |w_{(ij)Q} - w_{(ij)\text{final}}| \leq \varepsilon \end{cases}
\] (5.23)

where \( \varepsilon \) is a small real number (we choose \( \varepsilon = 0.01 \) in this work). Using (5.23) we can obtain a sensitivity matrix or a "shadow array" [124] of \( \{S_{ij}\} \), in which each element is corresponding to a synaptic weight connection. The sensitivity matrix keeps track of the sensitivities of all the synaptic weights in a network.

Note that the definition of (5.23) is different from that defined in [125], where the sensitivity is estimated using

\[
\hat{S}_{ij} = -\frac{\partial E}{\partial w_{ij}} \bigg|_{w_{ij}} \]

(5.24)

The problem with this estimate, as pointed out in [125], is that the partial derivative tends to zero when error decreases. While this is a desirable property of the gradient descent learning method, it results in poor sensitivity estimation. In our present derivation, the purpose of weight sensitivity estimation is to select the connections that are particularly
sensitive to the weight quantization shifting. Thus the synaptic weights with small
sensitivity values are basically not of interest.

Upon the completion of one of the training phases using continuous
representations for the weights, some synaptic weights may happen to be in positions of
the available sizes, or within small vicinities of the available sizes. In this case, the
denominator of $\Delta E_{ij}/(w_{(ij)Q} - w_{(ij)\text{final}})$ will tend to be zero. This problem is
circumvented by introducing a small real number $\epsilon$ to (5.23) where the denominator is
made to be not less than $\epsilon$. In this case, the calculated sensitivities of the weights by
using (5.23) may turn out to be smaller than their actual values. This problem can be
overcome by properly choosing the real number $\epsilon$ (less than 0.01). In most cases, the
weight sensitivity estimation doesn’t need a very high accuracy. Moreover even if the
sensitivities of some weights are wrongly estimated, it hardly matters since the retraining
procedure (which will be elucidated in the next subsection) will alter the distribution of
the weights and all the weight sensitivities will be realigned and recalculated. From this
viewpoint, our method is quite robust to the problems related to accuracy.

5.7.3 Perturbing the Weights of High Sensitivity

We then rank the synaptic weights in a list in terms of the magnitudes of their
sensitivity values, which are calculated using (5.22) and (5.23). A synaptic weight with a
higher sensitivity value contributes to increasing the total error more than a weight with
lower sensitivity does when the weight quantization procedure is applied. In fact, the
difference of cost function between two points (Q and M, which denotes quantization and
minimization) in weight space (Euclidean space) can be found by the line integral of the
partial derivative of cost function with respect to the weight of interest, as shown in
equation (5.25). This integral is related to the weight sensitivity defined in equation (5.22).

\[ E(w_{ij}^Q) - E(w_{ij}^M) = \int_0^M \frac{\partial E}{\partial w_{ij}} dw_{ij} \approx \int_0^M \frac{\Delta E}{\Delta w_{ij}} dw_{ij} = \int_0^M S_{ij} dw_{ij} \]  

(5.25)

Therefore, if perturbing the weights of high sensitivity while keeping other weights intact, one can eventually come up with an optimal set of weights by training the perturbed set of weights repeatedly. After the weight perturbation is done, we then use the newly obtained set of weights as the initial weights to perform the training procedure once more for the neural network. We also refer to this training procedure as retraining procedure. Note that all the training/retraining procedures are carried out using continuous representations for the weights; the quantized weights are only used when checking the output errors resulting from the weight quantization. Since in this way the weights more sensitive to weight quantization are particularly readjusted, the retraining procedure using the newly obtained weights as initial weights is more likely to result in another solution which is more robust to the effects of weight quantization.

The next problem is how to define a specific weight as a 'sensitive' or an 'insensitive' one. The extreme case here is to perturb all the weights. If the perturbation is performed in such a way that all the weights deviate from the original values significantly, then the following retraining process is tantamount to starting from another set of initial weights. In this case, it does not make sense since the training/retraining process is not making progress in a gradual manner. Conversely, if only a small percentage of the weights are perturbed, the next retraining process will most probably result in the identical set of weights as those in the previous time. The experiments show
that when perturbing 30% of the most sensitive weights, a retraining process would make progress on the top of the progress made in the previous trail.

It is noted that all the weights obtained from a training process are optimal in terms of the given training criterion. Thus if changing the weights with small perturbation magnitudes (comparable to the size of quantization resolution), it is highly probable that we obtain a set of weights that deviate only a little from the original set of weights and the weight search still cannot escape from a local minimum point. The result is that the retraining process recovers the weights before perturbation. Thus the condition for the perturbation size is

$$|\delta(w_i)| > r_i$$

(5.26)

As pointed out earlier, weight quantization procedure generates many local minima, much more than the local minima when using continuous weight. And the purpose of the weight perturbation here, after all, is to force the weight search to get out of these local minima by introducing noise [107] into the training process. To add noise to the process while considering the lower bound in (5.26), we define

$$\delta(w_i) = [\text{sign}(1 - R_1)](1 + \lambda R_2) r_i$$

(5.27)

where $R_1$ and $R_2$ are both random numbers with a range of [0,1], and $\lambda$ is a scale factor that controls the perturbation depth. We choose $\lambda=3$ in the following simulation. Note that the term $\text{sign}(1-R_1)$ determines the sign for the perturbation and the probabilities of generating the plus and minus signs are both 0.5.

5.7.4 The Training Algorithm and Its Flow Chart

The training algorithm is described as follows:
(a) Determine the weight range and weight resolution using the formula (5.1) and (5.2) described in the first section. For example, for digital weights with a resolution of five bits, we may choose the continuous weight range to be \([-1.5, +1.5]\) with a resolution 0.1, which corresponds to the 31 levels in a digital register.

(b) Set the iteration index \(k=1\). Initialize the complete neural network (which has been embedded with the subnets) with random continuous weights \(W^k_{\text{initial}}\). The maximum number of trials for iteration \(k_{\text{max}}\) is to be set (\(k_{\text{max}} = 50\) in this work). The minimum number of the output error is also needed to be defined, and we use a big number, i.e., \(E^{k}_{\text{min}} = 10^5\).

(c) Train/Retrain the network using a variant of the standard BP algorithm described in Chapter 3, until either a preset error function criterion is achieved or the maximum allowed number of training epochs goes beyond a preset maximum number \(N_\alpha\). Here we also employ the idea presented in [63] that is to reset the learning rate when it becomes a very low value due to a series of successive failure of the trails for reducing the total error. We obtain a weight matrix \(W^k_{\text{final}}\) upon the completion of the training/retraining. More specifically,

\[
W^k_{\text{final}} = f(W^k_{\text{initial}}, A_F, \xi, O)
\]  

(5.28)

where \(A_F\) represents the activation function. Note that we have used the terms training and retraining interchangeably. While the two terms are referred to the identical concept, we only use the term retraining to distinguish from the conventional concept since the training process is usually repeated a number of times in this algorithm.
(d) Conduct the weight quantization calculation using the formula (5.1) and (5.2). As described earlier, this procedure may include the process of cropping some of the weight values of large magnitudes. We express this step by

\[
\begin{align*}
\Delta W^k &= f(W_{\text{final}}^k, B_n) \\
W_Q^k &= W_{\text{final}}^k + \Delta W^k
\end{align*}
\]

(5.29)

(e) Evaluate the error output \( E_Q^k \) using the obtained quantized weights \( W_Q^k \). Similar to (5.28), this can be expressed by

\[
E(W_Q^k) = E(W_Q^k, A_F, \xi, O)
\]

(5.30)

(f) Compare the current error function output \( E(W_Q^k) \) with the minimum error output obtained so far \( E_{\text{MIN}}^k \). If the former is less than the latter \( E(W_Q^k) < E_{\text{MIN}}^k \), continue to the next step. If the former is not less than the latter \( E(W_Q^k) \geq E_{\text{MIN}}^k \), which means the current trial has failed), then go to the step (h).

(g) Since we have found that \( E(W_Q^k) < E_{\text{MIN}}^k \), we are ready to set or update the minimum output error and the set of weights associated with the error. We express this by

\[
\begin{align*}
W_{\text{MIN}}^k &= W_Q^k \\
E_{\text{MIN}}^k &= E(W_Q^k)
\end{align*}
\]

(5.31)

(h) Calculate the weight sensitivity for each synaptic weight to the operation of weight quantization using the equations (5.22) and (5.23), or write them in simplified form as

\[
S = \left| \frac{E(W_{\text{MIN}}^k) - E(W_{\text{final}}^k)}{W_{\text{MIN}}^k - W_{\text{final}}^k} \right|
\]

(5.32)
(i) Perturb the weights with high sensitivity values. We choose to perturb the top 30% of the most sensitive weights. If we have determined the sensitivity threshold $s_T$ for the perturbation, then the perturbation procedure is expressed by equations (5.33) and (5.34):

$$
\delta(w^k_{ij}) = f(s_{ij}, \hat{R}) = \begin{cases} 
\left[\text{sign}(1 - R_i)\right](1 + \lambda R_j) & \text{if } s_{ij} \geq s_T \\
0 & \text{if } s_{ij} < s_T 
\end{cases} \quad (5.33)
$$

where $\hat{R} = (R_1, R_2, \lambda)$, and

$$
\begin{align*}
\delta W^k &= \left\{ \delta(w^k_{ij}) \right\} \\
W^{k}_{\text{final}} &= W^{k}_{\text{MIN}} + \delta W^k
\end{align*} \quad (5.34)
$$

Note that the schemes of choosing the weights to be perturbed and perturbing the weights are not limited to what have been given here. From our point of view, the schemes here are the most intuitive and most straightforward ways. The other schemes are subject to further studies.

(j) Now the index $k$ is incremented by 1, i.e., $k \leftarrow k + 1$, and check if total number of iterations has gone beyond the preset maximum bound $k_{\text{max}}$. If $k \leq k_{\text{max}}$, go back to step (c); and if $k > k_{\text{max}}$, then stop the whole process where an optimal set of weights has been found.

A flow-chart of the entire algorithm is shown in Figure 5.2.
Figure 5.2: The Flow-Chart of the Proposed Algorithm for Dealing With Digital Weights.
5.8 Experimental Results

In the first experiment, once again, we use the neural network-based intelligent sensor, which we have called the smart sensor, as the prototype of neural network hardware to test our algorithm. The network was chosen as \( 64,8,4 \) which means the network includes 64 input nodes, 8 hidden neurons and 4 output units. The training patterns, which have been shown in Figure 2.8, are chosen to be those employed in [10]. The sum-squared error (SSE) goal is chosen as 0.02 for the training when using continuous weights. Note that no error goal can be given for digitized version of weights since the optimally achievable error goal can not be known prior to the training in this case. The other parameters (cf. Chapter 3) are chosen as follows: \( \eta(0) = 0.9, \varepsilon_1 = 0.3, \varepsilon_2 \)

![Graph showing a training/retraining curve](image)

*Figure 5.3: A Training/Retraining Curve of the Proposed Algorithm Which Includes 20 Times of Training/Retraining Procedures.*
= 0.04 and $\chi = 1.04$. We have tested our algorithm a number of times using the identical patterns. Learning rate reset occurs when consecutive 20 trials fail to reduce the total error.

![A Close-Up View of the Training Curve](image)

Figure 5.4: A Close-Up View of the Training/Retraining Curve Shown in Figure 5.3.

One of the obtained training curves is given in Figure 5.3 that includes 20 training/retraining procedures described in last section. For scrutinizing an individual training/retraining procedure, some close-up views of the general training curve may be needed. A close-up view for the training curve shown in Figure 5.3 is given in Figure 5.4. At each conjunction point of the training curve, one can find a sharp jumping which reflects the effect of the weight perturbation associated with each retraining procedure. Note that for continuous weights the total error (SSE) can be reduced to below the SSE error goal quickly in each retraining. What makes difference is that when these continuous weights are quantized into discretized weights the corresponding SSE values
could be quite different from each other. This result can be observed in Figure 5.5 where the SSE error curves are illustrated with respect to the above-mentioned 20 training/retraining procedures. In Figure 5.5, the dashed line represents the real errors during the training, while the solid line represents the minimal errors that the algorithm has found up to the current retraining procedure.

![Graph showing SSE for quantized weights](image)

Figure 5.5: The Sum-Squared Errors for Quantized Weights With Respect to the 20 Training/Retraining Procedures.

It should be remarked here that upon conducting the same training task using different initial random weights the obtained sets of quantized weights are usually not the same but they can always yield an approximately identical minimal error. We have used 10 different sets of initial random weights to train the same neural network with the same training task. The minimal errors that the 10 sets of obtained quantized weights yield are approximately the same (=1.18) although the 10 training curves are totally different. To be specific, two of the said training curves are shown in Figure 5.6.
5.9 Comparison with Simulated Annealing Method

We have described a new algorithm for training a VLSI neural network hardware with limited digital weights. Our experimental results indicate that the algorithm is feasible and particularly applicable to the training of the neural network implementation with digital weights of low resolution. More importantly, this method can be used to find out a possibly optimal (not in a strict sense) error goal, which is contingent on the classification task as well as the network hardware structure. It should be noted that our method belongs to trial and error strategy, since the neural network we discuss in this chapter is an interconnected distributed nonlinear system imposed with quantized values. Thus it is too hard to establish a mathematical formulation to solve the training problem which is a rather complex optimization problem. A good side-effect associated with our method is that the weight perturbation based on weight sensitivity is helpful in getting the training out of local minima traps. Thus this method can also be used in a training

![Figure 5.6. The Two Curves Obtained by Conducting the Same Training Task Using Different Initial Random Weights.](image)
process where the convergence speed is very slow. From this perspective, our proposed method takes some resemblance with simulated annealing method [126] which was basically established on the idea that by adjusting the temperature the optimization search is able to do ‘hill climbing’ so that it can get out of local traps. We will discuss the relationship between our method and simulated annealing method in the following.

Simulated annealing is based on the methods of statistical mechanics used for analyzing the behavior of atoms in solids and liquids. The method centers on the notion that there should always be some probability of a change of state that increases the energy of a physical system. It should be pointed out that normally a physical system will move toward lower energy states. This probability is chosen so that states are visited in accordance with their Boltzmann probabilities [127], and is dependent not only on the change in energy involved but also on the physical temperature of the system.

Specifically, instead of using a deterministic threshold, neurons in the Boltzmann machine [127] have a probabilistic decision rule such that neuron $k$ has state $s_k = 1$ with probability

$$p_k = \frac{1}{1 + e^{-\Delta E_k/T}}$$

(5.35)

where $T$ is a parameter that acts like temperature in a physical system and $\Delta E_k$ is the energy gap or difference between a state with unit $k$ off and unit $k$ on. The output of the neuron is always either 0 or 1 but its probability distribution is sigmoidal so, on the average, its output looks like the sigmoidal function. Note that as $T$ approaches 0, this distribution reduces to a step function. This rule allows the system to jump occasionally to a higher energy configuration and therefore to escape from local minima. The
machine gets its name from the mathematical properties in thermodynamics set forth by Boltzmann and useful in optimization.

The Boltzmann machine uses simulated annealing to reach a global energy minimum since the relative probability of two global states $\alpha$ and $\beta$ follows the Boltzmann distribution:

$$
\frac{P_\alpha}{P_\beta} = e^{-\frac{(E_\alpha - E_\beta)}{T}}
$$

(5.36)

and thus the lowest energy state is most probable at any temperature. Since, at low temperature, the time to thermal equilibrium is long, it is advisable to anneal by starting at a high temperature and then gradually reduce it [128].

This is completely analogous to the physical process of annealing damage in a crystal energy state within the crystal lattice. As the temperature is reduced the atoms lock into their proper places. The computation of such annealing is tedious for two reasons. First, the calculation involves imposing probability distributions and physical laws on the motion of particles. Second, the calculation is serial. The physical crystal’s atoms naturally obey physical laws without calculation and they all obey these laws in parallel. For the same reasons, Boltzmann machine simulations on digital computers are also tedious.

In order to make the process of training a neural network analogous to the process of annealing a physical system, we may divide a set of weights of the network into a number of vectors. The dimension of these vectors can be assigned to be anything, depending on dimension of the space in which we want to visualize. Choosing the number of dimension is trivial but two- or three-dimensional space, no doubt, offers better visualization capabilities. For instance, in the smart sensor, the number of the total
weights is $64 \times 8 + 8 \times 4 = 544$, we can choose to divide the weights into 272 vectors with dimension of two. Alternatively, we can also divide the weights into 182 vectors with dimension of three, with the last vector having two dummy elements. We use this three-dimension scheme in the following discussion. The above two schemes of dividing vectors are correspondent to two- and three-dimensional visualization. Note that the vectors can be configured with any combinations of the weight values. Now we can imagine that each vector corresponds to a particle in a physical system. The training process then can be imagined as moving these particles in the three-dimensional space while the energy of the physical system is being minimized with respect to the configuration of these particles.

The weight quantization procedure can be imagined as forcing all the particles to the cross points of a lattice in the 3-D space. Normally, in the standard BP algorithm the learning rate would become very small when the training is approaching to its final stage. Consequently, the steps of weight updating are also very small. They are so small that the particles cannot escape from its nearest cross points. This explains why a regular BP algorithm or any other floating points-based training algorithms are usually not suitable for the training with quantized weights.

In our proposed method, we first train the given network with BP algorithm. Once the energy (output of cost function) is reduced to a level less than a preset error goal, the obtained set of weights will be quantized and the energy function output is calculated. This is equivalent to assigning each particle a cross point in the 3-D lattice and measuring the output of energy function. The next thing that we do is to measure the sensitivity of each weight, or the position of each particle to the energy function.
Evidently, the energy function is more dependent on the particles with high sensitivity. In other words, to make the physical system to be in its lower energy state, one needs to move these particles to other places, the places where they are supposed to be. This is the reason that we have perturbed the weights of high sensitivity in our algorithm. After the perturbation procedure, the positions of the particles with high sensitivity are most probably changed. This procedure might increase the output of energy function of the physical system. However, another round of training using continuous weights can bring the system to a new structure in which the energy function is robust to the procedure of forcing the particles to their nearest cross points. It should be pointed out here that in the new structure some of the particles which used to be sensitive to the energy function may now become insensitive, while some which were insensitive may now become sensitive. These procedures are repeatedly conducted until a structure that is robust to the quantization procedure is obtained. The entire process is analogous to the process of repeatedly “heating” and “annealing” in metallurgical technology.

5.10 Summary

The use of programmable digital weights with low resolution brings about the problems in training that conventional training algorithms fail to converge due to the weight quantization effects. The weight quantization procedure is described at the beginning of this chapter. Then a mathematical analysis on weight quantization effects is given which indicates that the weight quantization effects could make training results totally unusable. Available training methods that deal with weight quantization effects are investigated. Normally, these training methods are only applicable to the neural network implementations of high weight resolution but cannot be used for the training
when low-resolution digital weights are employed in a VLSI implementation of neural network. A novel training algorithm that deals with the problems of the utility of low-resolution weights has been developed in this chapter. In this algorithm, the training is first performed using continuous weights until some training criterion is achieved, then the sensitivity of each weight to the quantization procedure is computed. Based on the sensitivity values, some part of the weights with high sensitivity are to be perturbed. Then the training is conducted again using this newly obtained set of weights. These procedures are repeated until a set of weights that are robust to the weight quantization procedure are obtained. A flow chart of the algorithm has been given which is followed some representative test results. In the last part of this chapter, the nature of the proposed algorithm has been analyzed by comparing to the simulated annealing method.
CHAPTER 6: SUMMARY AND CONCLUSIONS

6.1 Summary

The thesis presents a procedure for the in-the-loop training of VLSI implementations of neural networks with low resolution programmable digital weights.

In chapter three a method was developed to vary the network inputs and synaptic weights in a manner that allows one to obtain input-output data for a given neuron in the network. For a network with input nodes, one hidden layer of neurons, and output nodes each neuron can be readily modelled. One weight connected to the neuron input and one weight connected from the neuron output are set equal to zero. This creates a series signal path that allows input-output data to be obtained that characterizes the nonlinear activation function of the neuron being modelled. For a network with two hidden layers of neurons a series signal path from an input node to a neuron in the first layer to a neuron in the second layer to an output node is selected. The input node value and the first synaptic weight value are set so as to saturate the first neuron at its maximum output value. The synaptic weight connected to the input of the second neuron is then varied so as to sweep the input values such that input-output data that characterizes the neuron's activation function can be measured. During this procedure the final synaptic weight in the signal path had been set to unity.

Once the activation function of the second neuron is known the input-output data for the first neuron can be obtained. The input node value to the series signal path is varied in the ramp-like manner while the first synaptic weight and the second synaptic
weight in the path are set to unity. The resulting output at the output node at the end of the series signal path is then measured. The inverse of the activation function the second neuron is computed and used in conjunction with the input-output measurements of the total signal path to determine the activation function of the first neuron. The input-output information for all neurons in the network can be determined using this method.

The input-output data for each neuron can be used to obtain a model for the neuron based on training a small neural subnetwork with one hidden layer to approximate the activation function defined by the input-output values. The training of the subnetwork can be achieved using the standard backpropagation algorithm.

In the last section of chapter three a complete neural network model of the sensor is developed using both, a priori knowledge of the sensor architecture, together with the subnetwork models that have been obtained for the individual neurons. The resulting network consists of neural subnetworks interconnected by synaptic weights.

In chapter four the problem of how to train a network consisting of neural subnetworks interconnected by synaptic weights is dealt with. Since no explicit analytical function is available for any neuron's nonlinear activation function the standard backpropagation algorithm can not be used as the corresponding expression for the derivative of the activation function is not known. It was found during a large number of simulation trials that the actual gradient search direction in a high dimension optimization space is not as critical for final convergence. As with nonlinear systems in general it is difficult to guarantee convergence or the number of iterations required for convergence, however for sigmoidal-like nonlinearity's suitable gradient search directions can be
computed using certain approximations. The gradient search direction has been computed by modifying the standard backpropagation algorithm to utilize an approximation each neuron based on a weighted summation of sigmoidal functions together with using an analytical expression for the derivative of an ideal sigmoidal function. It was found during extensive simulation trials that satisfactory convergence was reached using these approximations, but on the average the number of iterations required to reach convergence was greater. The training procedure developed in this chapter is based on the assumption that the synaptic weights can be varied in a continuous manner as required to reach convergence. In chapter five the complex issues arising from the effects of weight quantization will be considered.

Chapter five deals with the derivation of an algorithm for the in-the-loop training of smart sensor structures as developed in chapter four when the synaptic weights are realized in terms of low resolution programmable digital weights employing digital multipliers. The network developed in chapter four is trained in-the-loop using continuous weights. Once convergence is obtained the resulting weight values are then changed to their nearest allowable quantized values, one at a time, in order to determine the individual weight's sensitivity with respect to the error function. Once the sensitivity of all the synaptic weights is known, thirty (30) percent of the total weights with the highest sensitivity are perturbed in a manner that generates a new value by taking one half (0.5) of the quantized value and adding it 0.5 of the of the quantized value multiplied by a uniformly distributed random process that generates numbers in range zero to one. The effect of this perturbation procedure is to balance the effect of previous weight training so as to minimize quantization error volatility and yet have a random weight
perturbation component that will prevent repeated convergence to the same local minimum.

The seventy (70) percent existing weights and the thirty (30) percent perturbed weights are then used as the initial weight values for the next iteration of in-the-loop training using continuous weights. This procedure of weight quantization followed by retraining with continuous weight variation is repeated until the desired convergence is reached.

6.2 Conclusions

- A procedure has been derived for the in-the-loop training of VLSI implementations of a class of smart sensors based on neural networks with low resolution programmable digital weights.

- In chapter three, an in-the-loop measurement procedure has been derived that allows one to determine the input-output characteristics for each neuron in a neural network have one or two hidden layers of neurons.

- The measured input-output values of the neurons can be used to obtain a model of the individual neuron based on a small ideal neural subnetwork by using the standard backpropagation algorithm.

- A complete neural network based model of the sensor has been developed that utilizes the known sensor architecture in conjunction with neural subnetworks being used in place of individual sensor neurons, and that can be trained using the algorithm developed in chapter four.
• In chapter four, a variation of the backpropagation algorithm has been derived that allows for the training of a neural network when no explicit analytical expression is available for any neuron activation function.

• In chapter five, an algorithm for training neural networks with low resolution programmable digital weights has been derived that measures the weight sensitivity of each weight with respect to the error function and then perturbs the weights with higher sensitivity in an iterative training/retraining procedure until the desired convergence is reached.

6.3 Future Work

This thesis has developed a theoretical basis for the in-the-loop training of a class of neural network based sensors with low-resolution digital weights. Appropriate future work of a theoretical nature would be derive the additional algorithms necessary so that the training algorithms derived in this these could be extended to the case where high-resolution programmable digital weights have been effectively reduced to low-resolution programmable digital weights due to some circuit fault conditions. In this case, the algorithms developed here could be extended to provide an adaptive fault tolerant strategy, for this type of failure, in a complex massively parallel smart sensor architecture with an on-chip learning capability. Appropriate future work of an experimental nature would be to build a training facility consisting of an in-the-loop fixture to hold an integrated circuit, together with the required computer interface, and then implement the algorithms derived in this thesis as part of an on-line system.
REFERENCES


Joint Conference on Neural Networks IJCNN'99 (electronic format), Washington, DC, 1999.


623-630.


from Custom Analog VLSI ‘Building Block’ Chips”, in Proc. Int. Joint Conf. on Neural 

Neural Network (ETANN) with 10240 ‘Floating Gate’ Synapses”, in Proc. Int. Joint 


Backpropagation Without Derivatives of the Activation Function”, IEEE Transactions on 

[109] B. Widrow and M.A. Lehr, “30 Years of Adaptive Neural Networks: Perceptron, 
Madaline, and Backpropagation”, Proceedings of the IEEE, 78 (9): 1415-1442.


descent method for learning in analog VLSI neural networks,” in Advances in Neural


and P. Gallinari, editors, *Proc. of the International Conference on Artificial Neural 


[144] M. Glessner and W. Pöchmüller, *An Overview of Neural Networks in VLSI*, Chapman and 


limited binary expressions", in *Proc. of Intl. Joint Conference on Neural Networks 
IJCNN'90*, vol. 2, pages 587-592, June 1990.

learning rule and its applications to optical neural networks", *Optical Computing and 

[148] R. Coggins, M. Jabri and Wattle, "A trainable gain analogue VLSI neural network", in 
*Advances in Neural Information Processing Systems NIPS'93*, vol. 6, pages 874-881, 

[149] T. A. Duong and T. Daud, "Cascade error projection: A learning algorithm for hardware 
implementation", in J. Mira and J. V. Sanchez-Andre'es, editors, *Foundations and Tools 
for Neural Modeling, Lecture Notes in Computer Science*, pp. 450-457. *Proc. of 

APPENDIX: MATLAB SOURCE CODE

1: Function approximation using a SNN

(1) File name: xapprl.m

x1 = -5:0.05:5; %y1=logsig(x1);

for i=1:100
    y1(i)=logsig(x1(i));
end
    for i=101:201
    y1(i)=0.5+(1/3.2)*sqrt(x1(i)/3);
end

%x=rand(1,100);
%x=x*10^-5;
% y = logsig(x);
%for i=1:100
% if x(i)<0
%     y(i)=logsig(x(i));
% else
%     y(i)=0.5+(1/3.2)*sqrt(x(i)/3);
% end
% end
plot(x1,y1)
pause
%INPAT=x;
%T=y;
INPAT=x1;
T=y1;
TP(1) = 25;
TP(2) = 2000;
TP(3) = 0.004;
TP(4) = 0.01;

[W1,B1,W2,B2,W3,B3]=initff(INPAT,20,'tansig',5,'tansig',1,'purelin');
figure
[W1, B1, W2, B2, W3, B3, TE, TR]=trainbpa(W1, B1,
'tansig',W2,B2,'tansig',W3,B3,'purelin',INPAT,T,TP);
pause
figure
a=simuff(x1, W1, B1, 'tansig',W2,B2,'tansig',W3,B3,'purelin');
plot(x1,y1,'y-',x1,a,'c-');

172
(2) File name: xfappr2.m

x1 = -5:0.05:5;

%f1:
for i=1:110
    y1(i)=tansig(x1(i));
end
initv=tansig(x1(110));
for i=111:201
    y1(i)=initv-1.5*logsig(x1(i))+1.5*logsig(x1(i));
end

%f2:
for i=1:120
    y2(i)=2*logsig(x1(i))-1;
end
initv=2*logsig(x1(120))-1;
for i=121:201
    y2(i)=initv-(1/1.6)*sqrt(x1(121)/2)+(1/1.6)*sqrt(x1(i)/2);
end

%f1 and f2 series’ output: (input x1)
for i=1:201
    if y1(i)<=-5+120*0.05
        y12(i)=2*logsig(y1(i))-1;
    else
        initv=2*logsig(x1(120))-1;
    end
end
plot(x1,y1,'c-';x1,y2,'y+');x1,y12,'ro');
plot(x1,y2,'w-'); pause

%Change to random order
xy2=[x1;y2];
%xy2=xorder(xy2);

%This is to find the inverse of f2;
INPAT=xy2(:,2)';
T=xy2(:,1)';
TP(1) = 25;
TP(2) = 2000;
TP(3) = 0.004;
TP(4) = 0.01;

[W1,B1,W2,B2,W3,B3]=initff(INPAT,15,'tansig',5,'tansig',1,'purelin');
[W1,B1,W2,B2]=initff(INPAT,15,'tansig',1,'purelin');
figure
[W1, B1, W2, B2, W3, B3, TE, TR]=trainbpa(W1, B1, 'tansig',W2,B2,'tansig',W3,B3,'purelin',INPAT,T,TP);
[W1, B1, W2, B2, TE, TR]=trainbpa(W1, B1, 'tansig',W2,B2,'purelin',INPAT,T,TP);
%pause
figure
%a=simuff(INPAT, W1, B1, 'tansig', W2, B2, 'tansig', W3, B3, 'purelin');
a=simuff(INPAT, W1, B1, 'tansig', W2, B2, 'purelin');
plot(INPAT, a, 'c.');
%pause
outs=simuff(y12, W1, B1, 'tansig', W2, B2, 'purelin');
T=outs;
figure
[W11, B11, W21, B21]=initff(x1, 20, 'tansig', 1, 'purelin');
[W11, B11, W21, B21, TE, TR]=trainbpa(W11, B11,
'tansig', W21, B21, 'purelin', x1, T, TP);
a=simuff(x1, W1, B1, 'tansig', W2, B2, 'purelin');
figure
plot(x1, y1, 'c-', x1, a, 'w. ');

(3) File name: xfappr3.m

A=dlmread('c:\yang\testdata\test1.txt', ' ');
% plot(A(:,1),A(:,2))
INPAT=A(:,1)'*10000;
TT=A(:,2)';

%new
INPAT=[-3; -2.5; INPAT'; 2.5; 3]';
TT=[0; 0; TT'; 5; 5]';
%TT=TT+0.2*sin(INPAT*2*pi);
plot(INPAT, TT)
pause
TP(1) = 25;
TP(2) = 3000;
TP(3) = 0.004;
TP(4) = 0.01;

%[W1, B1, W2, B2, W3, B3]=initff(INPAT, 5, 'tansig', 20, 'tansig', 1, 'purelin');
[W1, B1, W2, B2]=initff(INPAT, 25, 'tansig', 1, 'purelin');
figure
%[W1, B1, W2, B2, W3, B3, TE, TR]=trainbpa(W1, B1,
'tansig', W2, B2, 'tansig', W3, B3, 'purelin', INPAT, TT, TP);
[W1, B1, W2, B2, TE, TR]=trainbpa(W1, B1,
'tansig', W2, B2, 'purelin', INPAT, TT, TP);
pause
figure
x=-2:0.1:2;
%a=simuff(x, W1, B1, 'tansig', W2, B2, 'tansig', W3, B3, 'purelin');
a=simuff(x, W1, B1, 'tansig', W2, B2, 'purelin');
plot(INPAT, TT, 'y+', x, a, 'c.');
%save wb12 W1 B1 W2 B2

(4) File name: xfappr4.m

A=dlmread('c:\user\yjm\excel\test1.txt', ' ');
% plot(A(:,1),A(:,2))
INPAT=A(:,1)'*10000;
TT=A(:,2)';
% TT=TT+0.2*sin(INPAT*2*pi);
plot(INPAT,TT)
pause
    TP(1) = 25;
    TP(2) = 2000;
    TP(3) = 0.004;
    TP(4) = 0.01;
base=1800;

for iii=5:2:40
    %[W1,B1,W2,B2,W3,B3]=initff(INPAT,5,'tansig',20,'tansig',1,'purelin');
    [W1,B1,W2,B2]=initff(INPAT,iii,'tansig',1,'purelin');
    %[W1, B1, W2, B2, W3, B3, TE, TR]=trainbpa(W1, B1, 'tansig', W2, B2, 'tansig', W3, B3, 'purelin', INPAT, TT, TP);
    [W1, B1, W2, B2, TE, TR]=trainbpa(W1, B1, 'tansig', W2, B2, 'purelin', INPAT, TT, TP);
    for i=1:2000-base
        x_sse(i)=TR(1,base+i);
    end
    p_sse=[min(x_sse); max(x_sse); iii];

    if iii==5
        rsse=[p_sse];
    else
        rsse=[rsse p_sse];
    end
end

close
[m,n]=size(rsse);
for i=1:n
    plot([rsse(3,i) rsse(3,i)],[rsse(1,i) rsse(2,i)],'co',[rsse(3,i) rsse(3,i)],[rsse(1,i) rsse(2,i)],'y-')
    hold on
end
pause
figure
x=-2:0.01:2;
%a=simuff(x, W1, B1, 'tansig', W2, B2, 'tansig', W3, B3, 'purelin');
a=simuff(x, W1, B1, 'tansig', W2, B2, 'purelin');
plot(INPAT,TT,'y-',x,a,'c-');

(5) File name: xfappr5.m

INPAT=-2:.05:2;
% TT=5./(1+exp(-3*INPAT));
TT=xneuron(INPAT);
plot(INPAT,TT)
pause
    TP(1) = 25;
    TP(2) = 2000;
    TP(3) = 0.004;
    TP(4) = 0.01;

    %[W1,B1,W2,B2,W3,B3]=initff(INPAT,5,'tansig',20,'tansig',1,'purelin');
2: The autocorrelation test on error signals

File name: xntest.m

input=(rand(1,300)-0.5)*4;
%input=-2:0.01:2;
td=xneuron(input);
plot(input,td,'c.');
%pause
%td=5./(1+exp(-3*input));
out=simuff(input, W1, B1, 'tansig',W2,B2,'tansig',W3,B3,'purelin');
tv=out-td;
%pause
%tv=2*(rand(1,300)-0.5);
[m,n]=size(tv);
r0=sum(tv.^2)/n;
cnst=10;
for tau=1:cnst:10*cnst
    sss=0;
    for j=1:n-tau
        sss=sss+tv(j+tau)*tv(j);
    end
    tu=floor(tau/cnst)+1;
    r(tau+1)=sss/n;
end
r(1)=r0;
ttt=r/r0;
tau1=1:1:11;
hh=0:0.2:12;
g=ones(size(hh));
plot(tau1,ttt,'k--',hh,0.05*g,'b.-',hh,-0.05*g,'b.-')
xlabel('tau');
ylabel('Normalized covariance function x_t');

%Calculate the sign changes of tv
pause
[m,n]=size(tv);
sss=0;
for i=1:n-1
    if tv(i)*tv(i+1)<0
        sss=sss+1;
    end
3: Program for observing weight quantization effects

File name: xweff.m

a=[0 0 0 0 0 0 0 0];
b=[0 0 0 0 1 0 0 0];
c=[0 0 1 1 1 1 0 0];
d=[0 0 1 0 0 1 0 0];
e=[0 0 0 0 0 1 0 0];
f=[0 0 1 0 0 0 0 0];
g=[0 0 0 1 1 1 0 0];
h=[0 0 0 0 0 1 1 0];
i=[0 0 0 0 1 1 1 0];
j=[0 0 0 0 0 1 0 0];
k=[0 0 1 1 1 1 1 0];

pt1=[a b b b b b a];
pt2=[a c d e c f c a];
pt3=[a c d e g e c a];
pt4=[a d d d k e e a];
pt5=[a c f c e d c a];
pt6=[a c f c d d c a];
pt7=[a c e h b b b a];
pt8=[a c d c d d c a];
pt9=[a c d c e d c a];
pt0=[a c d d d d c a];
pta=[a c d d c d d a];
ptb=[a f f c d d c a];
ptc=[a c d f f d c a];
ptd=[a e e c d d c a];
pte=[a c f c f c a a];
ptf=[a c f c f f f a];

INPAT={pt1, pt2, pt3, pt4, pt5, pt6, pt7, pt8, pt9, pt0, pta, ptb, ptc, ptd, pte, ptf}';
T=[0 0 0 0;... 0 0 0 1;... 0 0 1 0;... 0 0 1 1;... 0 1 0 0;... 0 1 0 1;... 0 1 1 0;... 0 1 1 1;... 1 0 0 0;... 1 0 0 1;... 1 0 1 0;... 1 0 1 1;... 1 1 0 0;... 1 1 0 1;... 1 1 1 0;... 1 1 1 1}';
function pattern processing:
\[ [m, n] = \text{size} (\text{INPAT}); \]
\[ \text{tv} = \text{zeros} (m, 1); \]
for \( i = 1 : m \)
  if \( (\text{sum} (\text{INPAT}(i, :)) == n) | (\text{sum} (\text{INPAT}(i, :)) == 0) \)
    \( \text{tv}(i, 1) = 1; \)
  end
end

function new INPAT: IV
\[ \text{IV} = []; \]
for \( i = 1 : m \)
  if \( \text{tv}(i, 1) == 0 \)
    \( \text{IV} = [\text{IV}; \text{INPAT}(i, :)]; \)
  end
end
\[ \text{INPUT} = \text{IV}; \]
\[ \text{clear IV}; \]
\[ \text{pause} \]
\[ \text{kk} = 0.8; \]

\[ \text{tn} = 20; \]
\[ \text{jjj} = 1; \]
\[ \text{sseout} = \text{zeros} (\text{tn}, 2); \]

while \( \text{j jj} < \text{tn} + 1 \)

\[ [\text{W1}, \text{B1}, \text{W2}, \text{B2}] = \text{initff} (\text{INPAT}, 8, 'tansig', 4, 'tansig'); \]

\[ \text{out} = \text{simuff} (\text{INPAT}, \text{W1}, \text{B1}, 'tansig', \text{W2}, \text{B2}, 'tansig'); \]
\[ \text{sseout} (\text{j jj}, 1) = \text{sumsqr} (\text{out} - \text{T}); \]

\[ \text{wl} = \text{xquntz} (\text{W1}); \]
\[ \text{w2} = \text{xquntz} (\text{W2}); \]
\[ \text{b2} = \text{xquntz} (\text{B2}); \]
\[ \text{bl} = \text{xquntz} (\text{B1}); \]
\[ \text{outq} = \text{simuff} (\text{INPAT}, \text{wl}, \text{bl}, 'tansig', \text{w2}, \text{b2}, 'tansig'); \]
\[ \text{sseout} (\text{j jj}, 2) = \text{sumsqr} (\text{outq} - \text{T}); \]

\[ \text{pause} \]
\[ \text{j jj} = \text{j jj} + 1; \]
end

\[ \text{The following statement can save the matrix sseout} \]
\[ \text{and the file with the name 'xsseout' can be read by Excel} \]
\[ \text{save xsseout sseout -ascii -double -tabs} \]

4:  Programs for training a LNN
File name: `xtrain1.m`

```matlab
a=[1 0 0 0 0 0 0];

pt1=[0 0 0 0 0 0 0 a a a a a a a a a 1 1 1 1 1 1 1 1];
pt2=[a a a a a ... 
    0 1 0 0 0 0 0 0 ...
    0 1 0 0 0 0 0 0 ...
    0 1 1 0 0 0 0 0 ...
    0 0 0 0 1 1 1 1];
pt3=[a a a ... 
    0 1 0 0 0 0 0 0 ...
    0 1 0 0 0 0 0 0 ...
    0 1 1 0 0 0 0 0 ...
    0 0 1 1 0 0 0 0 ...
    0 0 0 0 0 1 1 1];
pt4=[a a ... 
    0 1 0 0 0 0 0 0 ...
    0 1 0 0 0 0 0 0 ...
    0 0 1 0 0 0 0 0 ...
    0 0 0 1 0 0 0 0 ...
    0 0 0 0 1 1 0 0 ...
    0 0 0 0 0 0 1 1];
pt5=[a ... 
    0 1 0 0 0 0 0 0 ...
    0 0 1 0 0 0 0 0 ...
    0 0 0 1 0 0 0 0 ...
    0 0 0 0 1 0 0 0 ...
    0 0 0 0 0 1 0 0 ...
    0 0 0 0 0 0 0 1];
pt6=[1 1 0 0 0 0 0 0 ...
    0 0 1 1 0 0 0 0 ...
    0 0 0 0 1 0 0 0 ...
    0 0 0 0 0 1 0 0 ...
    0 0 0 0 0 0 1 0 ...
    0 0 0 0 0 0 0 1];
pt7=[1 1 1 0 0 0 0 0 ...
    0 0 0 1 1 1 0 0 ...
    0 0 0 0 0 1 1 0 ...
    0 0 0 0 0 0 1 0 ...
    0 0 0 0 0 0 0 1 ...
    0 0 0 0 0 0 0 1];
pt8=[1 1 1 1 0 0 0 0 ...
    0 0 0 0 1 1 1 0 ...
    0 0 0 0 0 0 1 0 ...
    0 0 0 0 0 0 0 1 ...
    0 0 0 0 0 0 0 1 ...
    0 0 0 0 0 0 0 1];
pt9=[1 1 1 1 1 1 1 1 ... ```
INPAT=[pt1; pt2; pt3; pt4; pt5; pt6; pt7; pt8; pt9];
T=[1 0 0 0; 1 1 0 0; 1 0 1 0; 1 1 1 0; ... 
   1 1 1 1; 0 1 1 1; 0 1 0 1; 0 0 1 1; 0 0 0 1];

%input pattern processing:
[m,n]=size(INPAT);
tv=zeros(m,1);
for i=1:m
  if (sum(INPAT(i,:))==n) || (sum(INPAT(i,:))==0)
    tv(i,1)=1;
  end
end

%new INPAT: IV
IV=[];
for i=1:m
  if tv(i,1)==0
    IV=[IV; INPAT(i,:)];
  end
end
INPAT=IV;
clear IV;

[W1,B1,W2,B2]=initff(INPAT,8,'tansig',4,'tansig');

%[W1,B1,W2,B2]=initff(INPAT,10,'purelin',16,'purelin');
%[W1, B1, W2, B2, TE, TR]=trainbp(W1, B1, 'tansig',W2,B2,'purelin',INPAT,T,TP);
%[W1, B1, W2, B2, TE, TR]=trainbpa(W1, B1, 'tansig',W2,B2,'purelin',INPAT,T);
%With quantized weights
[W1, B1, W2, B2, TE, TR,sw1,sw2]=xtrainbp(W1, B1, 'tansig',W2,B2,'tansig',INPAT,T);

%[W1, B1, W2, B2,TE, TR]=trainbpx(W1, B1, 'purelin',W2,B2,'purelin',INPAT,T, TP);
%pause
%a=simuff(INPAT, W1, B1, 'tansig',W2,B2,'purelin');
%8 hidden neurons
ab=rand(8,2);
alpha=0.5+(3-0.5)*ab(:,1);
beta=-0.20+0.4*ab(:,2);

%Outputs with all standard activation functions;
%out1=simuff(INPAT, W1, B1, 'tansig',W2,B2,'purelin');
%Outputs with activation functions which have variations;
%out2=xsimuff1(INPAT,W1,B1,'sigml',alpha,beta,W2,B2,'purelin');

180
a=[0 0 0 0 0 0 0 0];
b=[0 0 0 0 1 0 0 0];
c=[0 0 1 1 1 1 0 0];
d=[0 0 1 0 0 1 0 0];
e=[0 0 0 0 0 1 0 0];
f=[0 0 1 0 0 0 0 0];
g=[0 0 0 0 1 1 1 0];
h=[0 0 0 0 1 1 1 0];
i=[0 0 0 0 1 1 1 0];
j=[0 0 0 0 0 1 0 0];
k=[0 0 1 1 1 1 1 0];

pt1=[a b b b b b b a];
pt2=[a c d e c f c a];
pt3=[a c d e g e c a];
pt4=[a d d d k e e a];
pt5=[a c f c e d c a];
pt6=[a c f c d d c a];
pt7=[a c e h b b b b a];
pt8=[a c d c d d c a];
pt9=[a c d c e d c a];
pt0=[a c d d d d c a];
pta=[a c d d c d d a];
ptb=[a f f c d d c a];
ptc=[a c d f f d c a];
ptd=[a e e e c d d c a];
pte=[a c f c f c a a];
ptf=[a c f c f f f f a];

INPAT=[pt1; pt2; pt3; pt4; pt5; pt6; pt7; pt8; pt9; pt0; pta; ptb; ptc; ptd; pte; ptf]'';
T=[0 0 0 0;... 0 0 0 1;... 0 0 1 0;... 0 0 1 1;... 0 1 0 0;... 0 1 0 1;... 0 1 1 0;... 0 1 1 1;... 1 0 0 0;... 1 0 0 1;... 1 0 1 0;... 1 0 1 1;... 1 1 0 0;... 1 1 0 1;... 1 1 1 0;... 1 1 1 1]'';

%input pattern processing:
[m,n]=size(INPAT);
tv=zeros(m,1);%for i=1:m
if (sum(INPAT(i,:))==(n)|(sum(INPAT(i,:))==(0)
tv(i,1)=1;
end

end

%new INPAT: IV
IV=[];
for i=1:m
    if tv(i,1)==0
        IV=[IV;INPAT(i,:)];
    end
end

INPAT=IV;
clear IV;
pause

TP(1)=25;
TP(2)=2000;
TP(3)= 0.02;
TP(4)=0.02; %0.01
TP(5)= 1.05;
TP(6)= 0.7;
TP(7)= 1.04;

[W1,B1,W2,B2]=initff(INPAT,8,'tansig',4,'purelin');
%[W1,B1,W2,B2]=initff(INPAT,10,'purelin',16,'purelin');
%[W1, B1, W2, B2, TE, TR]=trainbp(W1, B1, 'tansig', W2, B2, 'purelin', INPAT', T, TP);
[W1, B1, W2, B2, TE, TR]=trainbpa(W1, B1, 'tansig', W2, B2, 'purelin', INPAT, T, TP);
%[W1, B1, W2, B2, TE, TR]=trainbps(W1, B1, 'purelin', W2, B2, 'purelin', INPAT, T, TP);
pause

%a=simuff(INPAT, W1, B1, 'tansig', W2, B2, 'purelin');
%8 hidden neurons
ab=rand(8,2);
alpha=0.5+(3-0.5)*ab(:,1);
beta=-0.20+0.4*ab(:,2);

%Outputs with all standard activation functions;
out1=xsimuff(INPAT, W1, B1, 'tansig', W2, B2, 'purelin');
%Outputs with activation functions which have variations;
out2=xsimuff1(INPAT,W1,B1,'xsig1',alpha,beta,W2,B2,'purelin');

%Plot the comparisons of out1 and out2
[x1,y1]=bar(out1(1,:));
[x2,y2]=bar(out2(1,:));
y3=zeros(1,16);
[x3,y3]=bar(y3);
plot(x1,y1,'y-',x2,y2,'r-',x3,y3,'w-')
axis([0 17 -0.2 1.5]);

(3) File name: xtrain3.m

flag=1;
a=[1 0 0 0 0 0 0];
pt1=[0 0 0 0 0 0 0 a a a a a a a a 1 1 1 1 1 1 1 1 1];
\[
\begin{align*}
pt2 &= [a \ a \ a \ a \ldots \\
0 &\ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ ]; \\
pt3 &= [a \ a \ a \ldots \\
0 &\ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ ]; \\
pt4 &= [a \ a \ldots \\
0 &\ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ ]; \\
pt5 &= [a \ldots \\
0 &\ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ ]; \\
pt6 &= [1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ ]; \\
pt7 &= [1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ ]; \\
pt8 &= [1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ ]; \\
pt9 &= [1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ \ldots \\
0 &\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ ] \\
\end{align*}
\]
0 0 0 0 0 0 0 0

INPAT=[pt1; pt2; pt3; pt4; pt5; pt6; pt7; pt8; pt9]';
T=[1 0 0 0; 1 1 0 0; 1 0 1 0; 1 1 1 0; ...
  1 1 1 1; 0 1 1 1; 0 1 0 1; 0 0 1 1; 0 0 0 1]';

%m,n=size(INPAT);
tv=zeros(m,1);
for i=1:m
  if (sum(INPAT(i,:))==n) | (sum(INPAT(i,:))==0)
    tv(i,1)=1;
  end
end

%new INPAT:IV
IV=[];
for i=1:m
  if tv(i,1)==0
    IV=[IV; INPAT(i,:)];
  end
end
INPAT=IV;
clear IV;
%pause
kk=0.7;
smn=0.001;

[Wz1, Bz1, Wz2, Bz2]=initff(INPAT, 8, 'tansig', 4, 'purelin');

[%W1, B1, W2, B2]=initff(INPAT, 10, 'purelin', 16, 'purelin');
%[W1, B1, W2, B2, TE, TR]=trainbp(W1, B1,
  'tansig', W2, B2, 'purelin', INPAT, T, TP);
%[W1, B1, W2, B2, TE, TR]=trainbpaa(W1, B1,
  'tansig', W2, B2, 'purelin', INPAT, T);

tn=20;
jjj=1;
while jjj<tn+1
  %With quantized weights
  [Wz1, Bz1, Wz2, Bz2, TE, sw1, sw2]=xtrainb3(Wz1, Bz1,
    'tansig', Wz2, Bz2, 'purelin', INPAT, T, jjj);
  %xtrainb()
  out=xsimuff3(INPAT, Wz1, Bz1, 'tansig', Wz2, Bz2, 'purelin');

  %Quantized weights effect
  %wl=xquntz(Wz1);
  %w2=xquntz(Wz2);
  %b2=xquntz(Bz2);
  %bl=xquntz(Bz1);
  w1=xquntz1(Wz1);
  w2=xquntz1(Wz2);
  b2=xquntz1(Bz2);
  bl=xquntz1(Bz1);
  outq=xsimuff3(INPAT, w1, bl, 'tansig', w2, b2, 'purelin');

end
\texttt{sse=sumsqr(outq-T)}

\textbf{if} j\texttt{jj}>1  \\
\texttt{sse_r2(jjj)=sse;  \\
sse_r(jjj)=min(sse,sse_r(jjj-1));  \\
if sse-sse_r(jjj-1)>0.001  \\
\textbf{if} flag==1  \\
\hspace{1em}flag=2;  \\
\textbf{elseif} flag==2;  \\
\hspace{1em}flag=1;  \\
\textbf{end}  \\
\textbf{end}  \\
\textbf{else}  \\
\hspace{1em}sse_r(jjj)=sse;  \\
\hspace{1em}sse_r2(jjj)=sse;  \\
\textbf{end}  \\
\texttt{flag=2;}  \\
\textbf{if} flag==2  \\
\textbf{pause}  \\
[m,n]=\texttt{size(Wz1);}  \\
sw1=\texttt{zeros(m,n+1);}  \\
%m,n]=\texttt{size(Wz2);}  \\
%sw2=\texttt{zeros(m,n+1);}  \\
dw1=Wz1-w1;  \\
dw2=Wz2-w2;  \\
db2=Bz2-b2;  \\
db1=Bz1-b1;  \\
twl=Wz1;  \\
\textbf{for} i=1:m  \\
\hspace{1em}twl(i,:)=w1(i,:);  \\
\hspace{1em}\texttt{tswl(i,:)=twl(i,:)+0.005;w1(i,:);  \\
\hspace{1em}outl=xsimuff3(INPAT, twl, Bz1, 'tansig',Wz2,Bz2,'purelin');  \\
\hspace{1em}sse1=sumsqr(outl-out);  \\
\hspace{1em}\textbf{if} abs(dw1(i,:))>smn \hspace{1em}\%new  \\
\hspace{1em}\hspace{1em}sw1(i,:)= sse1/abs(dw1(i,:)); \hspace{1em}%abs(sse1/0.005);\hspace{1em}abs(sse1/dw1(i,:));  \\
\hspace{1em}\textbf{else}  \\
\hspace{1em}\hspace{1em}sw1(i,:)= sse1/smn; \hspace{1em}\%new  \\
\hspace{1em}\textbf{end}  \\
\hspace{1em}twl=Wz1;  \\
\textbf{end}  \\
\textbf{end}  \\
twl=Bz1;  \\
\textbf{for} i=1:m  \\
\hspace{1em}twl(i,:)=b1(i,:);  \\
\hspace{1em}\texttt{tswl(i,:)=twl(i,:)+0.005;b1(i,:);  \\
\hspace{1em}outl=xsimuff3(INPAT, Wz1, twl, 'tansig',Wz2,Bz2,'purelin');  \\
\hspace{1em}sse1=sumsqr(outl-out);  \\
\hspace{1em}\textbf{if} abs(db1(i,:))>smn  \\
\hspace{1em}\hspace{1em}sw1(i,:)= sse1/abs(db1(i,:)); \hspace{1em}%abs(sse1/0.005); \hspace{1em}abs(sse1/db1(i,:));  \\
\hspace{1em}\textbf{else}  \\
\hspace{1em}\hspace{1em}sw1(i,:)= sse1/smn;  \\
\hspace{1em}\textbf{end}  \\
\hspace{1em}twl=Bz1;  \\
\textbf{end}  \\
%\hspace{1em}\texttt{newly added}
sw2=zeros(m,n+1);
tw1=Wz2;
for i=1:m
    for j=1:n
        tw1(i,j)=w2(i,j);%tw1(i,j)+0.005;%w1(i,j);
        out1=xsimuff3(INPAT, Wz1, Bz1, 'tansig',tw1,Bz2,'purelin');
        ssel=sumsqr(out1-out);
        if abs(dw2(i))>smn
            sw2(i,j)= ssel/abs(dw2(i)); %abs(ssel/0.005);%abs(ssel/dw1(i,j));
        else
            sw2(i,j)= ssel/smnn
        end
        tw1=Wz2;
    end
end
tw1=Bz2;
for i=1:m
    tw1(i)=b2(i); %tw1(i)+0.005;%b1(i);
    out1=xsimuff3(INPAT, Wz1, Bz1, 'tansig',Wz2,tw1,'purelin');
    ssel=sumsqr(out1-out);
    if abs(db2(i))>smn
        sw2(i,n+1)= ssel/abs(db2(i)); %abs(ssel/0.005); %abs(ssel/db1(i));
    else
        sw2(i,n+1) = ssel/smnn;
    end
    tw1=Bz2;
end
swmax1=max(max(sw1));
swmax2=max(max(sw2));

temp_Wz1=Wz1;
[m,n]=size(Wz1);
for i=1:m
    for j=1:n+1
        if (sw1(i,j)>0.7*swmax1)&(j<n+1)
            Wz1(i,j)=kk*Wz1(i,j)+(1-kk)*rand;
        elseif (sw1(i,j)>0.7*swmax1)&(j==n+1)
            Bz1(i)=kk*Bz1(i)+(1-kk)*rand;
        end
    end
end

[m,n]=size(Wz2);

for i=1:m
    for j=1:n+1
        if (sw2(i,j)>0.7*swmax2)&(j<n+1)
            Wz2(i,j)=kk*Wz2(i,j)+(1-kk)*rand;
        elseif (sw2(i,j)>0.7*swmax2)&(j==n+1)
            Bz2(i)=kk*Bz2(i)+(1-kk)*rand;
        end
    end
end

% Wz1=xquntz(Wz1);
% Bz1=xquntz(Bz1);
%Wz1=xquntz(Wz1);
%Wz2=xquntz(Wz2);
%Bz2=xquntz(Bz2);
%Bz1=xquntz(Bz1);
Wz1=xquntz1(Wz1);
Wz2=xquntz1(Wz2);
Bz2=xquntz1(Bz2);
Bz1=xquntz1(Bz1);

end if flag==2?

jjj=jjj+1;
end

save sser sse_r;
pause
bar(sse_r);
axis([0 21 0 3]);
xlabel('The Epoch of Retraining')
ylabel('Sum-Squared Error')
pause
load sse_r
x=1:1:20;
y=0:0.05:20;
yy=1.3*ones(size(y));
plot(x,sse_r2,'yo',x,sse_r2,'c-',y,yy,'w.');
axis([0 20 0 5])
xlabel('The Epoch of Retraining')
ylabel('Sum-Squared Error for Quantized Weights')

(4) File name: xtrain4.m

a=[1 0 0 0 0 0 0 0];
% pt1=[0 0 0 0 0 0 0 0 a a a a a a a 1 1 1 1 1 1 1 1];
pt2=[a a a ...
0 1 0 0 0 0 0 ... 
0 1 0 0 0 0 0 ... 
0 1 1 1 0 0 0 ... 
0 0 0 0 1 1 1 ];
pt3=[a a a ...
0 1 0 0 0 0 0 ... 
0 1 0 0 0 0 0 ... 
0 1 1 0 0 0 0 ... 
0 0 1 1 1 0 0 ... 
0 0 0 0 0 1 1 ];
pt4=[a a ...
0 1 0 0 0 0 0 ... 
0 1 0 0 0 0 0 ... 
0 0 1 0 0 0 0 ... 
0 0 0 1 0 0 0 ... 
0 0 0 0 1 1 0 ...
pt5=[a 
 0 1 0 0 0 0 0 0 ...
 0 0 1 0 0 0 0 0 ...
 0 0 0 1 0 0 0 0 ...
 0 0 0 0 1 0 0 0 ...
 0 0 0 0 0 1 0 0 ...
 0 0 0 0 0 0 1 0 ...
 0 0 0 0 0 0 0 1 ];

pt6=[1 1 0 0 0 0 0 0 ...
 0 0 1 1 0 0 0 0 ...
 0 0 0 0 1 0 0 0 ...
 0 0 0 0 0 1 0 0 ...
 0 0 0 0 0 0 1 0 ...
 0 0 0 0 0 0 0 1 ...
 0 0 0 0 0 0 0 1 ];

pt7=[1 1 1 0 0 0 0 0 ...
 0 0 0 1 1 1 0 0 ...
 0 0 0 0 0 1 1 0 ...
 0 0 0 0 0 0 1 0 ...
 0 0 0 0 0 0 0 1 ...
 0 0 0 0 0 0 0 1 ...
 0 0 0 0 0 0 0 1 ];

pt8=[1 1 1 1 0 0 0 0 ...
 0 0 0 0 1 1 1 0 ...
 0 0 0 0 0 0 1 0 ...
 0 0 0 0 0 0 0 1 ...
 0 0 0 0 0 0 0 1 ...
 0 0 0 0 0 0 0 1 ...
 0 0 0 0 0 0 0 1 ];

pt9=[1 1 1 1 1 1 1 1 ...
 0 0 0 0 0 0 0 1 ...
 0 0 0 0 0 0 0 1 ...
 0 0 0 0 0 0 0 1 ...
 0 0 0 0 0 0 0 1 ...
 0 0 0 0 0 0 0 1 ...
 0 0 0 0 0 0 0 1 ];

INPAT=[pt1; pt2; pt3; pt4; pt5; pt6; pt7; pt8; pt9]';
T=[1 0 0 0; 1 1 0 0; 1 0 1 0; 1 1 1 0; ...
 1 1 1 1; 0 1 1 1; 0 1 0 1; 0 0 1 1; 0 0 0 1]';
TP=[25 1000 0.02 0.01 1.05 0.7 0.9 1.04];

%input pattern processing:
[m,n]=size(INPAT);
tv=zeros(m,1);
for i=1:m
  if (sum(INPAT(i,:))==n)|(sum(INPAT(i,:))==0)
    tv(i,1)=1;
  end
end

%new INPAT:IV
IV=[];
for i=1:m
    if cv(i,1)==0
        IV=[IV;INPAT(i,:)];
    end
end
INPAT=IV;
clear IV;
%pause
kk=0.8;

[Wx1,Bx1,Wx2,Bx2]=initff(INPAT,8,'tansig',4,'tansig');

%[W1,B1,W2,B2]=initff(INPAT,10,'purelin',16,'purelin');
%[W1, B1, W2, B2, TE, TR]=trainbp(W1, B1,
  'tansig','W2,B2','purelin',INPAT,T,TP);
%[W1, B1, W2, B2, TE, TR]=trainbpa(W1, B1,
  'tansig','W2,B2','purelin',INPAT,T);

[Wx1, Bx1, Wx2, Bx2, TE, TR,sw1,sw2]=xtrainbp(Wx1, Bx1,
  'tansig',Wx2,Bx2,'tansig',INPAT,T);

out=simuff(INPAT, W1, B1, 'tansig',W2,B2,'purelin');

pause
bar(sse_r1);
axis([0 21 0 3]);
xlabel('The Epoch of Retraining')
ylabel('Sum-Squared Error')

(5)  File name: xtrain5.m

a=[1 0 0 0 0 0 0 0 0];

% pt1=[0 0 0 0 0 0 0 a a a a a a a 1 1 1 1 1 1 1 1];
pt2=[a a a a ...
    0 1 0 0 0 0 0 ...  
    0 1 0 0 0 0 0 ...  
    0 1 1 1 0 0 0 0 ... 
    0 0 0 0 1 1 1 1 ];
pt3=[a a a ...
    0 1 0 0 0 0 0 ...  
    0 1 0 0 0 0 0 ...  
    0 1 1 0 0 0 0 ...  
    0 0 1 1 1 0 0 0 ... 
    0 0 0 0 0 1 1 1 ];
pt4=[a a ...
    0 1 0 0 0 0 0 ...  
    0 1 0 0 0 0 0 ...  
    0 0 1 0 0 0 0 ...  
    0 0 0 1 0 0 0 0 ... 
    0 0 0 0 1 1 0 0 ...  
    0 0 0 0 0 0 1 1 ];
pt5=[a ...
    0 1 0 0 0 0 0 ...  
    0 1 0 0 0 0 0 ... ]
```
0 0 0 1 0 0 0 0 ...
0 0 0 0 1 0 0 0 ...
0 0 0 0 0 1 0 0 ...
0 0 0 0 0 0 1 0 ...
0 0 0 0 0 0 0 1 ];
pt6=[1 1 0 0 0 0 0 0 ...
0 0 1 1 0 0 0 0 ...
0 0 0 0 1 0 0 0 ...
0 0 0 0 0 1 0 0 ...
0 0 0 0 0 0 1 0 ...
0 0 0 0 0 0 0 1 ...
0 0 0 0 0 0 0 1 ];
pt7=[1 1 1 0 0 0 0 0 ...
0 0 0 1 1 0 0 0 ...
0 0 0 0 0 1 0 0 ...
0 0 0 0 0 0 1 0 ...
0 0 0 0 0 0 0 1 ...
0 0 0 0 0 0 0 1 ...
0 0 0 0 0 0 0 1 ];
pt8=[1 1 1 1 0 0 0 0 ...
0 0 0 0 1 1 0 0 ...
0 0 0 0 0 0 1 0 ...
0 0 0 0 0 0 0 1 ...
0 0 0 0 0 0 0 1 ...
0 0 0 0 0 0 0 1 ...
0 0 0 0 0 0 0 1 ];
pt9=[1 1 1 1 1 1 1 1 ...
0 0 0 0 0 0 0 1 ...
0 0 0 0 0 0 0 1 ...
0 0 0 0 0 0 0 1 ...
0 0 0 0 0 0 0 1 ...
0 0 0 0 0 0 0 1 ...
0 0 0 0 0 0 0 1 ];

INPAT=[pt1; pt2; pt3; pt4; pt5; pt6; pt7; pt8; pt9]’;
T=[1 0 0 0; 1 1 0 0; 1 0 1 0; 1 1 1 0; ...
   1 1 1 1; 0 1 1 1; 0 1 0 1; 0 0 1 1; 0 0 0 1]’;

%input pattern processing:
[m,n]=size(INPAT);
tv=zeros(m,1);
for i=1:m
    if (sum(INPAT(i,:))==n) || (sum(INPAT(i,:))==0)
        tv(i,1)=1;
    end
end

%new INPAT:IV
IV=[];
for i=1:m
    if tv(i,1)==0
        IV=[IV; INPAT(i,:)];
```
end
end
INPAT=IV;
clear IV;
%pause
kk=0.7;

flag=1; %
load wb12.mat

[Wz1,Bz1,Wz2,Bz2]=initff(INPAT, 8, 'purelin', 4, 'purelin');
[W1,B1,W2,B2]=initff(INPAT, 16, 'purelin', 16, 'purelin');
[W1, B1, W2, B2, TE, TR]=trainbp(W1, B1, 'tansig', W2, B2, 'purelin', INPAT, T, TP);
[W1, B1, W2, B2, TE, TR]=trainbp(W1, B1, 'tansig', W2, B2, 'purelin', INPAT, T);

tn=1;
jjj=1;
while j jj<tn+1
  %With quantized weights
  [Wz1, Bz1, Wz2, Bz2, TE, TR, sw1, sw2]=xtrainbp(Wz1, Bz1, 'purelin', Wz2, Bz2, 'purelin', INPAT, T, flag);

  flag=flag+1;
  %out=simuff(INPAT, Wz1, Bz1, 'tansig', Wz2, Bz2, 'purelin');

  a1 = xmaf(W1,B1,W2,B2,Wz1*INPAT,Bz1);
  out = xmaf(W1,B1,W2,B2,Wz2*a1,Bz2);
%
  Quantized weights effect
  w1=xquntz(Wz1);
  w2=xquntz(Wz2);
  b2=xquntz(Bz2);
  b1=xquntz(Bz1);
%
  % outq=simuff(INPAT, w1, b1, 'tansig', w2, b2, 'purelin');
  % a1 = xmaf(W1,B1,W2,B2,w1*INPAT,b1);
  % outq = xmaf(W1,B1,W2,B2,w2*a1,b2);
%
  % sse=sumsqr(outq-T)
  %sse_r(jjj)=sse;
  %pause
  [m,n]=size(Wz1);
  sw1=zeros(m,n+1);
  [m,n]=size(Wz2);
  %sw2=zeros(m,n+1);
  dw1=Wz1-w1;
  dw2=Wz2-w2;
  db2=Bz2-b2;
  db1=Bz1-b1;
  twl=Wz1; %twl;
  for i=1:m
    for j=1:n
      % dwl(i,j)=0.1*sign(rand-0.5);
      % twl(i,j)=Wz1(i,j)+dwl(i,j);

191
twl(i,j)=wl(i,j);

% out1=simuff(INPAT, twl, B1, 'tansig', W2, B2, 'purelin');
% aal = xmaf(W1,B1,W2,B2,twl*INPAT,Bzl);
% out1 = xmaf(W1,B1,W2,B2,Wz2*aal,Bz2);

ssel=sumsqr(out1-out);
swl(i,j) = abs(ssel/dwl(i,j));
twl=Wzl;
end
end
twl=Bzl;%bl;
for i=1:m
   % dbl(i)=0.1*sign(rand-0.5);
   % twl(i)=Bzl(i)+dbl(i);
   twl(i)=b(i);
% out1=simuff(INPAT, W1, twl, 'tansig', W2, B2, 'purelin');
% aal = xmaf(W1,B1,W2,B2,Wzl*INPAT,twl);
% out1 = xmaf(W1,B1,W2,B2,Wz2*aal,Bz2);

ssel=sumsqr(out1-out);
swl(i,n+1)= abs(ssel/dbl(i));
twl=Bzl;
end
swmax=max(max(swl));
[m,n]=size(Wzl);
for i=1:m
    for j=1:n
        if (swl(i,j)>0.6*swmax)
            Wzl(i,j)=kk*Wzl(i,j)+(1-kk)*sign(rand-0.5);%rand;
        end
    end
end
jjj=jjj+1;
end

pause
pause
bar(sse_Rl);
axis([0 21 0 3]);
xlabel('The Epoch of Retraining')
ylabel('Sum-Squared Error')

pause
aal = xmaf(W1,B1,W2,B2,Wzl*INPAT,Bzl);
out = xmaf(W1,B1,W2,B2,Wz2*aal,Bz2);

twi=Wzl;
i=1;
for wi=-2.0:0.05:2.0
    Wzl(5,13)=wi;
    aal = xmaf(W1,B1,W2,B2,Wzl*INPAT,Bzl);
    out1 = xmaf(W1,B1,W2,B2,Wz2*aal,Bz2);
    ssel(i)=sumsqr(out1-out);
end
Wzl=twl;
i=i+1;
end
wi=-2.0:0.05:2.0;
plot(wi,sse1);

(6) File name: xtrain6.m

flag1=1;

INPAT=[0 0; 0 1; 1 0; 1 1]';
T=[0 1 1 0];

%input pattern processing:
[m,n]=size(INPAT);
tv=zeros(m,1);
for i=1:m
  if (sum(INPAT(i,:))==n)||(sum(INPAT(i,:))==0)
    tv(i,1)=1;
  end
end

%new INPAT:IV
IV=[];
for i=1:m
  if tv(i,1)==0
    IV=[IV;INPAT(i,:)];
  end
end
INPAT=IV;
clear IV;
%pause
kk=0.7;

flag=1; %
load wbl2.mat

%if flag1==1
[Wz1,Bz1,Wz2,Bz2]=initff(INPAT,4,'tansig',1,'tansig');
%save wblinit Wz1 Bz1 Wz2 Bz2
%else
%load wblinit
%end

%[W1,B1,W2,B2]=initff(INPAT,10,'purelin',16,'purelin');
%[W1, B1, W2, B2, TE, TR]=trainbp(W1, B1, 'tansig',W2,B2,'purelin',INPAT',T,TP);
%[W1, B1, W2, B2, TE, TR]=trainbpa(W1, B1, 'tansig',W2,B2,'purelin',INPAT,T);

tn=1;
jjj=1;
while jiii<tn+1
  %With quantized weights
  [Wz1, Bz1, Wz2, Bz2, TE, TR, sw1,sw2]=xtrnbp6(Wz1, Bz1, 'tansig',Wz2,Bz2,'tansig',INPAT,T,flag);

  flag=flag+1;


193
```matlab
%out=simuff(INPAT, Wz1, Bz1, 'tansig', Wz2, Bz2, 'purelin');

aal = xmaf(W1, B1, W2, B2, Wz1*INPAT, Bz1);
out = xmaf(W1, B1, W2, B2, Wz2*aal, Bz2);

% Quantized weights effect
w1=xquntz(Wz1);
w2=xquntz(Wz2);
b2=xquntz(Bz2);
b1=xquntz(Bz1);

% outq=xsimuff(INPAT, W1, B1, 'tansig', W2, B2, 'purelin');
% aal = xmaf(W1, B1, W2, B2, w1*INPAT, b1);
% outq = xmaf(W1, B1, W2, B2, w2*aal, b2);

% sse=sumsqr(outq-T)
% sse_r(rjj)=sse;
% pause
[m,n]=size(Wz1);
swl=zeros(m,n+1);
%[m,n]=size(Wz2);
%sw2=zeros(m,n+1);
dwl=Wz1-w1;
dw2=Wz2-w2;
db2=Bz2-b2;

twl=Wz1;%w1;
for i=1:m
    for j=1:n
        dwl(i,j)=0.1*sign(rand-0.5);
        twl(i,j)=Wz1(i,j)+dwl(i,j);
        twl(i,j)=w1(i,j);
    end
end
tw1=Wz1;
for i=1:m
    for j=1:n
        dbl(i)=0.1*sign(rand-0.5);
        twl(i)=Bz1(i)+dbl(i);
        twl(i)=b1(i);
    end
end
tw1=Bz1;%b1;
for i=1:m
    for j=1:n
        aal = xmaf(W1, B1, W2, B2, tw1*INPAT, Bz1);
        outl = xmaf(W1, B1, W2, B2, Wz2*aal, Bz2);
        ssel=sumsqr(outl-out);
        swl(i,j)=abs(ssel/dwl(i,j));
        twl=Wz1;
    end
end
swmax=max(max(swl));
[m,n]=size(Wz1);
for i=1:m
```

194
for j=1:n
    if (swl(i,j)>0.6*swmax)
        Wz1(i,j)=kk*Wz1(i,j)+(1-kk)*sign(rand-0.5);%rand;
    end
end
end
jjj=jjj+1;
end

pause
pause
bar(sse_r1);
axis([0 21 0 3]);
xlabel('The Epoch of Retraining')
ylabel('Sum-Squared Error')

pause
    aal = xmaf(W1,B1,W2,B2,Wz1*INPAT,Bz1);
    out = xmaf(W1,B1,W2,B2,Wz2*aal,Bz2);
tw1=Wz1;
i=1;
for wi=-2.0:0.05:2.0
    Wz1(5,13)=wi;
    aal = xmaf(W1,B1,W2,B2,Wz1*INPAT,Bz1);
    out1 = xmaf(W1,B1,W2,B2,Wz2*aal,Bz2);
    ssel(i)=sumsqr(out1-out);
Wz1=tw1;
i=i+1;
end
wi=-2.0:0.05:2.0;
plot(wi,ssel);

5:  Subroutines

(1)  File name: xmaf.m

function [a]=xmaf(yw1,yb1,yw2,yb2,input,bias)
%This is the modelled activation function

    if nargin < 1,error('Not enough input arguments'),end
    [nr,nc]=size(input);
    if nargin == 6
        input=input+bias*ones(1,nc);
    end
    %input=input*0.5;
    %a=2./(1+exp(-2*input))-1;
% i = find(~finite(a));
% a(i) = sign(input(i));
%ccc1=0;
%ccc2=0;
%load wb12
for i=1:nr

195
for j=1:nc
    if input(i,j)>2
        input(i,j)=2;
        % ccc1=ccc1+1;
    else if input(i,j)<-2
        input(i,j)=-2;
        % ccc2=ccc2+1;
    end
end

a(i,j)=0.4*simuff(input(i,j), yw1, yb1, 'tansig',yw2,yb2,'purelin')-1;
end
end

%c1
%c2
%pause

(2) File name: xmdelta.m

function d = xmdelta(yw1,yb1,yw2,yb2,a,d,w)
%DELTATAN Delta function for TANSIG neurons.
%
% DELTATAN(A,E)
% A - S1xQ matrix of output vectors
% E - S1xQ matrix of associated errors
% Returns an SxQ matrix of output layer derivatives.
%
% DELTATAN(A,D,W)
% D - S2xQ matrix of next layer delta vectors
% W - S2xS1 weight matrix between layers.
% Returns an SxQ matrix of hidden layer derivatives.
%
% DELTATAN(A)
% Returns derivatives of outputs A (not error) for TRAINLM.
%
% See also NNTRANS, BACKPROP, TANSIG, DELTALOG, DELTALIN.
%
% Jinming Yang
% Revised
% 1998
% $Revision: 1.2 $ $Date: 1994/01/11 16:24:08 $

kk=0.02;
if nargin < 1,error('Not enough input arguments'),end

if nargin == 5
    d = 1-(a.*a);
    d=(xmaf(yw1,yb1,yw2,yb2,a+kk)-xmaf(yw1,yb1,yw2,yb2,a-kk))/(2*kk);
elseif nargin == 6
    d =((xmaf(yw1,yb1,yw2,yb2,a+kk)-xmaf(yw1,yb1,yw2,yb2,a-kk))/(2*kk)).*d;
else
    d = ((xmaf(yw1,yb1,yw2,yb2,a+kk)-xmaf(yw1,yb1,yw2,yb2,a-kk))/(2*kk)).*(w'*d);
end
(3) File name: xneuron.m

function [out]=xneuron(neuron_in)
%neuron_in: input vector in a range [-2,2]
%Constraint: Vbias1+VTP=Vbias2-VTN

[m,n]=size(neuron_in);
m=max(m,n);
iv=neuron_in*50;

Vbias1=1.8;
Vbias2=3.2;
VTN=0.7;
VTP=0.7;
if=-100:1:100;
k1=0.020;

I0=((Vbias2-Vbias1)^2/(2*k1));

V01=(Vbias1+Vbias2)/2-VTN;
V02=Vbias2-VTN;
V03=Vbias1+VTP;
V04=(Vbias1+Vbias2)/2+VTP;

for iii=1:m
    if iv(iii)<=-(100-I0)
        V(iii)=V01-sqrt(k1*(-iv(iii)))-((Vbias2-Vbias1)^2)/4;
    elseif (iv(iii)>-100+I0)&(iv(iii)<=0)
        V(iii)=V02-sqrt(2*k1*(-iv(iii)));
    elseif (iv(iii)>0)&(iv(iii)<=100+I0)
        V(iii)=V03+sqrt(2*k1*iv(iii));
    elseif (iv(iii)>100+I0)
        V(iii)=V04+sqrt(k1*iv(iii))-((Vbias2-Vbias1)^2)/4;
    end

    if V(iii)>=2.5
        V(iii)=2.5+(V(iii)-2.5)*5/4;
    else
        V(iii)=2.5-(2.5-V(iii))*5/4;
    end

end

plot(iv,V,'c.')
pause
out=V;

(4) File name: xorder.m

function a=xorder(n)
%ORDER: Reordering the row order of the vector a so that
%its components are rearranged randomly.
%
%Jinming Yang; Dec. 15, 1997
if nargin==1, error('Not correct argument.'); end

[nr, nc]= size(n);

ref=rand(nr,1);
na=zeros(nr,nc);

for j=1:nr
    [x,i]=max(ref);
    na(j,:) = n(i,:);
    ref(i,1)=-1;
end

a=na;

(5) File name: xquntz.m

function w=xquntz(inw)
    % Quantize the weight
    rail=1;
    [m,n]=size(inw);
    for i=1:m
        for j=1:n
            if inw(i,j)>rail
                inw(i,j)=rail;
            elseif inw(i,j)<-rail
                inw(i,j)=-rail;
            end
        end
    end
    qq=round(16/rail);
    w=round(inw*qq)/qq;
    % w=round(inw*10)/10;

(6) File name: xrange.m

function [nw]=xrange(iw,ur,lr)
    % limit the matrix within [ur,lr]
    [m,n]=size(iw);
    for i=1:m
        for j=1:n
            if iw(i,j)>ur
                iw(i,j)=ur;
            else if iw(i,j)<lr
                iw(i,j)=lr;
            end
        end
    end
    nw=iw;

(7) File name: xsigml.m
function a = xsigml(n,bias,alpha,beta)

% TANSIG Hyperbolic tangent sigmoid transfer function
% with variations by the parameters: alpha and beta
% 
% alpha: 0.5--3.0
% beta:  + or - 0.20
% EXAMPLE: n = -10:0.1:10;
% a = tansig(n);
% plot(n,a)

% Jinming Yang; Dec. 10, 1997
if nargin ~=4, error('Not enough arguments.'); end

[nr,nc] = size(n);
 n = n + bias*ones(1,nc);

 a = 2 ./ (1 + exp(-diag(alpha)*n)) - 1+beta*ones(1,nc);

[m,n]=size(a);
for i=1:m
  for j=1:n
    if a(i,j)>1
      a(i,j)=1;
    else if a(i,j)<-1
      a(i,j)=-1;
    end
  end
end
end

(8) File name: xsimuff1.m

function [a1,a2,a3] = 
xsimuff1(p,w1,b1,f1,f1a,f1b,w2,b2,f2,f2a,f2b,\n\nw3,b3,f3)

% SIMUFF Simulate feed-forward network.
% 
% SIMUFF will simulate networks with up to 3 layers.
% 
% SIMUFF(P,W1,B1,'F1',...,Wn,Bn,'Fn')
% P - Matrix of input (column) vectors.
% Wi - Weight matrix of the ith layer.
% Bi - Bias (column) vector of the ith layer.
% Fi - Transfer function of the ith layer (string).
% Returns output of nth layer.
% 
% [A1,A2,...] = SIMUFF(P,W1,B1,'F1',...,Wn,Bn,'Fn')
% Returns:
% Ai - Output of the ith layer.
% 
% EXAMPLE: [w1,b1,w2,b2] = initff([0 10; -5 5],3,'tansig','purelin');
% p = [2; -3];
% a = simuff(p,w1,b1,'tansig',w2,b2,'purelin')
% 
% See also NNSIM, BACKPROP, INITFF, TRAINBPX, TRAINLM.
% Jinming Yang
% 1998
% $Revision: 1.3 $ $ 

if all([4 9 14] == nargin),error('Wrong number of input arguments'),end

if nargin == 4
    a1 = feval(f1,w1*p,b1);

elseif nargin == 9
    a1 = feval(f1,w1*p,b1,f1a,f1b);
    a2 = feval(f2,w2*a1,b2);
    if nargout <= 1, a1 = a2; end

elseif nargin == 14
    a1 = feval(f1,w1*p,b1,f1a,f1b);
    a2 = feval(f2,w2*a1,b2,f2a,f2b);
    a3 = feval(f3,w3*a2,b3);
    if nargout <= 1, a1 = a3; end
end

(9) File name: xsimuff3.m

function [a1,a2,a3] = xsimuff3(p,w1,b1,f1,w2,b2,f2,w3,b3,f3)
SIMUFF Simulate feed-forward network.

% SIMUFF will simulate networks with up to 3 layers.
% SIMUFF(P,W1,B1,'F1',...,Wn,Bn,'Fn')
% P - Matrix of input (column) vectors.
% Wi - Weight matrix of the ith layer.
% Bi - Bias (column) vector of the ith layer.
% Fi - Transfer function of the ith layer (string).
% Returns output of nth layer.
% [A1,A2,...] = SIMUFF(P,W1,B1,'F1',...,Wn,Bn,'Fn')
% Returns:
% A1 - Output of the ith layer.
% EXAMPLE: [w1,b1,w2,b2] = initff([0 10; -5 5],3,'tansig',2,'purelin');
% p = [2; -3];
% a = simuff(p,w1,b1,'tansig',w2,b2,'purelin')
% See also NNSIM, BACKPROP, INITFF, TRAINBPX, TRAINLM.

% Jinming Yang
% 1998
% $Revision: 1.5 $ $ 

if all([4 7 10] == nargin),error('Wrong number of input arguments'),end

load wb12.mat

if nargin == 4
    a1 = feval(f1,w1*p,b1);
elseif nargin == 7
    a1 = feval(f1,w1*p,b1);
    a2 = feval(f2,w2*a1,b2);
    a1 = xmaf(W1,B1,W2,B2,w1*p,b1);
    a2 = xmaf(W1,B1,W2,B2,w2*a1,b2);
    if nargout <= 1, a1 = a2; end
end

elseif nargin == 10
    a1 = feval(f1,w1*p,b1,f1a,f1b);
    a2 = feval(f2,w2*a1,b2,f2a,f2b);
    a3 = feval(f3,w3*a2,b3);
    if nargout <= 1, a1 = a3; end
end

(10)  File name: xtbpx2.m

function
[w1,b1,w2,b2,i,tr,sw1,sw2]=xtbpx2(w1,b1,f1,w2,b2,f2,p,t,flag,tp)
%XTBPX2 Train 2-layer feed-forward network w/fast backpropagation.

% [W1,B1,W2,B2,TE,TR]=TBPX2(W1,B2,F1,W1,B1,F2,P,T,TP)
% W1 - Weight matrix for the ith layer.
% B1 - Bias vector for the ith layer.
% F1 - Transfer function (string) for the ith layer.
% P - RxQ matrix of input vectors.
% T - SxQ matrix of target vectors.
% TP - Training parameters (optional).
% Returns:
% Wi - new weights.
% Bi - new biases.
% TE - the actual number of epochs trained.
% TR - training record: [row of errors]
% Training parameters are:
% TP(1) - Epochs between updating display, default = 25.
% TP(2) - Maximum number of epochs to train, default = 1000.
% TP(3) - Sum-squared error goal, default = 0.02.
% TP(4) - Learning rate, 0.01.
% TP(5) - Learning rate increase, default = 1.05.
% TP(6) - Learning rate decrease, default = 0.7.
% TP(7) - Momentum constant, default = 0.9.
% TP(8) - Maximum error ratio, default = 1.04.
% Missing parameters and NaN's are replaced with defaults.

% Jinming Yang
% Revised
% 1998
% $Revision: 1.3 $ $

if nargin < 8, error('Not enough arguments.'); end

if TRAINING PARAMETERS
if nargin == 9, tp = []; end
    tp = mndef(tp,[25 1000 0.02 0.01 1.05 0.7 0.9 1.04]);
    df = tp(1);
me =1000; % tp(2);
eg = tp(3);
lr = 0.1; % tp(4);
im = tp(5);
dm = tp(6);
mc = tp(7);
er = tp(8);
df1 = feval(f1,'delta');
df2 = feval(f2,'delta');
dw1 = w1'0;
db1 = b1'0;
dw2 = w2'0;
db2 = b2'0;
MC = 0;

load wbl2.mat

% PRESENTATION PHASE
% a1 = feval(f1,w1*p,b1);
% a2 = feval(f2,w2*a1,b2);
a1 = xmaf(W1,B1,W2,B2,w1*p,b1);
a2 = xmaf(W1,B1,W2,B2,w2*a1,b2);
e = t-a2;
SSE = sumsqr(e);

% TRAINING RECORD
tr = zeros(2,me+1);
tr(1:2,1) = [SSE, lr];

% PLOTTING FLAG
[r,q] = size(p);
[s,q] = size(t);
plotttype = (max(r,s) == 1);

% PLOTTING
%c
message = sprintf('TRAINBPX: %g/%g epochs, lr = %g, SSE = %g.\n',me);
fprintf(message,0,lr,SSE)

%flag=1; %&&&&&&&&
if flag=1
  if plotttype % if plotttype
    h = plotfa(p,t,p,a2);
  else
    h = plottr(tr(1:2,1),eg);
  end
else
  load handle
end

% BACKPROPAGATION PHASE
d2 = feval(df2,a2,e);
d1 = feval(df1,a1,d2,w2);
%d2 = xmdelta(W1,B1,W2,B2,a2,e);
%d1 = xmdelta(W1,B1,W2,B2,a1,d2,w2);
These are sensitivity parameters:
sssw1=zeros(size(w1));
sssw2=zeros(size(w2));
% initial weights
w1=w1;
w2=w2;

% for i=1:me
for i=1+(flag-1)*me:flag*me

% CHECK PHASE
if SSE < eg, i=i-1; break, end

% LEARNING PHASE
[dw1,db1] = learnbpm(p,d1,lr,MC,dw1,db1);
dw2,d2] = learnbpm(a1,d2,lr,MC,dw2,db2);
MC = mc;
new_w1 = w1 + dw1; new_b1 = b1 + db1;
new_w2 = w2 + dw2; new_b2 = b2 + db2;

% Quantize the weights; This doesn’t work!
%new_w1=xquntz(new_w1);
%new_w2=xquntz(new_w2);
%new_b1=xquntz(new_b1);
%new_b2=xquntz(new_b2);

% sensitivity
sssw1=sssw1+dw1.*dw1;
sssw2=sssw2+dw2.*dw2;

% PRESENTATION PHASE
%new_a1 = feval(f1,new_w1*p,new_b1);
%new_a2 = feval(f2,new_w2*new_a1,new_b2);
new_a1 = xmaf(W1,B1,W2,B2,new_w1*p,new_b1);
new_a2 = xmaf(W1,B1,W2,B2,new_w2*new_a1,new_b2);
new_e = t-new_a2;
new_SSE = sumsqrr(new_e);

% MOMENTUM & ADAPTIVE LEARNING RATE PHASE
if new_SSE > SSE*er
lr = lr * dm;
MC = 0;
cd=cd+1;
else
if new_SSE < SSE
lr = lr * im;
cd=0;
end
w1 = new_w1; b1 = new_b1; a1 = new_a1;
w2 = new_w2; b2 = new_b2; a2 = new_a2;
e = new_e; SSE = new_SSE;

% BACKPROPAGATION PHASE
d2 = feval(df2,a2,e);
d1 = feval(df1,a1,d2,w2);
d2 = xmdelta(W1,B1,W2,B2,a2,e);
%d1 = xmdelta(W1,B1,W2,B2,a1,d2,w2);

end

% TRAINING RECORD
tr(:,i+1) = [SSE; lr];
%tr(:,i+1) = [SSE; cd];
% PLOTTING
if rem(i,df) == 0
fprintf(message,i,lr,SSE)
if plottype
delete(h);
    h = plot(p,a2);
drawnow
else
    h = plottr(tr(1:2,1:(i+1)),eg,h);
end
end

% TRAINING RECORD
tr = tr(1:2,1:(i+1));

% PLOTTING
if rem(i,df) ~= 0
fprintf(message,i,lr,SSE)
if plottype
    delete(h);
    plot(p,a2);
drawnow
else
    plottr(tr,eg,h);
end

sw1=100*(sssw1.*w1)./(w1-wi1);
sw2=100*(sssw2.*w2)./(w2-wi2);

% WARNINGS
if SSE > eg
    disp('')
    disp('TRAINBPXX: Network error did not reach the error goal.
    disp(' Further training may be necessary, or try different')
    disp(' initial weights and biases and/or more hidden neurons.
    disp('')
end
save handle h tr
pause

(11) File name: xtbpx26.m

function
[w1,b1,w2,b2,i, tr, sw1,sw2]=xtbpx26(w1,b1,f1,w2,b2,f2,p,t,flag,tp)
%XTBFX26 Train 2-layer feed-forward network w/ fast backpropagation.
%
% [W1,B1,W2,B2,TE,TR] = TBPX2(W1,B2,F1,W1,B1,F2,P,T,TP)
% W1 - Weight matrix for the ith layer.
% B1 - Bias vector for the ith layer.
% F1 - Transfer function (string) for the ith layer.
% P - RxQ matrix of input vectors.
% T - SxQ matrix of target vectors.
% TP - Training parameters (optional).
% Returns:
% W1 - new weights.
% B1 - new biases.
% TE - the actual number of epochs trained.
% TR - training record: [row of errors]
%
% Training parameters are:
% TP(1) - Epochs between updating display, default = 25.
% TP(2) - Maximum number of epochs to train, default = 1000.
% TP(3) - Sum-squared error goal, default = 0.02.
% TP(4) - Learning rate, 0.01.
% TP(5) - Learning rate increase, default = 1.05.
% TP(6) - Learning rate decrease, default = 0.7.
% TP(7) - Momentum constant, default = 0.9.
% TP(8) - Maximum error ratio, default = 1.04.
% Missing parameters and NaN's are replaced with defaults.
%
% Revised
% 1998
% $Revision: 1.8 $

if nargin < 8, error('Not enough arguments.'), end

% TRAINING PARAMETERS
if nargin == 9, tp = []; end
tp = nndef(tp,[25 1000 0.02 0.01 1.05 0.7 0.9 1.04]);
df = 1; %tp(1);
m = 1000; % tp(2);
e = 0.004; %tp(3);
lr = 0.1; %tp(4);
im = tp(5);
dm = tp(6);
mc = tp(7);
er = tp(8);

df1 = feval(f1,'delta');
df2 = feval(f2,'delta');

dw1 = w1*0;
db1 = b1*0;
dw2 = w2*0;
db2 = b2*0;
MC = 0;

urail=1.6;
lrail=1.6;
ttt=[0 0];
load wb12.mat
% PRESENTATION PHASE
% a1 = feval(f1,wi*p,b1);
% a2 = feval(f2,w2*a1,b2);
  a1 = xmaf(W1,B1,W2,B2,w1*p,b1);
  a2 = xmaf(W1,B1,W2,B2,w2*a1,b2);
  e = t-a2;
  SSE = sumsqr(e);

% TRAINING RECORD
  tr = zeros(2,me+1);
  tr(1:2,1) = [SSE; lr];

% PLOTTING FLAG
  [r,q] = size(p);
  [s,q] = size(t);
  plottype = (max(r,s) == 1);

% PLOTTING
% clg
message = sprintf('TRAINBFX: %g/%g epochs, lr = %g, SSE = %g.
' n',me);
fprintf(message,0,lr,SSE)

%flag=1; %&%%%%%%%
if flag==1
  if plottype % if plottype
    h = plotfa(p,t,p,a2);
  else
    h = plottr(tr(1:2,1),eg);
  end
else
  load handle
end

% BACKPROPAGATION PHASE
% d2 = feval(df2,a2,e);
% d1 = feval(df1,a1,d2,w2);
  d2 = xmdelta(W1,B1,W2,B2,a2,e);
  d1 = xmdelta(W1,B1,W2,B2,a1,d2,w2);
  %These are sensitivity parameters:
  sssw1=zeros(size(w1));
  sssw2=zeros(size(w2));
  %initial weights
  w1=1;w2;

  cd=0;
  %for i=1:me
  for i=1+(flag-1)*me:flag*me

% CHECK PHASE
  if SSE < eg, i=i-1; break, end

% LEARNING PHASE
  [dw1,db1] = learnbpm(p,d1,lr,MC,dw1,db1);
  [dw2,db2] = learnbpm(a1,d2,lr,MC,dw2,db2);
  MC = mc;
new_w1 = w1 + dw1; new_b1 = b1 + db1;  
new_w2 = w2 + dw2; new_b2 = b2 + db2;

% Quantize the weights; This doesn't work!
% new_w1=xquntz(new_w1);
% new_w2=xquntz(new_w2);
% new_b1=xquntz(new_b1);
% new_b2=xquntz(new_b2);

% sensitivity
swwl=swwl+dwl.*dw1;  
sww2=sww2+dw2.*dw2;

% PRESENTATION PHASE
%new_a1 = feval(f1,new_w1*p,new_b1);
%new_a2 = feval(f2,new_w2*new_a1,new_b2);
new_a1 = xmaf(W1,B1,W2,B2,new_w1*p,new_b1);
new_a2 = xmaf(W1,B1,W2,B2,new_w2*new_a1,new_b2);
new_e = t-new_a2;
new_SSE = sumsq(new_e);

% MOMENTUM & ADAPTIVE LEARNING RATE PHASE
if new_SSE > SSE*er  
    lr = lr * dm;
    MC = 0;
    cd=cd+1;
else  
    if new_SSE < SSE  
        lr = lr * im;
    end  
    cd=0;
end
w1 = new_w1; b1 = new_b1; a1 = new_a1;
w2 = new_w2; b2 = new_b2; a2 = new_a2;
%wl=xrange(w1,urail,lrail);
%w2=xrange(w2,urail,lrail);
%bl=xrange(b1,urail,lrail);
%b2=xrange(b2,urail,lrail);

e = new_e; SSE = new_SSE;

% BACKPROPAGATION PHASE
%d2 = feval(df2,a2,e);
%d1 = feval(df1,a1,d2,w2);
d2 = xmdelta(W1,B1,W2,B2,a2,e);
d1 = xmdelta(W1,B1,W2,B2,a1,d2,w2);
end

% TRAINING RECORD
tr(:,i+1) = [SSE; lr];
%tr(:,i+1) = [SSE; cd];  
% PLOTTING
if rem(i,df) == 0  
fprintf(message,i,lr,SSE)
if i==1
ttt=[w2(1) w2(2) w2(3)];
else
ttt=[ttt; w2(1) w2(2) w2(3)];
end

if plottype
    delete(h);
    h = plot(p,a2);
drawnow
else
    h = plottr(tr(1:2,1:(i+1)),eg,h);
end
end

% TRAINING RECORD
tr = tr(1:2,1:(i+1));

% PLOTTING
if rem(i,df) == 0
    fprintf(message,i,lr,SSE)
    if plottype
        delete(h);
        plot(p,a2);
drawnow
    else
        plottr(tr,eg,h);
    end
end

sw1=100*(sssw1.*w1)./(w1-wi1);
sw2=100*(sssw2.*w2)./(w2-wi2);

% WARNINGS
if SSE > eg
    disp('')
    disp('TRAINBPX: Network error did not reach the error goal.')
disp(' ' Further training may be necessary, or try different')
disp(' ' initial weights and biases and/or more hidden neurons.')
disp(' ')
end

%figure
plot(ttt(:,1),ttt(:,2),'ko')
xlabel('W2(1)')
ylabel('W2(2)')
save ttt ttt
pause
save handle h tr

(12) File name: xtbpx3.m

function
[w1,b1,w2,b2,i,tr,sw1,sw2]=xtbpx3(w1,b1,f1,w2,b2,f2,p,t,flag1,tp)

208
%XTBPX3 Train 2-layer feed-forward network w/fast backpropagation.
%
% [W1,B1,W2,B2,TE,TR] = TBPX2(W1,B2,F1,W1,B1,F2,P,T,TP)
% Wi - Weight matrix for the ith layer.
% Bi - Bias vector for the ith layer.
% Fi - Transfer function (string) for the ith layer.
% P - RxQ matrix of input vectors.
% T - SxQ matrix of target vectors.
% TP - Training parameters (optional).
% Returns:
% Wi - new weights.
% Bi - new biases.
% TE - the actual number of epochs trained.
% TR - training record: [row of errors]
%
% Training parameters are:
% TP(1) - Epochs between updating display, default = 25.
% TP(2) - Maximum number of epochs to train, default = 1000.
% TP(3) - Sum-squared error goal, default = 0.02.
% TP(4) - Learning rate, 0.01.
% TP(5) - Learning rate increase, default = 1.05.
% TP(6) - Learning rate decrease, default = 0.7.
% TP(7) - Momentum constant, default = 0.9.
% TP(8) - Maximum error ratio, default = 1.04.
% Missing parameters and NaN's are replaced with defaults.

% J. Yang
% Revised
%
% $Revision: 1.3$

if nargin < 8, error('Not enough arguments.'); end

% TRAINING PARAMETERS
if nargin == 9, tp = []; end
tp = rndef(tp,[25 1000 0.02 0.01 1.05 0.7 0.9 1.04]);
df = tp(1);
me =1000; % tp(2);
eg = tp(3);
lr = 0.1; %tp(4);
im = tp(5);
dm = tp(6);
mc = tp(7);
er = tp(8);
df1 = feval(f1,'delta');
df2 = feval(f2,'delta');

dw1 = w1*0;
db1 = b1*0;
dw2 = w2*0;db2 = b2*0;
MC = 0;

load wbl2.mat

% PRESENTATION PHASE
% al = feval(f1,w1*p,b1);
a2 = feval(f2,w2*a1,b2);
a1 = xma(x,W1,B1,W2,B2,w1*p,b1);
a2 = xma(x,W1,B1,W2,B2,w2*a1,b2);
e = t-a2;
SSE = sumsqr(e);

% TRAINING RECORD
tr = zeros(2,me+1);
tr(1:2,1) = [SSE; lr];

% PLOTTING FLAG
[r,q] = size(p);
[s,q] = size(t);
plottype = (max(r,s) == 1);

% PLOTTING
clg
message = sprintf('TRAINBPX: %g/%g epochs, lr = %g, SSE = %g',

message,0,lr,SSE)
flag=1;  %&&&&&&&&
if flag==1
  if plottype  % if plottype
    h = plotfa(p,t,p,a2);
  else
    h = plottr(tr(1:2,1),eg);
  end
else
  load handle
end

% BACKPROPAGATION PHASE
d2 = feval(df2,a2,e);
d1 = feval(df1,a1,d2,w2);
d2 = xmdelta(W1,B1,W2,B2,a2,e);
d1 = xmdelta(W1,B1,W2,B2,a1,d2,w2);
%These are sensitivity parameters:
sssw1=zeros(size(w1));
sssw2=zeros(size(w2));
%initial weights
w1 = w1;
w2 = w2;

cd=0;
over_flag=0;
%for i=1:me
for i=1+(flag-1)*me:flag*me

% CHECK PHASE
if SSE < eg, i=i-1; break, end
if over_flag>=20, lr=0.8, over_flag=0; end

% LEARNING PHASE
[dw1,db1] = learnbp(p,d1,lr,MC,dw1,db1);
[dw2,db2] = learnbp(a1,d2,lr,MC,dw2,db2);
MC = mc;
new_w1 = w1 + dw1; new_b1 = b1 + db1;
new_w2 = w2 + dw2; new_b2 = b2 + db2;

% Quantize the weights; This doesn't work!
% It will quickly get stuck in some local minima
% new_w1=xquantz(new_w1);
% new_w2=xquantz(new_w2);
% new_b1=xquantz(new_b1);
% new_b2=xquantz(new_b2);

% sensitivity
sssw1=sssw1+dw1.*dw1;
sssw2=sssw2+dw2.*dw2;

% PRESENTATION PHASE
% new_a1 = feval(f1,new_w1*p,new_b1);
% new_a2 = feval(f2,new_w2*new_a1,new_b2);
new_a1 = xmaf(W1,B1,W2,B2,new_w1*p,new_b1);
new_a2 = xmaf(W1,B1,W2,B2,new_w2*new_a1,new_b2);
new_e = t-new_a2;
new_SSE = sumsqr(new_e);

if new_SSE-SSE>=0
  over_flag=over_flag+1;
end

% MOMENTUM & ADAPTIVE LEARNING RATE PHASE
if new_SSE > SSE*er
  lr = lr * dm;
  MC = 0;
cd=cd+1;
else
  if new_SSE < SSE
    lr = lr * im;
    cd=0;
  end
  w1 = new_w1; b1 = new_b1; a1 = new_a1;
  w2 = new_w2; b2 = new_b2; a2 = new_a2;
  e = new_e; SSE = new_SSE;
end

% BACKPROPAGATION PHASE
d2 = feval(df2,a2,e);
d1 = feval(df1,a1,d2,w2);
%d2 = xmdelta(W1,B1,W2,B2,a2,e);
%d1 = xmdelta(W1,B1,W2,B2,a1,d2,w2);

end

% TRAINING RECORD
tr(:,i+1) = [SSE; lr];
%tr(:,i+1) = [SSE; cd];
% PLOTTING
if rem(i,df) == 0
  fprintf(message,i,lr,SSE)
  if plottype
delete(h); 
  h = plot(p,a2);
drawnow
else
    h = plottr(tr(1:2,1:(i+1)),eg,h);
end
end
end

% TRAINING RECORD
tr = tr(1:2,1:(i+1));

% PLOTTING
if rem(i,df) == 0
    fprintf(message,i,lr,SSE)
    if plottype
        delete(h);
        plot(p,a2);
        drawnow
    else
        plottr(tr,eg,h);
    end
end

sw1=100*(sssw1.*w1)./(w1-wi1);
sw2=100*(sssw2.*w2)./(w2-wi2);

% WARNINGS
if SSE > eg
    disp('''
    disp(''TRAINBPX: Network error did not reach the error goal.'')
    disp('' Further training may be necessary, or try different'')
    disp('' initial weights and biases and/or more hidden neurons."
    disp('' '"
end

%pause
if flag1==1
    save trfile tr
else
    ttr=tr;
    load trfile
    tr=[tr ttr];
    save trfile tr
end

(13) File name: xtrainbp.m

function [a,b,c,d,e,f,g,h] = xtrainbp (i,j,k,l,m,n,o,p,q,r,s,t)
%XTRAINBP Train feed-forward network with bp + adaptive learning.
% XTRAINBP can be called with 1, 2, or 3 sets of weights
% and biases to train up to 3 layer feed-forward networks.
% [W1,B1,W2,B2,...,TE,TR] = TRAINBPA(W1,B1,F1,W2,B2,F2,...,P,T,TP)
% W1 - SixR weight matrix for the ith layer.
% Bi - SixI bias vector for the ith layer.
% Fi - Transfer function (string) for the ith layer.
% P - RxQ matrix of input vectors.
% T - SxQ matrix of target vectors.
% TP - Training parameters (optional).
% Returns new weights and biases and
% Wi - new weights.
% Bi - new biases.
% TE - the actual number of epochs trained.
% TR - training record: [row of errors]
%
% Training parameters are:
% TP(1) - Epochs between updating display, default = 25.
% TP(2) - Maximum number of epochs to train, default = 1000.
% TP(3) - Sum-squared error goal, default = 0.02.
% TP(4) - Learning rate, 0.01.
% TP(5) - Learning rate increase, default = 1.05.
% TP(6) - Learning rate decrease, default = 0.7.
% TP(7) - Maximum error ratio, default = 1.04.11;1k
% Missing parameters and NaN's are replaced with defaults.

% See also NNTRAIN, BACKPROP, INITFF, SIMFF, TRAINBPX, TRAINLM.

% J. Yang
% Revised.
% 1998
% $Revision: 1.6$

if all([6 7 9 10 12 13] == nargin), error('Wrong number of input arguments'), end

if nargin == 6
   [a,b,c,d] = tbpx1(i,j,k,l,m,[NaN NaN NaN NaN NaN 0 NaN]);
elseif nargin == 7
   n = [n NaN+zeros(1,8) 0];
   len = length(n);
   n = n([1 2 3 4 5 6 len 7]);
   [a,b,c,d] = tbpx1(i,j,k,l,m,n);
elseif nargin == 9
   [a,b,c,d,e,f,ssw1,ssw2] = xtbpx2(i,j,k,l,m,n,o,p,q);%
   % [NaN NaN NaN NaN NaN NaN NaN 0 NaN]);
   g=ssw1;
   h=ssw2;
elseif nargin == 10
   q = [q NaN+zeros(1,8) 0];
   len = length(q);
   q = q([1 2 3 4 5 6 len 7]);
   [a,b,c,d,e,f] = xtbpx2(i,j,k,l,m,n,o,p,q);
elseif nargin == 12
   [a,b,c,d,e,f,g,h] = tbpx3(i,j,k,l,m,n,o,p,q,r,s,...
   [NaN NaN NaN NaN NaN NaN 0 NaN]);
elseif nargin == 13
   t = [t NaN+zeros(1,8) 0];
   len = length(t);
\[ t = t([1 \ 2 \ 3 \ 4 \ 5 \ 6 \ \text{len} \ 7]); \]
\[ [a,b,c,d,e,f,g,h] = \text{tbpx3}(i,j,k,l,m,n,o,p,q,r,s,t); \]
end

(14) File name: xtnbp6.m

function [a,b,c,d,e,f,g,h] = xtnbp6(i,j,k,l,m,n,o,p,q,r,s,t)
% XTRANBP Train feed-forward network with bp + adaptive learning.
% XTRANBP can be called with 1, 2, or 3 sets of weights
% and biases to train up to 3 layer feed-forward networks.
% [W1,B1,W2,B2,...,TE,TR] = TRAINBPA(W1,B1,F1,W2,B2,F2,...,P,T,TP)
% Wi - SxR weight matrix for the ith layer.
% Bi - Sx1 bias vector for the ith layer.
% Fi - Transfer function (string) for the ith layer.
% P - RxQ matrix of input vectors.
% T - SxQ matrix of target vectors.
% TP - Training parameters (optional).
% Returns new weights and biases and
% Wi - new weights.
% Bi - new biases.
% TE - the actual number of epochs trained.
% TR - training record: [row of errors]
% Training parameters are:
% TP(1) - Epochs between updating display, default = 25.
% TP(2) - Maximum number of epochs to train, default = 1000.
% TP(3) - Sum-squared error goal, default = 0.02.
% TP(4) - Learning rate, 0.01.
% TP(5) - Learning rate increase, default = 1.05.
% TP(6) - Learning rate decrease, default = 0.7.
% TP(7) - Maximum error ratio, default = 1.04.
% Missing parameters and NaN’s are replaced with defaults.
% See also NNTRAIN, BACKPROP, INITFF, SIMFF, TRAINBPX, TRAINLM.
% J.J. Yang
% Revised.
% $Revision: 1.2$

if all([6 7 9 10 12 13] == nargin), error('Wrong number of input arguments'), end

if nargin == 6
\[ [a,b,c,d] = \text{tbpx1}(i,j,k,l,m,[\text{NaN NaN NaN NaN NaN NaN 0 NaN}]); \]
elseif nargin == 7
n = [n NaN+zeros(1,8) 0];
len = length(n);
n = n([1 2 3 4 5 6 len 7]);
[a,b,c,d] = \text{tbpx1}(i,j,k,l,m,n);
elseif nargin == 9
\[ [a,b,c,d,e,f,ssw1,ssw2] = \text{xtbpx26}(i,j,k,l,m,n,o,p,q); \]
end

214
% [NaN NaN NaN NaN NaN NaN NaN 0 NaN]);

g=ssw1;
h=ssw2;

elseif nargin == 10
    q = [q NaN+zeros(1,8) 0];
    len = length(q);
    q = q([1 2 3 4 5 6 len 7]);
    [a,b,c,d,e,f] = xtbpx2(i,j,k,l,m,n,o,p,q);

elseif nargin == 12
    [a,b,c,d,e,f,g,h] = xtbpx3(i,j,k,l,m,n,o,p,q,r,s,...
    [NaN NaN NaN NaN NaN NaN NaN 0 NaN]);

elseif nargin == 13
    t = [t NaN+zeros(1,8) 0];
    len = length(t);
    t = t([1 2 3 4 5 6 len 7]);
    [a,b,c,d,e,f,g,h] = xtbpx3(i,j,k,l,m,n,o,p,q,r,s,t);
end

(15)  File name: xtrans.m

% Constraint: Vbias1+VTP=Vbias2+VTN
Vbias1=1.8;
Vbias2=3.2;
VTN=0.7;
VTP=0.7;
I=-100:1:100;
k1=0.01;

for jj=1:4
    k1=k1+(jj-1)*0.004;
    I0=round(((Vbias2-Vbias1)^2/(2*k1));

    V01=(Vbias1+Vbias2)/2-VTN;
    for i=1:(100-I0)
        V(i)=V01-sqrt(k1*(-I(i))-((Vbias2-Vbias1)^2)/4);
    end

    V02=Vbias2-VTN;
    for i=100-I0:101
        V(i)=V02-sqrt(2*k1*(-I(i)));
    end

    V03=Vbias1+VTP;
    for i=101:100+I0
        V(i)=V03+sqrt(2*k1*I(i));
    end

    V04=(Vbias1+Vbias2)/2+VTP;
    for i=100+I0:201
        V(i)=V04+sqrt(k1*I(i)-((Vbias2-Vbias1)^2)/4);
    end

215
VV(jj,:) = V;

end

%alpha = 0.05
%y = 5./(1+exp(-alpha*I));

%plot(I,VV(1,:),'c--',I,VV(2,:),'y-',I,VV(3,:),'r-',I,VV(4,:),'w-')
plot(I,VV(1,:))
VITA AUCTORIS

NAME: Jinming Jim Yang

PLACE OF BIRTH: Jinxi, Liaoning Province, China

YEAR OF BIRTH: 1965

EDUCATION: Liaoning Shi-Yan Middle School, Shenyang, China
1978-1982

Tsinghua University, Beijing, China
1982-1987 B. Sc.

Tsinghua University, Beijing, China
1987-1989 M. A. Sc.

University of Windsor, Windsor, Ontario, Canada
1994-2000 Ph. D.