

Introduction to Artificial Neural Systems

مرکز تحقیقات مخابرات ایران



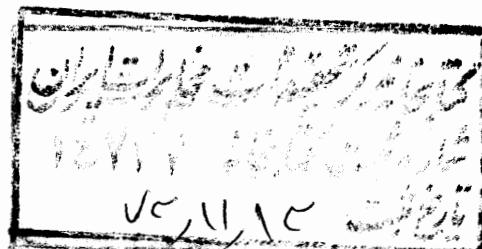
کتابهای لاتین

2 1 4 7 2 7

I.T.R.C.

JACEK M. ZURADA

Professor of Electrical Engineering and of
Computer Science and Engineering



WEST PUBLISHING COMPANY

St. Paul New York Los Angeles San Francisco

QA
76.87
Z87
1992

Text Design: Geri Davis, Quadrata
Composition and Art: Technique Typesetting
Copyediting: Loretta Palagi
Indexing: Barbara Farabaugh

WEST'S COMMITMENT TO THE ENVIRONMENT

In 1906, West Publishing Company began recycling materials left over from the production of books. This began a tradition of efficient and responsible use of resources. Today, up to 95 percent of our legal books and 70 percent of our college texts are printed on recycled, acid-free stock. West also recycles nearly 22 million pounds of scrap paper annually—the equivalent of 181,717 trees. Since the 1960s, West has devised ways to capture and recycle waste inks, solvents, oils, and vapors created in the printing process. We also recycle plastics of all kinds, wood, glass, corrugated cardboard, and batteries, and have eliminated the use of styrofoam book packaging. We at West are proud of the longevity and the scope of our commitment to our environment.

Production, Prepress, Printing and Binding by West Publishing Company.

COPYRIGHT ©1992

By WEST PUBLISHING COMPANY
610 Opperman Drive
P.O. Box 64526
St. Paul, MN 55164-0526

All rights reserved

Printed in the United States of America

99 98 97 96 95 94 93 92

8 7 6 5 4 3 2 1 0

Library of Congress Cataloging-in-Publication Data

Zurada, Jacek M.

Introduction to artificial neural systems / Jacek M. Zurada
p. cm.

Includes index.

ISBN 0-314-93391-3 (alk. paper)

1. Neural networks (Computer science) I. Title

QA76.87.Z87 1992

006.3-dc20

92-712
© CIP

To Anna, Joanna, and Mark



Contents

Preface

1	Artificial Neural Systems: Preliminaries	1
1.1	Neural Computation: Some Examples and Applications	3
	Classifiers, Approximators, and Autonomous Drivers	4
	Simple Memory and Restoration of Patterns	10
	Optimizing Networks	14
	Clustering and Feature Detecting Networks	16
1.2	History of Artificial Neural Systems Development	17
1.3	Future Outlook	21
	References	22

2	Fundamental Concepts and Models of Artificial Neural Systems	25
2.1	Biological Neurons and Their Artificial Models	26
	Biological Neuron	27
	McCulloch-Pitts Neuron Model	30
	Neuron Modeling for Artificial Neural Systems	31
2.2	Models of Artificial Neural Networks	37
	Feedforward Network	37
	Feedback Network	42
2.3	Neural Processing	53
2.4	Learning and Adaptation	55
	Learning as Approximation or Equilibria Encoding	55
	Supervised and Unsupervised Learning	56
2.5	Neural Network Learning Rules	59
	Hebbian Learning Rule	60
	Perceptron Learning Rule	64
	Delta Learning Rule	66
	Widrow-Hoff Learning Rule	69
	Correlation Learning Rule	69
	Winner-Take-All Learning Rule	70
	Outstar Learning Rule	71
	Summary of Learning Rules	72
2.6	Overview of Neural Networks	74
2.7	Concluding Remarks	76
	Problems	78
	References	89
3	Single-Layer Perceptron Classifiers	93
3.1	Classification Model, Features, and Decision Regions	94
3.2	Discriminant Functions	99
3.3	Linear Machine and Minimum Distance Classification	106
3.4	Nonparametric Training Concept	114
3.5	Training and Classification Using the Discrete Perceptron: Algorithm and Example	120
3.6	Single-Layer Continuous Perceptron Networks for Linearly Separable Classifications	132
3.7	Multicategory Single-Layer Perceptron Networks	142

3.8 Concluding Remarks 152

Problems 153

References 161

4 Multilayer Feedforward Networks 163**4.1 Linearly Nonseparable Pattern Classification 165****4.2 Delta Learning Rule for Multiperceptron Layer 175****4.3 Generalized Delta Learning Rule 181****4.4 Feedforward Recall and Error Back-Propagation Training 185**

Feedforward Recall 185

Error Back-Propagation Training 186

Example of Error Back-Propagation Training 190

Training Errors 195

Multilayer Feedforward Networks as
Universal Approximators 196**4.5 Learning Factors 206**

Initial Weights 208

Cumulative Weight Adjustment versus
Incremental Updating 208

Steepness of the Activation Function 209

Learning Constant 210

Momentum Method 211

Network Architectures Versus Data Representation 214

Necessary Number of Hidden Neurons 216

4.6 Classifying and Expert Layered Networks 220

Character Recognition Application 221

Expert Systems Applications 225

Learning Time Sequences 229

4.7 Functional Link Networks 230**4.8 Concluding Remarks 234**

Problems 235

References 248

5 Single-Layer Feedback Networks 251**5.1 Basic Concepts of Dynamical Systems 253****5.2 Mathematical Foundations of Discrete-Time
Hopfield Networks 254**

5.3	Mathematical Foundations of Gradient-Type Hopfield Networks	264
5.4	Transient Response of Continuous-Time Networks	276
5.5	Relaxation Modeling in Single-Layer Feedback Networks	283
5.6	Example Solutions of Optimization Problems	287
	Summing Network with Digital Outputs	287
	Minimization of the Traveling Salesman Tour Length	294
5.7	Concluding Remarks	299
	Problems	301
	References	310
6	Associative Memories	313
6.1	Basic Concepts	314
6.2	Linear Associator	320
6.3	Basic Concepts of Recurrent Autoassociative Memory	325
	Retrieval Algorithm	327
	Storage Algorithm	328
	Performance Considerations	336
6.4	Performance Analysis of Recurrent Autoassociative Memory	339
	Energy Function Reduction	342
	Capacity of Autoassociative Recurrent Memory	343
	Memory Convergence versus Corruption	345
	Fixed Point Concept	349
	Modified Memory Convergent Toward Fixed Points	351
	Advantages and Limitations	354
6.5	Bidirectional Associative Memory	354
	Memory Architecture	355
	Association Encoding and Decoding	357
	Stability Considerations	359
	Memory Example and Performance Evaluation	360
	Improved Coding of Memories	363
	Multidirectional Associative Memory	368
6.6	Associative Memory of Spatio-temporal Patterns	370
6.7	Concluding Remarks	375
	Problems	377
	References	386

7 Matching and Self-Organizing Networks 389

7.1	Hamming Net and MAXNET	391
7.2	Unsupervised Learning of Clusters	399
	Clustering and Similarity Measures	399
	Winner-Take-All Learning	401
	Recall Mode	406
	Initialization of Weights	406
	Separability Limitations	409
7.3	Counterpropagation Network	410
7.4	Feature Mapping	414
7.5	Self-Organizing Feature Maps	423
7.6	Cluster Discovery Network (ART1)	432
7.7	Concluding Remarks	444
	Problems	445
	References	452

8 Applications of Neural Algorithms and Systems 455

8.1	Linear Programming Modeling Network	456
8.2	Character Recognition Networks	464
	Multilayer Feedforward Network for Printed Character Classification	464
	Handwritten Digit Recognition: Problem Statement	476
	Recognition Based on Handwritten Character Skeletonization	478
	Recognition of Handwritten Characters Based on Error Back-propagation Training	482
8.3	Neural Networks Control Applications	485
	Overview of Control Systems Concepts	485
	Process Identification	489
	Basic Nondynamic Learning Control Architectures	494
	Inverted Pendulum Neurocontroller	499
	Cerebellar Model Articulation Controller	504
	Concluding Remarks	511
8.4	Networks for Robot Kinematics	513
	Overview of Robot Kinematics Problems	514
	Solution of the Forward and Inverse Kinematics Problems	516

	Comparison of Architectures for the Forward Kinematics Problem	519
	Target Position Learning	523
8.5	Connectionist Expert Systems for Medical Diagnosis	527
	Expert System for Skin Diseases Diagnosis	528
	Expert System for Low Back Pain Diagnosis	532
	Expert System for Coronary Occlusion Diagnosis	537
	Concluding Remarks	539
8.6	Self-Organizing Semantic Maps	539
8.7	Concluding Remarks	546
	Problems	548
	References	559

9 Neural Networks Implementation 565

9.1	Artificial Neural Systems: Overview of Actual Models	566
	Node Numbers and Complexity of Computing Systems	567
	Neurocomputing Hardware Requirements	569
	Digital and Analog Electronic Neurocomputing Circuits	575
9.2	Integrated Circuit Synaptic Connections	585
	Voltage-controlled Weights	587
	Analog Storage of Adjustable Weights	592
	Digitally Programmable Weights	595
	Learning Weight Implementation	605
9.3	Active Building Blocks of Neural Networks	608
	Current Mirrors	610
	Inverter-based Neuron	613
	Differential Voltage Amplifiers	617
	Scalar Product and Averaging Circuits with Transconductance Amplifiers	624
	Current Comparator	626
	Template Matching Network	628
9.4	Analog Multipliers and Scalar Product Circuits	630
	Depletion MOSFET Circuit	631
	Enhancement Mode MOS Circuit	636
	Analog Multiplier with Weight Storage	638
	Floating-Gate Transistor Multipliers	640

9.5	Associative Memory Implementations	644
9.6	Electronic Neural Processors	652
9.7	Concluding Remarks	663
	Problems	666
	References	679

Appendix

A1	Vectors and Matrices	A1
A2	Quadratic Forms and Definite Matrices	A10
A3	Time-Varying and Gradient Vectors, Jacobian and Hessian Matrices	A13
A4	Solution of Optimization Problems	A17
A5	Stability of Nonlinear Dynamical Systems	A21
A6	Analytic Geometry in Euclidean Space in Cartesian Coordinates	A29
A7	Selected Neural Network Related Concepts	A38
A8	MOSFET Modeling for Neural Network Applications	A39
A9	Artificial Neural Systems (ANS) Program Listing	A43

Index

Preface

MOTIVATION FOR THE BOOK

The recent resurgence of interest in neural networks has its roots in the recognition that the brain performs computations in a different manner than do conventional digital computers. Computers are extremely fast and precise at executing sequences of instructions that have been formulated for them. A human information processing system is composed of neurons switching at speeds about a million times slower than computer gates. Yet, humans are more efficient than computers at computationally complex tasks such as speech understanding. Moreover, not only humans, but even animals, can process visual information better than the fastest computers.

The question of whether technology can benefit from emulating the computational capabilities of organisms is a natural one. Unfortunately, the understanding of biological neural systems is not developed enough to address the issues of functional similarity that may exist between the biological and man-made neural

systems. As a result, any major potential gains derived from such functional similarity, if they exist, have yet to be exploited.

This book introduces the foundations of artificial neural systems. Much of the inspiration for such systems comes from neuroscience. However, we are not directly concerned with networks of biological neurons in this text. Although the newly developed paradigms of artificial neural networks have strongly contributed to the discovery, understanding, and utilization of potential functional similarities between human and artificial information processing systems, many questions remain open. Intense research interest persists and the area continues to develop. The ultimate research objective is the theory and implementation of massively parallel interconnected systems which could process the information with an efficiency comparable to that of the brain.

To achieve this research objective, we need to define the focus of the study of artificial neural systems. *Artificial neural systems, or neural networks, are physical cellular systems which can acquire, store, and utilize experiential knowledge.* The knowledge is in the form of stable states or mappings embedded in networks that can be recalled in response to the presentation of cues. This book focuses on the foundations of such networks. The fundamentals of artificial neural systems theory, algorithms for information acquisition and retrieval, examples of applications, and implementation issues are included in our discussion. We explore a rather new and fundamentally different approach to computing and information processing.

Programmed computing, which has dominated information processing for more than four decades, is based on decision rules and algorithms encoded into the form of computer programs. The algorithms and program-controlled computing necessary to operate conventional computers have their counterparts in the learning rules and information recall procedures of a neural network. These are not exact counterparts, however, because neural networks go beyond digital computers since they can progressively alter their processing structure in response to the information they receive.

The purpose of this book is to help the reader understand the acquisition and retrieval of experiential knowledge in densely interconnected networks containing cells of processing elements and interconnecting links. Some of the discussed networks can be considered adaptive; others acquire knowledge *a priori*. Retrieval of knowledge is termed by some authors as neural computation. However, "neural computation" is only a segment of the artificial neural systems focus. This book also addresses the concepts of parallel machines that are able to acquire knowledge, and the corresponding issues of implementation.

I believe that the field of artificial neural systems has evolved to a point where a course on the subject is justified. Unfortunately, while the technical literature is full of reports on artificial neural systems theories and applications, it is hard to find a complete and unified description of techniques. The beginning student is likely to be bewildered by different levels of presentations, and

a widespread spectrum of metaphors and approaches presented by authors of diverse backgrounds.

This book was conceived to satisfy the need for a comprehensive and unified text in this new discipline of artificial neural systems. It brings students a fresh and fascinating perspective on computation. The presentation of the material focuses on basic system concepts and involves learning algorithms, architectures, applications, and implementations. The text grew out of the teaching effort in artificial neural systems offered for electrical engineering and computer science and engineering senior and graduate students at the University of Louisville. In addition, the opportunity to work at Princeton University has considerably influenced the project.

This book is designed for a one semester course. It is written at a comprehensible level for students who have had calculus, linear algebra, analytical geometry, differential equations, and some exposure to optimization theory. As such, the book is suitable for senior-level undergraduate or beginning graduate students. Due to the emphasis on systems, the book should be appropriate for electrical, computer, industrial, mechanical, and manufacturing engineering students as well as for computer and information science, physics, and mathematics students. Those whose major is not electrical or computer engineering may find Sections 9.2 to 9.5 superfluous for their study. They can also skip Sections 5.4 to 5.6 without any loss of continuity.

With the mathematical and programming references in the Appendix, the book is self-contained. It should also serve already-practicing engineers and scientists who intend to study the neural networks area. In particular, it is assumed that the reader has no experience in neural networks, learning machines, or pattern recognition.

PHILOSOPHY

The fundamentals of artificial neural systems are the main concern of this book. Concepts are introduced gradually. The basic neural network paradigms are conveyed using a system perspective and mathematics exposition suitable for thorough understanding. Although only some of the algorithms are provided with proofs, the book uses mathematical exposition at the depth essential for artificial neural systems implementation and simulation, and later, initiation of research in the area. The reader is not only shown how to get a result, but also how to think about and possibly extend it.

Neural network processing typically involves dealing with large-scale problems in terms of dimensionality, amount of data handled, and the volume of simulation or neural hardware processing. This large-scale approach is both essential and typical for real-life applications. Although the study of such applications

show dramatically *what* has been done, a student needs to find out *how* and *why* it has happened. Peeling off the complexity of large-scale approaches, diversity of applications, and overlapping aspects can facilitate the study.

When serious readers look behind the easy articles on neural networks in popular books or magazines, they may notice an involved mosaic of theories taken from mathematics, physics, neurobiology, and engineering. Fortunately, there are unified and pedagogical approaches to studying such complex subjects as neural networks. In order to understand digital computer organization and architecture, for example, one needs to study basic axioms of Boolean algebra, binary arithmetic, sequential circuits, control organization, microprogramming, etc. A similar decomposition of the neural network knowledge domain into basic but coherent parts should simplify the study. Such an approach is employed in this text.

An additional difficulty in studying the field arises from the large number of publications appearing in the neural network field. They make use of a variety of mathematical approaches, notations, and terminologies. Authors with diverse backgrounds bring a myriad of perspectives from numerous disciplines, ranging from neurobiology and physics to engineering and computer science. While the field has benefitted from having experts with a variety of backgrounds and perspectives, the resulting differences in approach have made it difficult to see the underlying similarities of substance. To blend interdisciplinary aspects of the discipline, I present a unified perspective, and also link the approaches and terminologies among them.

As the reader will hopefully notice, the fundamental concepts of neural processing are often rather lucid. Because they combine simplicity and power, they are also appealing. The approach often taken in this text is to reduce both high dimensionality and data volume whenever possible without loss of generality, and to demonstrate algorithms in two- or three-dimensional space and with a small amount of data. An algorithm is often introduced along with an explanatory example showing how and why it works. Later on, the algorithm is formally outlined and formulated for any size of the problem. Some proofs of algorithms are included; others are omitted for simplicity. Some proofs of neural algorithms, however, have yet to be constructed.

Examples within chapters usually follow this philosophy of material presentation. I feel strongly that illustrative examples need to be made available to readers to develop intuitive understanding and to facilitate study. The end-of-chapter problems typically also have a reduced size and complexity. They focus on enhancing the understanding of principles; some of them are short proofs. I also believe that solving problems should enhance the understanding of algorithm principles and related concepts. Since the use of existing algorithms does not contribute much to the understanding of the subject, only a limited number of end of chapter problems is devoted to the “simulate and see” approach.

At the same time, I realize that the practical significance of neural computation becomes apparent for large or very large-scale tasks. To illustrate neural networks practical applications, computationally intensive problems are also included at the ends of most chapters. Such problems are highlighted with asterisks and can be considered mini-projects. Their size is usually between the "pencil and paper" problem size and the large-scale application. Because some of these problems are rather computationally intensive, the coding of algorithms or the use of the ANS program (available with this text) or other software is recommended for their solution. Unless software is available, the mini-project problems usually take much more time to solve than other problems.

Problems marked with asterisks also tend to overlap somewhat throughout the chapters with regard to input data, training sets, and similar aspects, but the algorithms used differ among problems in various chapters. As the course has been developed, students should have created an integrated program package for selected neural algorithms by the end of a semester. The input of the package includes user-specified problem size, choice of an algorithm, entering training data, training the network, and commands for running the program for test data in the recall mode. Obviously, neither the package code nor the applications of the algorithms are optimized, but they are not meant to be, since they address a spectrum of neurocomputing paradigms for project work.

Rather than focusing on a number of algorithms, another alternative for project content of the course is proposed. Students can be assigned one large-scale project, which can be the outgrowth, or fusion, of the computationally intensive end-of-chapter problems. In this mode, instead of using the spectrum of several algorithms, one neural processing algorithm is covered in depth. Alternatively, one practical application problem can be studied and approached using more than one paradigm. Study of the literature referenced at the end of each chapter, or extension of applications covered in Chapter 8, can provide the reader with more guidelines for further work.

SUMMARY BY CHAPTERS

Chapter 1 presents a number of neural network examples and applications. It provides an overview of what neural networks can accomplish. Brief historical remarks of artificial neural network development are presented, followed by a future outlook of the discipline.

Chapter 2 introduces the terminology and discusses the most important neural processing paradigms, network properties, and learning concepts. Basic assumptions, definitions, and models of neurons and neural architectures are introduced in this chapter. The chapter concludes with the analysis of networks and their learning rules.

The foundations of supervised learning machines are presented in Chapter 3. Feedforward single-layer architectures are studied. The concept of the nonparametric training of a linear machine composed of discrete perceptrons is introduced. The training is discussed in geometric terms, and the steepest descent minimization of the classification error function is accomplished for a network with continuous perceptrons.

The concept of error function minimization is extended in Chapter 4 to multilayer feedforward neural networks. Emphasis is placed on the pattern-to-image space mapping provided through the first network layer. The generalized delta learning rule is derived based on the steepest descent minimization rule. The error back-propagation algorithm is discussed and illustrated with examples. Application examples of multilayer feedforward networks are also covered. The chapter ends with presentation of functional link networks.

Chapter 5 covers the foundations of recurrent networks. Discrete-time and continuous-time gradient-type networks are introduced. The time-domain behavior of single-layer feedback networks is emphasized. Particular attention is paid to the energy minimization property during the evolution of the system in time, and to the use and interpretation of trajectories in the output space. A comprehensive example involving analog-to-digital converters and an optimization problem solving using gradient-type networks are presented to illustrate the theory.

Associative memories are introduced in Chapter 6. The exposition proceeds from the linear associator to single-layer recurrent networks. Two-layer recurrent networks, known as bidirectional associative memories, are introduced. They are then generalized to perform as multidirectional and spatio-temporal memories. Performance evaluation of single- and double-layer memory is covered and illustrated with examples. Memory capacity and retrieval performance versus distances between stored patterns are discussed.

Chapter 7 covers networks that for the most part learn in an unsupervised environment. These networks typically perform input vector matching, clustering or feature extraction. Diverse architectures covered in this chapter include the Hamming networks, MAXNET, clustering and counterpropagation networks, self-organizing feature mapping networks, and adaptive resonance networks.

Chapter 8 provides examples of neural network applications for computation, control, and information processing. An optimization network for a linear programming task is discussed. Character recognition architectures using hidden-neuron layers are described, and applications in robotics and control systems are reviewed. Other application examples treat the implementation of connectionist expert systems for medical diagnosis. Such systems first acquire and then use the knowledge base to diagnose a patient's disease when provided with a list of symptoms. The application of a self-organizing feature mapping network for semantic map processing is covered.

The final chapter, Chapter 9, is devoted to hardware implementation issues. A brief review of digital neurocomputing and simulation issues is followed by an

extensive discussion of microelectronic implementation techniques. The emphasis of this chapter is on adaptive analog and learning circuit configurations which can be implemented in MOS technology.

The Appendix contains essential mathematical references needed for the understanding of neural network paradigms covered in the book. It also contains the listing of the main procedures of the program ANS in Pascal, which is capable of running on IBM-PC compatible microcomputers. The program implements most computational algorithms discussed in the book, and can provide numerical solutions to a number of examples and problems contained in the text.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my wife, Anna, for her encouragement and cooperation during the time I spent writing the book. I was introduced to the subject in the outstanding lectures of Professor J. Seidler at the Technical University of Gdansk, Poland, who had an early influence on this project. Also, I am indebted to a number of individuals who, directly or indirectly, assisted in the preparation of the text. In particular, I wish to extend my appreciation to Professors M. Bialko of the Technical University of Gdansk, Poland, D. Mlynski of the Technical University of Karlsruhe, Germany, M. H. Hassoun of Wayne State University, S. K. Park of Tennessee Technical University, B. Dickinson of Princeton University; J. Starzyk of Ohio University; R. W. Newcomb of the University of Maryland; and P. B. Aronhime and J. H. Lilly of the University of Louisville whose criticism, comments, and suggestions have been very helpful. The list of individuals who have contributed to the development of this text include graduate students whose effort I gratefully acknowledge. I also wish to thank Diane S. Eiland for her commitment to and skillful effort of processing the manuscript in many iterations.

I would like to thank Peter Gordon, Engineering Editor at West Publishing, for sharing his enthusiasm and supporting the evolving project.

I also thank Sharon Adams, Developmental Editor, Mélina Brown, Production Editor, and Jo Anne Bechard at West Publishing.

The following colleagues were extremely helpful in offering comments and suggestions:

James Anderson
Brown University

Donald Glaser
University of California-Berkeley

Leonard Myers
California Polytechnic University-San Luis Obispo

Bahram Nabet
Drexel University
Edgar Sanchez-Sinencio
Texas A&M University
Gene A. Tagliarini
Clemson University
John Uhran
University of Notre Dame.

Jacek M. Zurada

Notation and Abbreviations

ANS	— artificial neural systems simulation program from appendix page A9
ART	— adaptive resonance theory
BAM	— bidirectional associative memory
C	— capacitance matrix of Hopfield network (diagonal)
CPS	— connections per second
d	— desired output vector of a trained network
D	— digital control word
$\det \mathbf{A}$	— determinant of matrix \mathbf{A}
e	— error
E	— error function to be minimized during learning (for feedforward networks)
E	— energy function to be minimized during recall (for feedback networks)
EBPTA	— (E can be termed as a loss, or an objective function)
$f(\cdot)$	— error back-propagation training algorithm
$f(\text{net})$	— neuron's activation operator
$f^{(i)}$	— activation function
$f_{\text{sat}+}, f_{\text{sat}-}$	— memory forced response vector
F	— operational amplifier saturation voltages
\mathbf{F}	— memory forced response matrix
$g(\mathbf{x})$	— discriminant function
G	— conductance matrix of Hopfield network (diagonal)
$h(\mathbf{x})$	— multivariable function to be approximated
\mathbf{H}	— Hessian matrix (also ∇^2)
HD	— Hamming Distance
$H(\mathbf{w}, \mathbf{x})$	— neural network-produced approximation of function $h(\mathbf{x})$
$i_o(\mathbf{x})$	— decision function
\mathbf{i}	— network input current vector (bias vector)
I	— number of input nodes of multilayer feedforward network
\mathbf{I}	— identity matrix
I_+	— excitatory current
I_-	— inhibitory current
I_{ds}	— drain-to-source current
J	— number of hidden nodes of multilayer feedforward network
\mathbf{J}	— Jacobian matrix
k	— device transconductance of MOS transistor
k'	— process transconductance of MOS transistor
K	— number of output nodes of multilayer feedforward network
KCL	— Kirchhoff's current law
L	— channel length of MOS transistor

MDAC	— multiplying digital-to-analog converter
n	— normal vector to hyperplane
net	— activation vector for neuron layer (general notation)
<i>net</i>	— activation value of a neuron
o	— output vector of a neuron layer
P	— number of patterns in the training set
r	— learning signal
r	— unit normal vector to hyperplane
RAMSRA	— recurrent associative memory storage and retrieval algorithm
RDPTA	— <i>R</i> -category discrete perceptron training algorithm
R_{ds}	— tunable weight resistance produced by MOS transistor
$r(t)$	— reference signal (Chapter 8)
$\text{sgn}(x)$	— signum function
$s^{(i)}$	— vector stored in memory (or classifier)
S	— vectors stored in memory, arranged in matrix
SCPTA	— single continuous perceptron training algorithm
SDPTA	— single discrete perceptron training algorithm
T	— neuron (or perceptron) threshold
TLU	— threshold logic unit
u	— activation vector (only for Hopfield layer)
v	— output vector of continuous-time feedback or memory network
V	— weight matrix (v_{ij} is the weight from node <i>j</i> toward node <i>i</i>)
V_{ds}, V_{gs}	— drain-to-source and gate-to-source voltage of MOS transistor, respectively
v_o	— output voltage
V_{th}, V_{dep}	— threshold, depletion voltages of MOS transistor, respectively
VTC	— voltage transfer characteristics
w	— weight vector (or augmented weight vector)
W	— channel width of MOS transistor
W	— weight matrix (w_{ij} is the weight from node <i>j</i> toward node <i>i</i>)
w'	— updated weight vector (or augmented weight vector)
\hat{w}	— normalized weight vector
x	— input pattern vector
y	— input vector, or augmented input vector, also output of hidden layer neuron
y^k	— matching score at input of MAXNET
z	— input vector for multilayer network

Greek

α	— learning constant
β	— learning constant

$\Gamma[\cdot]$	— nonlinear diagonal matrix operator with operators $f(\cdot)$ on the diagonal
δ	— error signal vector
δ_y	— error signal vector of hidden layer
δ_o	— error signal vector of output layer
δ_{ij}	— Kronecker delta function
η	— learning constant
η	— memory output noise vector
θ_1, θ_2	— joint angles of planar robot manipulator
λ	— gain (steepness) factor of the continuous activation function, also used in Chapter 9 to denote the channel length modulation factor
ρ	— vigilance test level (Chapter 7) (also used as measure of distance in Chapter 2, and as radius of attraction Chapter 6)
$\nabla_x E(\mathbf{x})$	— gradient of a multivariable function $E(\mathbf{x})$ with respect to \mathbf{x}
$\nabla_x^2 E(\mathbf{x})$	— Hessian matrix of a multivariable function $E(\mathbf{x})$ with respect to \mathbf{x}

ARTIFICIAL NEURAL SYSTEMS: PRELIMINARIES

Happy those who are convinced so as to be of the general opinions.

LORD HALIFAX

- 1.1 Neural Computation: Some Examples and Applications
- 1.2 History of Artificial Neural Systems Development
- 1.3 Future Outlook

For many centuries, one of the goals of humankind has been to develop machines. We envisioned these machines as performing all cumbersome and tedious tasks so that we might enjoy a more fruitful life. The era of machine making began with the discovery of simple machines such as lever, wheel and pulley. Many equally congenial inventions followed thereafter. Nowadays engineers and scientists are trying to develop intelligent machines. Artificial neural systems are present-day examples of such machines that have great potential to further improve the quality of our life.

As mentioned in the preface, people and animals are much better and faster at recognizing images than most advanced computers. Although computers outperform both biological and artificial neural systems for tasks based on precise and fast arithmetic operations, artificial neural systems represent the promising new generation of information processing networks. Advances have been made in applying such systems for problems found intractable or difficult for traditional computation. Neural networks can supplement the enormous processing power

of the von Neumann digital computer with the ability to make sensible decisions and to learn by ordinary experience, as we do.

A neural network's ability to perform computations is based on the hope that we can reproduce some of the flexibility and power of the human brain by artificial means. Network computation is performed by a dense mesh of computing nodes and connections. They operate collectively and simultaneously on most or all data and inputs. The basic processing elements of neural networks are called *artificial neurons*, or simply *neurons*. Often we simply call them *nodes*. Neurons perform as summing and nonlinear mapping junctions. In some cases they can be considered as threshold units that fire when their total input exceeds certain bias levels. Neurons usually operate in parallel and are configured in regular architectures. They are often organized in layers, and feedback connections both within the layer and toward adjacent layers are allowed. Each connection strength is expressed by a numerical value called a *weight*, which can be modified.

We have just introduced several new words describing some of the attributes of neural processing. These, and other notions introduced later in this chapter, are not yet rigorously defined. For now we will use them to discuss qualitatively several introductory examples and applications of neurocomputing. A more detailed exposition of concepts and terminology is provided in Chapter 2. Let us only mention that the artificial neural systems field alone goes under the guise of many names in the literature. The effort is not only called neural networks, but also neurocomputing, network computation, connectionism, parallel distributed processing, layered adaptive systems, self-organizing networks, or neuromorphic systems or networks. The reader may not need to assimilate all new technical jargon terms related to neurocomputing. Rather, the variety of names indicates how many different perspectives can be taken when studying a single subject of interest such as neural networks.

Artificial neural systems function as parallel distributed computing networks. Their most basic characteristic is their architecture. Only some of the networks provide instantaneous responses. Other networks need time to respond and are characterized by their time-domain behavior, which we often refer to as *dynamics*. Neural networks also differ from each other in their learning modes. There are a variety of learning rules that establish when and how the connecting weights change. Finally, networks exhibit different speeds and efficiency of learning. As a result, they also differ in their ability to accurately respond to the cues presented at the input.

In contrast to conventional computers, which are programmed to perform specific tasks, most neural networks must be taught, or trained. They learn new associations, new patterns, and new functional dependencies. *Learning corresponds to parameter changes*. As will be shown in subsequent chapters, learning rules and algorithms used for experiential training of networks replace the programming required for conventional computation. Neural network users do not specify an algorithm to be executed by each computing node as would programmers of a more traditional machine. Instead, they select what in their view is the

best architecture, specify the characteristics of the neurons and initial weights, and choose the training mode for the network. Appropriate inputs are then applied to the network so that it can acquire knowledge from the environment. As a result of such exposure, the network assimilates the information that can later be recalled by the user.

Artificial neural system computation lies in the middle ground between engineering and artificial intelligence. The mathematical techniques utilized, such as mean-square error minimization, are engineering-like, but an experimental *ad hoc* approach is often necessary. Heuristic methodology and “quasi-rigorous” techniques of network learning are needed, since often no theory is available for selection of the appropriate neural system for a specific application.

Much of this book is devoted to answering questions, such as the following: How can a network be trained efficiently and why does it learn? What models of neurons need to be used for good performance? What are the best architectures for certain classes of problems? What are the best ways of extracting the knowledge stored in a network? What are the typical applications of neural computation and how accurate are they? How can a specific task be accomplished if it is modeled through a neural network? How can artificial neural systems be implemented as application-specific integrated circuits? We realize that our main objective of study is to design efficient neural systems. However, a strong understanding of the principles and behavior of networks is invaluable if a good design is to be developed.

Neural networks have attracted the attention of scientists and technologists from a number of disciplines. The discipline has attracted neuroscientists who are interested in modeling biological neural networks, and physicists who envisage analogies between neural network models and the nonlinear dynamical systems they study. Mathematicians are fascinated by the potential of mathematical modeling applied to complex large systems phenomena. Electrical and computer engineers look at artificial neural systems as computing networks for signal processing. They are also interested in building electronic integrated circuit-based intelligent machines. Psychologists look at artificial neural networks as possible prototype structures of human-like information processing. Finally, computer scientists are interested in opportunities that are opened by the massively parallel computational networks in the areas of artificial intelligence, computational theory, modeling and simulation, and others.

1.1

NEURAL COMPUTATION: SOME EXAMPLES AND APPLICATIONS

In the previous section we read about interesting and novel aspects of neural computation. We now focus on the applicability and usefulness of neural network

technology in the solving of real-world problems. Below, we demonstrate sample applications and relate them to simplified neural network examples. Because mathematical exposition and more precise explanation of concepts are addressed starting in Chapter 2, this discussion is descriptive rather than quantitative.

Classifiers, Approximators, and Autonomous Drivers

This portion of our neural networks discussion introduces networks that respond instantaneously to the applied input. Let us try to inspect the performance of a simple *classifier*, and then also of similar networks that are far more complex, but can be designed as extensions of the classifier.

Assume that a set of eight points, P_0, P_1, \dots, P_7 , in three-dimensional space is available. The set consists of all vertices of a three-dimensional cube as follows:

$$\{P_0(-1, -1, -1), P_1(-1, -1, 1), P_2(-1, 1, -1), P_3(-1, 1, 1), \\ P_4(1, -1, -1), P_5(1, -1, 1), P_6(1, 1, -1), P_7(1, 1, 1)\}$$

Elements of this set need to be classified into two categories. The first category is defined as containing points with two or more positive ones; the second category contains all the remaining points that do not belong to the first category. Accordingly, points P_3, P_5, P_6 , and P_7 belong to the first category, and the remaining points to the second category.

Classification of points P_3, P_5, P_6 , and P_7 can be based on the summation of coordinate values for each point evaluated for category membership. Notice that for each point $P_i(x_1, x_2, x_3)$, where $i = 0, \dots, 7$, the membership in the category can be established by the following calculation:

$$\text{If } \operatorname{sgn}(x_1 + x_2 + x_3) = \begin{cases} 1, & \text{then category 1} \\ -1, & \text{then category 2} \end{cases} \quad (1.1)$$

Expression (1.1) describes the decision function of the classifier designed by inspection of the set that needs to be partitioned. No design formulas or training have been required to accomplish the design, however. The resulting neural network shown in Figure 1.1(a) is extremely simple. It implements expression (1.1) by summing the inputs x_1, x_2 , and x_3 with unity weighting coefficients. The weighted sum undergoes a thresholding operation and the output of the unit is 1 for $x_1 + x_2 + x_3 > 0$, otherwise it is -1. We thus achieve the desired classification using a single *unit*, or computing node. The unit implements summation with respective weights followed by the thresholding operation.

Looking at the network just discussed and taking a different perspective, we may notice that in addition to classification, the network performs three-dimensional Cartesian space partitioning as illustrated in Figure 1.1(b). The

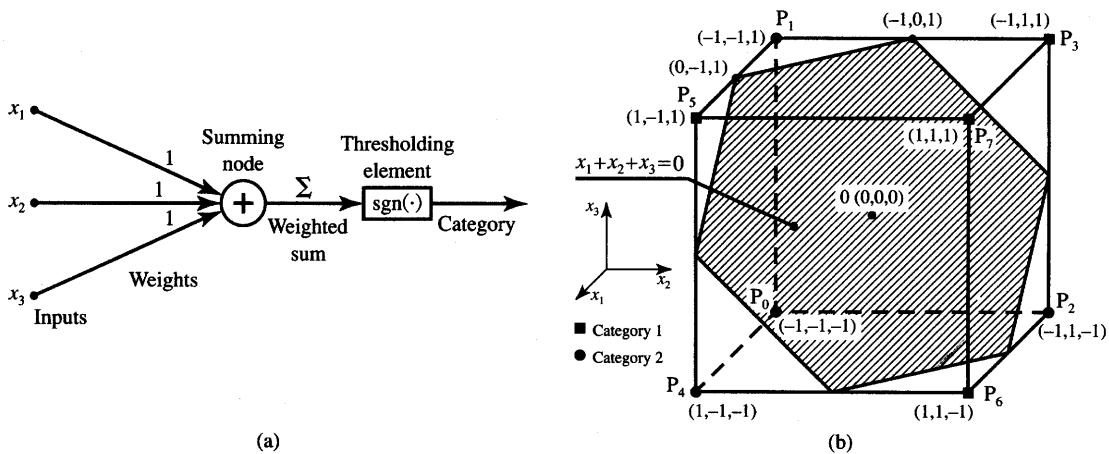


Figure 1.1 Partitioning of the set of cube vertices: (a) single-unit diagram and (b) implemented partitioning.

partitioning plane is $x_1 + x_2 + x_3 = 0$. All points above it are mapped into $+1$ and all below it are mapped into -1 . Although this observation is not crucial for our understanding of classifier design, insight into the geometry of mapping will prove to be very valuable for more complex cases.

Design of neural network classifiers becomes far more involved and intriguing when requirements for membership in categories become complicated. No single-unit classifier exists, for example, that would implement assignment of P_2 , P_3 , and P_5 into a single category, and of the remaining five points of the set into the other category. The details of the classification discussion, however, will be postponed until Chapter 3, since they require more formal coverage.

As stated, the unit from Figure 1.1(a) maps the entire three-dimensional space into just two points, 1 and -1 . A question arises as to whether a unit with a “squashed” sgn function rather than a regular sgn function could prove more advantageous. Assuming that the “squashed” sgn function has the shape as in Figure 1.2, notice that now the outputs take values in the range $(-1, 1)$ and are generally more discernible than in the previous case. Using units with continuous characteristics offers tremendous opportunities for new tasks that can be performed by neural networks. Specifically, the fine granularity of output provides more information than the binary ± 1 output of the thresholding element.

An important new feature of networks with units having continuous characteristics is that they can be trained independent of their architecture. This has not been the case for many networks with units implementing a sgn function. The principles of this training are explained beginning in Section 2.4. Meanwhile, we concentrate on what trained networks could compute for us, rather than how to

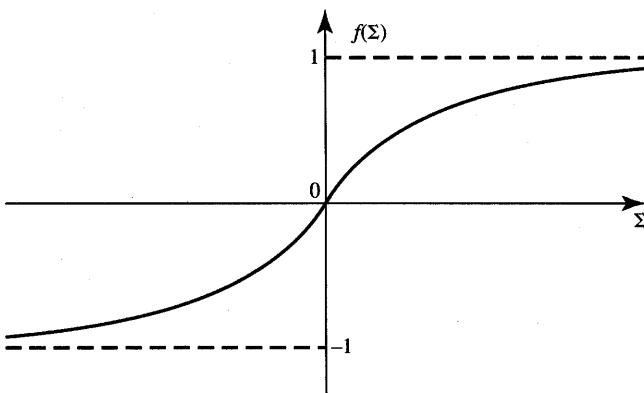


Figure 1.2 “Squashed” sgn function.

design them. As an example, let us look at a neural network classifier applied to the solution of a real-world problem of biomedical signal processing.

Neurologists often need to evaluate the electroencephalogram (EEG) signals of a patient. The EEG waveforms contain important information about the patient's abnormal brain function. The interpretation of EEG signals is used by neurologists when diagnosing the risk of seizures and deciding on corrective measures. The recordings are taken from the patient's scalp and are multichannel. Up to 64 electrodes are used to monitor fully electrical impulses generated by the patient's brain.

Evaluation of the multichannel EEG data is rather difficult and therefore done most often by qualified neurologists. In addition, large amounts of data must be handled because of the round-the-clock monitoring of each patient. To reduce the costs of monitoring and data interpretation, the EEG waves need to be processed by a computing device that provides on-line detection of signals of interest. One of the most important signal shapes that needs to be discriminated is an EEG spike which indicates an imminent epileptic seizure.

EEG spike detection using a neural network classifier was recently developed (Eberhart and Dobbins 1990). Data are monitored in four channels of interest. The EEG waves are sampled 200 or 250 times per second within a 240-ms time window. This yields 48 or 60 data samples available from each channel to be evaluated. The data samples from each channel are fed to the neural network of interconnected units with squashed characteristics as in Figure 1.2. The total of 41 units arranged in three layers does the processing of the data. Two output units are provided for the purpose of spike identification. When the first output unit responds close to unity, this identifies a spike; when the second output yields high response, the network identifies the present input as nonspike.

The network was designed by a team of signal processing engineers who developed the system and the design methodology, and by a team of four to

six neurologists who provided expert identification of spikes and nonspikes. The network was extensively trained using both spike and nonspike data.

The test results for the trained network are impressive. Six series of comprehensive tests were reported. First, the network was tested with spike/nonspike waves used earlier for training. All spikes used for training were positively identified by the network. Only 2 nonspikes of the total of 260 nonspikes were classified mistakenly as spikes. Following the test of the classification with the previously used training data, the classification of entirely new spike/nonspike wave data was performed. Again, all spikes were correctly identified. However, there were also some new nonspikes incorrectly classified by the network as spikes. Despite some number of false alarms, the performance of the network has been found to be significantly better than that required for practical application in hospitals.

While classification of input data is of tremendous importance and will be revisited in more detail in subsequent chapters, let us turn our attention to other forms of mapping. An example of function approximation performed by a neural network is shown in Figure 1.3(a). The network's input is a scalar variable x , and its output is a scalar function $o(x)$. It can be seen from the figure that each of the n units operates with input that is equal to weighted input x plus individual bias that enters each of the summing nodes. Each of the n nonlinear mapping units produces the output. The outputs are then weighted and summed again. Then the nonlinear mapping is performed by the output unit and the network implements the function $o(x)$. The function $o(x)$ depends on a number of weights and biases as marked on the figure.

Notice that the network configuration shown in Figure 1.3(a) can be used to approximate a function. Figure 1.3(b) illustrates a specific example of approximation. The dashed line denotes the function that needs to be approximated and is known at P example points only. The exact function is

$$h(x) = 0.8 \sin \pi x, \quad -1 \leq x \leq 1 \quad (1.2)$$

Choosing $P = 21$, we obtain the data for function approximation as shown by the circles on the dashed curve.

Approximation of function $h(x)$ as in Equation (1.2) through function $o(x)$ can be implemented using the network from Figure 1.3(a) for $n = 10$. Parameters of the neural network have been found as a result of the training using 21 points shown as training data. The reader may notice that the number of parameters for the discussed network is 31, with 20 of them being weights and 11 being bias levels. The mapping $o(x)$ performed by the network is shown by a continuous line. Although the two functions $h(x)$ and $o(x)$ do not exactly overlap, the example shows the potential that exists for function approximation using neural networks.

While neural networks for function approximation seem useful for the purpose of mathematical modeling, we may need a more convincing argument for how networks of this class may be used in practical applications. A sample

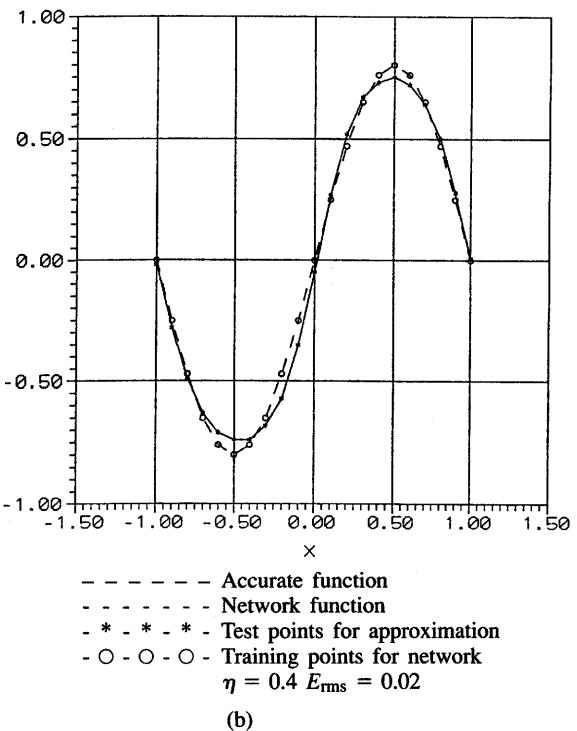
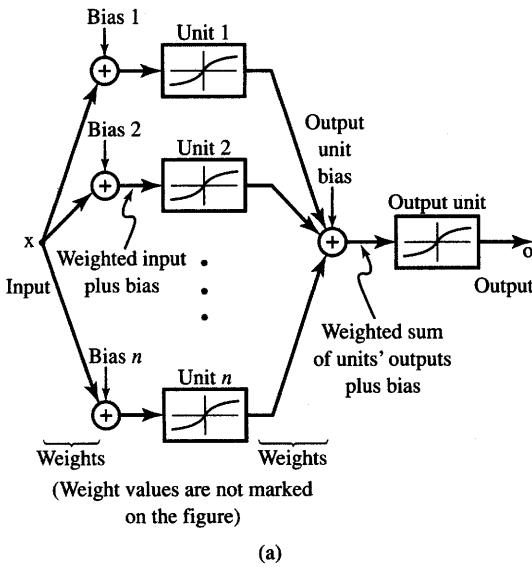


Figure 1.3 Neural network with continuous units as function approximator: (a) block diagram and (b) example approximation of Equation (1.2).

network designed for the task of road following can offer us additional insight into the applied nature and potential of artificial neural systems. Consider, for example, ALVINN (Autonomous Land Vehicle In a Neural Network) project reported by Pomerleau (1989). The ALVINN network takes road images from a camera and a laser range finder as input. It produces as output the direction the vehicle should travel in order to follow the road. Let us look at some interesting details of this system.

The architecture of ALVINN is shown in Figure 1.4. Inputs to the network consist of video and range information. The video information is provided by the 30×32 retina, which depicts the road scene. The resulting 960 segments of the scene are each coded into input proportional to the intensity of blue color. The blue band of the color image is used because it provides the highest contrast between the road and nonroad. The distance information is available at the input from a second retina consisting of 8×32 units and receiving the signal from a laser range finder. The resulting 256 inputs are each coded proportionally to the proximity of the corresponding area in the image. An additional single unit is made available in the network to indicate whether the road is lighter or darker than the nonroad in the previous image.

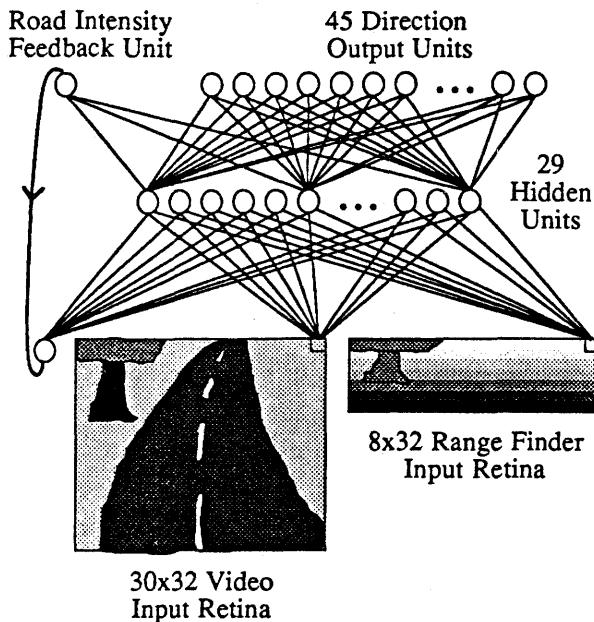


Figure 1.4 Architecture of autonomous vehicle driver. SOURCE: (Pomerleau 1989) © Morgan-Kaufmann; reprinted with permission.

The 1217 inputs are fed to the layer consisting of 29 units with squashed sgn characteristics. Each of the 29 units is connected to each of the 46 output units. A single output unit provides feedback input information about the road intensity, and the remaining 45 units are directional. The middle unit indicating high output gives the signal to drive straight ahead. Units to the left and right represent the curvature of the necessary turn that the vehicle should perform to stay at the road center. The extreme left and right units correspond to the sharpest left and right turns, respectively.

ALVINN has been developed using computer-generated road images. The images have been used as training exemplars. The training set of different road images has involved 1200 different combinations of scenes, curvatures, lighting conditions and distortion levels. The actual neural network driver has been implemented using an on-board computer and a modified Chevy van. Performance of the network as reported by Pomerleau (1989) has been comparable to that achieved by the best traditional vision-based navigation systems evaluated under similar conditions.

It seems that the salient image features important for accurate driving have been acquired by ALVINN not from the programmer or the algorithm, but from the data environment presented to the network. As a result, the system mastered the art of driving within a half-hour training session. It would take

many months of algorithm development and parameter tuning related to vision and pattern recognition work to develop a conventional autonomous driver of comparable efficiency.

Interesting observations were reported following the development of the project. The network performed much better after it had been trained to recover after making mistakes during driving. This training apparently provided the network with instructions on how to develop corrective driving measures. Presently, the project is aimed toward dealing more sensibly with road forks and intersections.

Simple Memory and Restoration of Patterns

This part of our neural network discussion introduces networks that respond, in time, to a presented pattern. Since they do so in a very characteristic way through a gradual reconstruction of a stored pattern, we call such networks simply *memories*. Only introductory discussion is presented below, with more in-depth coverage of memory analysis and design techniques to follow in Chapters 5 and 6.

Let us begin with an analysis of a very simple network as shown in Figure 1.5(a). The network consists of three units computing values of $\text{sgn}(\cdot)$, three summing nodes, and six weights of values +1 or -1. Signals passing through

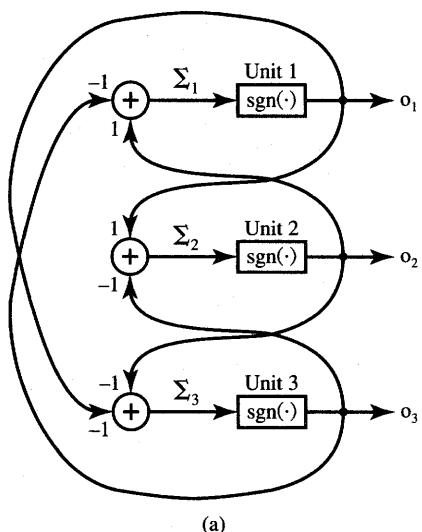


Figure 1.5 Simple neural network memory: (a) network diagram and (b) listing of updates.

Case	Unit Number	Present Output O	Σ	$\text{sgn } \Sigma$	Next Output O
1	1	1	0	x	1
	2	1	0	x	1
	3	1	-2	-1	(-1)
2	1	1	2	1	1
	2	1	2	1	1
	3	-1	-2	-1	-1
3	1	-1	2	1	(1)
	2	1	0	x	1
	3	-1	0	x	-1
4	1	1	0	x	1
	2	-1	2	1	(1)
	3	-1	0	x	-1

x stands for $\text{sgn}(0)$.

Encircled are updated outputs.

(a)

(b)

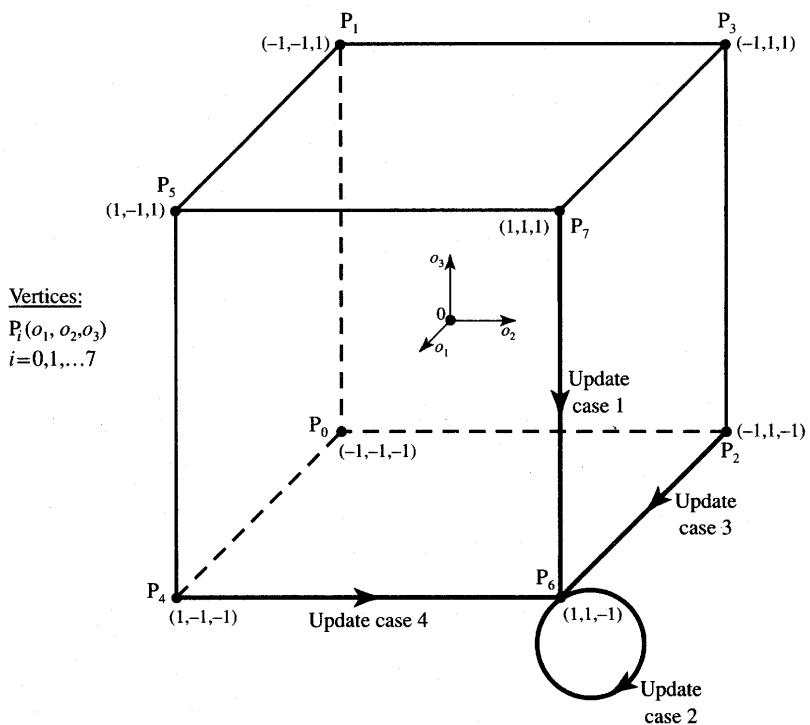


Figure 1.6 Graphical interpretation of the memory network from Figure 1.5.

weights are multiplied by the weight values. Assume that the network is initialized at its output and the initial output is $o_1 = 1$, $o_2 = 1$, $o_3 = 1$. When the network is allowed to compute, we notice that inputs to units 1 and 2 are exactly zero, while the input to unit 3 is equal to -2 . As a result, o_1 and o_2 do not update their respective values since $\text{sgn}(0)$ is an undetermined value. However, o_3 updates to -1 after it leaves the initial state of 1. Therefore, single updating activity across the network results in $o_1 = o_2 = 1$ and $o_3 = -1$.

The next computing cycle does not bring about any change since inputs to units 1, 2, and 3 are 2, 2, and -2 , respectively, thus yielding $o_1 = o_2 = 1$ and $o_3 = -1$ again. The listing of discussed updates is shown in Figure 1.5(b) as cases 1 and 2. Transitions from initial output $o_1 = -1$, $o_2 = 1$, $o_3 = -1$, and $o_1 = 1$, $o_2 = -1$, $o_3 = -1$ are also displayed as cases 3 and 4, respectively. Cases 1, 3, and 4 result in a single unit update and they all terminate in the same final output of $o_1 = 1$, $o_2 = 1$, and $o_3 = -1$. In addition, analysis of case 2 indicates that this output is, in fact, a terminal one, no matter how many update cycles are allowed to take place thereafter.

Figure 1.6 provides a geometrical interpretation of updates performed by the simple memory network from Figure 1.5. It can be seen that $P_6(1,1,-1)$ is the stable output of the discussed memory. When a single entry of the initializing

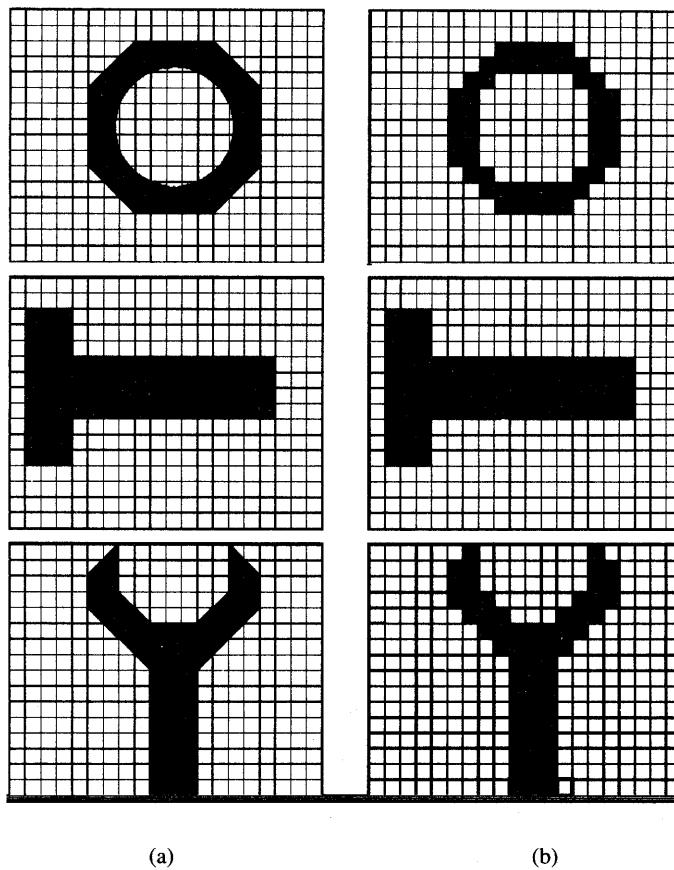


Figure 1.7 Memory network applications: (a) images of three mechanical parts and (b) images converted to bit-map forms.

binary output vector does not agree with P_6 , the network responds with the corrected entry. This output is sustained thereafter.

The memory concept demonstrated in this example can be easily extended to real-world applications. Assume that three images as shown in Figure 1.7(a) need to be stored in a memory network. They are a nut, a bolt, and a wrench as displayed on the 20×16 grid. The network, which can store any bit maps placed on the grid shown, has 320 units, each computing the sgn function. Essentially, the network is an extension of memory from Figure 1.5(a). The first preparatory step that needs to be taken is to represent the actual images through grid elements that are entirely white or black. In this way we obtain a set of three approximated

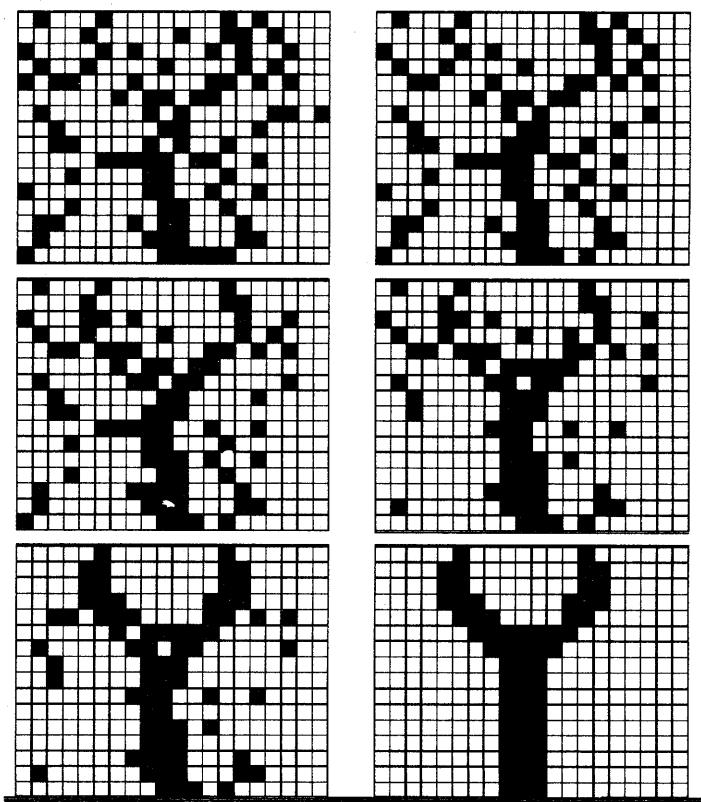


Figure 1.8 Restoration of the wrench image.

images. Examples of such images obtained from Figure 1.7(a) are illustrated in Figure 1.7(b).

A neural network memory can now be designed using the methods explained in Chapter 6. The memory holds each of the three binary patterns corresponding to bit maps from Figure 1.7(b). Once the memory is initialized at its output, it starts computing. After the evolution through intermediate states, the memory should terminate at one of its originally stored outputs. Figure 1.8 illustrates a sequence of outputs when a distorted image of the wrench is used to initialize the computation. The sequence consists of three snapshots taken during the computation performed by the memory network. It can be seen that the correct wrench shape is restored after a number of correctional transitions. This application indicates the potential of neural networks memory to restore heavily distorted images.

Optimizing Networks

Optimization is one of the most important objectives of engineering. The purpose of optimization is to minimize certain cost functions, usually defined by the user. Neural computation methods are successfully applied, among others, to the solution of some optimization problems. A more formal discussion of neural computation for optimization is covered in Chapter 5. Below we discuss only introductory concepts.

Assume that an analog value x , $0 \leq x \leq 3.5$, needs to be converted to a two-bit binary number v_1v_0 such that

$$x \cong 2v_1 + v_0 \quad (1.3)$$

where $v_1, v_0 = 0, 1$. Obviously, for the conditions stated in Equation (1.3) four possible solutions to the problem exist: 00, 01, 10, 11. A simple network similar to the one depicted in Figure 1.5(a) can be designed to solve the conversion problem stated in Equation (1.3). However, the network consists of only two units described by "squashed" characteristics, each responding in the range (0, 1). The block diagram of the network is shown in Figure 1.9. In addition to the two units, the network contains a number of interconnecting elements not shown on the figure.

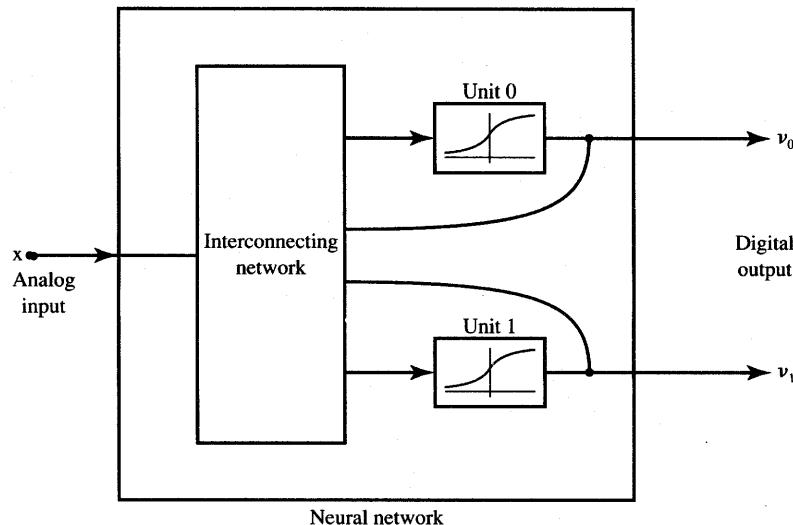


Figure 1.9 Block diagram of a two-bit A/D converter.

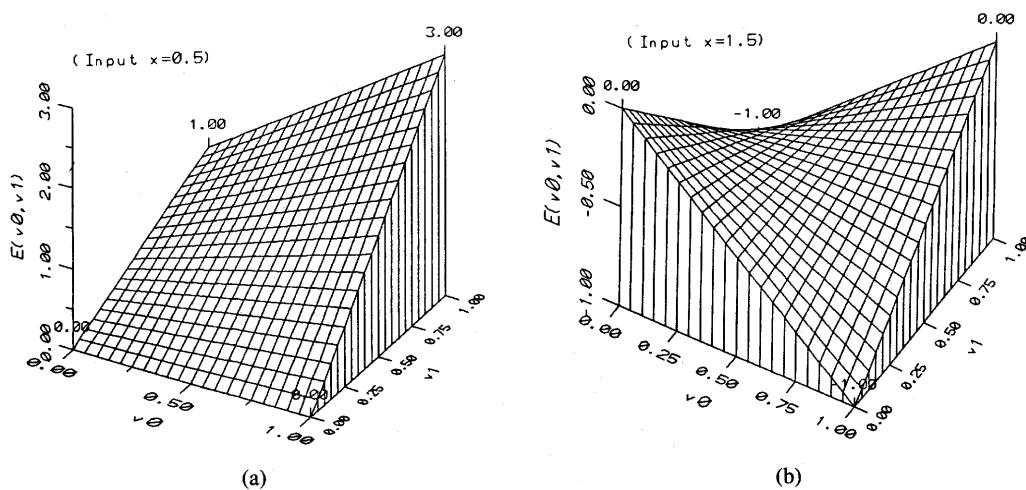


Figure 1.10 Energy function for a two-bit A/D conversion problem: (a) $x = 0$ and (b) $x = 1$.

Note that the conversion expressed by Equation (1.3) corresponds to the minimization of analog-to-digital conversion error, where

$$\text{error} = (x - 2v_1 - v_0)^2 \quad (1.4)$$

in the presence of such constraints that v_1 , v_0 can take values of 0 or 1. This constrained minimization problem can be fit into a certain class of neural networks. The class has the unique property such that the so-called energy function in these networks undergoes minimization while the network is computing.

The property of energy minimization is of great importance. It states that the network seeks out, by itself, its energy minimum and then settles there. A number of optimization problems can be translated directly into the minimization of a neural network's energy function. Once the translation is accomplished and the neural network designed accordingly, the optimization task is transferred to the neural network for actual solution. Minimization of energy by the network can be considered analogous to the minimization of error expressed by Equation (1.4).

Figure 1.10 illustrates two cases of minimization of the energy function $E(v_0, v_1)$. Figure 1.10(a) shows that for input $x = 0$ the energy minimum is located at $v_0 = v_1 = 0$. Furthermore, the network output should always stabilize at that corner, since the energy function is monotonic. Similarly, convergence of the network output is guaranteed toward $v_0 = 1$ and $v_1 = 0$ for input $x = 1$. This can be seen from Figure 1.10(b). In actual networks, output approaches a corner but never reaches it since $0 < v_0 < 1$ and $0 < v_1 < 1$.

The class of neural networks exemplified here using a simple two-bit analog-to-digital (A/D) converter is very useful for the solution of a number of problems. In particular, it is useful for the solution of combinational optimization tasks. They are characterized by $N!$ or e^N possible solutions of problems that are of size N . Understandably, evaluation of the cost function at each potential solution point of such problems can be very computationally intensive. Therefore, other ways of combinational optimization are studied, and neural networks are among them.

One of the problems that can be successfully solved by this class of networks is called *job-shop scheduling*. Job-shop scheduling is a resource allocation problem, subject to allocation and sequencing constraints. The resources are typically machines and the jobs are the basic tasks that need to be accomplished using the machines. Each task may consist of several subtasks related by certain precedence restrictions. The job-shop problem is solved when the starting times of all jobs are determined and the sequencing rules are not violated. The cost function to be minimized is defined for this problem as a sum of the starting times of all jobs subject to compliance with precedence constraints (Foo and Takefuji 1988). Problems that are somewhat simpler but belong to the same group of optimization tasks include scheduling classrooms to classes, hospital patients to beds, workers to tasks they can accomplish best, etc.

Clustering and Feature Detecting Networks

A rather important group of neural networks can be used for detecting clusters of data. Such networks are tuned to certain similarity aspects that are of interest in the data being evaluated. For example, we may be interested in grouping certain measurement results together to suppress any systematic errors that may have occurred during measurements. We may also be interested in detecting regularly appearing components of inputs. These components may indicate the true signal components as opposed to the noise that is random and would not form any clusters.

Clustering and feature detecting networks exhibit remarkable properties of self-organization. These properties are closely related to the formation of knowledge representation in artificial intelligence and information theory. The networks of this class usually have a simple architecture; considerable subtleties arise mainly during the self-organization process, which is discussed in more detail in Chapter 7.

Another important function of neural networks can be feature detection. Feature detection is usually related to the dimensionality reduction of data. Some networks provide impressive planar mapping of features that are a result of multi-dimensional inputs of a fairly complex structure. Some of the many applications reported are discussed in Chapters 7 and 8 and one is especially intriguing. It involves mapping speech features as described below.

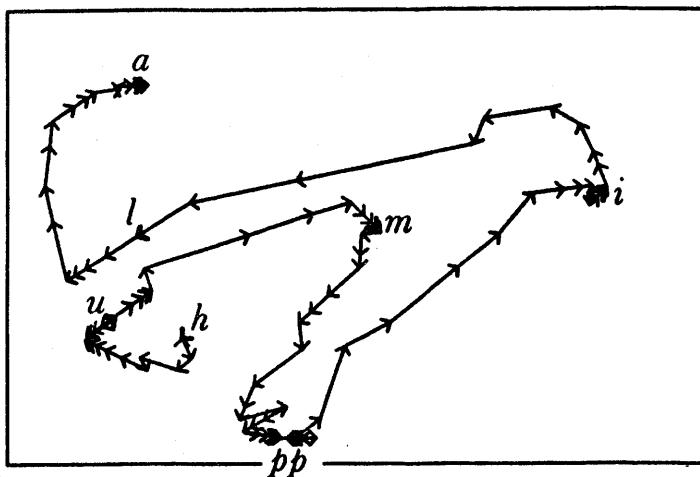


Figure 1.11 Phonotopic map of the Finnish word *humpilla* as an example of extracted features of speech. SOURCE: (Kohonen, 1988) © IEEE; reprinted with permission.

Speech processing studies indicate that speech signal consists of 15 frequency channels within the audio-frequency band. Elementary components of speech, called *phonemes*, could thus be displayed in the 15-dimensional spectral space of speech signals. The problem with such representation of phonemes is that it is of no use for us because our powers of visualization are limited to three dimensions. Through appropriate neural network mapping, however, it is possible to “flatten” the 15-dimensional spectrum of speech onto a constrained planar array. Figure 1.11 shows a so-called “phonotopic” map of a spoken word. The word *humpilla* has been used to produce the map. The rectangle is filled with units (not shown) that are selectively activated during speech. The sequence of word phonemes forms a trajectory that provides an impressive visualization of the continuous speech signal. Phonotopic maps may be invaluable for building phonetic typewriters, for speech training, and for therapy. Profoundly deaf people could have visual feedback from their speech.

1.2

HISTORY OF ARTIFICIAL NEURAL SYSTEMS DEVELOPMENT

Artificial neural systems development has an interesting history. Since it is not possible to cover this history in depth in a short introductory chapter, only major achievements are mentioned. This glimpse at the field’s past milestones

should provide the reader with an appreciation of how contributions to the field have led to its development over the years. The historical summary below is not exhaustive; some milestones are omitted and some are mentioned only briefly.

The year 1943 is often considered the initial year in the development of artificial neural systems. McCulloch and Pitts (1943) outlined the first formal model of an elementary computing neuron. The model included all necessary elements to perform logic operations, and thus it could function as an arithmetic-logic computing element. The implementation of its compact electronic model, however, was not technologically feasible during the era of bulky vacuum tubes. The formal neuron model was not widely adopted for the vacuum tube computing hardware description, and the model never became technically significant. However, the McCulloch and Pitts neuron model laid the groundwork for future developments.

Influential researchers of that time suggested that research in design of brain-like processing might be interesting. To quote John von Neumann's (1958) observations on the "brain language":

We have now accumulated sufficient evidence to see that whatever language the central nervous system is using, it is characterized by less logical and arithmetical depth than what we are normally used to. The following is an obvious example of this: the retina of the human eye performs a considerable reorganization of the visual image as perceived by the eye. Now this reorganization is effected on the retina, or to be more precise, at the point of entry of the optic nerve by means of the successive synapses only, i.e. in terms of three consecutive logical steps. The statistical character of the message system used in the arithmetics of the central nervous system and its low precision, described earlier, cannot proceed very far in the message systems involved. Consequently, there exist here different logical structures from the ones we are ordinarily used to in logics and mathematics. They are characterized by less logical and arithmetical depth than we are used to under otherwise similar circumstances. Thus logic and mathematics in the central nervous system, when viewed as languages, must structurally be essentially different from those languages to which our common experience refers.

Donald Hebb (1949) first proposed a learning scheme for updating neuron's connections that we now refer to as the *Hebbian learning rule*. He stated that the information can be stored in connections, and postulated the learning technique that had a profound impact on future developments in this field. Hebb's learning rule made primary contributions to neural networks theory.

During the 1950s, the first neurocomputers were built and tested (Minsky 1954). They adapted connections automatically. During this stage, the neuron-like element called a *perceptron* was invented by Frank Rosenblatt in 1958. It was a

trainable machine capable of learning to classify certain patterns by modifying connections to the threshold elements (Rosenblatt 1958). The idea caught the imagination of engineers and scientists and laid the groundwork for the basic machine learning algorithms that we still use today.

In the early 1960s a device called *ADALINE* (for ADaptive LINEar combiner) was introduced, and a new, powerful learning rule called the *Widrow-Hoff learning rule* was developed by Bernard Widrow and Marcian Hoff (1960, 1962). The rule minimized the summed square error during training involving pattern classification. Early applications of ADALINE and its extensions to MADALINE (for Many ADALINES) include pattern recognition, weather forecasting, and adaptive controls. The monograph on learning machines by Nils Nilsson (1965) clearly summarized many of the developments of that time. That book also formulates inherent limitations of learning machines with modifiable connections.

Despite the successes and enthusiasm of the early and mid-1960s, the existing machine learning theorems of that time were too weak to support more complex computational problems. Although the bottlenecks were exactly identified in Nilsson's work and the neural network architectures called *layered networks* were also known, no efficient learning schemes existed at that time that would circumvent the formidable obstacles (see Section 4.1). Neural network research entered into the stagnation phase. Another reason that contributed to this research slowdown at that time was the relatively modest computational resources available then.

The final episode of this era was the publication of a book by Marvin Minsky and Seymour Papert (1969) that gave more doubt as to the layered learning networks' potential. The stated limitations of the perceptron-class networks were made public; however, the challenge was not answered until the mid-1980s. The discovery of successful extensions of neural network knowledge had to wait until 1986. Meanwhile, the mainstream of research flowed toward other areas, and research activity in the neural network field, called at that time *cybernetics*, had sharply decreased. The artificial intelligence area emerged as a dominant and promising research field, which took over, among others, many of the tasks that neural networks of that day could not solve.

During the period from 1965 to 1984, further pioneering work was accomplished by a handful of researchers. The study of learning in networks of threshold elements and of the mathematical theory of neural networks was pursued by Sun-Ichi Amari (1972, 1977). Also in Japan, Kunihiko Fukushima developed a class of neural network architectures known as *neocognitrons* (Fukushima and Miyaka 1980). The neocognitron is a model for visual pattern recognition and is concerned with biological plausibility. The network emulates the retinal images and processes them using two-dimensional layers of neurons.

Associative memory research has been pursued by, among others, Tuevo Kohonen in Finland (1977, 1982, 1984, 1988) and James A. Anderson (Anderson et al. 1977). Unsupervised learning networks were developed for feature mapping

into regular arrays of neurons (Kohonen 1982). Figure 1.11 demonstrates one of the practical uses of this class of networks. Stephen Grossberg and Gail Carpenter have introduced a number of neural architectures and theories and developed the theory of adaptive resonance networks (Grossberg 1974, 1982).

During the period from 1982 until 1986, several seminal publications were published that significantly furthered the potential of neural networks. The era of renaissance started with John Hopfield (1982, 1984) introducing a recurrent neural network architecture for associative memories. His papers formulated computational properties of a fully connected network of units. Figures 1.5 through 1.10 illustrate basic concepts, examples and applications of fully connected networks originally introduced by Hopfield.

Another revitalization of the field came from the publication in 1986 of two volumes on parallel distributed processing, edited by James McClelland and David Rumelhart (1986). The new learning rules and other concepts introduced in this work have removed one of the most essential network training barriers that grounded the mainstream efforts of the mid-1960s. The publication by McClelland and Rumelhart opened a new era for the once-underestimated computing potential of layered networks. The function approximator, EEG spike detector and autonomous driver discussed in the previous section provide examples facilitated by the new learning rules.

Although the mathematical framework for the new training scheme of layered networks was discovered in 1974 by Paul Werbos, it went largely unnoticed at that time (Werbos 1974). According to the most recent statement (Dreyfus 1990), the first authors of the optimization approach for multilayer feedforward systems were Bryson (Bryson and Ho 1969) and Kelley (Kelley 1969) who obtained a gradient solution for multistage network training. In 1962, Dreyfus used a simple, new recursive derivation based on the chain-rule of differentiation to prove the Bryson-Kelley results and dealt explicitly with the optimal control problem in its discrete-stage form (Dreyfus 1962). Their work, however, has not been carried to maturity and adopted for neural network learning algorithms. Most of the developments covered in this historical overview are looked at in detail in subsequent chapters.

Beginning in 1986-87, many new neural networks research programs were initiated. The intensity of research in the neurocomputing discipline can be measured by a quickly growing number of conferences and journals devoted to the field. In addition to many edited volumes that contain collections of papers, several books have already appeared. The list of applications that can be solved by neural networks has expanded from small test-size examples to large practical tasks. Very-large-scale integrated neural network chips have been fabricated. At the time of this writing, educational offerings have been established to explore the artificial neural systems science. Although neurocomputing has had an interesting history, the field is still in its early stages of development.

1.3

FUTURE OUTLOOK

The current explosion of interest in artificial neural systems is based on a number of scientific and economic expectations. One should be aware, however, that some of the popular expectations may be exaggerated and are not realistic. The tendency for overexpectations comes to the field of artificial neural systems naturally through its name and, sometimes, through the terminology it uses. It is somewhat tempting for a human being to envisage a human-like machine with human or near-human perceptual, reasoning, and movement capabilities. Anthropomorphic terminology, inspirations, and the sensory processing applications of neural network technology may be blamed partially for linking such human-like man-made machines with what is merely a technical discipline. Such "links" should be dismissed with good humor.

We can be quite sure that neural networks will not replace conventional computers. Basic reasons preclude neurocomputers replacing conventional ones for most tasks. Conventional computers are now very inexpensive to make. They are extremely fast and accurate for executing mathematical subroutines, text processing, computer-aided design, data processing and transfer, and many other tasks. In general, conventional digital computers outperform any other computing devices in numerical calculations.

One possibility for the use of artificial neural systems is to simulate physical systems that are best expressed by massively parallel networks. Also, low-level signal processing might best be done by parallel analog or analog and digital neural networks. Perhaps the most likely applications for neural networks will be those involving classification, association, and reasoning rather than sequential arithmetic computing. Not surprisingly, the best hope for the widespread use of artificial neural systems, or neurocomputing, is in computationally intensive areas that are not successfully attacked by conventional computers. It seems that the areas requiring human-like inference and perception of speech and vision are the most likely candidates for applications. If these succeed, there will be more applications in real-time control of complex systems and other applications that we cannot yet anticipate.

Neural networks are also expected to be widely applied in expert systems and in a variety of signal processors. At present, such systems are available as aids for medical diagnosis, financial services, stock price prediction, solar flare forecasting, radar pulse identification, and other applications. As most researchers agree, future artificial neural systems are not going to replace computational and artificial intelligence simulations on conventional computers either. Rather, they will offer a complementary technology. The ultimate goal may be to exploit both technologies under the same roof, while presenting a single, flexible interface to the user.

Today's neurocomputers are often conventional programmable computers running neural simulation software. Such simulated neurocomputers provide suitable but conventional input and output interfaces. Some neurocomputers employ dedicated architectures, or neurocomputing boards, that make the neural network simulations operate faster and more accurately when compared to standard architectures. Understandably, as neurocomputers become faster, smaller, and more efficient, the limits of applications will expand beyond simulated solutions. Furthermore, it seems certain that the proliferation of application-specific very-large-scale integrated (VLSI) neural networks is of crucial importance for the long-term success of the technology.

Fortunately, artificial neural systems fabrication follows the invented computing paradigms. Optical neural network implementations are expected to become directly applicable. Since 1986, when AT&T's first neural integrated circuit memory was fabricated, we have seen the proliferation of microelectronic neural network chips. VLSI neural systems with modifiable weights have reached the market. However, general-purpose trainable neural network chips with on-chip learning are probably several years off.

In the long term, we could expect that artificial neural systems will be used in applications involving vision, speech, decision-making, and reasoning, but also as signal processors such as filters, detectors, and quality control systems. Applications are expected especially for processing large amounts of data. Also, neural networks may offer solutions for cases in which a processing algorithm or analytical solutions are hard to find, hidden, or nonexistent. Such cases include modeling complex processes, extracting properties of large sets of data, and providing identification of plants that need to be controlled. A robot system already exists that combines vision with robot arm motion control and learns hand-eye coordination. Such systems can be expected to be extended over the next few years to produce a sophisticated robotic system. Continuous, speaker-independent speech recognition systems are being developed. They are likely to be several years off, but the limited vocabulary systems may be commercially available sooner.

Artificial neural networks technology is still very new and is developing quickly. We are witnessing fast expansion of neural network-based intelligent machines. Hopefully, this expansion will enhance the quality of our lives and make many difficult tasks easier to accomplish. Further speculations may not be needed, because the technological achievements and available products will speak for themselves.

REFERENCES

- Alexander, I., ed. 1989. *Neural Computing Architectures—The Design of Brain-Like Machines*. Cambridge, Mass.: MIT Press.

- Amari, S. I. 1972. "Learning Patterns and Pattern Sequences by Self-Organizing Nets of Threshold Elements," *IEEE Trans. Computers* C-21: 1197-1206.
- Amari, S. I. 1977. "Neural Theory of Association and Concept Formation," *Biol. Cybern.* 26: 175-185.
- Anderson, J. A., J. W. Silverstein, S. A. Rite, and R. S. Jones, 1977. "Distinctive Features, Categorical Perception, and Probability Learning: Some Applications of a Neural Model," *Psych. Rev.* 84: 413-451.
- Bryson, A. E., and Y. C. Ho. 1969. *Applied Optimal Control*. Waltham, Mass.: Blaisdell, 43-45.
- Dayhoff, J. 1990. *Neural Network Architectures—An Introduction*. New York: Van Nostrand Reinhold.
- Dreyfus, S. 1962. "The Numerical Solution of Variational Problems," *Math. Anal. Appl.* 5(1): 30-45.
- Dreyfus, S. E. 1990. "Artificial Neural Networks, Back Propagation and the Kelley-Bryson Gradient Procedure," *J. Guidance, Control Dynamics*. 13(5): 926-928.
- Eberhart, R. C., and R. W. Dobbins. 1990. "Case Study I: Detection of Electroencephalogram Spikes," in *Neural Networks PC Tools*. ed. R. C. Eberhart, R. W. Dobbins. San Diego, Calif.: Academic Press.
- Foo, Y. P. S. and Y. Takefuji. 1988. "Integer Linear Programming Neural Networks for Job-Shop Scheduling," *Proc. 1988 Intern. IEEE Conf. Neural Networks*, San Diego, Calif.
- Fukushima, K. and S. Miyaka. 1980. "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position," *Biol. Cybern.* 36(4): 193-202.
- Grossberg, S. 1977. "Classical and Instrumental Learning by Neural Networks," in *Progress in Theoretical Biology*. vol. 3. New York: Academic Press, 51-141.
- Grossberg, S. 1982. *Studies of Mind and Brain: Neural Principles of Learning Perception, Development, Cognition, and Motor Control*. Boston: Reidel Press.
- Hebb, D. O. 1949. *The Organization of Behavior, A Neuropsychological Theory*. New York: John Wiley.
- Hopfield, J. J. 1982. "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," *Proc. Natl. Acad. Sci.* 79: 2554-58.
- Hopfield, J. J. 1984. "Neurons with Graded Response Have Collective Computational Properties Like Those of Two State Neurons," *Proc. Natl. Acad. Sci.* 81: 3088-3092.
- Kelley, H. J. 1960. "Gradient Theory of Optimal Flight Path," *ARS Journal*, 30(10): 947-954.

- Kirrmann, H. (June) 1989. "Neural Computing: The New Gold Rush in Informatics," *IEEE Microworld*.
- Kohonen, T. 1977. *Associative Memory: A System-Theoretical Approach*, Berlin: Springer-Verlag.
- Kohonen, T. 1982. "A Simple Paradigm for the Self-Organized Formation of Structured Feature Maps," in *Competition and Cooperation in Neural Nets*. ed. S. Amari, M. Arbib. vol. 45. Berlin: Springer-Verlag.
- Kohonen, T. 1984. *Self-Organization and Associative Memory*. Berlin: Springer Verlag.
- Kohonen, T. 1988. "The 'Neural' Phonetic Typewriter," *IEEE Computer* 27(3): 11-22.
- McClelland, J. L., D. E. Rumelhart 1986. *Parallel Distributed Processing*. Cambridge: MIT Press and the PDP Research Group.
- McCulloch, W. S. and W. H. Pitts. 1943. "A Logical Calculus of the Ideas Imminent in Nervous Activity," *Bull. Math. Biophys.* 5:115-133.
- Minsky, M. and S. Papert, 1969. *Perceptrons*. Cambridge, Mass.: MIT Press.
- Minsky, M. 1954. "Neural Nets and the Brain," *Doctoral Dissertation*, Princeton University, NJ.
- Nilsson, N. J. 1965. *Learning Machines: Foundations of Trainable Pattern Classifiers*, New York: McGraw Hill; also republished as *The Mathematical Foundations of Learning Machines*, Morgan-Kaufmann Publishers, San Mateo, Calif. 1990.
- Pomerleau, D. A. 1989. "ALVINN: An Autonomous Land Vehicle in a Neural Network," in *Advances in Neural Information Processing*. ed. D. Touretzky. vol. 1, San Mateo, Calif.: Morgan-Kaufmann Publishers.
- Rosenblatt, F. 1958. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psych. Rev.* 65: 386-408.
- Shriver, B. D. 1988. "Artificial Neural Systems," *IEEE Computer* (March): 8-9.
- von Neumann, J. 1958. *The Computer and the Brain*. New Haven, Conn.: Yale University Press, 87.
- Werbos, P. J. 1974. "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," *Doctoral Dissertation, Appl. Math.*, Harvard University, Mass.
- Widrow, B. 1962. "Generalization and Information Storage in Networks of Adaline 'Neurons,'" in *Self-organizing Systems*. M. C. Jovitz, G. T. Jacobi, and G. Goldstein. eds., Washington, D.C.: Spartan Books, 435-461.
- Widrow, B. and M. E. Hoff, Jr. 1960. "Adaptive Switching Circuits," 1960 IRE Western Electric Show and Convention Record, part 4 (Aug. 23): 96-104.

FUNDAMENTAL CONCEPTS AND MODELS OF ARTIFICIAL NEURAL SYSTEMS

*Nothing happens in the universe
that does not have a sense of either
certain maximum or minimum.*

L. EULER

- 2.1 Biological Neurons and Their Artificial Models
- 2.2 Models of Artificial Neural Networks
- 2.3 Neural Processing
- 2.4 Learning and Adaptation
- 2.5 Neural Network Learning Rules
- 2.6 Overview of Neural Networks
- 2.7 Concluding Remarks

There are a number of different answers possible to the question of how to define neural networks. At one extreme, the answer could be that neural networks are simply a class of mathematical algorithms, since a network can be regarded essentially as a graphic notation for a large class of algorithms. Such algorithms produce solutions to a number of specific problems. At the other end, the reply may be that these are synthetic networks that emulate the biological neural networks found in living organisms. In light of today's limited knowledge of biological neural networks and organisms, the more plausible answer seems to be closer to the algorithmic one.

In search of better solutions for engineering and computing tasks, many avenues have been pursued. There has been a long history of interest in the biological sciences on the part of engineers, mathematicians, and physicists endeavoring to gain new ideas, inspirations, and designs. Artificial neural networks have undoubtedly been biologically inspired, but the close correspondence between them and real neural systems is still rather weak. Vast discrepancies exist between both the architectures and capabilities of artificial and natural neural networks. Knowledge about actual brain functions is so limited, however, that

there is little to guide those who would try to emulate them. No models have been successful in duplicating the performance of the human brain. Therefore, the brain has been and still is only a metaphor for a wide variety of neural network configurations that have been developed (Durbin 1989).

Despite the loose analogy between artificial and natural neural systems, we will briefly review the biological neuron model. The synthetic neuron model will subsequently be defined in this chapter and examples of network classes will be discussed. The basic definitions of neuron and elementary neural networks will also be given. Since no common standards are yet used in the technical literature, this part of the chapter will introduce notation, graphic symbols, and terminology used in this text. The basic forms of neural network processing will also be discussed.

The reader may find this initially surprising, but the majority of this book's content is devoted to network learning and, specifically, to experiential training. In this chapter basic learning concepts are characterized and defined. While the exposition of artificial neural network learning is developed gradually here and throughout subsequent chapters, the reader should find it intriguing and appealing to discover how powerful and useful learning techniques can be. Since neural networks represent a rich class of learning algorithms and architectures, a taxonomy of the most important networks is presented.

2.1

BIOLOGICAL NEURONS AND THEIR ARTIFICIAL MODELS

A human brain consists of approximately 10^{11} computing elements called *neurons*. They communicate through a connection network of axons and synapses having a density of approximately 10^4 synapses per neuron. Our hypothesis regarding the modeling of the natural nervous system is that neurons communicate with each other by means of electrical impulses (Arbib 1987). The neurons operate in a chemical environment that is even more important in terms of actual brain behavior. We thus can consider the brain to be a densely connected electrical switching network conditioned largely by the biochemical processes. The vast neural network has an elaborate structure with very complex interconnections. The input to the network is provided by sensory receptors. Receptors deliver stimuli both from within the body, as well as from sense organs when the stimuli originate in the external world. The stimuli are in the form of electrical impulses that convey the information into the network of neurons. As a result of information processing in the central nervous systems, the effectors are controlled and give human responses in the form of diverse actions. We thus have a three-stage system, consisting of receptors, neural network, and effectors, in control of the organism and its actions.

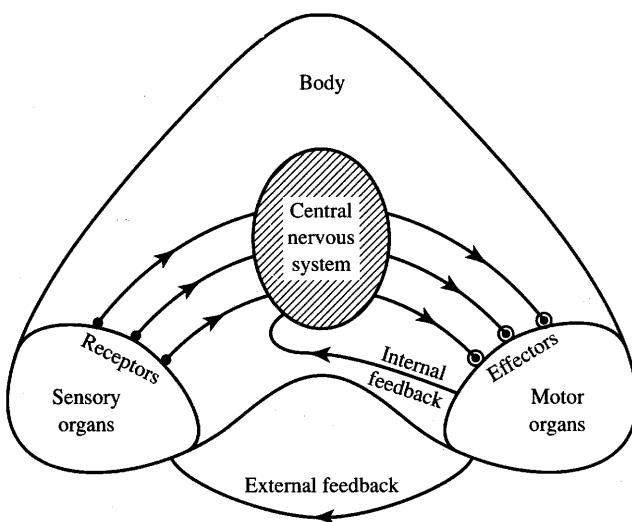


Figure 2.1 Information flow in nervous system.

A lucid, although rather approximate idea, about the information links in the nervous system is shown in Figure 2.1. As we can see from the figure, the information is processed, evaluated, and compared with the stored information in the central nervous system. When necessary, commands are generated there and transmitted to the motor organs. Notice that motor organs are monitored in the central nervous system by feedback links that verify their action. Both internal and external feedback control the implementation of commands. As can be seen, the overall nervous system structure has many of the characteristics of a closed-loop control system.

Biological Neuron

The *elementary nerve cell*, called a *neuron*, is the fundamental building block of the biological neural network. Its schematic diagram is shown in Figure 2.2. A typical cell has three major regions: the cell body, which is also called the *soma*, the *axon*, and the *dendrites*. Dendrites form a dendritic tree, which is a very fine bush of thin fibers around the neuron's body. Dendrites receive information from neurons through axons—long fibers that serve as transmission lines. An axon is a long cylindrical connection that carries impulses from the neuron. The end part of an axon splits into a fine arborization. Each branch of it terminates in a small endbulb almost touching the dendrites of neighboring neurons. The axon-dendrite contact organ is called a *synapse*. The synapse is where the neuron introduces its signal to the neighboring neuron. The signals reaching a synapse and received

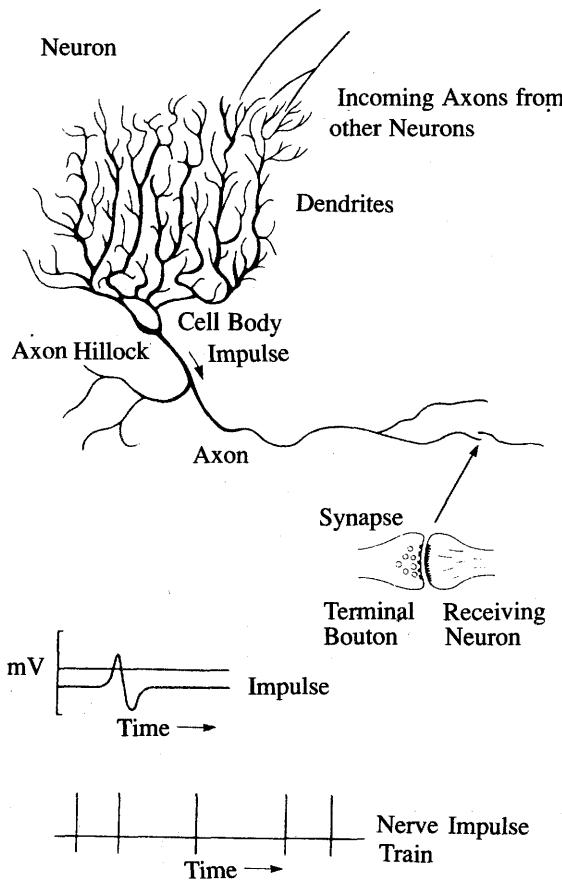


Figure 2.2 Schematic diagram of a neuron and a sample of pulse train. (Adapted from (Dayhoff 1990), © Van Nostrand Reinhold; with permission.)

by dendrites are electrical impulses. The interneuronal transmission is sometimes electrical but is usually effected by the release of chemical transmitters at the synapse. Thus, terminal boutons generate the chemical that affects the receiving neuron. The receiving neuron either generates an impulse to its axon, or produces no response.

The neuron is able to respond to the total of its inputs aggregated within a short time interval called the *period of latent summation*. The neuron's response is generated if the total potential of its membrane reaches a certain level. The membrane can be considered as a shell, which aggregates the magnitude of the incoming signals over some duration. Specifically, the neuron generates a pulse response and sends it to its axon only if the conditions necessary for firing are fulfilled.

Let us consider the conditions necessary for the firing of a neuron. Incoming impulses can be *excitatory* if they cause the firing, or *inhibitory* if they hinder the firing of the response. A more precise condition for firing is that the excitation should exceed the inhibition by the amount called the *threshold* of the neuron, typically a value of about 40 mV (Arbib 1987). Since a synaptic connection causes the excitatory or inhibitory reactions of the receiving neuron, it is practical to assign positive and negative unity weight values, respectively, to such connections. This allows us to reformulate the neuron's firing condition. The neuron fires when the total of the weights to receive impulses exceeds the threshold value during the latent summation period.

The incoming impulses to a neuron can only be generated by neighboring neurons and by the neuron itself. Usually, a certain number of incoming impulses are required to make a target cell fire. Impulses that are closely spaced in time and arrive synchronously are more likely to cause the neuron to fire. As mentioned before, observations have been made that biological networks perform temporal integration and summation of incoming signals. The resulting spatio-temporal processing performed by natural neural networks is a complex process and much less structured than digital computation. The neural impulses are not synchronized in time as opposed to the synchronous discipline of digital computation. The characteristic feature of the biological neuron is that the signals generated do not differ significantly in magnitude; the signal in the nerve fiber is either absent or has the maximum value. In other words, information is transmitted between the nerve cells by means of binary signals.

After carrying a pulse, an axon fiber is in a state of complete nonexcitability for a certain time called the *refractory period*. For this time interval the nerve does not conduct any signals, regardless of the intensity of excitation. Thus, we may divide the time scale into consecutive intervals, each equal to the length of the refractory period. This will enable a discrete-time description of the neurons' performance in terms of their states at discrete time instances. For example, we can specify which neurons will fire at the instant $k + 1$ based on the excitation conditions at the instant k . The neuron will be excited at the present instant if the number of excited excitatory synapses exceeds the number of excited inhibitory synapses at the previous instant by at least the number T , where T is the neuron's threshold value.

The time units for modeling biological neurons can be taken to be of the order of a millisecond. However, the refractory period is not uniform over the cells. Also, there are different types of neurons and different ways in which they connect. Thus, the picture of real phenomena in the biological neural network becomes even more involved. We are dealing with a dense network of interconnected neurons that release asynchronous signals. The signals are then fed forward to other neurons within the spatial neighborhood but also fed back to the generating neurons.

The above discussion is extremely simplified when seen from a neurobiological point of view, though it is valuable for gaining insight into the principles of "biological computation." Our computing networks are far simpler than their biological counterparts. Let us examine an artificial neuron model that is of special, historical significance.

McCulloch-Pitts Neuron Model

The first formal definition of a synthetic neuron model based on the highly simplified considerations of the biological model described in the preceding section was formulated by McCulloch and Pitts (1943). The McCulloch-Pitts model of the neuron is shown in Figure 2.3a. The inputs x_i , for $i = 1, 2, \dots, n$, are 0 or 1, depending on the absence or presence of the input impulse at instant k . The neuron's output signal is denoted as o . The firing rule for this model is defined as follows

$$o^{k+1} = \begin{cases} 1 & \text{if } \sum_{i=1}^n w_i x_i^k \geq T \\ 0 & \text{if } \sum_{i=1}^n w_i x_i^k < T \end{cases}$$

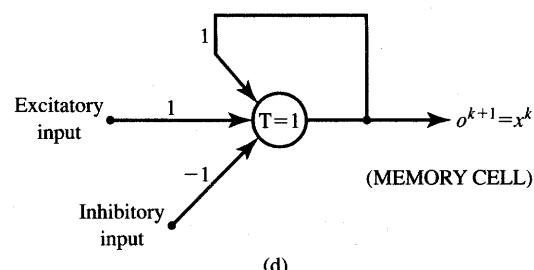
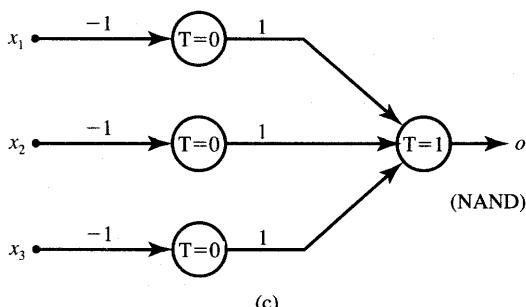
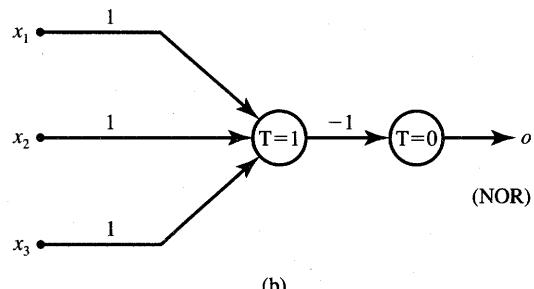
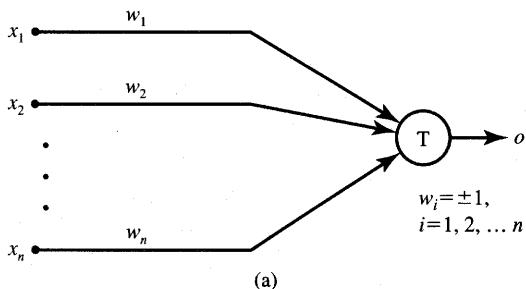


Figure 2.3 McCulloch-Pitts model neuron and elementary logic networks: (a) model diagram, (b) NOR gate, (c) NAND gate, and (d) memory cell.

where superscript $k = 0, 1, 2, \dots$ denotes the discrete-time instant, and w_i is the multiplicative weight connecting the i 'th input with the neuron's membrane. In further discussion, we will assume that a unity delay elapses between the instants k and $k + 1$. Note that $w_i = +1$ for excitatory synapses, $w_i = -1$ for inhibitory synapses for this model, and T is the neuron's threshold value, which needs to be exceeded by the weighted sum of signals for the neuron to fire.

Although this neuron model is very simplistic, it has substantial computing potential. It can perform the basic logic operations NOT, OR, and AND, provided its weights and thresholds are appropriately selected. As we know, any multivariable combinational function can be implemented using either the NOT and OR, or alternatively the NOT and AND, Boolean operations. Examples of three-input NOR and NAND gates using the McCulloch-Pitts neuron model are shown in Figure 2.3(b) and (c). The reader can easily inspect the implemented functions by compiling a truth table for each of the logic gates shown in the figure.

Both the neuron model and the example logic circuits discussed so far have been combinational and little attention has been paid to the inherent delay involved in their operation. However, the unity delay property of the McCulloch-Pitts neuron model makes it possible to build sequential digital circuitry.

First note that a single neuron with a single input x and with the weight and threshold values both of unity, computes $o^{k+1} = x^k$. Such a simple network thus behaves as a single register cell able to retain the input for one period elapsing between two instants. As a consequence, once a feedback loop is closed around the neuron as shown in Figure 2.3(d), we obtain a memory cell. An excitatory input of 1 initializes the firing in this memory cell, and an inhibitory input of 1 initializes a nonfiring state. The output value, at the absence of inputs, is then sustained indefinitely. This is because the output of 0 fed back to the input does not cause firing at the next instant, while the output of 1 does.

Thus, we see that digital computer hardware of arbitrary complexity can be constructed using an artificial neural network consisting of elementary building blocks as shown in Figure 2.3. Our purpose, however, is not to duplicate the function of already efficient digital circuitry, but rather to assess and exploit the computational power that is manifested by interconnected neurons subject to the experiential learning process.

Neuron Modeling for Artificial Neural Systems

The McCulloch-Pitts model of a neuron is characterized by its formalism and its elegant, precise mathematical definition. However, the model makes use of several drastic simplifications. It allows binary 0, 1 states only, operates under a discrete-time assumption, and assumes synchrony of operation of all neurons

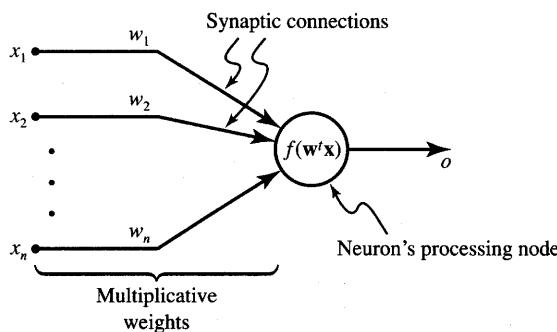


Figure 2.4 General symbol of neuron consisting of processing node and synaptic connections.

in a larger network. Weights and the neurons' thresholds are fixed in the model and no interaction among network neurons takes place except for signal flow. Thus, we will consider this model as a starting point for our neuron modeling discussion. Specifically, the artificial neural systems and computing algorithms employ a variety of neuron models that have more diversified features than the model just presented. Below, we introduce the main artificial neuron models that will be used later in this text.

Every neuron model consists of a processing element with synaptic input connections and a single output. The signal flow of neuron inputs, x_i , is considered to be unidirectional as indicated by arrows, as is a neuron's output signal flow. A general neuron symbol is shown in Figure 2.4. This symbolic representation shows a set of weights and the *neuron's processing unit*, or *node*. The neuron output signal is given by the following relationship:

$$o = f(\mathbf{w}^t \mathbf{x}), \text{ or} \quad (2.1a)$$

$$o = f \left(\sum_{i=1}^n w_i x_i \right) \quad (2.1b)$$

where \mathbf{w} is the *weight vector* defined as

$$\mathbf{w} \stackrel{\Delta}{=} [w_1 \ w_2 \ \cdots \ w_n]^t$$

and \mathbf{x} is the *input vector*:

$$\mathbf{x} \stackrel{\Delta}{=} [x_1 \ x_2 \ \cdots \ x_n]^t$$

(All vectors defined in this text are column vectors; superscript t denotes a transposition.) The function $f(\mathbf{w}^t \mathbf{x})$ is often referred to as an *activation function*. Its domain is the set of activation values, *net*, of the neuron model, we thus often use this function as $f(\text{net})$. The variable *net* is defined as a scalar product of the weight and input vector

$$\text{net} \stackrel{\Delta}{=} \mathbf{w}^t \mathbf{x} \quad (2.2)$$

The argument of the activation function, the variable net , is an analog of the biological neuron's membrane potential. Note that temporarily the threshold value is not explicitly used in (2.1) and (2.2), but this is only for notational convenience. We have momentarily assumed that the modeled neuron has $n - 1$ actual synaptic connections that come from actual variable inputs x_1, x_2, \dots, x_{n-1} . We have also assumed that $x_n = -1$ and $w_n = T$. Since threshold plays an important role for some models, we will sometimes need to extract explicitly the threshold as a separate neuron model parameter.

The general neuron symbol, shown in Figure 2.4 and described with expressions (2.1) and (2.2), is commonly used in neural network literature. However, different artificial neural network classes make use of different definitions of $f(net)$. Also, even within the same class of networks, the neurons are sometimes considered to perform differently during different phases of network operation. Therefore, it is pedagogically sound to replace, whenever needed, the general neuron model symbol from Figure 2.4 with a specific $f(net)$ and a specific neuron model. The model validity will then usually be restricted to a particular class of network. Two main models introduced below are often used in this text.

Acknowledging the simplifications that are necessary to model a biological neuron network with artificial neural networks, we introduce in this text the following terminology: (1) *neural networks are meant to be artificial neural networks consisting of neuron models* and (2) *neurons are meant to be artificial neuron models* as defined in this chapter.

Observe from (2.1) that the neuron as a processing node performs the operation of summation of its weighted inputs, or the scalar product computation to obtain net . Subsequently, it performs the nonlinear operation $f(net)$ through its activation function. Typical activation functions used are

$$f(net) \triangleq \frac{2}{1 + \exp(-\lambda net)} - 1 \quad (2.3a)$$

and

$$f(net) \triangleq \text{sgn}(net) = \begin{cases} +1, & net > 0 \\ -1, & net < 0 \end{cases} \quad (2.3b)$$

where $\lambda > 0$ in (2.3a) is proportional to the neuron gain determining the steepness of the continuous function $f(net)$ near $net = 0$. The continuous activation function is shown in Figure 2.5(a) for various λ . Notice that as $\lambda \rightarrow \infty$, the limit of the continuous function becomes the $\text{sgn}(net)$ function defined in (2.3b). Activation functions (2.3a) and (2.3b) are called *bipolar continuous* and *bipolar binary functions*, respectively. The word "bipolar" is used to point out that both positive and negative responses of neurons are produced for this definition of the activation function.

By shifting and scaling the bipolar activation functions defined by (2.3), *unipolar continuous* and *unipolar binary activation functions* can be obtained,

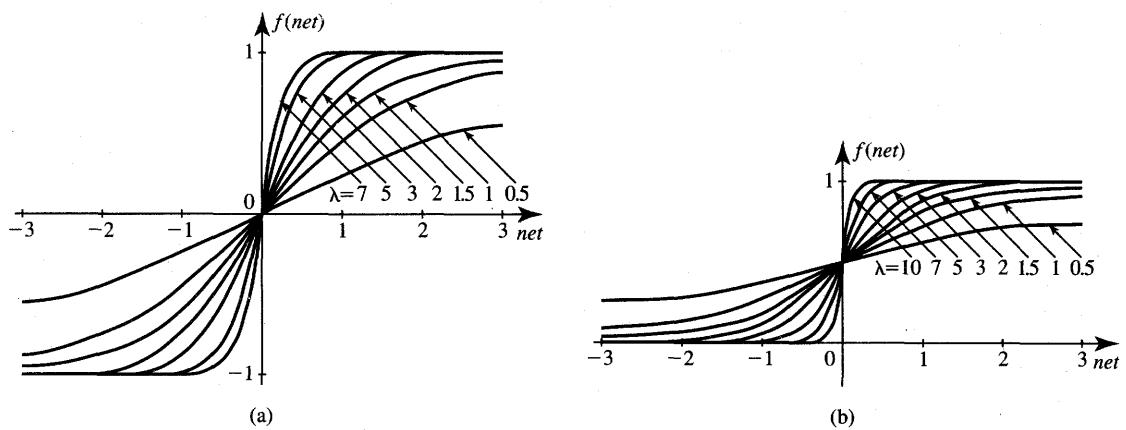


Figure 2.5 Activation functions of a neuron: (a) bipolar continuous and (b) unipolar continuous.

respectively, as

$$f(\text{net}) \triangleq \frac{1}{1 + \exp(-\lambda \text{net})} \quad (2.4a)$$

and

$$f(\text{net}) \triangleq \begin{cases} 1, & \text{net} > 0 \\ 0, & \text{net} \leq 0 \end{cases} \quad (2.4b)$$

Function (2.4a) is shown in Figure 2.5(b). Again, the unipolar binary function is the limit of $f(\text{net})$ in (2.4a) when $\lambda \rightarrow \infty$. The soft-limiting activation functions (2.3a) and (2.4a) are often called *sigmoidal characteristics*, as opposed to the *hard-limiting activation functions* given in (2.3b) and (2.4b). Hard-limiting activation functions describe the discrete neuron model.

Most neurons in this text employ bipolar activation functions. Some neural network architectures or applications do, however, specifically require the unipolar neuron responses. If this is the case, appropriate qualification for the type of activation function used is made. Essentially, any function $f(\text{net})$ that is monotonically increasing and continuous such that $\text{net} \in R$ and $f(\text{net}) \in (-1, 1)$ can be used instead of (2.3a) in neural modeling. A few neural models that often involve some form of feedback require the use of another type of nonlinearity than that defined in (2.3) and (2.4). An example activation function that is a unipolar ramp is shown in Figure 2.6.

If the neuron's activation function has the bipolar binary form of (2.3b), the symbol of Figure 2.4 can be replaced by the diagram shown in Figure 2.7(a), which is actually a discrete neuron functional block diagram showing summation performed by the summing node and the hard-limiting thresholding performed by the *threshold logic unit* (TLU). This model consists of the synaptic weights, a summing node, and the TLU element.

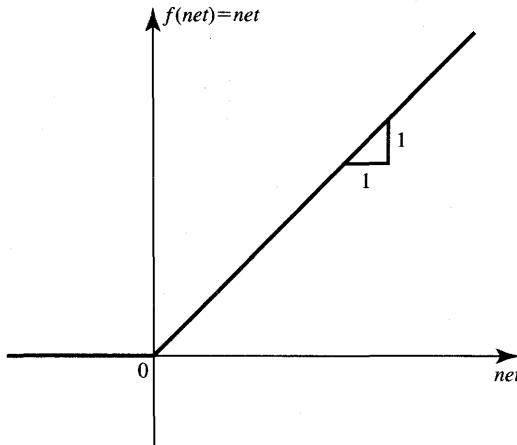


Figure 2.6 Unipolar ramp activation function.

In the case of the continuous activation function as in (2.3a), the model used is shown in Figure 2.7(b). The neuron is depicted as a summing high-gain saturating amplifier which amplifies its input signal $\mathbf{w}'\mathbf{x}$. The models in Figure 2.7(a) and (b) can be called *discrete (binary)* and *continuous perceptrons*, respectively. The discrete perceptron, introduced by Rosenblatt (1958), was the first learning machine. It can be viewed as a precursor of many of the neural network models that we use today. Also, its study provides considerable insight into the nature of artificial neural systems.

As has been shown, neuron outputs are either discrete (binary) or continuous. Given a layer of m neurons, their output values o_1, o_2, \dots, o_m can be arranged in a layer's output vector:

$$\mathbf{o} \triangleq [o_1 \quad o_2 \quad \cdots \quad o_m]^t \quad (2.5)$$

where o_i is the output signal of the i 'th neuron. The domains of vectors \mathbf{o} are defined in m -dimensional space as follows for $i = 1, 2, \dots, m$ (Hecht-Nielsen 1990):

$$(-1, 1)^m \equiv \{\mathbf{o} \in R^m, \quad o_i \in (-1, 1)\} \quad (2.6a)$$

or

$$(0, 1)^m \equiv \{\mathbf{o} \in R^m, \quad o_i \in (0, 1)\} \quad (2.6b)$$

for bipolar and unipolar continuous activation functions defined as in (2.3a) and (2.4a), respectively. It is evident that the domain of vector \mathbf{o} is, in this case, the interior of either the m -dimensional cube $(-1, 1)^m$ or of the cube $(0, 1)^m$.

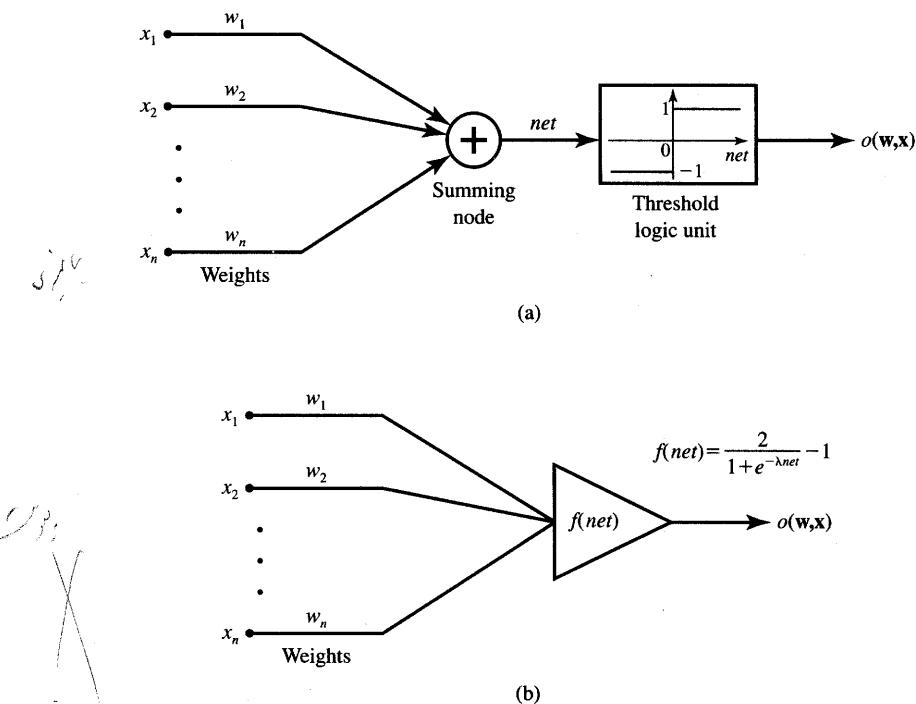


Figure 2.7 Common models of neurons with synaptic connections: (a) hard-limiting neuron (binary perceptron) and (b) soft-limiting neuron (continuous perceptron).

For binary-valued outputs o_i , the domains of \mathbf{o} in m -dimensional space for $i = 1, 2, \dots, m$, are

$$\{-1, 1\}^m \equiv \{\mathbf{o} \in R^m, o_i \in \{-1, 1\}\} \quad (2.7a)$$

or

$$\{0, 1\}^m \equiv \{\mathbf{o} \in R^m, o_i \in \{0, 1\}\} \quad (2.7b)$$

for bipolar and unipolar binary activation functions defined as in (2.3b) and (2.4b), respectively. The domains of vector \mathbf{o} are in this case vertices of either the m -dimensional cube $[-1, 1]^m$ or of the cube $[0, 1]^m$. Each vector \mathbf{o} having binary-valued entries can thus assume 2^m different values. Input vectors \mathbf{x} and their domains can be described similarly.

Artificial neural systems using the models defined by (2.1) through (2.4) do not involve the biological neuron features of delay, refractory period, or discrete-time operation. In fact, the neuron models listed so far in this section represent instantaneous, memoryless networks; i.e. they generate the output response determined only by the present excitation. A delay feature can be added to the instantaneous neuron model by adding an external delay element to make the ensemble of neurons operate with memory, as shown in the next section.

2.2

MODELS OF ARTIFICIAL NEURAL NETWORKS

Our introductory definition of artificial neural networks as physical cellular networks that are able to acquire, store, and utilize experiential knowledge has been related to the network's capabilities and performance. At this point, knowing the definition of the artificial neural network neuron model, we may benefit from another definition. The neural network can also be defined as *an interconnection of neurons, as defined in (2.1) through (2.4), such that neuron outputs are connected, through weights, to all other neurons including themselves; both lag-free and delay connections are allowed.* As research efforts continue, new and extended definitions may be developed, but this definition is sufficient for the introductory study of artificial neural architectures and algorithms found in this text.

Feedforward Network

Let us consider an elementary *feedforward architecture* of m neurons receiving n inputs as shown in Figure 2.8(a). Its output and input vectors are, respectively

$$\begin{aligned}\mathbf{o} &= [o_1 \quad o_2 \quad \cdots \quad o_m]^t \\ \mathbf{x} &= [x_1 \quad x_2 \quad \cdots \quad x_n]^t\end{aligned}\tag{2.8}$$

Weight w_{ij} connects the i 'th neuron with the j 'th input. The double subscript convention used for weights in this book is such that *the first and second subscript denote the index of the destination and source nodes, respectively.* We thus can write the activation value for the i 'th neuron as

$$net_i = \sum_{j=1}^n w_{ij}x_j, \quad \text{for } i = 1, 2, \dots, m\tag{2.9}$$

The following nonlinear transformation [Equation (2.10)] involving the activation function $f(net_i)$, for $i = 1, 2, \dots, m$, completes the processing of \mathbf{x} . The transformation, performed by each of the m neurons in the network, is a strongly nonlinear mapping expressed as

$$o_i = f(\mathbf{w}_i^t \mathbf{x}), \quad \text{for } i = 1, 2, \dots, m\tag{2.10}$$

where weight vector \mathbf{w}_i contains weights leading toward the i 'th output node and is defined as follows

$$\mathbf{w}_i \triangleq [w_{i1} \quad w_{i2} \quad \cdots \quad w_{in}]^t\tag{2.11}$$

Introducing the nonlinear matrix operator Γ , the mapping of input space x to

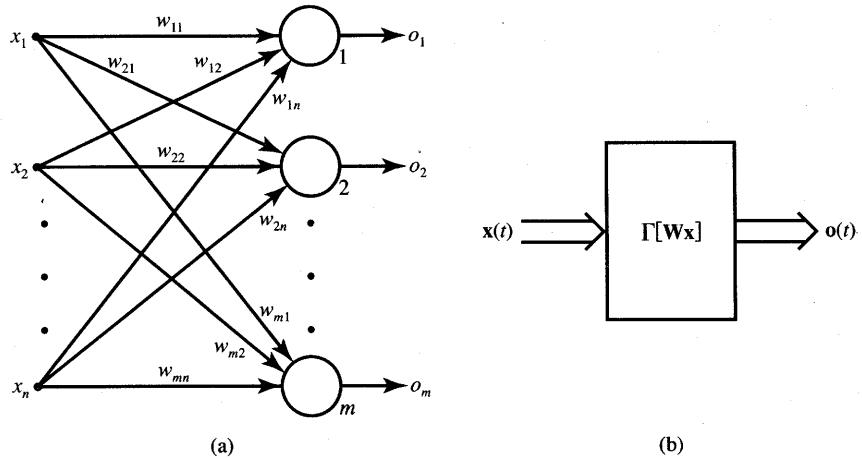


Figure 2.8 Single-layer feedforward network: (a) interconnection scheme and (b) block diagram.

output space o implemented by the network can be expressed as follows

$$\mathbf{o} = \Gamma[\mathbf{Wx}] \quad (2.12a)$$

where \mathbf{W} is the *weight matrix*, also called the *connection matrix*:

$$\mathbf{W} \triangleq \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix} \quad (2.12b)$$

and

$$\Gamma[\cdot] \triangleq \begin{bmatrix} f(\cdot) & 0 & \cdots & 0 \\ 0 & f(\cdot) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f(\cdot) \end{bmatrix} \quad (2.12c)$$

Note that the nonlinear activation functions $f(\cdot)$ on the diagonal of the matrix operator Γ operate componentwise on the activation values *net* of each neuron. Each activation value is, in turn, a scalar product of an input with the respective weight vector.

The input and output vectors \mathbf{x} and \mathbf{o} are often called *input* and *output patterns*, respectively. The mapping of an input pattern into an output pattern as shown in (2.12) is of the feedforward and instantaneous type, since it involves no time delay between the input \mathbf{x} , and the output \mathbf{o} . Thus, we can rewrite (2.12a)

in the explicit form involving time t as

$$\mathbf{o}(t) = \Gamma[\mathbf{Wx}(t)] \quad (2.13)$$

Figure 2.8(b) shows the block diagram of the feedforward network. As can be seen, the generic feedforward network is characterized by the lack of feedback. This type of network can be connected in cascade to create a multilayer network. In such a network, the output of a layer is the input to the following layer. Even though the feedforward network has no explicit feedback connection when $\mathbf{x}(t)$ is mapped into $\mathbf{o}(t)$, the output values are often compared with the “teacher’s” information, which provides the desired output value, and also an error signal can be employed for adapting the network’s weights. We will postpone more detailed coverage of feedforward network learning through adaptation to Sections 2.4 and 2.5.

EXAMPLE 2.1

This example presents the analysis of a two-layer feedforward network using neurons having the bipolar binary activation function given in (2.3b). The network to be analyzed is shown in Figure 2.9(a). Our purpose is to find output o_5 for a given network and input pattern. Each of the network layers is described by the formula (2.12a)

$$\mathbf{o} = \Gamma[\mathbf{Wx}] \quad (2.14)$$

By inspection of the network diagram, we obtain the output, input vectors, and the weight matrix for the first layer, respectively, as

$$\begin{aligned}\mathbf{o} &= [o_1 \ o_2 \ o_3 \ o_4]^t \\ \mathbf{x} &= [x_1 \ x_2 \ -1]^t \\ \mathbf{W}_1 &= \begin{bmatrix} 1 & 0 & 1 \\ -1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & -1 & -3 \end{bmatrix}\end{aligned}$$

Similarly, for the second layer we can write

$$\begin{aligned}\mathbf{o} &= [o_5] \\ \mathbf{x} &= [o_1 \ o_2 \ o_3 \ o_4 \ -1]^t \\ \mathbf{W}_2 &= [1 \ 1 \ 1 \ 1 \ 3.5]\end{aligned}$$

The response of the first layer can be computed for bipolar binary activation functions as below

$$\mathbf{o} = [\operatorname{sgn}(x_1 - 1) \ \operatorname{sgn}(-x_1 + 2) \ \operatorname{sgn}(x_2) \ \operatorname{sgn}(-x_2 + 3)]^t$$

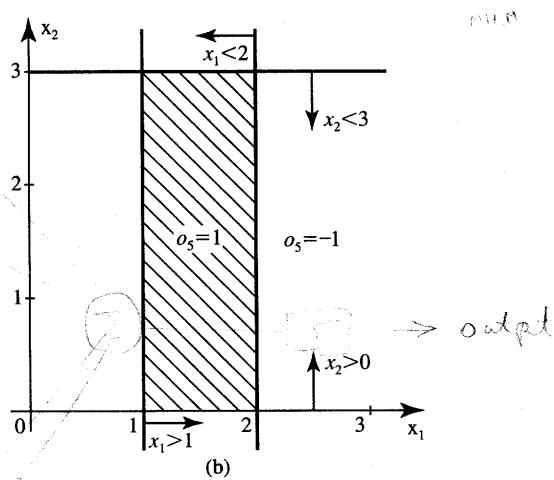
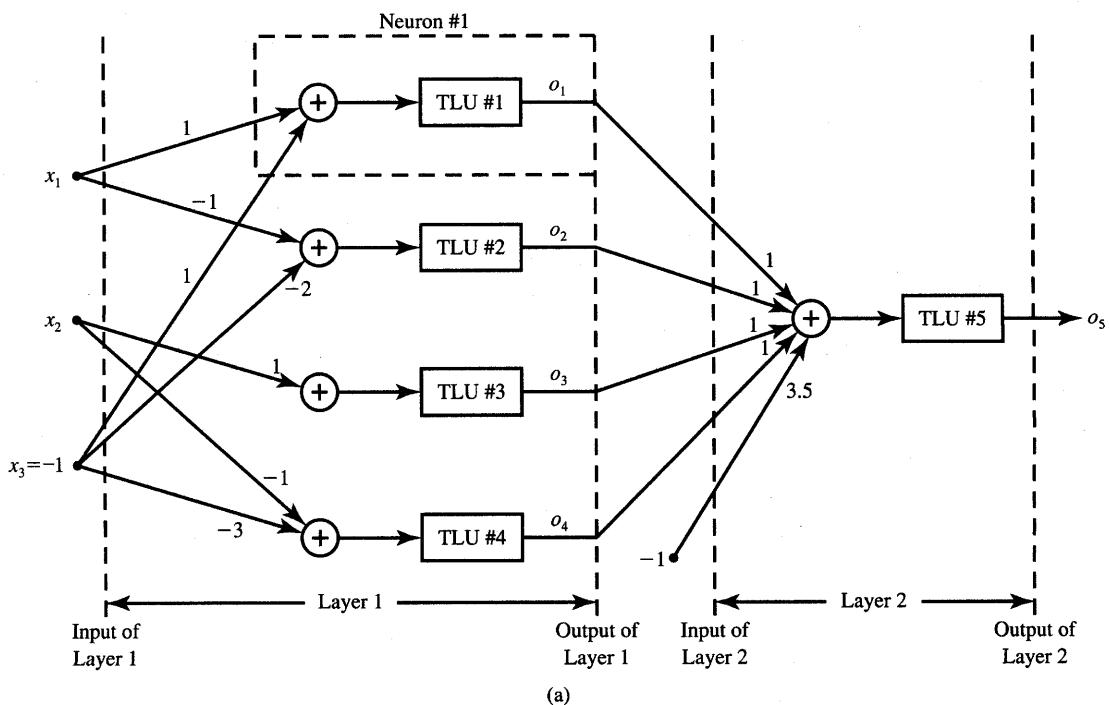
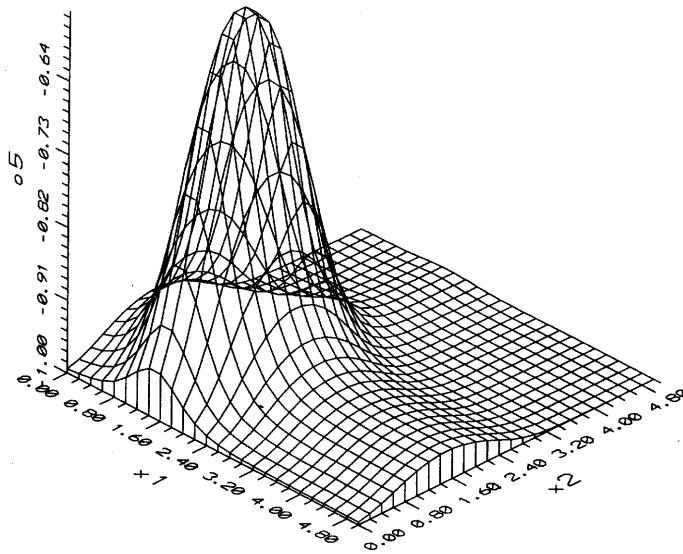


Figure 2.9a,b Example of two-layer feedforward network: (a) diagram and (b) two-dimensional space mapping (discrete activation function).

Let us discuss the mapping performed by the first layer. Note that each of the neurons 1 through 4 divides the plane x_1, x_2 into two half-planes. The half-planes where the neurons' responses are positive (+1) have been marked on Figure 2.9(b) with arrows pointing toward the positive response



(c)

Figure 2.9c Example of two-layer feedforward network (continued): (c) two-dimensional space mapping (continuous activation function, $\lambda = 2.5$).

half-plane. The response of the second layer can be easily obtained as

$$o_5 = \text{sgn}(o_1 + o_2 + o_3 + o_4 - 3.5)$$

Note that the fifth neuron responds $+1$ if and only if $o_1 = o_2 = o_3 = o_4 = 1$. It therefore selects the intersection of four half-planes produced by the first layer and designated by the arrows. Figure 2.9(b) shows that the network maps the shaded region of plane x_1, x_2 into $o_5 = 1$, and it maps its complement into $o_5 = -1$. In summary, the network of Figure 2.9(a) provides mapping of the entire x_1, x_2 plane into one of the two points ± 1 on the real number axis.

Let us look at the mapping provided by the same architecture but with neurons having sigmoidal characteristics. For the continuous bipolar activation function given in (2.3a), we obtain for the first layer

$$\mathbf{o} = \begin{bmatrix} \frac{2}{1 + \exp(1 - x_1)\lambda} - 1 \\ \frac{2}{1 + \exp(x_1 - 2)\lambda} - 1 \\ \frac{2}{1 + \exp(-x_2)\lambda} - 1 \\ \frac{2}{1 + \exp(x_2 - 3)\lambda} - 1 \end{bmatrix}$$

and for the second layer

$$o_5 = \frac{2}{1 + \exp(3.5 - o_1 - o_2 - o_3 - o_4)\lambda} - 1$$

The network with neurons having sigmoidal activation functions performs mapping as shown in Figure 2.9(c). The figure reveals that although some similarity exists with the discrete neuron case, the mapping is much more complex. The example shows how the two-dimensional space has been mapped into the segment of one-dimensional space. In summary, the network of Figure 2.9(a) with bipolar continuous neurons provides mapping of the entire x_1, x_2 plane into the interval $(-1, 1)$ on the real number axis. Similar mapping was shown earlier in Figure 1.3. In fact, neural networks with as few as two layers are capable of universal approximation from one finite dimensional space to another. ■

Feedback Network

A feedback network can be obtained from the feedforward network shown in Figure 2.8(a) by connecting the neurons' outputs to their inputs. The result is depicted in Figure 2.10(a). The essence of closing the feedback loop is to enable

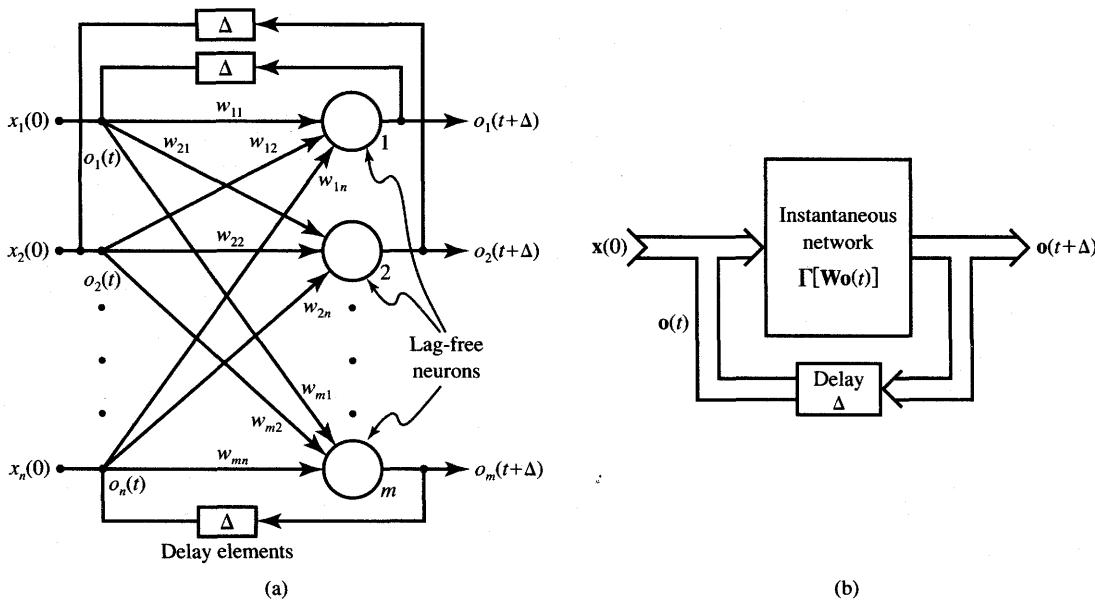


Figure 2.10 Single-layer discrete-time feedback network: (a) interconnection scheme and (b) block diagram.

control of output o_j through outputs o_j , for $j = 1, 2, \dots, m$. Such control is especially meaningful if the present output, say $\mathbf{o}(t)$, controls the output at the following instant, $\mathbf{o}(t + \Delta)$. The time Δ elapsed between t and $t + \Delta$ is introduced by the delay elements in the feedback loop as shown in Figure 2.10(a). Here the time delay Δ has a symbolic meaning; it is an analogy to the refractory period of an elementary biological neuron model. Using the notation introduced for feedforward networks, the mapping of $\mathbf{o}(t)$ into $\mathbf{o}(t + \Delta)$ can now be written as

$$\mathbf{o}(t + \Delta) = \Gamma[\mathbf{W}\mathbf{o}(t)] \quad (2.15)$$

This formula is represented by the block diagram shown in Figure 2.10(b). Note that the input $\mathbf{x}(t)$ is only needed to initialize this network so that $\mathbf{o}(0) = \mathbf{x}(0)$. The input is then removed and the system remains autonomous for $t > 0$. We thus consider here a special case of this feedback configuration, such that $\mathbf{x}(t) = \mathbf{x}(0)$ and no input is provided to the network thereafter, or for $t > 0$.

There are two main categories of single-layer feedback networks. If we consider time as a discrete variable and decide to observe the network performance at discrete time instants $\Delta, 2\Delta, 3\Delta, \dots$, the system is called *discrete-time*. For notational convenience, the time step in discrete-time networks is equated to unity, and the time instances are indexed by positive integers. Symbol Δ thus has the meaning of unity delay. The choice of indices as natural numbers is convenient since we initialize the study of the system at $t = 0$ and are interested in its response thereafter. For a discrete-time artificial neural system, we have converted (2.15) to the form

$$\mathbf{o}^{k+1} = \Gamma[\mathbf{W}\mathbf{o}^k], \quad \text{for } k = 1, 2, \dots \quad (2.16a)$$

where k is the instant number. The network in Figure 2.10 is called *recurrent* since its response at the $k + 1$ 'th instant depends on the entire history of the network starting at $k = 0$. Indeed, we have from (2.16a) a series of nested solutions as follows

$$\begin{aligned} \mathbf{o}^1 &= \Gamma[\mathbf{W}\mathbf{x}^0] \\ \mathbf{o}^2 &= \Gamma[\mathbf{W}\Gamma[\mathbf{W}\mathbf{x}^0]] \\ &\dots \\ \mathbf{o}^{k+1} &= \Gamma[\mathbf{W}\Gamma[\dots\Gamma[\mathbf{W}\mathbf{x}^0]\dots]] \end{aligned} \quad (2.16b)$$

Recurrent networks typically operate with a discrete representation of data; they employ neurons with a hard-limiting activation function. A system with discrete-time inputs and a discrete data representation is called an *automaton*. Thus, recurrent neural networks of this class can be considered as automatons.

Equations (2.16) describe what we call the *state* \mathbf{o}^k of the network at instants $k = 1, 2, \dots$, and they yield the sequence of *state transitions*. The network begins the state transitions once it is initialized at instant 0 with \mathbf{x}^0 , and it goes through

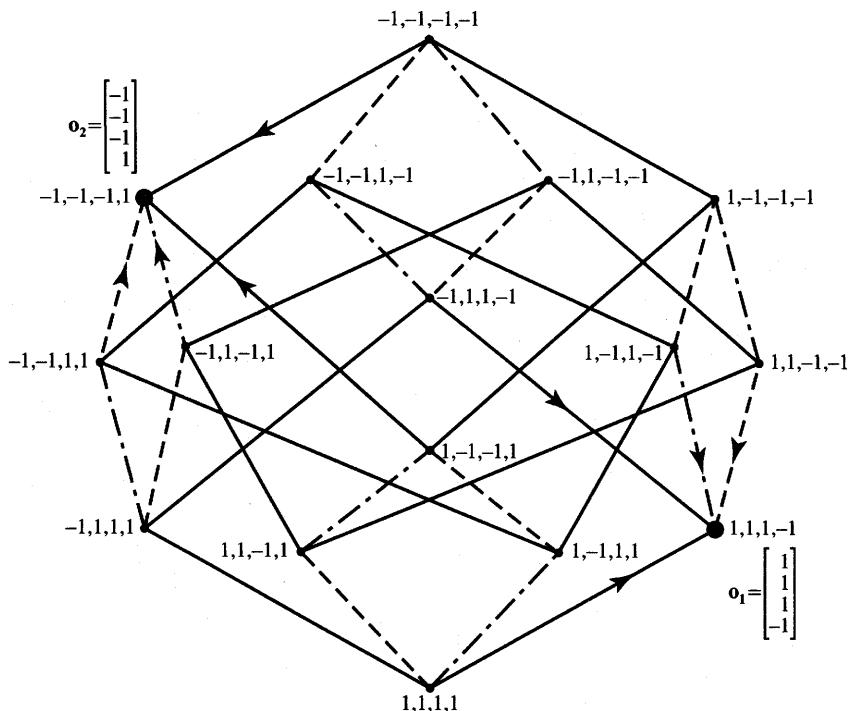


Figure 2.11 Four-dimensional hypercube with two equilibrium states in Example 2.2.

state transitions \mathbf{o}^k , for $k = 1, 2, \dots$, until it possibly finds an equilibrium state. This equilibrium state is often called an *attractor*. An attractor can consist of a single state or a limited number of states. The example network of Figure 1.5(a) can be considered as having a single-state attractor of value $\mathbf{o} = [1 \ 1 \ -1]'$. The sequence of states of a recurrent network is generally nondeterministic. In addition, there are often many equilibrium states that can potentially be reached by the network after a number of such nondeterministic transitions. These issues will be covered in more detail in Chapters 5 and 6.

EXAMPLE 2.2

This example covers the basic concept of state transitions and the analysis of a simple recurrent discrete-time network. Figure 2.11 shows a four-dimensional hypercube, the vertices of which represent four-dimensional bipolar binary vectors, or simply, binary numbers. The cube shown visualizes state vector values and possible transitions for the four-neuron network. Four edges terminate at each vertex, since vectors that are different by a

single bit component are connected with an edge. Thus, binary numbers joined by any single line can be considered to differ by only one digit. The vertices can be thought of as the output vector domain of a certain four-neuron discrete-time network with bipolar binary neurons.

Assume now that we know how to enforce the transition of states as marked by the arrows. We also can notice that each event of state transition ends at one of the two vectors as follows

$$\begin{aligned}\mathbf{o}_1 &= [1 \ 1 \ 1 \ -1]^t \\ \mathbf{o}_2 &= [-1 \ -1 \ -1 \ 1]^t\end{aligned}\quad (2.17)$$

which are apparently equilibria. Below is an example showing that a network can be devised that indeed has the equilibria as in (2.17).

Let us analyze an example of the discrete-time recurrent network shown in Figure 2.12. Its weight matrix can be set up by inspection as

$$\mathbf{W} = \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix} \quad (2.18)$$

First, it is easy to notice that none of the equilibrium states of (2.17) causes any further transitions in the network. Indeed, substituting $\mathbf{o}_1 = \mathbf{x}^0$ in (2.16b) and using (2.18) results in the first recurrence

$$\mathbf{o}^1 = \begin{bmatrix} \text{sgn}(\cdot) & 0 & 0 & 0 \\ 0 & \text{sgn}(\cdot) & 0 & 0 \\ 0 & 0 & \text{sgn}(\cdot) & 0 \\ 0 & 0 & 0 & \text{sgn}(\cdot) \end{bmatrix} \begin{bmatrix} \text{net}_1^0 \\ \text{net}_2^0 \\ \text{net}_3^0 \\ \text{net}_4^0 \end{bmatrix} \quad (2.19a)$$

or

$$\mathbf{o}^1 = [\text{sgn}(3) \ \text{sgn}(3) \ \text{sgn}(3) \ \text{sgn}(-3)]^t \quad (2.19b)$$

and thus no transitions take place thereafter since $\mathbf{o}^1 = \mathbf{o}^2 = \dots = \mathbf{o}_1$. The reader may easily verify that if $\mathbf{o}_2 = \mathbf{x}^0$ then $\mathbf{o}^1 = \mathbf{o}^2 = \dots = \mathbf{o}_2$, and no transitions take place either.

Assume now that the network is initialized at a state $\mathbf{x}^0 = [1 \ 1 \ 1 \ 1]^t$, which is adjacent to \mathbf{o}_1 . We may compute that the output \mathbf{o}^1 becomes

$$\mathbf{o}^1 = [\text{sgn}(1) \ \text{sgn}(1) \ \text{sgn}(1) \ \text{sgn}(-3)]^t \quad (2.20)$$

Therefore, the transition that takes place is

$$[1 \ 1 \ 1 \ 1] \rightarrow [1 \ 1 \ 1 \ -1]$$

and the network reaches its closest equilibrium state. The reader may easily

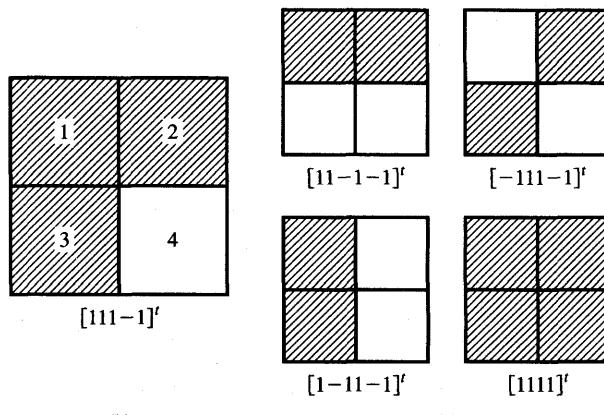
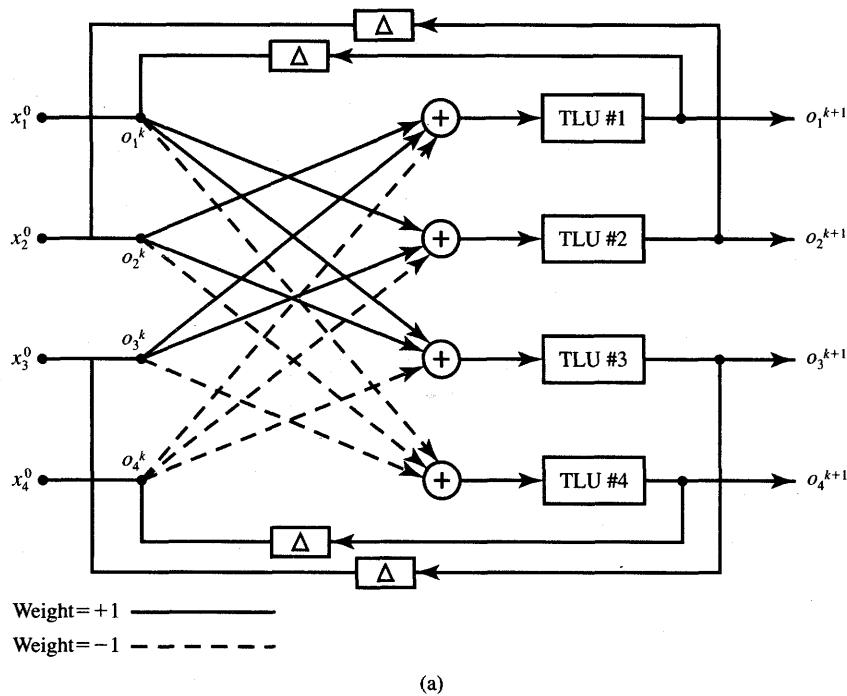


Figure 2.12 Recurrent network for two-equilibrium state diagram of Figure 2.11: (a) diagram, (b) bit map of equilibrium state vector \mathbf{o}_1 , and (c) example bit maps that converge to the bit map of part (b).

verify that for any of the initial states as below

$$\mathbf{x}^0 = [1 \quad 1 \quad -1 \quad -1]^t, \quad \mathbf{x}^0 = [1 \quad -1 \quad 1 \quad -1]^t, \text{ or}$$

$$\mathbf{x}^0 = [-1 \quad 1 \quad 1 \quad -1]^t$$

there will be subsequent transitions such that $\mathbf{o}^1 = \mathbf{o}_1$ and the network will always settle at \mathbf{o}_1 .

Bipolar binary vectors can be visualized in a bit-map form so that black and white elements of a bit map correspond to 1 and -1, respectively. The bit map in Figure 2.12(b) would thus represent \mathbf{o}_1 . Note that the four bit maps of Figure 2.12(c), if represented as vectors, can be regarded as the initial states. Transitions from these states end at the equilibrium point \mathbf{o}_1 . It is seen that the initial state converges in the network of Figure 2.12(a) to its closest equilibrium distant only by one bit. In cases when the initial state differs by two bits from both equilibria, the initializing input vector is equidistant to both \mathbf{o}_1 and \mathbf{o}_2 . Thus, convergence to either of the equilibria would be justifiable. This assumes that the network should seek out the output which is the most similar to the initializing binary vector. Although we have not shown yet how to encode the desired equilibrium states and how to control transitions, the discussion in this example helps in understanding the concept of state transitions and the analysis of recurrent networks. ■

In the discussion of the feedback concept, a discrete time delay Δ has been used between the input and output. The insertion of a unity delay element in the network feedback loop has made it possible to outline the formal definition of a recurrent network operating at discrete instants $k = 1, 2, \dots$, upon its initialization at $k = 0$. To generalize the ideas just discussed, note that the feedback concept can also be implemented with any infinitesimal delay between output and input introduced in the feedback loop. The consequence of an assumption of such delay between input and output is that the output vector can be considered to be a continuous-time function. As a result, the entire network operates in continuous time. It can be seen that a *continuous-time* network can be obtained by replacing delay elements in Figure 2.10 with suitable continuous-time lag producing components.

An example of such an elementary delay network is shown in Figure 2.13(a). It is a simple electric network consisting of resistance and capacitance. In fact, electric networks are very often used to model the computation performed by neural networks. Electric networks possess the flexibility to model all linear and nonlinear phenomena encountered in our neural network studies. Because of this flexibility, they will often represent working physical models of neural networks. The differential equation relating v_2 and v_1 , the output and input voltages, respectively, is

$$\frac{dv_2}{dt} + \frac{v_2}{RC} = \frac{v_1}{RC} \quad (2.21)$$

The change of output voltage Δv_2 occurring within the small time interval Δt can be approximately expressed from (2.21) as follows:

$$\Delta v_2 \cong \frac{\Delta t}{C} \cdot \frac{v_1 - v_2}{R} \quad (2.22)$$

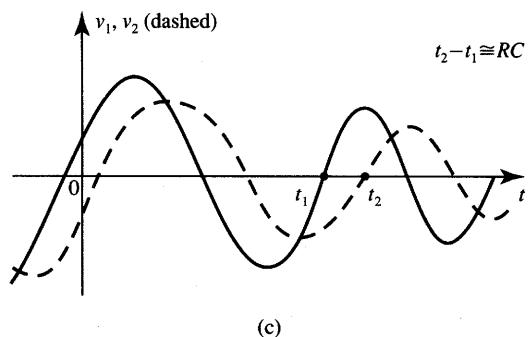
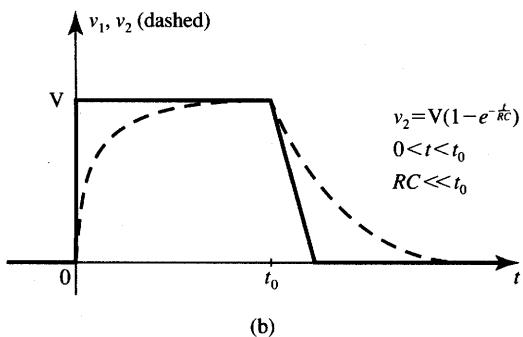
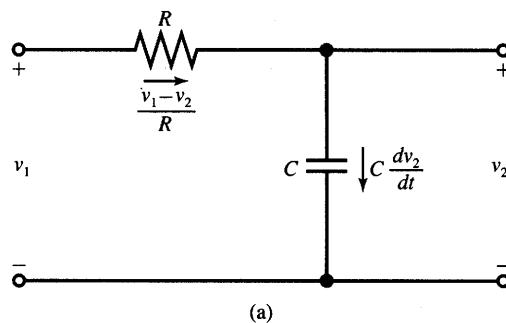


Figure 2.13 Feedback connection in continuous-time neural network: (a) elementary connecting circuit, (b) output response due to an impulse, and (c) output wave due to a damped harmonic input wave.

From (2.22) we see that for fixed C and Δt , increases of output voltage v_2 are proportional to the difference between v_1 and v_2 . If v_1 is kept constant or is varied slowly with respect to the time constant RC , v_2 approximately follows v_1 in time, with a small delay. Examples of input waveforms v_1 and output responses v_2 are shown in Figures 2.13(b) and (c). Figure 2.13(b) shows $v_2(t)$ resulting from $v_1(t)$ being an impulse excitation. Figure 2.13(c) shows $v_2(t)$ due to a damped harmonic input wave excitation. Although such a wave is not very likely to be observed in artificial neural systems, it shows the explicit delay of v_2 where v_2 lags v_1 by approximately RC .

Usually, continuous-time networks employ neurons with continuous activation functions. An elementary synaptic connection using the delay network given in Figure 2.13(a) is shown in Figure 2.14. The resistance R_{ij} serves as a weight from the output of the j 'th neuron to the input of the i 'th neuron. Using the finite time interval Δt , Equation (2.21) can be discretized as

$$\frac{\text{net}_i^{k+1} - \text{net}_i^k}{\Delta t} \cong \frac{1}{R_{ij}C_i} (o_j^k - \text{net}_i^k) \quad (2.23a)$$

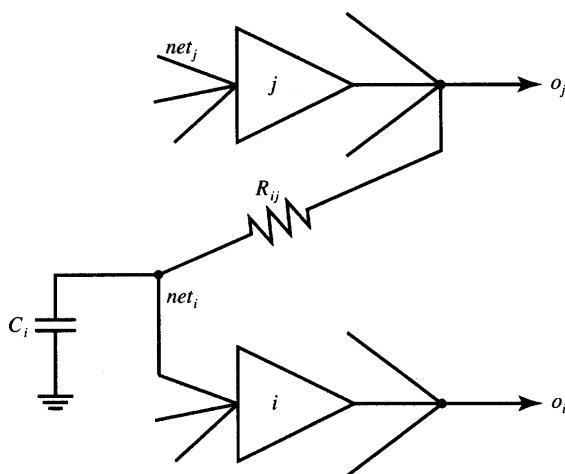


Figure 2.14 Elementary synaptic connection within continuous-time network.

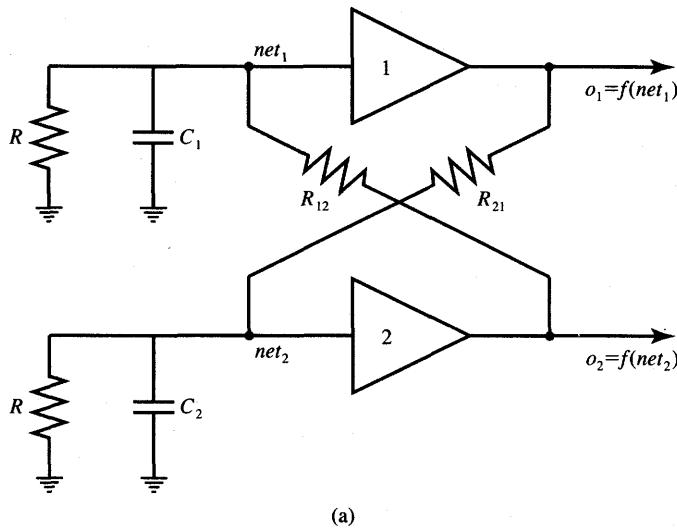
The activation of the *i*'th neuron at the instant $k + 1$ can be expressed as

$$net_i^{k+1} \cong net_i^k + \frac{\Delta t}{R_{ij}C_i} (o_j^k - net_i^k) \quad (2.23b)$$

As can be seen, the contribution to net_i by the *j*'th neuron is distributed in time according to (2.23b), where Δt denotes an infinitesimal time step. When n neurons are connected to the input of the *i*'th neuron as shown in Figure 2.14, expression (2.23b) needs to be computed for $j = 1, 2, \dots, n$ and summed. Numerous studies indicate that for n neurons interconnected as shown, fairly complex dynamic behavior of the network is usually obtained. The accurate description of transitions, which have been very simple for the recurrent discrete-time networks as demonstrated in Example 2.2, requires solving nonlinear differential equations for continuous-time networks. A more detailed discussion of properties of this class of networks is presented in Chapter 5. The introductory example below illustrates the main features and time-domain performance of continuous-time feedback networks.

EXAMPLE 2.3

Let us consider an electric continuous-time network of two neurons as shown in Figure 2.15(a). The matrix form description, such as (2.12) for feedforward networks and (2.15) for discrete-time feedback networks, will be discussed in Chapter 5. In this example, we obtain the network equations by inspection of its diagram. We also discuss the network's main properties in terms of its dynamical behavior.



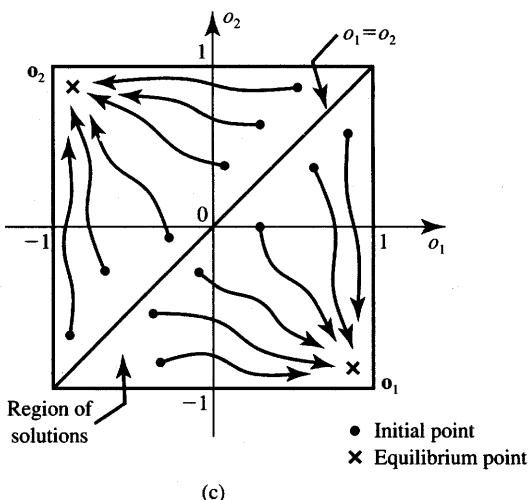
(a)

Simulation Results

Step	Run 1		Run 2		Run 3		Run 4	
	o_1	o_2	o_1	o_2	o_1	o_2	o_1	o_2
0	0.245	-0.124	0.462	-0.245	0.555	-0.555	0.704	-0.848
25	0.264	-0.146	0.493	-0.284	0.598	-0.598	0.749	-0.875
50	0.284	-0.170	0.526	-0.326	0.642	-0.642	0.790	-0.899
75	0.307	-0.195	0.560	-0.370	0.685	-0.685	0.827	-0.919
100	0.332	-0.224	0.596	-0.416	0.727	-0.727	0.860	-0.937
125	0.360	-0.254	0.633	-0.464	0.767	-0.767	0.888	-0.951
150	0.389	-0.287	0.671	-0.513	0.804	-0.804	0.912	-0.963
175	0.421	-0.323	0.708	-0.563	0.838	-0.838	0.932	-0.973
200	0.456	-0.361	0.745	-0.612	0.868	-0.868	0.948	-0.980
225	0.492	-0.402	0.781	-0.661	0.894	-0.894	0.961	-0.986
250	0.530	-0.445	0.814	-0.707	0.916	-0.916	0.971	-0.990
275	0.570	-0.489	0.845	-0.751	0.935	-0.935	0.979	-0.993
300	0.611	-0.535	0.872	-0.791	0.950	-0.950	0.985	-0.995
325	0.652	-0.582	0.897	-0.828	0.963	-0.963	0.989	-0.997
350	0.692	-0.628	0.918	-0.860	0.972	-0.972	0.993	-0.998
375	0.732	-0.674	0.936	-0.888	0.980	-0.980	0.995	-0.999
400	0.770	-0.718	0.951	-0.912	0.986	-0.986	0.997	-0.999
425	0.806	-0.760	0.963	-0.931	0.990	-0.990	0.998	-0.999
450	0.838	-0.798	0.973	-0.948	0.993	-0.993	0.999	-1.000
475	0.868	-0.833	0.980	-0.960	0.995	-0.995	0.999	-1.000
500	0.893	-0.864	0.986	-0.971	0.997	-0.997	0.999	-1.000

(b)

Figure 2.15a,b Continuous-time network from Example 2.3: (a) diagram, and (b) sample iterations for o_1 , o_2



(c)

Figure 2.15c Continuous-time network from Example 2.3 (continued): (c) illustration for equilibrium search.

The neuron outputs are connected to their inputs using a simple RC delay network, as in Figure 2.13(a). In addition, resistances R between each neuron input and ground are added to account for the neurons' finite input resistances. Such resistances model nonzero input currents absorbed by actual neurons. The differential equations obtained by summing currents in input nodes of the network being input neuron nodes are

$$\begin{aligned} C_1 \frac{dnet_1}{dt} &= \frac{o_2 - net_1}{R_{12}} - \frac{net_1}{R} \\ C_2 \frac{dnet_2}{dt} &= \frac{o_1 - net_2}{R_{21}} - \frac{net_2}{R} \end{aligned} \quad (2.24)$$

where $o_i = f(net_i)$, $i = 1, 2$, and $f(net)$ is assumed to be given by (2.3a). We will also assume that $C_1 = C_2$ and $R_{12} = R_{21} < 0$ for this network. Although this version of the circuit involves negative resistances, we will show in Chapter 5 that they will not be needed to build actual networks. Before an analysis of this network is pursued, let us try to approach the problem intuitively.

Assume that this circuit has been initialized by storing a charge at the input capacitances, and it is allowed thereafter to seek its own equilibrium. Our initial assumption is that starting with observations of the network at $t = 0$, the value o_1 has slightly increased. This would, after a small delay Δt introduced by the RC network, lower the value of net_2 since R_{21} is negative. Since $f(net_2)$ is a monotonically increasing function, o_2 would also decrease within the $(0, \Delta t)$ interval. This in turn would increase net_1 and, as a final

consequence, the original initial increase of o_1 would be enhanced due to the feedback interaction described. We have the cause-and-effect relationship, where the effect increases the initial cause in a closed, positive feedback, loop. If this interaction is allowed to continue for some time, the increases of o_1 will eventually slow down, since $f(\text{net})$ flattens out far from the origin, when $o_1 \rightarrow 1^-$. Noticeably, as o_1 increases, o_2 decreases symmetrically toward -1^+ (a^+ , a^- are "right" and "left" neighborhoods, respectively, of the point a while the point a is excluded). This discussion indicates that the network of Figure 2.15(a) for high-gain neurons would seek its equilibrium, or $[o_1 \ o_2] \rightarrow [1^- \ -1^+]$.

The correct conclusion, however, is not unique. If the initial hypothesis on the increase of o_1 is negated, the conclusion from a similar discussion as above is just the opposite and the network seeks its equilibrium such that $[o_1 \ o_2] \rightarrow [-1^+ \ 1^-]$. Obviously, the network cannot simultaneously be seeking two different solutions and move along opposite paths. Either the first or the second hypothesis must be true, but it is somewhat difficult to pick the right one at this point.

To gain better insight into the dynamics of the network, the equations (2.24) must be solved numerically since the closed-form solutions do not exist. Discretizing the differential equation (2.24) similar to (2.23), we obtain

$$\begin{aligned} \text{net}_1^{k+1} &\equiv \text{net}_1^k + \frac{\Delta t}{R_{12}C_1} (o_2^k - \text{net}_1^k) - \frac{\text{net}_1^k}{RC_1} \Delta t \\ \text{net}_2^{k+1} &\equiv \text{net}_2^k + \frac{\Delta t}{R_{21}C_2} (o_1^k - \text{net}_2^k) - \frac{\text{net}_2^k}{RC_2} \Delta t \end{aligned} \quad (2.25)$$

Notice that the numerical integration method chosen in (2.25) (forward Euler) is simple but usually not recommended for more sophisticated numerical computation. However, it is adequate for the purpose of this example and is also consistent with formulas (2.23). Sample results of simulations are partially listed in Figure 2.15(b) linking o_1 and o_2 as a function of k . Conditions chosen for the simulation are $\lambda = 2.5$, $dt = 0.002$, $R = 10$, and $R_{12}C_1 = R_{21}C_2 = -1$. Initial conditions are listed for $k = 0$ in the first row of the table for each of the four simulations shown. All of the partially listed runs converge to the solution $[o_1 \ o_2] = [1^- \ -1^+]$.

The schematic illustration for more general cases of the equilibrium search is shown in Figure 2.15(c) for $-1 \leq o_1 \leq 1$ and $-1 \leq o_2 \leq 1$. The trajectories indicate the computed solution functions $o_2(o_1)$ with time being a parameter. The line $o_2 = o_1$ separates the two regions of attractions. For initial condition $o_1^0 > o_2^0$, $[1^- \ -1^+]$ is an equilibrium solution, otherwise the solution is at $[-1^+ \ 1^-]$. Note that this network maps the entire two-dimensional input space into two points, which are roughly expressed as $\mathbf{o}_1 \approx \sqrt{2} \angle -45^\circ$ and $\mathbf{o}_2 \approx \sqrt{2} \angle 135^\circ$. In contrast to feedforward networks, as shown in Figure 2.9(a), feedback networks require time to provide response to the mapping problem.

2.3

NEURAL PROCESSING

The material of the previous section has focused mainly on the computation of response \mathbf{o} for a given input \mathbf{x} for several important classes of neural networks. The analysis of three example networks shown in Figures 2.9(a), 2.12(a), and 2.15(a) has also been discussed. The process of computation of \mathbf{o} for a given \mathbf{x} performed by the network is known as *recall*. Recall is the proper processing phase for a neural network, and its objective is to retrieve the information. Recall corresponds to the decoding of the stored content which may have been encoded in a network previously. In this section we are mainly concerned with the recall mode and the tasks it can accomplish. Based on the preliminary observations made so far, we now outline the basic forms of neural information processing.

Assume that a set of patterns can be stored in the network. Later, if the network is presented with a pattern similar to a member of the stored set, it may associate the input with the closest stored pattern. The process is called *autoassociation*. Typically, a degraded input pattern serves as a cue for retrieval of its original form. This is illustrated schematically in Figure 2.16(a). The figure shows a distorted square recalling the square encoded. Another example of auto-association is provided in the Figure 1.8.

Associations of input patterns can also be stored in a *heteroassociation* variant. In heteroassociative processing, the associations between pairs of patterns are stored. This is schematically shown in Figure 2.16(b). A square input pattern presented at the input results in the rhomboid at the output. It can be inferred that the rhomboid and square constitute one pair of stored patterns. A distorted input pattern may also cause correct heteroassociation at the output as shown with dashed line.

Classification is another form of neural computation. Let us assume that a set of input patterns is divided into a number of classes, or categories. In response to an input pattern from the set, the classifier is supposed to recall the information regarding class membership of the input pattern. Typically, classes are expressed by discrete-valued output vectors, and thus output neurons of classifiers would employ binary activation functions. The schematic diagram illustrating the

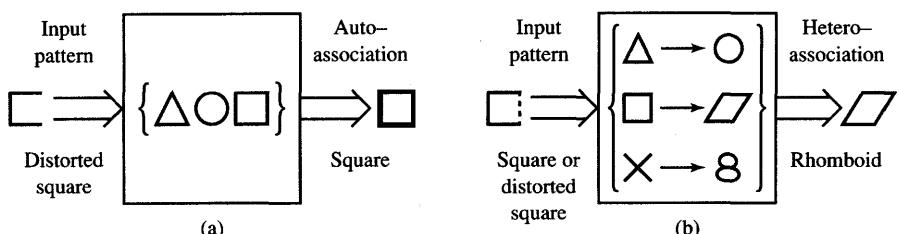


Figure 2.16 Association response: (a) autoassociation and (b) heteroassociation.

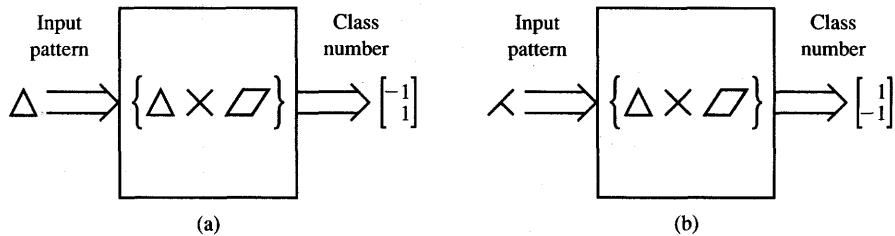


Figure 2.17 Classification response: (a) classification and (b) recognition.

classification response for patterns belonging to three classes is shown in Figure 2.17(a).

Interestingly, classification can be understood as a special case of heteroassociation. The association is now between the input pattern and the second member of the heteroassociative pair, which is supposed to indicate the input's class number. If the network's desired response is the class number but the input pattern does not exactly correspond to any of the patterns in the set, the processing is called *recognition*. When a class membership for one of the patterns in the set is recalled, recognition becomes identical to classification. Recognition within the set of three patterns is schematically shown in Figure 2.17(b). This form of processing is of particular significance when an amount of noise is superimposed on input patterns.

One of the distinct strengths of neural networks is their ability to generalize. The network is said to generalize well when it sensibly interpolates input patterns that are new to the network. Assume that a network has been trained using the data x_1 through x_5 , as shown in Figure 2.18. The figure illustrates bad and good generalization examples at points that are new and are between the training points. Neural networks provide, in many cases, input-output mappings with good generalization capability.

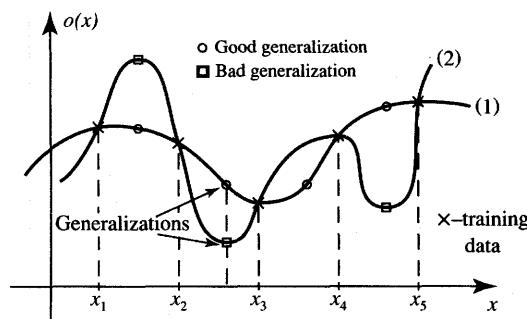


Figure 2.18 Generalization example.

When reviewing the different aspects of neural processing in this section, the assumption has been made that neural networks are able to store data. So far, we discussed the retrieval of network responses without covering any possible methods of data storage. Data are stored in a network as a result of its learning. The following section introduces basic concepts of learning.

2.4

LEARNING AND ADAPTATION

Each of us acquires and then hones our skills and abilities through the basic phenomenon of learning. Learning is a fundamental subject for psychologists, but it also underlies many of the topics in this book. In general, learning is a relatively permanent change in behavior brought about by experience. Learning in human beings and animals is an inferred process; we cannot see it happening directly and we can assume that it has occurred by observing changes in performance. Learning in neural networks is a more direct process, and we typically can capture each learning step in a distinct cause-effect relationship. To perform any of the processing tasks discussed in the previous section, neural network learning of an input-output mapping from a set of examples is needed. Designing an associator or a classifier can be based on learning a relationship that transforms inputs into outputs given a set of examples of input-output pairs. A classical framework for this problem is provided by approximation theory (Poggio and Girosi 1990).

Learning as Approximation or Equilibria Encoding

Approximation theory focuses on approximating a continuous, multivariable function $h(\mathbf{x})$ by another function $H(\mathbf{w}, \mathbf{x})$, where $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_n]^t$ is the input vector and $\mathbf{w} = [w_1 \ w_2 \ \dots \ w_m]^t$ is a parameter (weight) vector. In the approach below, we will look at a class of neural networks as systems that can learn approximation of relationships. The learning task is to find \mathbf{w} that provides the best possible approximation of $h(\mathbf{x})$ based on the set of training examples $\{\mathbf{x}\}$. An important choice that needs to be made is which approximation function $H(\mathbf{w}, \mathbf{x})$ to use. An ill-chosen, nonsmooth, approximation function example is shown in Figure 2.18 as curve (2). Even with the best choice of parameters for an ill-chosen function, the approximation is inaccurate between successive data points. The choice of function $H(\mathbf{w}, \mathbf{x})$ in order to represent $h(\mathbf{x})$ is called a *representation* problem. Once $H(\mathbf{w}, \mathbf{x})$ has been chosen, the network learning algorithm is applied for finding optimal parameters \mathbf{w} . A more precise

formulation of the learning problem can be stated as calculation involving w^* such that (Poggio and Girosi 1990):

$$\rho[H(w^*, x), h(x)] \leq \rho[H(w, x), h(x)] \quad (2.26)$$

where $\rho[H(w, x), h(x)]$, or distance function, is a measure of approximation quality between $H(w, x)$ and $h(x)$. When the fit is judged according to the sum of squared differences taken for the set of training examples $\{x\}$, the distance has a form of sum of squared errors. As will be shown in Chapters 3 and 4, the feed-forward networks, both single-layer and multilayer, can be taught to perform the desired mappings as described. In this chapter, only the main learning principles will be introduced.

In contrast to feedforward networks, which store mapping that can be recalled instantaneously, feedback networks are dynamical systems. The mappings in feedback networks are encoded in the equilibrium states. Similar to approximation learning, weights also determine the properties of feedback networks. Learning in feedback networks corresponds to equilibria encoding. Usually this is accomplished by a so-called “recording process,” but stepwise learning approaches have also been developed for this class of networks. Equilibrium states learning will be discussed in more detail in Chapters 5 and 6.

Supervised and Unsupervised Learning

Under the notion of learning in a network, we will consider a process of forcing a network to yield a particular response to a specific input. A particular response may or may not be specified to provide external correction. Learning is necessary when the information about inputs/outputs is unknown or incomplete *a priori*, so that no design of a network can be performed in advance. The majority of the neural networks covered in this text requires training in a supervised or unsupervised learning mode. Some of the networks, however, can be designed without incremental training. They are designed by *batch learning* rather than stepwise training.

Batch learning takes place when the network weights are adjusted in a single training step. In this mode of learning, the complete set of input/output training data is needed to determine weights, and feedback information produced by the network itself is not involved in developing the network. This learning technique is also called *recording*. Learning with feedback either from the teacher or from the environment rather than a teacher, however, is more typical for neural networks. Such learning is called incremental and is usually performed in steps.

The concept of feedback plays a central role in learning. The concept is highly elusive and somewhat paradoxical. In a broad sense it can be understood as an introduction of a pattern of relationships into the cause-and-effect path. We will distinguish two different types of learning: learning *with* supervision versus

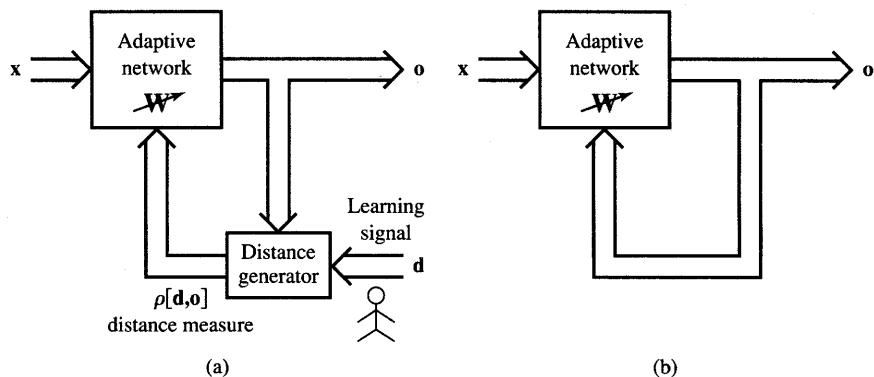


Figure 2.19 Block diagram for explanation of basic learning modes: (a) supervised learning and (b) unsupervised learning.

learning *without* supervision. The learning types block diagrams are illustrated in Figure 2.19.

In *supervised learning* we assume that at each instant of time when the input is applied, the desired response \mathbf{d} of the system is provided by the teacher. This is illustrated in Figure 2.19(a). The distance $\rho[\mathbf{d}, \mathbf{o}]$ between the actual and the desired response serves as an error measure and is used to correct network parameters externally. Since we assume adjustable weights, the teacher may implement a reward-and-punishment scheme to adapt the network's weight matrix \mathbf{W} . For instance, in learning classifications of input patterns or situations with known responses, the error can be used to modify weights so that the error decreases. This mode of learning is very pervasive. Also, it is used in many situations of natural learning. A set of input and output patterns called a *training set* is required for this learning mode.

Typically, supervised learning rewards accurate classifications or associations and punishes those which yield inaccurate responses. The teacher estimates the negative error gradient direction and reduces the error accordingly. In many situations, the inputs, outputs and the computed gradient are deterministic, however, the minimization of error proceeds over all its random realizations. As a result, most supervised learning algorithms reduce to stochastic minimization of error in multi-dimensional weight space.

Figure 2.19(b) shows the block diagram of unsupervised learning. In *learning without supervision*, the desired response is not known; thus, explicit error information cannot be used to improve network behavior. Since no information is available as to correctness or incorrectness of responses, learning must somehow be accomplished based on observations of responses to inputs that we have marginal or no knowledge about. For example, unsupervised learning can easily result in finding the boundary between classes of input patterns distributed as shown

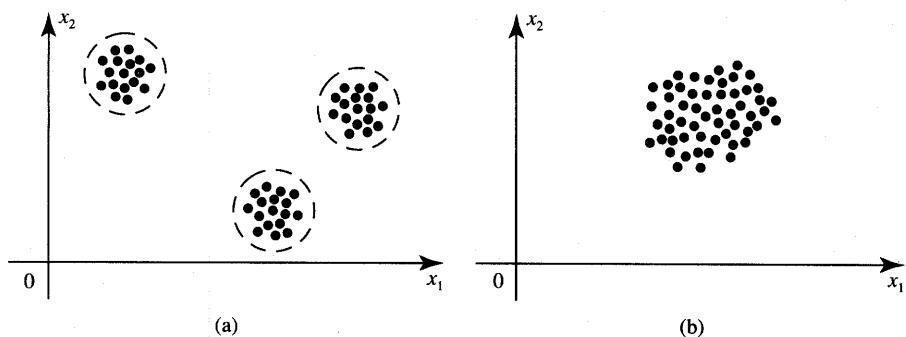


Figure 2.20 Two-dimensional patterns: (a) clustered and (b) no apparent clusters.

in Figure 2.20. In a favorable case, as in Figure 2.20(a), cluster boundaries can be found based on the large and representative sample of inputs. Suitable weight self-adaptation mechanisms have to be embedded in the trained network, because no external instructions regarding potential clusters are available. One possible network adaptation rule is: A pattern added to the cluster has to be closer to the center of the cluster than to the center of any other cluster.

Unsupervised learning algorithms use patterns that are typically redundant raw data having no labels regarding their class membership, or associations. In this mode of learning, the network must discover for itself any possibly existing patterns, regularities, separating properties, etc. While discovering these, the network undergoes change of its parameters, which is called *self-organization*.

The technique of unsupervised learning is often used to perform clustering as the unsupervised classification of objects without providing information about the actual classes. This kind of learning corresponds to minimal *a priori* information available. Some information about the number of clusters, or similarity versus dissimilarity of patterns, can be helpful for this mode of learning. Finally, learning is often not possible in an unsupervised environment, as would probably be true in the case illustrated in Figure 2.20(b) showing pattern classes not easily discernible even for a human.

Unsupervised learning is sometimes called learning without a teacher. This terminology is not the most appropriate, because learning without a teacher is not possible at all. Although, the teacher does not have to be involved in every training step, he has to set goals even in an unsupervised learning mode (Tsyplkin 1973). We may think of the following analogy. Learning with supervision corresponds to classroom learning with the teacher's questions answered by students and corrected, if needed, by the teacher. Learning without supervision corresponds to learning the subject from a videotape lecture covering the material but not including any other teacher's involvement. The teacher lectures directions and methods, but is not available. Therefore, the student cannot get explanations of unclear questions, check answers and become fully informed.

2.5

NEURAL NETWORK LEARNING RULES

Our focus in this section will be artificial neural network learning rules. A neuron is considered to be an adaptive element. Its weights are modifiable depending on the input signal it receives, its output value, and the associated teacher response. In some cases the teacher signal is not available and no error information can be used, thus the neuron will modify its weights based only on the input and/or output. This is the case for unsupervised learning.

Let us study the learning of the weight vector \mathbf{w}_i , or its components w_{ij} connecting the j 'th input with the i 'th neuron. The trained network is shown in Figure 2.21 and uses the neuron symbol from Figure 2.4. In general, the j 'th input can be an output of another neuron or it can be an external input. Our discussion in this section will cover single-neuron and single-layer network supervised learning and simple cases of unsupervised learning. Under different learning rules, the form of the neuron's activation function may be different. Note that the threshold parameter may be included in learning as one of the weights. This would require fixing one of the inputs, say x_n . We will assume here that x_n , if fixed, takes the value of -1 .

The following *general learning rule* is adopted in neural network studies (Amari 1990): *The weight vector $\mathbf{w}_i = [w_{i1} \ w_{i2} \ \cdots \ w_{in}]^t$ increases in proportion to the product of input \mathbf{x} and learning signal r .* The learning signal r is in general a function of \mathbf{w}_i , \mathbf{x} , and sometimes of the teacher's signal d_i . We thus have for the network shown in Figure 2.21:

$$r = r(\mathbf{w}_i, \mathbf{x}, d_i) \quad (2.27)$$

The increment of the weight vector \mathbf{w}_i produced by the learning step at time t according to the general learning rule is

$$\Delta\mathbf{w}_i(t) = cr [\mathbf{w}_i(t), \mathbf{x}(t), d_i(t)] \mathbf{x}(t) \quad (2.28)$$

where c is a positive number called the *learning constant* that determines the rate of learning. The weight vector adapted at time t becomes at the next instant, or learning step,

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + cr [\mathbf{w}_i(t), \mathbf{x}(t), d_i(t)] \mathbf{x}(t) \quad (2.29a)$$

The superscript convention will be used in this text to index the discrete-time training steps as in Eq. (2.29a). For the k 'th step we thus have from (2.29a) using this convention

$$\mathbf{w}_i^{k+1} = \mathbf{w}_i^k + cr(\mathbf{w}_i^k, \mathbf{x}^k, d_i^k) \mathbf{x}^k \quad (2.29b)$$

The learning in (2.29) assumes the form of a sequence of discrete-time weight modifications. Continuous-time learning can be expressed as

$$\frac{d\mathbf{w}_i(t)}{dt} = cr\mathbf{x}(t) \quad (2.30)$$

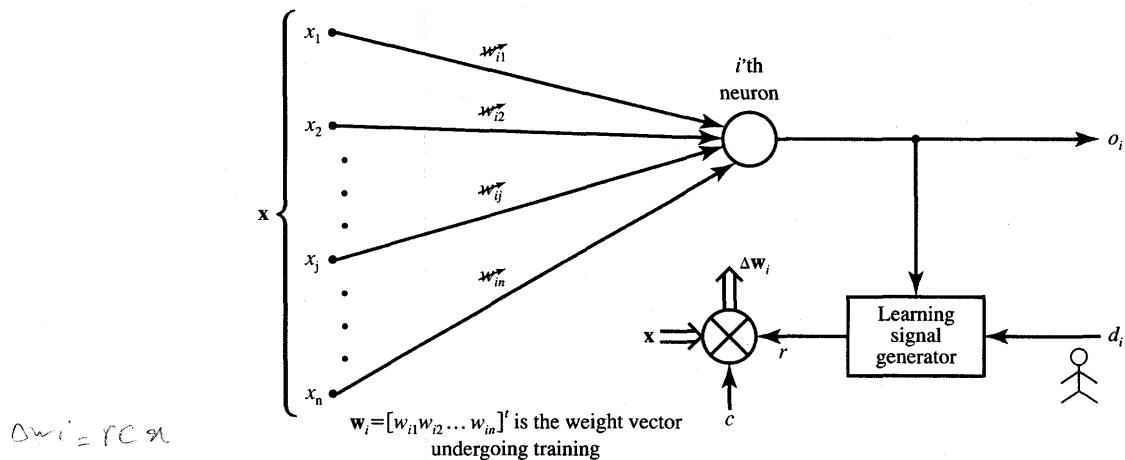


Figure 2.21 Illustration for weight learning rules (d_i provided only for supervised learning mode).

Discrete-time, or stepwise, learning is reviewed below. Weights are assumed to have been suitably initialized before each learning experiment started.

Hebbian Learning Rule

For the Hebbian learning rule the learning signal is equal simply to the neuron's output (Hebb 1949). We have

$$\cancel{\times} \quad r \triangleq f(\mathbf{w}_i^T \mathbf{x}) \quad (2.31)$$

The increment $\Delta \mathbf{w}_i$ of the weight vector becomes

$$\cancel{\times} \quad \Delta \mathbf{w}_i = cf(\mathbf{w}_i^T \mathbf{x})\mathbf{x} \quad (2.32a)$$

The single weight w_{ij} is adapted using the following increment:

$$\cancel{\times} \quad \Delta w_{ij} = cf(\mathbf{w}_i^T \mathbf{x})x_j \quad (2.32b)$$

This can be written briefly as

$$\Delta w_{ij} = co_i x_j, \quad \text{for } j = 1, 2, \dots, n \quad (2.32c)$$

This learning rule requires the weight initialization at small random values around $\mathbf{w}_i = \mathbf{0}$ prior to learning. The Hebbian learning rule represents a purely feedforward, unsupervised learning. The rule implements the interpretation of the classic statement: "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes place in firing it, some growth process or

metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased." (Hebb 1949.)

The rule states that if the crossproduct of output and input, or correlation term $o_i x_j$ is positive, this results in an increase of weight w_{ij} ; otherwise the weight decreases. It can be seen that the output is strengthened in turn for each input presented. Therefore, frequent input patterns will have most influence at the neuron's weight vector and will eventually produce the largest output.

Since its inception, the Hebbian rule has evolved in a number of directions. In some cases, the Hebbian rule needs to be modified to counteract unconstrained growth of weight values, which takes place when excitations and responses consistently agree in sign. This corresponds to the Hebbian learning rule with saturation of the weights at a certain, preset level. Throughout this text note that other learning rules often reflect the Hebbian rule principle. Below, most of the learning rules are illustrated with simple numerical examples. Note that the subscript of the weight vector is not used in the examples since there is only a single weight vector being adapted there.

EXAMPLE 2.4

This example illustrates Hebbian learning with binary and continuous activation functions of a very simple network. Assume the network shown in Figure 2.22 with the initial weight vector

$$\mathbf{w}^1 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix}$$

needs to be trained using the set of three input vectors as below

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ -2 \\ 1.5 \\ 0 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 1 \\ -0.5 \\ -2 \\ -1.5 \end{bmatrix}, \quad \mathbf{x}_3 = \begin{bmatrix} 0 \\ 1 \\ -1 \\ 1.5 \end{bmatrix}$$

for an arbitrary choice of learning constant $c = 1$. Since the initial weights are of nonzero value, the network has apparently been trained beforehand. Assume first that bipolar binary neurons are used, and thus $f(\text{net}) = \text{sgn}(\text{net})$.

Step 1 Input \mathbf{x}_1 applied to the network results in activation net^1 as below:

$$\text{net}^1 = \mathbf{w}^{1T} \mathbf{x}_1 = [1 \quad -1 \quad 0 \quad 0.5] \begin{bmatrix} 1 \\ -2 \\ 1.5 \\ 0 \end{bmatrix} = 3$$

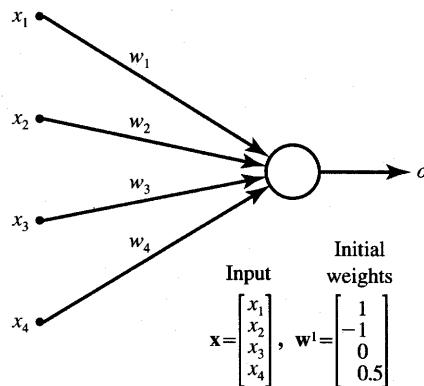


Figure 2.22 Network for training in Examples 2.4 through 2.6.

The updated weights are

$$w^2 = w^1 + \text{sgn}(net^1)x_1 = w^1 + x_1$$

and plugging numerical values we obtain

$$w^2 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} + \begin{bmatrix} 1 \\ -2 \\ 1.5 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ -3 \\ 1.5 \\ 0.5 \end{bmatrix}$$

where the superscript on the right side of the expression denotes the number of the current adjustment step.

Step 2 This learning step is with x_2 as input:

$$net^2 = w^{2t}x_2 = [2 \quad -3 \quad 1.5 \quad 0.5] \begin{bmatrix} 1 \\ -0.5 \\ -2 \\ -1.5 \end{bmatrix} = -0.25$$

The updated weights are

$$w^3 = w^2 + \text{sgn}(net^2)x_2 = w^2 - x_2 = \begin{bmatrix} 1 \\ -2.5 \\ 3.5 \\ 2 \end{bmatrix}$$

Step 3 For input x_3 , we obtain in this step

$$net^3 = w^{3t}x_3 = [1 \quad -2.5 \quad 3.5 \quad 2] \begin{bmatrix} 0 \\ 1 \\ -1 \\ 1.5 \end{bmatrix} = -3$$

The updated weights are

$$\mathbf{w}^4 = \mathbf{w}^3 + \text{sgn}(net^3)\mathbf{x}_3 = \mathbf{w}^3 - \mathbf{x}_3 = \begin{bmatrix} 1 \\ -3.5 \\ 4.5 \\ 0.5 \end{bmatrix}$$

It can be seen that learning with discrete $f(net)$ and $c = 1$ results in adding or subtracting the entire input pattern vectors to and from the weight vector, respectively. In the case of a continuous $f(net)$, the weight incrementing/decrementing vector is scaled down to a fractional value of the input pattern.

Revisiting the Hebbian learning example, with continuous bipolar activation function $f(net)$, using input \mathbf{x}_1 and initial weights \mathbf{w}^1 , we obtain neuron output values and the updated weights for $\lambda = 1$ as summarized in Step 1. The only difference compared with the previous case is that instead of $f(net) = \text{sgn}(net)$, now the neuron's response is computed from (2.3a).

Step 1

$$f(net^1) = 0.905$$

$$\mathbf{w}^2 = \begin{bmatrix} 1.905 \\ -2.81 \\ 1.357 \\ 0.5 \end{bmatrix}$$

Subsequent training steps result in weight vector adjustment as below:

Step 2

$$f(net^2) = -0.077$$

$$\mathbf{w}^3 = \begin{bmatrix} 1.828 \\ -2.772 \\ 1.512 \\ 0.616 \end{bmatrix}$$

Step 3

$$f(net^3) = -0.932$$

$$\mathbf{w}^4 = \begin{bmatrix} 1.828 \\ -3.70 \\ 2.44 \\ -0.783 \end{bmatrix}$$

Comparison of learning using discrete and continuous activation functions indicates that the weight adjustments are tapered for continuous $f(net)$ but are generally in the same direction.

Perceptron Learning Rule

For the perceptron learning rule, the learning signal is the difference between the desired and actual neuron's response (Rosenblatt 1958). Thus, learning is supervised and the learning signal is equal to

$$r \triangleq d_i - o_i \quad (2.33)$$

where $o_i = \text{sgn}(\mathbf{w}_i^T \mathbf{x})$, and d_i is the desired response as shown in Figure 2.23. Weight adjustments in this method, $\Delta \mathbf{w}_i$ and Δw_{ij} , are obtained as follows

$$\Delta \mathbf{w}_i = c [d_i - \text{sgn}(\mathbf{w}_i^T \mathbf{x})] \mathbf{x} \quad (2.34a)$$

$$\Delta w_{ij} = c [d_i - \text{sgn}(\mathbf{w}_i^T \mathbf{x})] x_j, \quad \text{for } j = 1, 2, \dots, n \quad (2.34b)$$

Note that this rule is applicable only for binary neuron response, and the relationships (2.34) express the rule for the bipolar binary case. Under this rule, weights are adjusted if and only if o_i is incorrect. Error as a necessary condition of learning is inherently included in this training rule. Obviously, since the desired response is either 1 or -1 , the weight adjustment (2.34a) reduces to

$$\Delta \mathbf{w}_i = \pm 2c \mathbf{x} \quad (2.35)$$

where a plus sign is applicable when $d_i = 1$, and $\text{sgn}(\mathbf{w}_i^T \mathbf{x}) = -1$, and a minus sign is applicable when $d_i = -1$, and $\text{sgn}(\mathbf{w}_i^T \mathbf{x}) = 1$. The reader should notice that the weight adjustment formula (2.35) cannot be used when $d_i = \text{sgn}(\mathbf{w}_i^T \mathbf{x})$. The weight adjustment is inherently zero when the desired and actual responses agree. As we will see throughout this text, the perceptron learning rule is of central importance for supervised learning of neural networks. The weights are initialized at any values in this method.

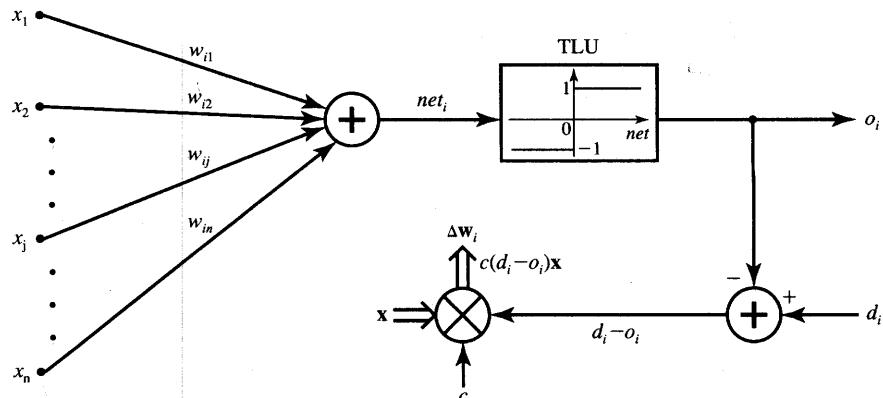


Figure 2.23 Perceptron learning rule.

EXAMPLE 2.5

This example illustrates the perceptron learning rule of the network shown in Figure 2.23. The set of input training vectors is as follows:

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}, \quad \mathbf{x}_3 = \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$$

and the initial weight vector \mathbf{w}^1 is assumed identical as in Example 2.4. The learning constant is assumed to be $c = 0.1$. The teacher's desired responses for $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ are $d_1 = -1$, $d_2 = -1$, and $d_3 = 1$, respectively. The learning according to the perceptron learning rule progresses as follows.

Step 1 Input is \mathbf{x}_1 , desired output is d_1 :

$$net^1 = \mathbf{w}^{1t}\mathbf{x}_1 = [1 \quad -1 \quad 0 \quad 0.5] \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = 2.5$$

Correction in this step is necessary since $d_1 \neq \text{sgn}(2.5)$. We thus obtain updated weight vector

$$\mathbf{w}^2 = \mathbf{w}^1 + 0.1(-1 - 1)\mathbf{x}_1$$

Plugging in numerical values we obtain

$$\mathbf{w}^2 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} - 0.2 \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$$

Step 2 Input is \mathbf{x}_2 , desired output is d_2 . For the present weight vector \mathbf{w}^2 we compute the activation value net^2 as follows:

$$net^2 = \mathbf{w}^{2t}\mathbf{x}_2 = [0 \quad 1.5 \quad -0.5 \quad -1] \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix} = -1.6$$

Correction is not performed in this step since $d_2 = \text{sgn}(-1.6)$

Step 3 Input is \mathbf{x}_3 , desired output is d_3 , present weight vector is \mathbf{w}^3 . Computing net^3 we obtain:

$$net^3 = \mathbf{w}^{3t}\mathbf{x}_3 = [-1 \quad 1 \quad 0.5 \quad -1] \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix} = -2.1$$

Correction is necessary in this step since $d_3 \neq \text{sgn}(-2.1)$. The updated weight values are

$$\mathbf{w}^4 = \mathbf{w}^3 + 0.1(1 + 1)\mathbf{x}_3$$

or

$$\mathbf{w}^4 = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix} + 0.2 \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.6 \\ -0.4 \\ 0.1 \\ 0.5 \end{bmatrix}$$

This terminates the sequence of learning steps unless the training set is recycled. It is not a coincidence that the fourth component of \mathbf{x}_1 , \mathbf{x}_2 , and \mathbf{x}_3 in this example is invariable and equal to -1 . Perceptron learning requires fixing of one component of the input vector, although not necessarily at the -1 level.

The reader may ask what this training has achieved, if anything. At this point the answer can be very preliminary. The real objectives of training and its significance will require more thorough study later in this text. Our preliminary answer for now is that if the same training set is resubmitted, the respective output error should be, in general, smaller. Since the binary perceptron does not provide fine output error information, we may look at its input. The reader may verify it to see that $\text{net}^4 = 0.9$ obtained after three training steps as a response to resubmitted pattern \mathbf{x}_1 is closer to $\text{net} < 0$ than the initial activation value in the first training step, $\text{net}^1 = 2.5$. By observing successive net values for continued training we would see the trend of improving answers. We show in Chapter 3 that, eventually, the network trained in this mode will stop committing any mistakes. ■

Delta Learning Rule

The delta learning rule is only valid for continuous activation functions as defined in (2.3a), (2.4a), and in the supervised training mode. The learning signal for this rule is called *delta* and is defined as follows

$$r \triangleq [d_i - f(\mathbf{w}_i^T \mathbf{x})]f'(\mathbf{w}_i^T \mathbf{x}) \quad (2.36)$$

The term $f'(\mathbf{w}_i^T \mathbf{x})$ is the derivative of the activation function $f(\text{net})$ computed for $\text{net} = \mathbf{w}_i^T \mathbf{x}$. The explanation of the delta learning rule is shown in Figure 2.24. This learning rule can be readily derived from the condition of least squared error between o_i and d_i . Calculating the gradient vector with respect to \mathbf{w}_i of the squared error defined as

$$E \triangleq \frac{1}{2}(d_i - o_i)^2 \quad (2.37a)$$

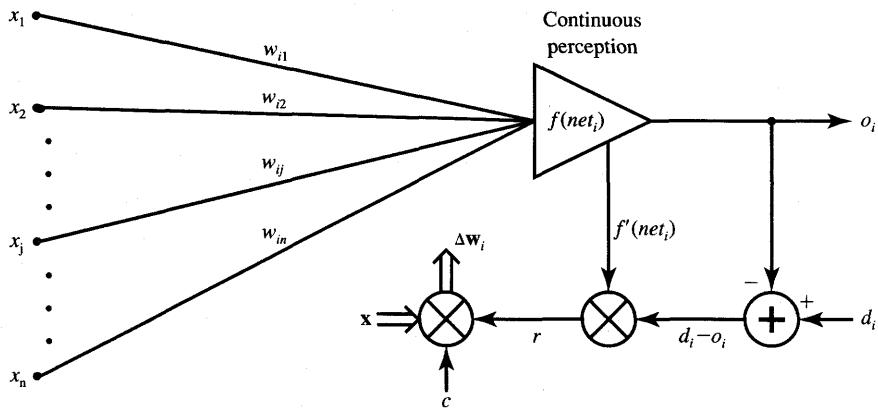


Figure 2.24 Delta learning rule.

which is equivalent to

$$E = \frac{1}{2} [d_i - f(\mathbf{w}_i^T \mathbf{x})]^2 \quad (2.37b)$$

we obtain the error gradient vector value

$$\nabla E = -(d_i - o_i) f'(\mathbf{w}_i^T \mathbf{x}) \mathbf{x} \quad (2.38a)$$

The components of the gradient vector are

$$\frac{\partial E}{\partial w_{ij}} = -(d_i - o_i) f'(\mathbf{w}_i^T \mathbf{x}) x_j, \quad \text{for } j = 1, 2, \dots, n \quad (2.38b)$$

Since the minimization of the error requires the weight changes to be in the negative gradient direction, we take

$$\Delta \mathbf{w}_i = -\eta \nabla E \quad (2.39)$$

where η is a positive constant. We then obtain from Eqs. (2.38a) and (2.39)

$$\Delta \mathbf{w}_i = \eta (d_i - o_i) f'(\mathbf{w}_i^T \mathbf{x}) \mathbf{x} \quad (2.40a)$$

or, for the single weight the adjustment becomes

$$\Delta w_{ij} = \eta (d_i - o_i) f'(\mathbf{w}_i^T \mathbf{x}) x_j, \quad \text{for } j = 1, 2, \dots, n \quad (2.40b)$$

Note that the weight adjustment as in (2.40) is computed based on minimization of the squared error. Considering the use of the general learning rule (2.28) and plugging in the learning signal as defined in (2.36), the weight adjustment becomes

$$\Delta \mathbf{w}_i = c (d_i - o_i) f'(\mathbf{w}_i^T \mathbf{x}) \mathbf{x} \quad (2.41)$$

Therefore, it can be seen that (2.41) is identical to (2.40), since c and η have

been assumed to be arbitrary constants. The weights are initialized at any values for this method of training.

The delta rule was introduced only recently for neural network training (McClelland and Rumelhart 1986). This rule parallels the discrete perceptron training rule. It also can be called the continuous perceptron training rule. The delta learning rule can be generalized for multilayer networks. This will be discussed in more detail in Chapter 3 and extended in Chapter 4.

EXAMPLE 2.6

This example discusses the delta learning rule as applied to the network shown in Figure 2.24. Training input vectors, desired responses, and initial weights are identical to those in Example 2.5. The delta learning requires that the value $f'(net)$ be computed in each step. For this purpose, we can use the following result derived in Equations (3.48) through (3.51):

$$f'(net) = \frac{1}{2}(1 - o^2)$$

valid for the bipolar continuous activation function. The result is useful since it expresses the slope of the activation function through the neuron's output signal. Using (2.41) for the arbitrarily chosen learning constant $c = 0.1$, and $\lambda = 1$ for the bipolar continuous activation function $f(net)$ as in (2.3a), the delta rule training can be summarized as follows.

Step 1 Input is vector x_1 , initial weight vector is w^1 :

$$\begin{aligned} net^1 &= w^{1t}x_1 = 2.5 \\ o^1 &= f(net^1) = 0.848 \\ f'(net^1) &= \frac{1}{2}[1 - (o^1)^2] = 0.140 \\ w^2 &= c(d_1 - o^1)f'(net^1)x_1 + w^1 \\ &= [0.974 \quad -0.948 \quad 0 \quad 0.526]^t \end{aligned}$$

Step 2 Input is vector x_2 , weight vector is w^2 :

$$\begin{aligned} net^2 &= w^{2t}x_2 = -1.948 \\ o^2 &= f(net^2) = -0.75 \\ f'(net^2) &= \frac{1}{2}[1 - (o^2)^2] = 0.218 \\ w^3 &= c(d_2 - o^2)f'(net^2)x_2 + w^2 \\ &= [0.974 \quad -0.956 \quad 0.002 \quad 0.531]^t \end{aligned}$$

Step 3 Input is x_3 , weight vector is \mathbf{w}^3 :

$$\begin{aligned} net^3 &= \mathbf{w}^{3t} \mathbf{x}_3 = -2.46 \\ o^3 &= f(net^3) = -0.842 \\ f'(net^3) &= \frac{1}{2}[1 - (o^3)^2] = 0.145 \\ \mathbf{w}^4 &= c(d_3 - o^3)f'(net^3)\mathbf{x}_3 + \mathbf{w}^3 \\ &= [0.947 \quad -0.929 \quad 0.016 \quad 0.505]^t \end{aligned}$$

Obviously, since the desired values are ± 1 in this example, the corrections will be performed in each step, since $d_i - f(net_i) \neq 0$ throughout the entire training. This method usually requires small c values, since it is based on moving the weight vector in the weight space in the negative error gradient direction.

Widrow-Hoff Learning Rule

The Widrow-Hoff learning rule (Widrow 1962) is applicable for the supervised training of neural networks. It is independent of the activation function of neurons used since it minimizes the squared error between the desired output value d_i and the neuron's activation value $net_i = \mathbf{w}_i^t \mathbf{x}$. The learning signal for this rule is defined as follows

$$r \triangleq d_i - \mathbf{w}_i^t \mathbf{x} \quad (2.42)$$

The weight vector increment under this learning rule is

$$\Delta \mathbf{w}_i = c(d_i - \mathbf{w}_i^t \mathbf{x})\mathbf{x} \quad (2.43a)$$

or, for the single weight the adjustment is

$$\Delta w_{ij} = c(d_i - \mathbf{w}_i^t \mathbf{x})x_j, \quad \text{for } j = 1, 2, \dots, n \quad (2.43b)$$

This rule can be considered a special case of the delta learning rule. Indeed, assuming in (2.36) that $f(\mathbf{w}_i^t \mathbf{x}) = \mathbf{w}_i^t \mathbf{x}$, or the activation function is simply the identity function $f(net) = net$, we obtain $f'(net) = 1$, and (2.36) becomes identical to (2.42). This rule is sometimes called the *LMS (least mean square) learning rule*. Weights are initialized at any values in this method.

Correlation Learning Rule

By substituting $r = d_i$ into the general learning rule (2.28) we obtain the correlation learning rule. The adjustments for the weight vector and the single

weights, respectively, are

$$\Delta w_i = cd_i \mathbf{x} \quad (2.44a)$$

$$\Delta w_{ij} = cd_i x_j, \quad \text{for } j = 1, 2, \dots, n \quad (2.44b)$$

This simple rule states that if d_i is the desired response due to x_j , the corresponding weight increase is proportional to their product. The rule typically applies to recording data in memory networks with binary response neurons. It can be interpreted as a special case of the Hebbian rule with a binary activation function and for $o_i = d_i$. However, Hebbian learning is performed in an unsupervised environment, while correlation learning is supervised. While keeping this basic difference in mind, we can observe that Hebbian rule weight adjustment (2.32a) and correlation rule weight adjustment (2.44a) become identical. Similar to Hebbian learning, this learning rule also requires the weight initialization $\mathbf{w} = \mathbf{0}$.

Winner-Take-All Learning Rule

This learning rule differs substantially from any of the rules discussed so far in this section. It can only be demonstrated and explained for an ensemble of neurons, preferably arranged in a layer of p units. This rule is an example of competitive learning, and it is used for unsupervised network training. Typically, winner-take-all learning is used for learning statistical properties of inputs (Hecht-Nielsen 1987). The learning is based on the premise that one of the neurons in the layer, say the m 'th, has the maximum response due to input \mathbf{x} , as shown in Figure 2.25. This neuron is declared the *winner*. As a result of this winning event, the weight vector \mathbf{w}_m

$$\mathbf{w}_m = [w_{m1} \quad w_{m2} \quad \cdots \quad w_{mn}]^t \quad (2.45)$$

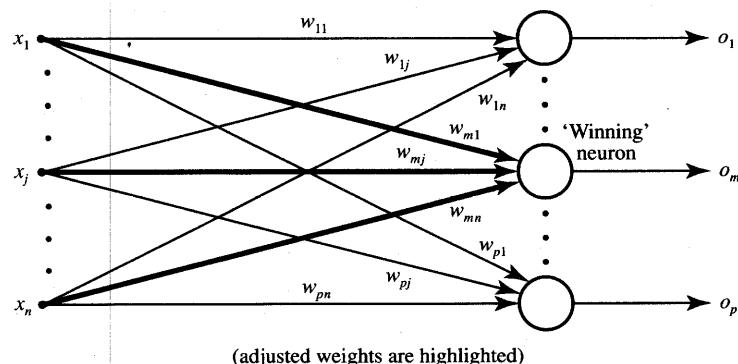


Figure 2.25 Competitive unsupervised "winner-take-all" learning rule.

containing weights highlighted in the figure is the only one adjusted in the given unsupervised learning step. Its increment is computed as follows

$$\Delta \mathbf{w}_m = \alpha(\mathbf{x} - \mathbf{w}_m) \quad (2.46a)$$

or, the individual weight adjustment becomes

$$\Delta w_{mj} = \alpha(x_j - w_{mj}), \quad \text{for } j = 1, 2, \dots, n \quad (2.46b)$$

where $\alpha > 0$ is a small learning constant, typically decreasing as learning progresses. The winner selection is based on the following criterion of maximum activation among all p neurons participating in a competition:

$$\mathbf{w}_m^t \mathbf{x} = \max_{i=1,2,\dots,p} (\mathbf{w}_i^t \mathbf{x}) \quad (2.47)$$

As shown later in Chapter 7, this criterion corresponds to finding the weight vector that is closest to the input \mathbf{x} . The rule (2.46) then reduces to incrementing \mathbf{w}_m by a fraction of $\mathbf{x} - \mathbf{w}_m$. Note that only the winning neuron fan-in weight vector is adjusted. After the adjustment, its fan-in weights tend to better estimate the input pattern in question. In this method, the winning neighborhood is sometimes extended beyond the single neuron winner so that it includes the neighboring neurons. Weights are typically initialized at random values and their lengths are normalized during learning in this method. More detailed justification and application of this rule is provided in Chapter 7.

Outstar Learning Rule

Outstar learning rule is another learning rule that is best explained when neurons are arranged in a layer. This rule is designed to produce a desired response \mathbf{d} of the layer of p neurons shown in Figure 2.26 (Grossberg 1974, 1982). The rule is used to provide learning of repetitive and characteristic properties of input/output relationships. This rule is concerned with supervised learning; however, it is supposed to allow the network to extract statistical properties of the input and output signals. The weight adjustments in this rule are computed as follows

$$\Delta \mathbf{w}_j = \beta(\mathbf{d} - \mathbf{w}_j) \quad (2.48a)$$

or, the individual weight adjustments are

$$\Delta w_{mj} = \beta(d_m - w_{mj}), \quad \text{for } m = 1, 2, \dots, p \quad (2.48b)$$

Note that in contrast to any learning rule discussed so far, the adjusted weights are fanning out of the j 'th node in this learning method and the weight vector in (2.48a) is defined accordingly as

$$\mathbf{w}_j \triangleq \begin{bmatrix} w_{1j} & w_{2j} & \cdots & w_{pj} \end{bmatrix}^t$$

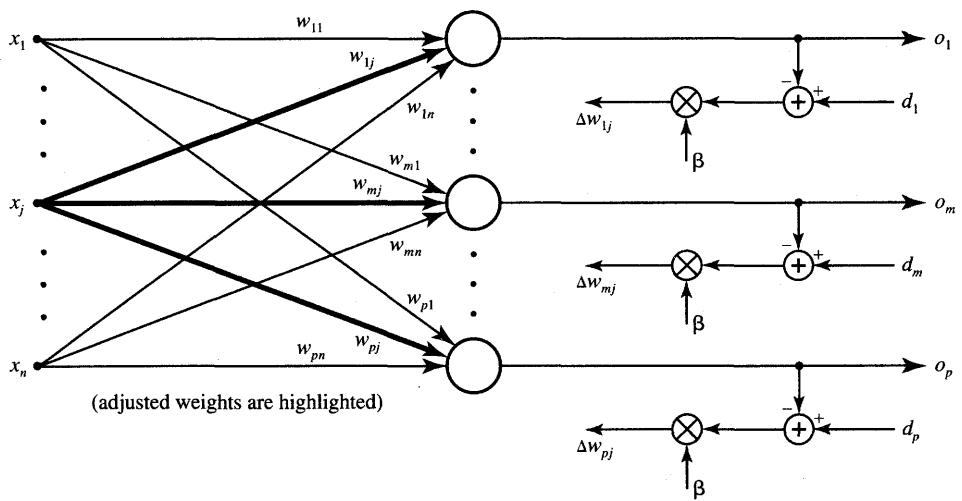


Figure 2.26 Outstar learning rule.

Arbitrarily selectable β is a small positive learning constant decreasing during training. The rule typically ensures that the output pattern becomes similar to the undistorted desired output after repetitively applying (2.48) on distorted output versions. More detailed discussion and application of the outstar learning rule is presented in Chapter 7.

Summary of Learning Rules

Table 2.1 provides the summary of learning rules and of their properties. Seven learning rules are tabulated and compared in terms of the single weight adjustment formulas, supervised versus unsupervised learning mode, weight initialization, and required neuron activation functions. Also, reference is made in the table regarding the conditions of learning. Most learning rules allow for learning of a single, isolated neuron's weight with exception of winner-take-all and outstar rules. Both of these rules require a layer of neurons in order for the weight adaptation to proceed.

So far we have assumed that desired output values are precisely known for each input pattern when a network is trained in the supervised mode. But in some situations fewer details are available. When only partial information is available about the correctness of the network's response, we have so-called reinforcement learning. In the extreme case, only "right" or "wrong" information is available to provide instruction to the network. The amount of feedback received from the environment is very limited in such cases since the teacher's signal is

TABLE 2.1

Summary of learning rules and their properties.

Learning rule	Single weight adjustment Δw_{ij}	Initial weights	Learning	Neuron characteristics	Neuron / Layer
Hebbian	$c o_i x_j$ $j = 1, 2, \dots, n$	0	U	Any	Neuron
Perceptron	$c [d_i - \text{sgn}(\mathbf{w}_i^T \mathbf{x})] x_j$ $j = 1, 2, \dots, n$	Any	S	Binary bipolar, or Binary unipolar*	Neuron
Delta	$c(d_i - o_i)f'(net_i)x_j$ $j = 1, 2, \dots, n$	Any	S	Continuous	Neuron
Widrow-Hoff	$c(d_i - \mathbf{w}_i^T \mathbf{x})x_j$ $j = 1, 2, \dots, n$	Any	S	Any	Neuron
Correlation	$c d_i x_j$ $j = 1, 2, \dots, n$	0	S	Any	Neuron
Winner-take-all	$\Delta w_{mj} = \alpha(x_j - w_{mj})$ $m\text{-winning neuron number}$ $j = 1, 2, \dots, n$	Random Normalized	U	Continuous	Layer of p neurons
Outstar	$\beta(d_i - w_{ij})$ $i = 1, 2, \dots, p$	0	S	Continuous	Layer of p neurons

 c, α, β are positive learning constants

S—supervised learning, U—unsupervised learning

*— Δw_{ij} not shown

rather evaluative than instructive. The reinforcement learning is sometimes called “learning with a critic” as opposed to learning with a teacher (Hertz, Krogh, and Palmer 1991).

The coverage of the seven most important learning rules in this section is by no means exhaustive. Only those rules that are used later in the text have been covered. However, a serious reader may at this point raise a question about validity, convergence, or practical importance of the rules. Fortunately, each of

the learning rules has its meaning and its own mathematical substantiation and applicability. Indeed, neural networks can be trained using these rules to perform useful tasks such as classification, recognition, or association. The outcome of the training is as usual for any training: Results are for the most part successful, but failures are also sometimes possible; however, they are the exception, rather than the rule. The exposition in later chapters provides more detailed answers to questions about why neural networks are capable of learning. We will also study how the network learning assists in solving problems that otherwise would have been difficult to solve.

2.6

OVERVIEW OF NEURAL NETWORKS

The review of introductory concepts of artificial neural systems in this chapter considers the main aspects of neural processing. Performance of neural networks during recall and their basic learning techniques are studied. We also address some elementary information processing tasks that neural networks can solve. The review would not be complete, however, without an overview and taxonomy of artificial neural systems architectures.

More than a dozen specific architectures are covered in this text. Let us attempt to group them into classes. There seem to be numerous ways of classifying artificial neural systems for the purposes of study, analysis, understanding, and utilization. One such way of characterizing neural networks makes use of the fact that the recall of information can be effected in two ways.

The recall can be performed in the feedforward mode, or from input toward output, only. Such networks are called feedforward and have no memory. Recall in such networks is instantaneous, thus the past time conditions are irrelevant for their computation. Feedforward network's behavior does not depend on what happened in the past but rather what happens now. Specifically, the network responds only to its present input.

The second group of networks performs recall computation with feedback operational. These networks can be considered as dynamical systems, and a certain time interval is needed for their recall to be completed. Feedback networks are also called recurrent. They interact with their input through the output. As we have seen, recurrent networks can operate either in a discrete- or continuous-time mode.

Another meaningful basis for classification is to differentiate neural networks by their learning mode. As discussed in Sections 2.4 and 2.5, supervised and unsupervised learning are the main forms of learning. In addition, there is a large group of networks that does not undergo supervised or unsupervised training, because the complete design information is available *a priori*. Many artificial neural memories fall into this category. They are designed by recording or encoding desired equilibria. We say that such networks are trained in a batch

mode. The continuous or discrete data that need to be associated, or hetero-associated, are used for batch mode learning. The weights of such networks remain fixed following the recording. The reader should be aware that because there are few standards for neural network terminology, some authors consider memories trained by recording, or in batch mode, as networks trained with supervision.

Table 2.2 summarizes the taxonomy of the most important artificial neural system architectures. Thirteen different architectures are listed in the first column

TABLE 2.2

Classification of the most important artificial neural networks according to their learning and recall modes.

Network Architecture	Learning Mode S, U, R	Recall Mode FF, REC	Recall Time Domain CT, DT
Single-layer Network of Discrete and Continuous Perceptrons (Figure 1.1)	S	FF	—
Multilayer Network of Discrete and Continuous Perceptrons (EEG spike detectors, ALVINN, Figures 1.3 and 2.9)	S	FF	—
Gradient-type Network (Figures 1.8 and 2.15)	R	REC	CT
Linear Associative Memory	R	FF	—
Autoassociative Memory [Figures 1.5, 1.7, 2.12, and 2.16(a)]	R	REC	DT or CT
Bidirectional Associative Memory [Figure 2.16(b)] (also Multidirectional Associative Memory)	R	REC	DT or CT
Temporal Associative Memory	R	REC	DT or CT
Hamming Network	R	FF	—
MAXNET	R (fixed)	REC	DT or CT
Clustering Network (Figure 2.25)	U	FF	—
Counterpropagation Network (Figure 2.25 and 2.26)	U + S	FF	—
Self-Organizing Neural Array (Figure 1.11)	U	FF	—
Adaptive Resonance Theory 1 Network	U	REC	DT or CT

Learning Mode

S—Supervised

U—Unsupervised

R—Recording (Batch)

Recall Mode

FF—Feedforward

REC—Recurrent

Recall Time Domain (only for recurrent networks)

CT—Continuous-time

DT—Discrete-time

of the table. Only basic network configurations covered in this text are included, therefore the table is by no means exhaustive. To link the discussed architectures with the examples of networks introduced in Chapters 1 and 2, a number of references are provided in the first column.

As stressed before, the learning and recall modes are the most important characteristics of the networks we study. Supervised/unsupervised/recording (batch) learning modes are specified in the table for each network of interest. Feedforward, or recurrent recall mode, is also highlighted. Note that feedforward networks provide instantaneous recall. Additionally, continuous- and discrete-time networks have been marked within the group of recurrent networks.

No judgment or opinion about comparative worth of networks is made here or at any other place in this book. Instead, the focus is on the mathematical and algorithmic principles of each network.

Neural networks can also be characterized in terms of their input and output values as discrete (binary) or continuous. As mentioned earlier in this chapter, diverse neuron models are used in various networks, sometimes even a combination of them. These aspects of artificial neural networks will be highlighted in the sections where they are discussed.

2.7

CONCLUDING REMARKS

At this point it may be desirable to reiterate and expand the introductory remarks made so far concerning the differences between the conventional and neural network computation. Table 2.3 provides a summary overview of the basic aspects of computation for each of the approaches. Inspection of the entries of the table indicates that neural networks represent collective, non-algorithmic, low-precision, nonlinear computing machines that learn during training from examples and are data-controlled. As such they are very different from programmable conventional computers. The reader is encouraged to look at the details provided in Table 2.3.

To gain better understanding of the presented concepts, let us look for analogies between the theory of learning neurons and the theory of learning of organisms. According to psychologists, there are two basic forms of learning. *Operant conditioning* is a form of learning that reinforces only the responses of an organism. The reinforcement is aimed at making the desired responses more likely and ignoring or punishing those that are not desirable. In contrast to this form of learning, *classical conditioning* is based on training for a stimulus-response sequence.

Perceptron or delta training rules are aimed at influencing responses so that they become more desirable when the same input is applied repetitively. These

TABLE 2.3

Comparison of conventional and neural network computation.

Task or performance aspect	Conventional computation	Neural network computation
Problem solving	Algorithm formulation	Selection of architecture and definition of the set of representative examples
Input data	Numerical form	Numerical, but also perceptual representation allowed
Knowledge acquisition	Programming	Training
Knowledge retrieval	Sequential computation	Recall in the form of collective processing
Computation	High-precision arithmetic	Low-precision, nonlinear mapping
Internal data	Internal representation in control of the algorithm	Internal representation in control of input data
Fixed- or intermediate-data storage	ROM, RAM-high-precision binary memories	Interconnecting weights of typically continuous values

rules will thus correspond to operant conditioning. The Hebbian learning rule couples the input and output and involves the relationship between the present stimulus-response pair, or input-output correlation. Thus, this learning rule corresponds to the classical conditioning organism learning.

This chapter treats introductory concepts and definitions used in artificial neural systems. It contains a discussion of relationships between real neural systems and their artificial counterparts. The chapter provides an overview of the

fundamental terminology. It also introduces a historical model of the artificial neuron developed by McCulloch and Pitts. Subsequently, a number of other artificial neuron models are discussed.

The basic taxonomy of neural networks as feedforward and recurrent recall systems is provided along with recall analysis examples. The most important features of the learning modes are formulated and discussed in this chapter. The phenomenon of network learning is explained based on the unifying general learning rule and on the so-called learning signal concept. This approach clarifies how neural networks learn. Learning rules for adapting neuron's weights as introduced in this chapter provide the reference framework needed for further study. In subsequent chapters our focus is on what the learning can accomplish, how it happens, and how to use it in the most efficient way.

PROBLEMS

Please note that problems highlighted with an asterisk (*) are typically computationally intensive and the use of programs is advisable to solve them.

- P2.1* The logic networks shown in Figure P2.1 use the McCulloch-Pitts model neuron from Figure 2.3. Find the truth tables and the logic functions that are implemented by networks (a), (b), (c), and (d).
- P2.2* Use McCulloch-Pitts neurons to design logic networks that implement the following functions. Use a single neuron in (a), and two neurons in cascade for (b) and (c). (A prime denotes a logic complement.)

$$(a) o^{k+1} = x_1^k x_2^k x_3'^k$$

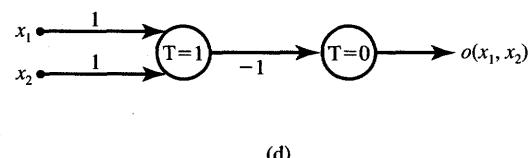
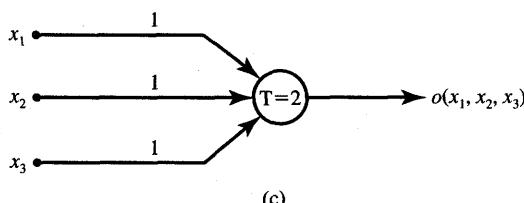
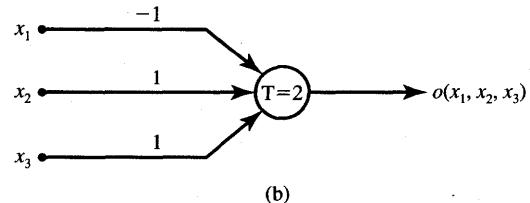
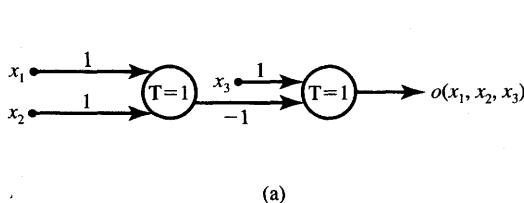


Figure P2.1 Neural networks for analysis of logic functions (neurons from Figure 2.3).

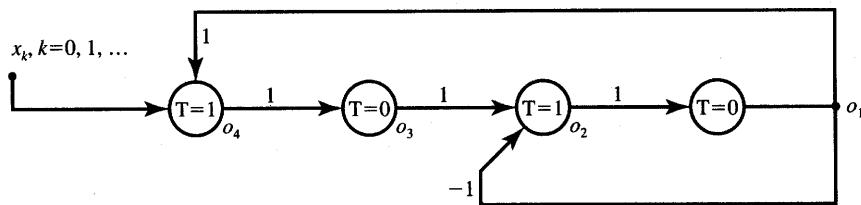


Figure P2.3 Randomizing shift register using neurons from Figure 2.3.

$$(b) \quad o^{k+2} = x_1^k x_2^k x_3^k$$

$$(c) \quad o^{k+2} = x_1^k x_2^k$$

- P2.3** The sequential circuit shown in Figure P2.3 contains a shift register, which processes the incoming impulse sequence and memorizes it using the register cell property of the McCulloch-Pitts neurons. Assume that input data are $x^0 = 1$, $x^1 = 1$, $x^2 = 0$, $x^3 = 1$, $x^4 = x^5 = x^6 = \dots = 0$. Compute the contents of the register $(o_1^8, o_2^8, o_3^8, o_4^8)$ after eight processing steps have elapsed. Assume all cells have initially cleared outputs at $k = 0$.
- P2.4** The feedforward network shown in Figure P2.4 using bipolar binary neurons is mapping the entire plane x_1, x_2 into a binary o value. Find the segment of the x_1, x_2 plane for which $o_4 = 1$, and its complement for which $o_4 = -1$.
- P2.5** Each of the two networks shown in Figure P2.5 implements an identical function on unipolar binary vectors in two-dimensional input space. Analyze the networks to show that they are equivalent. Assume

$$f(\text{net}) = \begin{cases} 0 & \text{net} \leq 0 \\ 1 & \text{net} > 0 \end{cases}$$

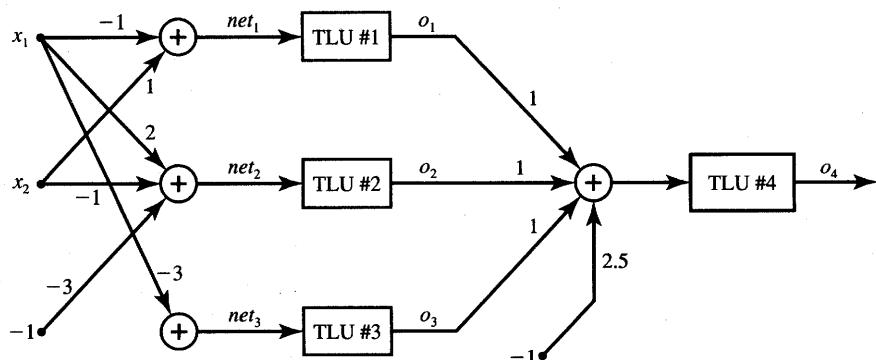


Figure P2.4 Feedforward network for Problem P2.4.

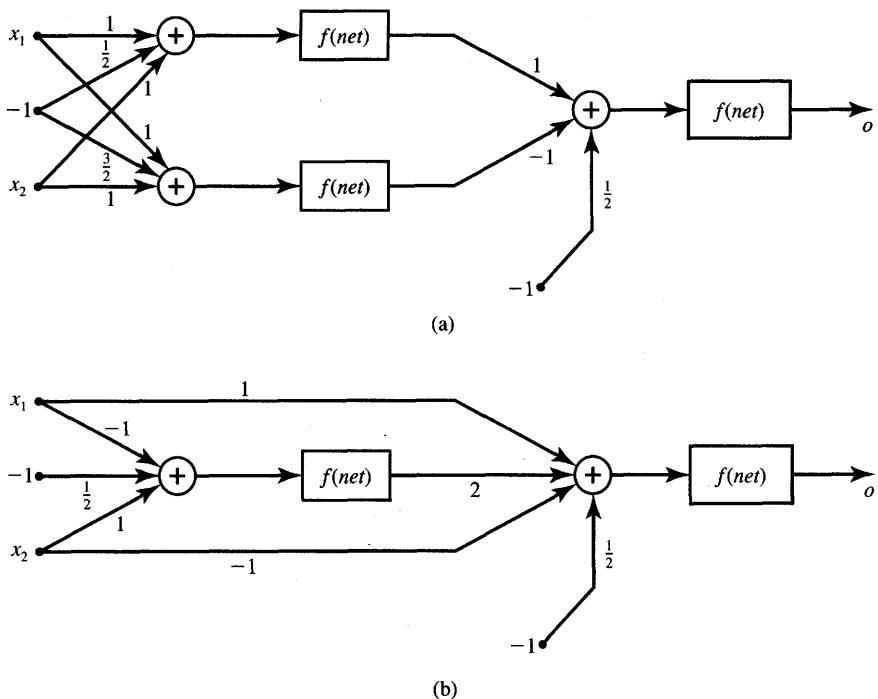


Figure P2.5 Feedforward network for analysis in Problem P2.5.

- P2.6 The network shown in Figure P2.6 is an analog-to-digital converter and can be used for coding a continuous value x into a 4-bit unipolar binary code $(o_3 \ o_2 \ o_1 \ o_0)$. Analyze the network and find each range of x that is converted in each of the binary codes $(0 \ 0 \ 0 \ 0)$, ..., $(1 \ 1 \ 1 \ 1)$. Assume $-1 \leq x \leq 16$ and unipolar binary neurons used as in Problem P2.5.
- P2.7 The feedforward network shown in Figure P2.7 has been designed to code the grey intensity of a pixel expressed on a continuous scale, $0 \leq x \leq 1$ (Ramacher 1989). The output binary code is $(q_3 \ q_2 \ q_1)$. Analyze the network and find each range of x that is converted into the binary codes $(0 \ 0 \ 0)$, ..., $(1 \ 1 \ 1)$. Assume unipolar binary neurons as in Problem P2.5.
- P2.8 The network shown in Figure P2.8 uses neurons with a continuous activation function as in (2.3a) with $\lambda = 1$. The neuron's output has been measured as $o_1 = 0.28$ and $o_2 = -0.73$. Find the input vector $\mathbf{x} = [x_1 \ x_2]^T$ that has been applied to the network. Also find the slope values of the activation function at the activations net_1 and net_2 .

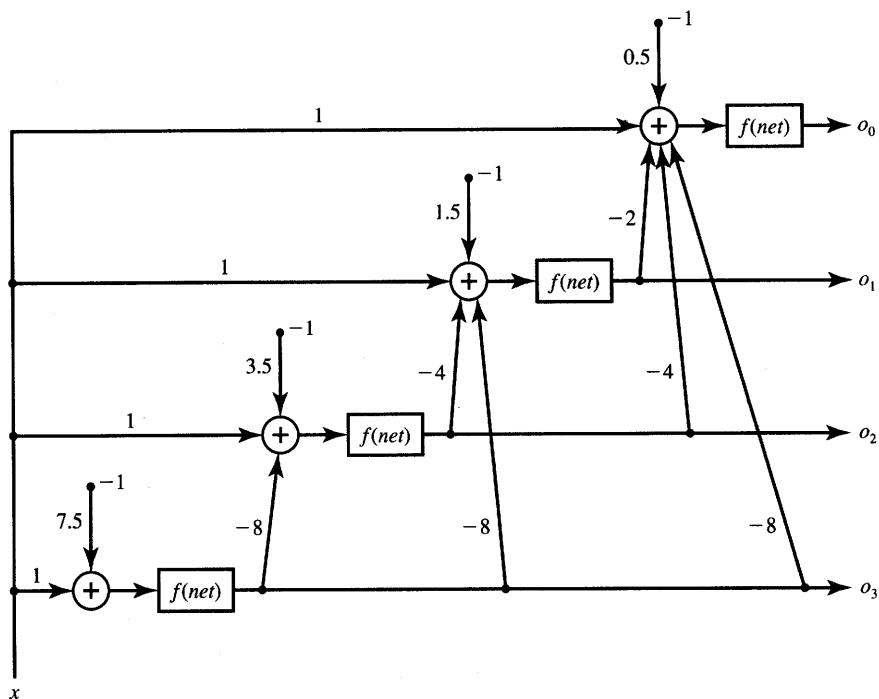


Figure P2.6 Feedforward converter for analysis in Problem P2.6.

- P2.9 The network shown in Figure P2.9 using neurons with $f(\text{net})$ as in (2.3a) has been designed to assign input vectors $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ to cluster 1 or 2. The cluster number is identical to the number of the neuron yielding the larger response. Determine the most likely cluster membership for each of the following three vectors. Assume $\lambda = 2$. The input vectors are

$$\mathbf{x}_1 = \begin{bmatrix} 0.866 \\ 0.5 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} -0.985 \\ -0.174 \end{bmatrix}, \quad \mathbf{x}_3 = \begin{bmatrix} 0.342 \\ -0.94 \end{bmatrix}$$

- P2.10 The two networks shown in Figure P2.10 implement partitioning of the plane x_1, x_2 and yield $o_1 = \pm 1$ (network a, bipolar binary output neuron), and $|o_1| < 1$ (network b, bipolar continuous output neuron). The neurons used in both networks are with bipolar characteristics as in (2.3). Assume for network b that $\lambda = 1$. For both networks:

- (a) Find o_1 as a function of x_1 with $x_2 = +2$.
- (b) Draw the line on x_1, x_2 plane separating positive and negative responses for $|x_1| < 5$ and $|x_2| < 5$.

- P2.11 The network shown in Figure P2.11 uses neurons with continuous bipolar characteristics with $\lambda = 1$. It implements partitioning of plane x_1, x_2 and

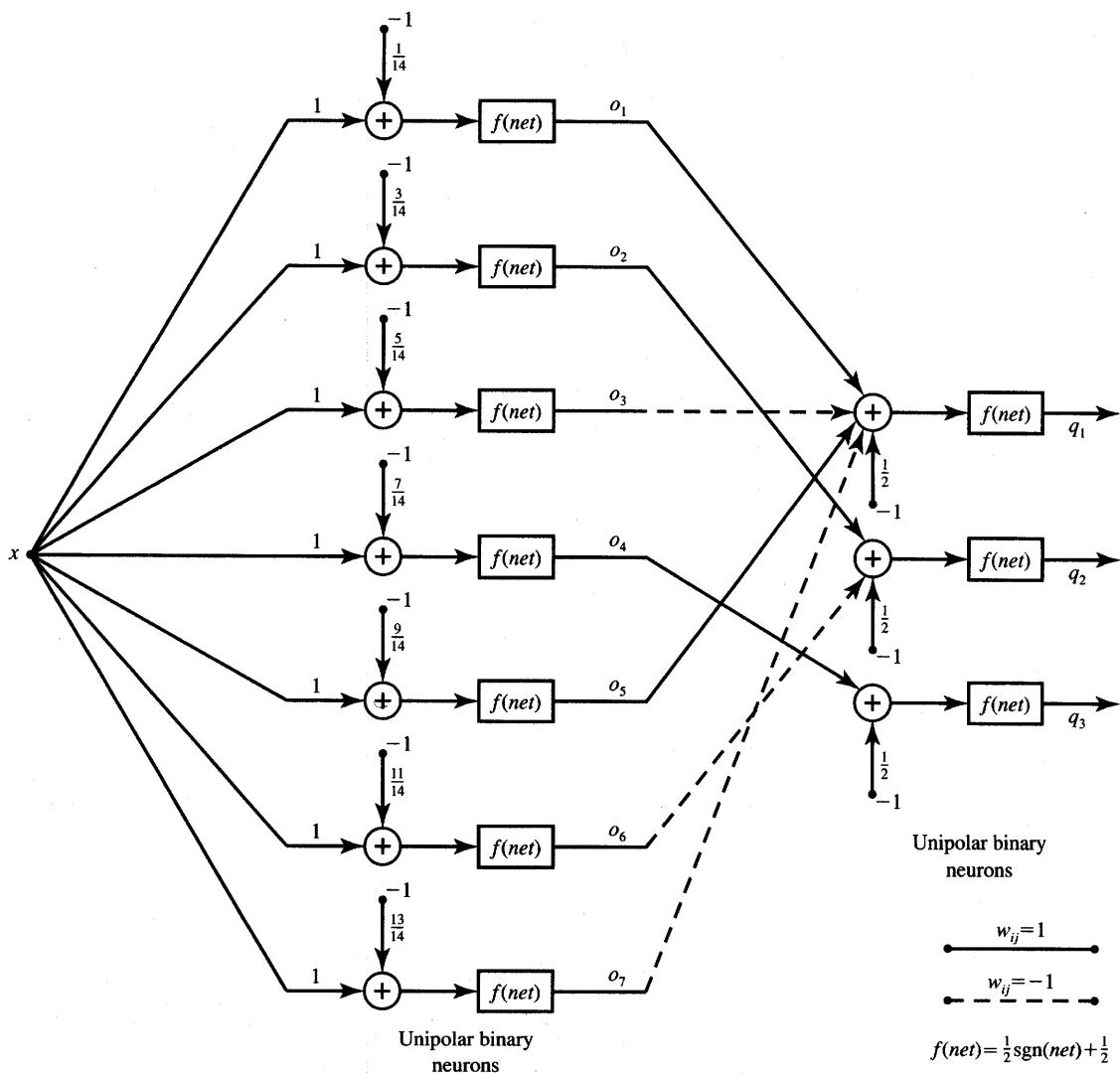


Figure P2.7 Feedforward encoding of an analog value.

maps it into $|o_1| < 1$. Analyze the network and obtain regions on the plane x_1, x_2 with positive and negative responses o_1 for $|x_1| < 5, |x_2| < 5$, and $T_1 = T_2 = 0$. Then simulate the network and tabulate the function $o_1(x_1, x_2)$ in the domain of interest.

- P2.12 The network shown in Figure P2.12 uses neurons with continuous bipolar characteristics with $\lambda = 5$. It implements mapping of plane x_1, x_2 into

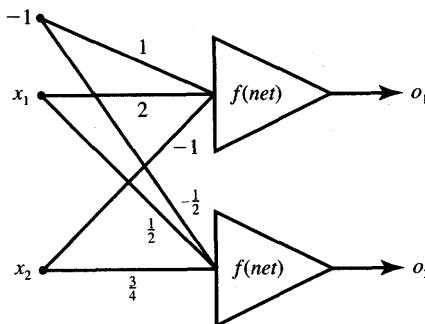


Figure P2.8 Network for Problem P2.8.

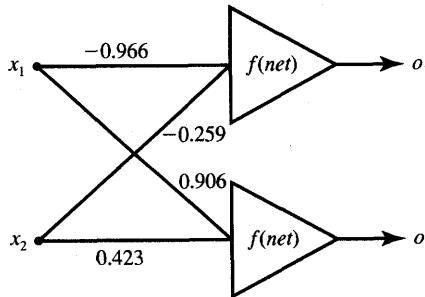


Figure P2.9 Network for cluster identification.

$|o_1| < 1$. Simulate the network and tabulate the function $o_1(x_1, x_2)$ for $|x_1| < 2.5$ and $|x_2| < 2.5$.

P2.13 Assume that the vertices of a three-dimensional bipolar binary cube are used to represent eight states of a recurrent neural network with three bipolar binary neurons. The equilibrium states are $\mathbf{o}_1 = [-1 \quad -1 \quad -1]^t$ and $\mathbf{o}_2 = [1 \quad 1 \quad 1]^t$. Sketch the desirable state transitions between the vertices.

P2.14 A discrete-time four-neuron recurrent network as in Figure 2.10 in the text with bipolar binary neurons has the weight matrix

$$\mathbf{W} = \begin{bmatrix} 0 & -1 & -1 & 1 \\ -1 & 0 & 1 & -1 \\ -1 & 1 & 0 & -1 \\ 1 & -1 & -1 & 0 \end{bmatrix}$$

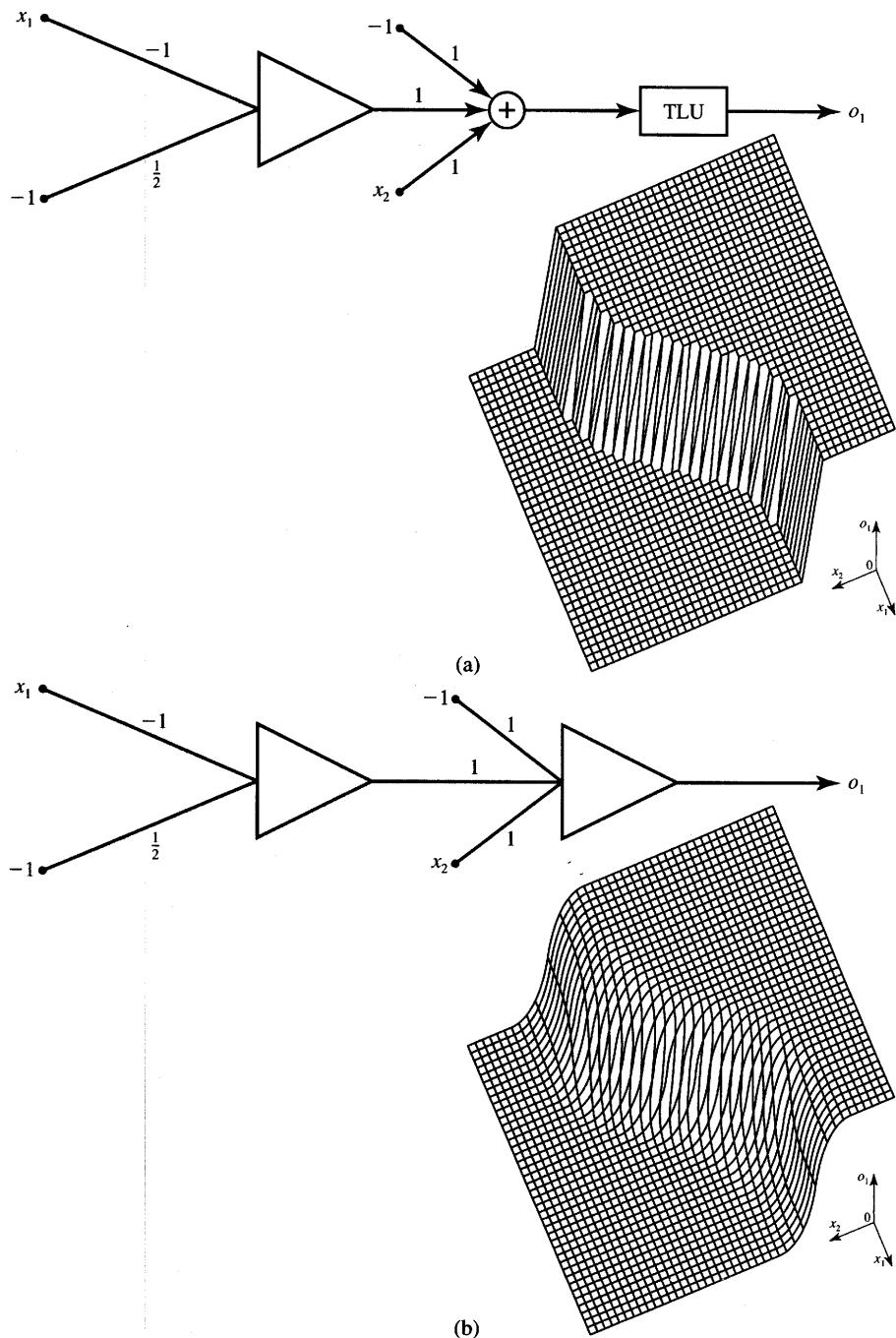


Figure P2.10 Networks for Problem P2.10 and implemented partitioning: (a) network a and (b) network b.

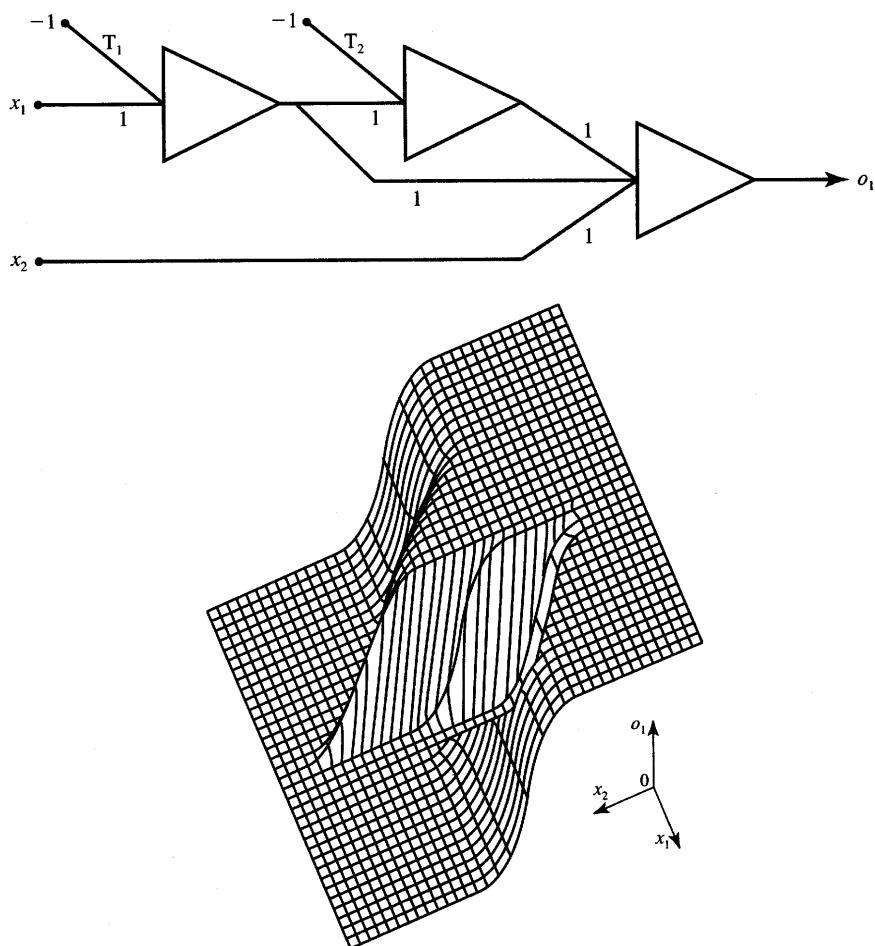


Figure P2.11 Network and implemented partitioning for Problem P2.11. (Partitioning shown for \$T_1 \neq 0\$ and \$T_2 \neq 0\$.)

Find the sequence of state transitions after the network has been initialized at $\mathbf{x}^0 = [-1 \quad -1 \quad -1 \quad 1]^t$. Repeat for three other initializing vectors

$$\mathbf{x}^0 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix}, \quad \mathbf{x}^0 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{x}^0 = \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix}$$

P2.15* For the continuous-time network using the bipolar neurons shown in Figure P2.15:

- (a) Obtain discretized differential equations.

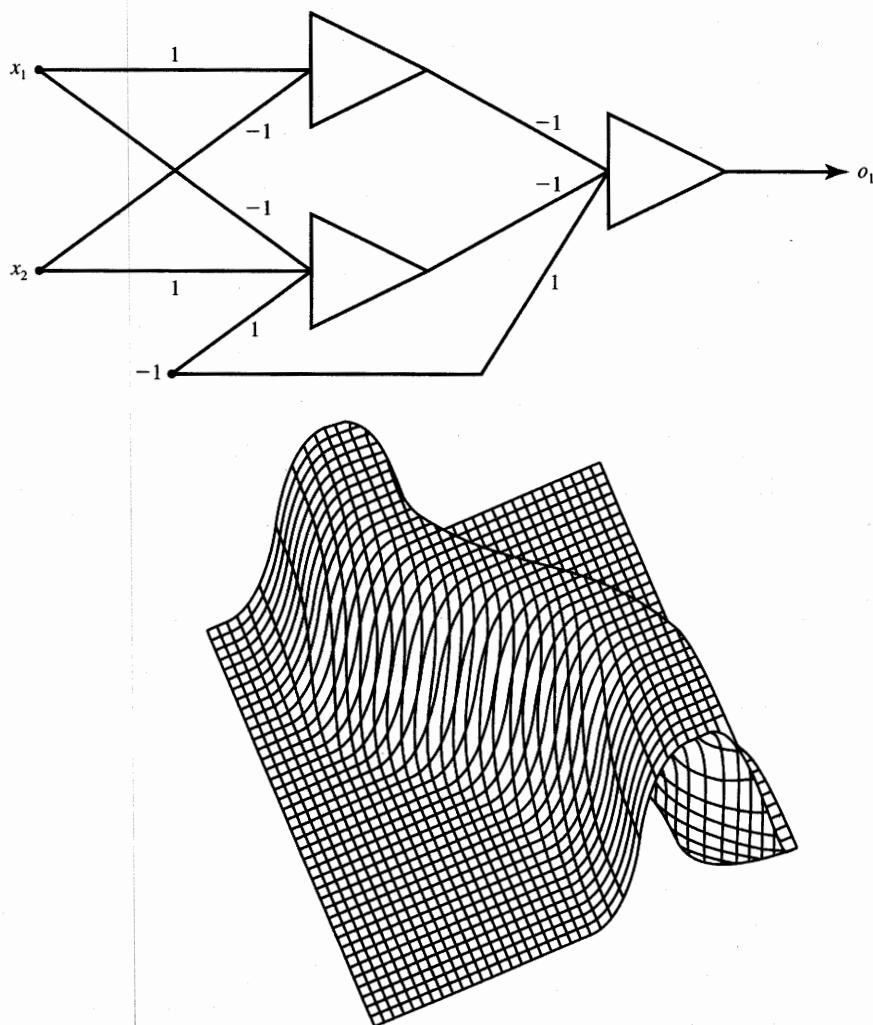


Figure P2.12 Network and implemented partitioning for Problem P2.12.

- (b) Perform sample simulations showing that $\mathbf{o}_1 = [1^- \ 1^- \ 1^-]^t$ and $\mathbf{o}_2 = [-1^+ \ -1^+ \ -1^+]^t$ are equilibrium points stored in the network shown in the figure.

P2.16 Four steps of Hebbian learning of a single-neuron network as in Figure 2.21 have been implemented starting with $\mathbf{w}^1 = [1 \ -1]^t$ for learning constant $c = 1$ using inputs as follows:

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathbf{x}_3 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad \mathbf{x}_4 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

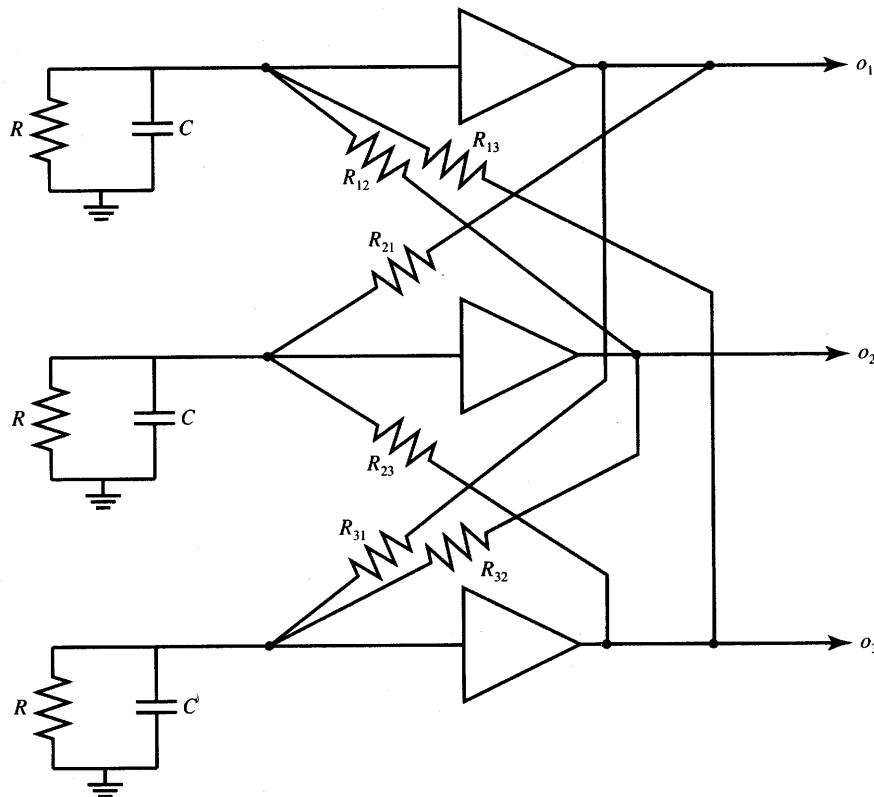


Figure P2.15 Continuous-time feedback network for Problem P2.15. ($R_{ij} = 1$, $C = 1$, $R = 10$, and $\lambda = 2.5$.)

Find final weights for:

- (a) bipolar binary $f(\text{net})$
- (b) bipolar continuous $f(\text{net})$, $\lambda = 1$.

P2.17 Implement the perceptron rule training of the network from Figure 2.23 using $f(\text{net}) = \text{sgn}(\text{net})$, $c = 1$, and the following data specifying the initial weights \mathbf{w}^1 , and the two training pairs

$$\mathbf{w}^1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \left(\mathbf{x}_1 = \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix}, d_1 = -1 \right), \left(\mathbf{x}_2 = \begin{bmatrix} 0 \\ -1 \\ -1 \end{bmatrix}, d_2 = 1 \right)$$

Repeat the training sequence (\mathbf{x}_1, d_1) , (\mathbf{x}_2, d_2) until two correct responses in a row are achieved. List the net^k values obtained during training.

- P2.18 A single-neuron network using $f(\text{net}) = \text{sgn}(\text{net})$ as in Figure 2.23 has been trained using the pairs of \mathbf{x}_i , d_i as shown below:

$$\left(\mathbf{x}_1 = \begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}, \quad d_1 = -1 \right), \quad \left(\mathbf{x}_2 = \begin{bmatrix} 0 \\ -1 \\ 2 \\ -1 \end{bmatrix}, \quad d_2 = 1 \right),$$

$$\left(\mathbf{x}_3 = \begin{bmatrix} -2 \\ 0 \\ -3 \\ -1 \end{bmatrix}, \quad d_3 = -1 \right)$$

The final weights obtained using the perceptron rule are

$$\mathbf{w}^4 = [3 \ 2 \ 6 \ 1]^t$$

Knowing that correction has been performed in each step for $c = 1$, determine the following weights:

- (a) \mathbf{w}^3 , \mathbf{w}^2 , \mathbf{w}^1 by back-tracking the training
- (b) \mathbf{w}^5 , \mathbf{w}^6 , \mathbf{w}^7 obtained for steps 4, 5, and 6 of training by reusing the sequence (\mathbf{x}_1, d_1) , (\mathbf{x}_2, d_2) , (\mathbf{x}_3, d_3) .

- P2.19 Perform two training steps of the network as in Figure 2.24 using the delta learning rule for $\lambda = 1$ and $c = 0.25$. Train the network using the following data pairs

$$\left(\mathbf{x}_1 = \begin{bmatrix} 2 \\ 0 \\ -1 \end{bmatrix}, \quad d_1 = -1 \right), \quad \left(\mathbf{x}_2 = \begin{bmatrix} 1 \\ -2 \\ -1 \end{bmatrix}, \quad d_2 = 1 \right)$$

The initial weights are $\mathbf{w}^1 = [1 \ 0 \ 1]^t$. [Hint: Use $f'(\text{net}) = (1/2)(1 - o^2)$ and $f(\text{net})$ as in (2.3a).]

- P2.20 A recurrent network with three bipolar binary neurons has been trained using the correlation learning rule with a single bipolar binary input vector in a single training step only. The training was implemented starting at $\mathbf{w}^0 = 0$, for $c = 1$. The resulting weight matrix is

$$\mathbf{W} = \begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & 1 \\ -1 & 1 & 1 \end{bmatrix}$$

Find the vectors \mathbf{x} and \mathbf{d} that have been used for training. (There are two different answers.)

- P2.21 Perform two training steps for the network as in Figure 2.24 using the Widrow-Hoff learning rule. Assume the same training data as in Problem P2.19.

P2.22* Write the program for analysis of two-layer feedforward networks. The user-specified parameters should include type of the activation function (2.3a-b), (2.4a-b), λ (if needed), the size of the network, and test input vectors. Verify your program using data from Problems P2.7 and P2.12, and from Example 2.1. Note that each neuron needs to have one fixed input (bias) with an associated weight.

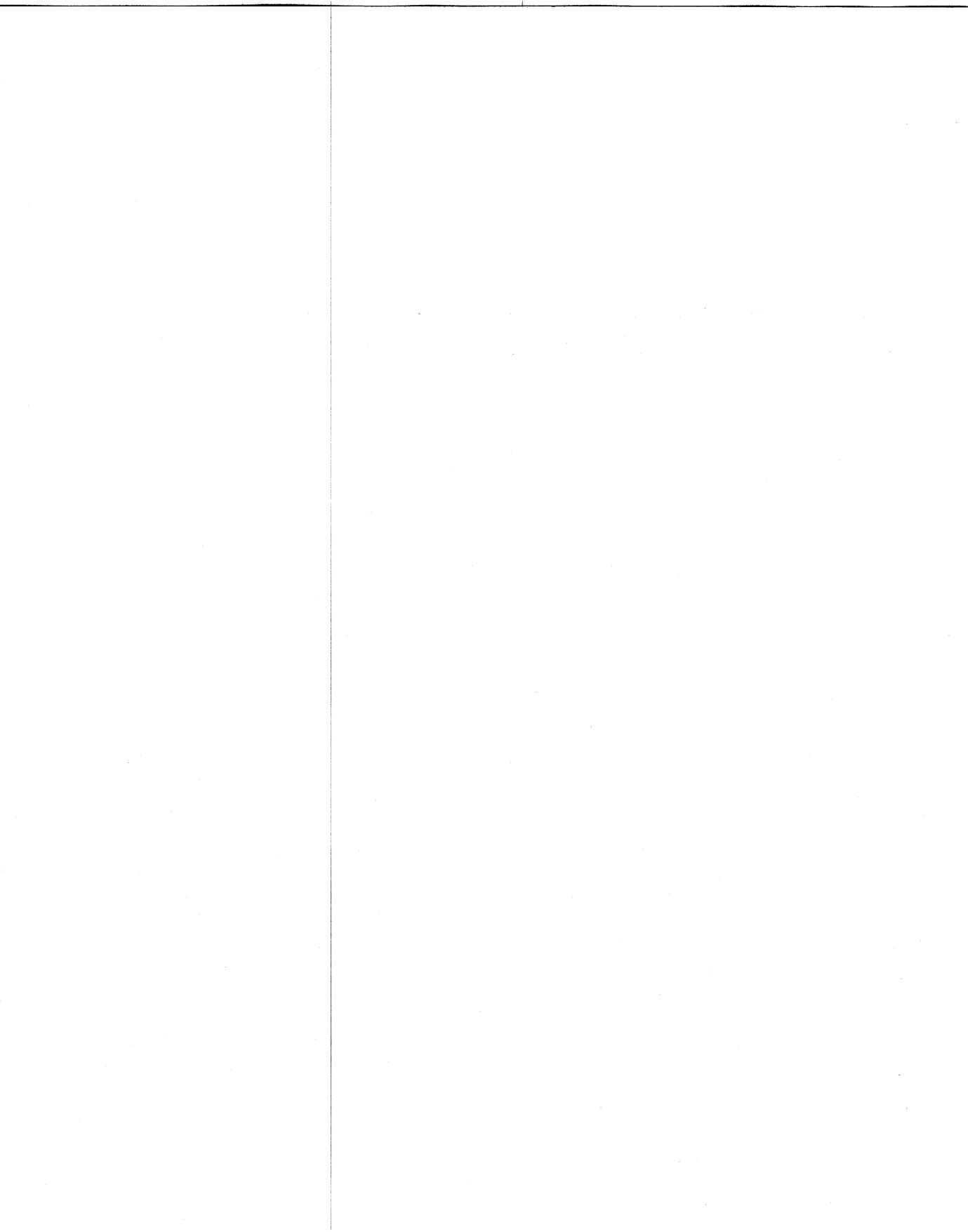
P2.23* Implement the learning algorithms by writing a program that can be used to train weights of a single neuron with up to six inputs. The program should allow the user to specify the learning rule, the type of activation function, λ (if needed), training data, and to perform the specified number of training steps. Rules to include are Hebbian, perceptron, delta, and Widrow-Hoff. Verify your program using data from Problems P2.16, P2.17, P2.19, and P2.21.

REFERENCES

- Amari, S. I. 1990. "Mathematical Foundations of Neurocomputing," *IEEE Proc.* 78(9): 1443–1463.
- Arbib, M. A. 1987. *Brains, Machines and Mathematics*, 2nd ed. New York: Springer Verlag.
- Bemasconi, J. 1988. "Analysis and Comparison of Different Learning Algorithms for Pattern Association Problems," in *Neural Information Processing Systems*, ed. D. Anderson. New York: American Institute of Physics.
- Carpenter, G. A. 1989. "Neural Network Models for Pattern Recognition and Associative Memory," *Neural Networks* 2: 243–257.
- Dayhoff, J. 1990. *Neural Network Architectures—An Introduction*. New York: Van Nostrand Reinhold.
- Durbin, R. 1989. "On the Correspondence Between Network Models and the Nervous System," in *The Computing Neuron*, ed. R. Durbin, C. Miall, G. Mitchison, Reading, Mass.: Addison-Wesley Publishing Co.
- Feldman, J. A., M. A. Fandy, and N. Goddard. 1988. "Computing with Structured Neural Networks," *IEEE Computer* (March): 91–103.
- Grossberg, S. 1977. *Classical and Instrumental Learning by Neural Networks in Progress in Theoretical Biology*, vol. 3. New York: Academic Press, 51–141.
- Grossberg, S. 1982. *Studies of Mind and Brain: Neural Principles of Learning Perception, Development, Cognition, and Motor Control*. Boston: Reidell Press.

- Hebb, D. O. 1949. *The Organization of Behavior, a Neuropsychological Theory*. New York: John Wiley.
- Hecht-Nielsen, R. 1987. "Counterpropagation Networks," *Appl. Opt.* 26(23): 4979-4984.
- Hecht-Nielsen, R. 1990. *Neurocomputing*. Reading, Mass.: Addison-Wesley Publishing Co.
- Hertz, J., A. Krogh, and R. G. Palmer. 1991. *Introduction to the Theory of Neural Computation*. Redwood City, Calif.: Addison-Wesley Publishing Co.
- Hopfield, J. J., and D. W. Tank. 1986. "Computing with Neural Circuits: A Model," *Science* 233: 625-633.
- Lerner, A. Ya. 1972. *Fundamentals of Cybernetics*. London: Chapman and Hall Ltd.
- Lippmann, R. P. 1987. "An Introduction to Computing with Neural Nets," *IEEE Magazine on Acoustics, Signal and Speech Processing* (April): 4-22.
- McClelland, J. L., D. E. Rumelhart, and the PDP Research Group. 1986. *Parallel Distributed Processing*. Cambridge: The MIT Press.
- McCulloch, W. S., and W. H. Pitts. 1943. "A Logical Calculus of the Ideas Imminent in Nervous Activity," *Bull Math. Biophys.* 5: 115-133.
- Mitchison, G. 1989. "Learning Algorithms and Networks of Neurons," in *The Computing Neuron*, ed. R. Durbin, C. Miall, G. Mitchison. Reading, Mass.: Addison-Wesley Publishing Co.
- Poggio, T., and F. Girosi. 1990. "Networks for Approximation and Learning," *Proc. IEEE* 78(9): 1481-1497.
- Ramacher, U., Wesseling, M. 1989. "A Geometrical Approach to Neural Network Design," *Proc. of the Joint Neural Networks Conf.*, Washington, D.C., pp. II, 147-152.
- Reilly, D. L., and L. N. Cooper. 1990. "An Overview of Neural Networks: Early Models to Real World Systems," in *Introduction of Neural and Electronic Networks*. New York: Academic Press.
- Rosenblatt, F. 1958. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psych. Rev.* 65: 386-408.
- Rumelhart, D. E. 1990. "Brain Style Computation: Learning and Generalization," in *Introduction to Neural and Electronic Networks*. New York: Academic Press.
- Szu, H. 1990. "Reconfigurable Neural Nets by Energy Convergence Learning Principles Based on Extended McCulloch-Pitts Neurons and Synapses," *Proc. IEEE Int. Neural Networks Conf.* pp. I-485-I-496.
- Tsyplkin, Ya. Z. 1973. *Foundations of the Theory of Learning Systems*. New York: Academic Press.

- Wasserman, P. D. 1989. *Neural Computing Theory and Practice*. New York: Van Nostrand Reinhold.
- Widrow, B. 1962. "Generalization and Information Storage in Networks of Adaline 'Neurons,'" in *Self-Organizing Systems 1962*, ed. M. C. Jovitz, G. T. Jacobi, G. Goldstein. Washington, D.C.: Spartan Books, 435-461.



SINGLE-LAYER PERCEPTRON CLASSIFIERS

A cat that once sat on a hot stove will never again sit on a hot stove or on a cold one either.

M. TWAIN

- 3.1 Classification Model, Features, and Decision Regions
- 3.2 Discriminant Functions
- 3.3 Linear Machine and Minimum Distance Classification
- 3.4 Nonparametric Training Concept
- 3.5 Training and Classification Using the Discrete Perceptron: Algorithm and Example
- 3.6 Single-Layer Continuous Perceptron Networks for Linearly Separable Classifications
- 3.7 Multicategory Single-Layer Perceptron Networks
- 3.8 Concluding Remarks

In this chapter, the foundations of trainable decision-making networks will be formulated. The principal function of a decision-making system is to yield decisions concerning the class membership of the input pattern with which it is confronted. Conceptually, the problem can be described as a transformation of sets, or functions, from the input space to the output space, which is called the classification space. In general, the transformations of input data into class membership are highly complex and noninvertible.

We will develop the expertise gradually. The linear discriminant functions will be introduced first, and the simple correction rule to perform network training will be devised. The training, or network adaptation, will be presented as a sequence of iterative weight adjustments. Strong emphasis will be put on the geometrical interpretation of the training procedure. This will provide the reader with necessary insight and, hopefully, with a better understanding of mathematical methods for neural network-based classifiers.

Starting with the definitions of basic concepts of classification and with examples of two-class classifiers using the hard-limiting thresholding device, the training rules will then be extended to the case of continuous error function

minimization. This extension will require the replacement of the summing and hard-limiting thresholding device, or a discrete perceptron, with a continuous perceptron. The chapter will conclude with preliminary discussion of multi-neuron multiclass single-layer networks and their training. The architectures discussed in this chapter are limited to single-layer feedforward networks. The chapter also provides an explanation and justification of perceptron and delta training rules introduced formally but without proof in the previous chapter.

The approach presented in this chapter is introductory and applicable mainly to the classification of linearly separable classes of patterns; thus it may be viewed as having somewhat limited practical importance. However, the same approach can easily be generalized and applied later for different network architectures. As the reader will see, such extension will prove to be much more powerful and useful. The discussion of these more versatile and complex architectures and relevant training approaches will be presented in subsequent chapters.

3.1

CLASSIFICATION MODEL, FEATURES, AND DECISION REGIONS

Our discussion of neural network classifiers and classification issues has so far been rather informal. A simplistic two-class classifier (Figure 1.1) was presented in Section 1.1. Also, classification was introduced in Section 2.3 as a form of neural computation and, specifically, as a form of information recall. This notion was illustrated in Figure 2.17(a). We now approach the classification issues in more detail.

One of the most useful tasks that can be performed by networks of interconnected nonlinear elements introduced in the previous chapter is pattern classification. A *pattern* is the quantitative description of an object, event, or phenomenon. The classification may involve spatial and temporal patterns. Examples of spatial patterns are pictures, video images of ships, weather maps, fingerprints, and characters. Examples of temporal patterns include speech signals, signals vs. time produced by sensors, electrocardiograms, and seismograms. Temporal patterns usually involve ordered sequences of data appearing in time.

The goal of pattern classification is to assign a physical object, event, or phenomenon to one of the prespecified *classes* (also called *categories*.) Despite the lack of any formal theory of pattern perception and classification, human beings and animals have performed these tasks since the beginning of their existence. Let us look at some of the classification examples.

The oldest classification tasks required from a human being have been classification of the human environment into such groups of objects as living species,

plants, weather conditions, minerals, tools, human faces, voices, or silhouettes, etc. The interpretation of data has been learned gradually as a result of repetitive inspecting and classifying of examples. When a person perceives a pattern, an inductive inference is made and the perception is associated with some general concepts or clues derived from the person's past experience. The problem of pattern classification may be regarded as one of discriminating the input data within object population via the search for invariant attributes among members of the population.

While some of the tasks mentioned above can be learned easily, the growing complexity of the human environment and technological progress has created classification problems that are diversified and also difficult. As a result, the use of various classifying aids became helpful and in some applications, even indispensable. Reading and processing bank checks exemplifies a classification problem that can be automated. It obviously can be performed by a human worker, however, machine classification can achieve much greater efficiency.

Extensive study of the classification process has led to the development of an abstract mathematical model that provides the theoretical basis for classifier design. Eventually, machine classification came to maturity to help people in their classification tasks. The electrocardiogram waveform, biomedical photograph, or disease diagnosis problem can nowadays be handled by machine classifiers. Other applications include fingerprint identification, patent searches, radar and signal detection, printed and written character classification, and speech recognition.

Figure 3.1(a) shows the block diagram of the recognition and classification system. As mentioned in Section 2.3, recognition is understood here as a class assignment for input patterns that are not identical to the patterns used for training of the classifier. Since the training concept has not been fully explained yet, we will focus first on techniques for classifying patterns.

The classifying system consists of an input transducer providing the input pattern data to the feature extractor. Typically, inputs to the feature extractor are sets of data vectors that belong to a certain category. Assume that each such set member consists of real numbers corresponding to measurement results for a given physical situation. Usually, the converted data at the output of the transducer can be compressed while still maintaining the same level of machine performance. The compressed data are called *features*. The feature extractor at the input of the classifier in Figure 3.1(a) performs the reduction of dimensionality. The feature space dimensionality is postulated to be much smaller than the dimensionality of the pattern space. The feature vectors retain the minimum number of data dimensions while maintaining the probability of correct classification, thus making handling data easier.

An example of possible feature extraction is available in the analysis of speech vowel sounds. A 16-channel filterbank can provide a set of 16-component spectral vectors. The vowel spectral content can be transformed into perceptual quality space consisting of two dimensions only. They are related to tongue height

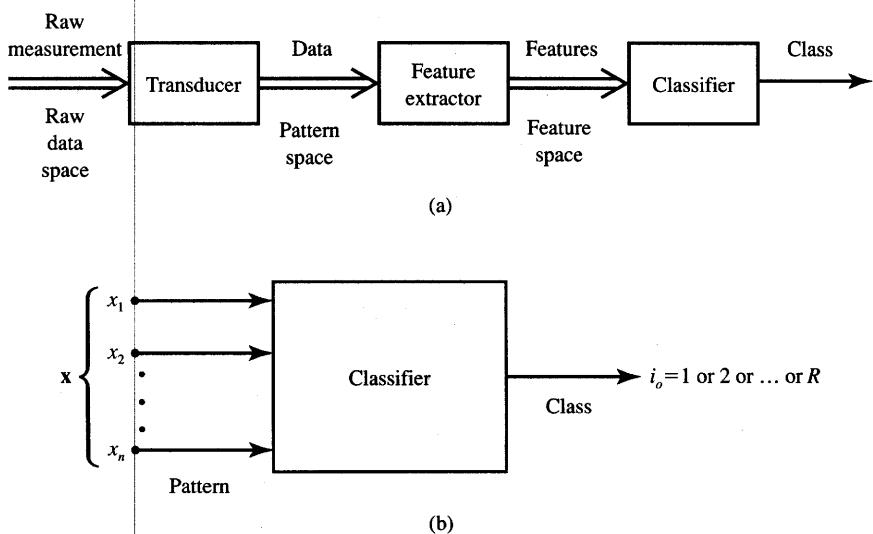


Figure 3.1 Recognition and classification system: (a) overall block diagram and (b) pattern classifier.

and retraction. Another example of dimensionality reduction is the projection of planar data on a single line, reducing the feature vector size to a single dimension. Although the projection of data will often produce a useless mixture, by moving and/or rotating the line it might be possible to find its orientation for which the projected data are well separated. In such a case, two-dimensional data are represented by single-dimensional features denoting the position of the projected points on the line.

It is beyond the scope of this chapter to discuss the selection of measurements or data feature extraction from the input pattern vector. We shall henceforth assume that the sets of extracted feature vectors yield the sets of pattern vectors to be classified and that the extraction, or selection, of input components to the classifier has been done as wisely as possible. Thus, the pattern vector \mathbf{x} shown in Figure 3.1(b) consists of components that may be features. Chapter 7 covers neural network architectures suitable for separate feature extractions.

However, the n -tuple vectors at the input to the classifier on Figure 3.1(b) may also be input pattern data when separate feature extraction does not take place. In such a case the classifier's function is to perform not only the classification itself but also to internally extract input pattern features. The rationale for this approach in our study is that neural networks can be successfully used for joint classification/recognition tasks and for feature extraction. For such networks, the feature extractor and classifier from Figure 3.1(a) can be considered merged to the single classifier network of Figure 3.1(b). Networks that operate on input data and perform no separate feature extraction when classifying patterns are discussed in Section 8.2.

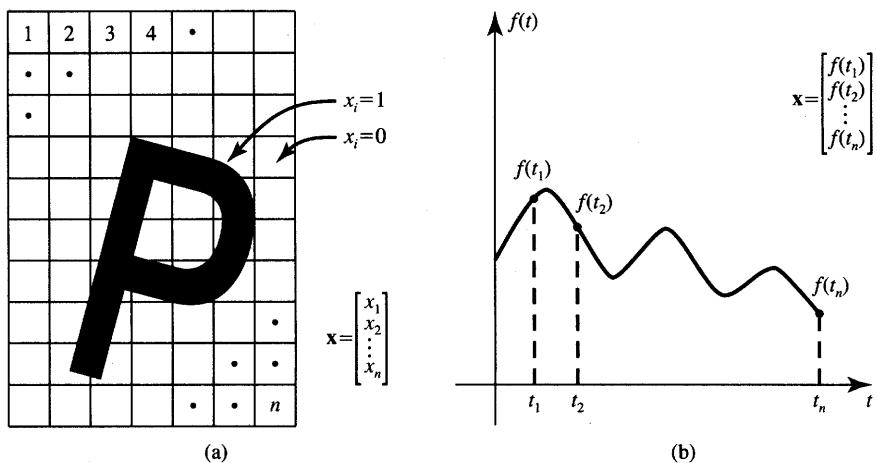


Figure 3.2 Two simple ways of coding patterns into pattern vectors: (a) spatial object and (b) temporal object (waveform).

In further discussion we will represent the classifier input components as a vector \mathbf{x} . The classification at the system's output is obtained by the classifier implementing the decision function $i_o(\mathbf{x})$. The discrete values of the response i_o are 1 or 2 or ... or R . The responses represent the categories into which the patterns should be placed. The classification (decision) function of Equation (3.1) is provided by the transformation, or mapping, of the n -component vector \mathbf{x} into one of the category numbers i_o as shown in Figure 3.1(b):

$$i_o = i_o(\mathbf{x}) \quad (3.1)$$

where

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Figure 3.2 depicts two simple ways to generate the pattern vector for cases of spatial and temporal objects to be classified. In the case shown in Figure 3.2(a), each component x_i of the vector $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_n]^t$ is assigned the value 1 if the i 'th cell contains a portion of a spatial object; otherwise, the value 0 (or -1) is assigned. In the case of a temporal object being a continuous function of time t , the pattern vector may be formed at discrete time instants t_i by letting $x_i = f(t_i)$, for $i = 1, 2, \dots, n$. This is shown in Figure 3.2(b).

Classification can often be conveniently described in geometric terms. Any pattern can be represented by a point in n -dimensional Euclidean space E^n called the *pattern space*. Points in that space corresponding to members of the pattern

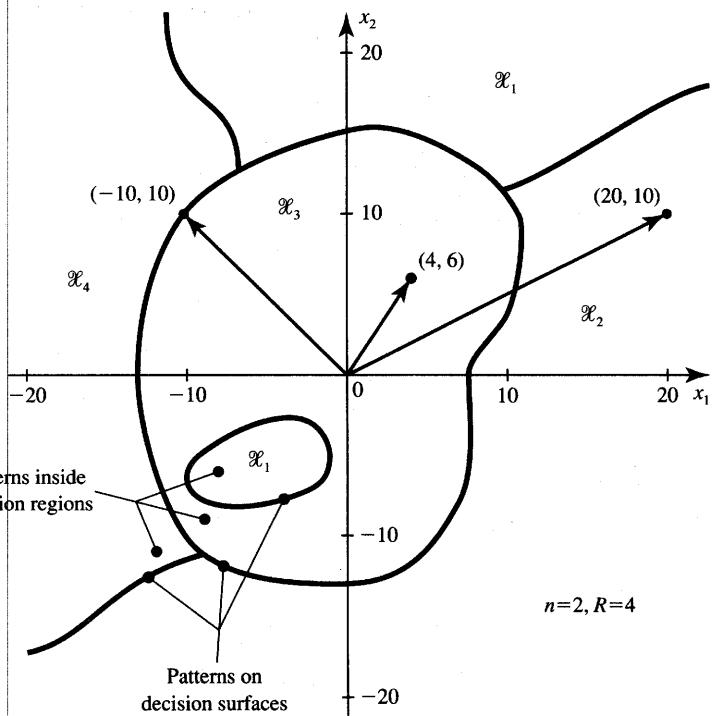


Figure 3.3 Decision regions example.

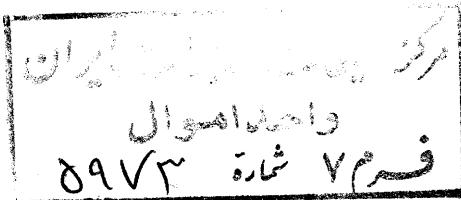
set are n -tuple vectors \mathbf{x} . A pattern classifier maps sets of points in E^n space into one of the numbers $i_o = 1, 2, \dots, R$, as described by the decision function (3.1). The sets containing patterns of classes $1, 2, \dots, R$ are denoted here by $\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_R$, respectively.

An example case for $n = 2$ and $R = 4$ is illustrated in Figure 3.3 showing disjoint regions $\mathcal{X}_1, \dots, \mathcal{X}_4$. Let us postulate for simplicity that the classifier's response as in (3.1) should be the class number. We now have the decision function for a pattern of class j yielding the following result:

$$i_o(\mathbf{x}) = j \quad \text{for all } \mathbf{x} \in \mathcal{X}_j, \quad j = 1, 2, 3, 4$$

Thus, the example vector $\mathbf{x} = [20 \ 10]^t$ belongs to \mathcal{X}_2 and is of class 2, vector $\mathbf{x} = [4 \ 6]^t$ belongs to \mathcal{X}_3 and is of class 3, etc.

The regions denoted \mathcal{X}_i are called *decision regions*. Regions \mathcal{X}_i are separated from each other by so-called *decision surfaces*. We shall assume that patterns located on decision surfaces do not belong to any category. In Figure 3.3 an example of such a pattern located on the boundary is $\mathbf{x} = [-10 \ 10]^t$. Note that the decision surfaces in two-dimensional pattern space E^2 are curved lines. For a more general case of space E^n they may be $(n-1)$ -dimensional hypersurfaces.



3.2

DISCRIMINANT FUNCTIONS

In this chapter, the assumption is made that both a set of n -dimensional patterns $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_P$ and the desired classification for each pattern are known. The size P of the pattern set is finite, and it is usually much larger than the dimensionality n of the pattern space. In many practical cases we will also assume that P is much larger than the number of categories R . Although the assumptions regarding n , P , and R are often valid for practical classification cases, they do not necessarily hold for our study of classification principles, nor do they limit the validity of our final conclusions.

We will first discuss classifiers that use the discriminant functions concept. This discussion will lead to interesting conclusions as to how neural network classifiers should be trained. The study will also explicitly produce some of the training rules introduced in Chapter 2.

Let us assume momentarily, and for the purpose of this presentation, that the classifier has already been designed so that it can correctly perform the classification tasks. During the classification step, the membership in a category needs to be determined by the classifier based on the comparison of R *discriminant functions* $g_1(\mathbf{x}), g_2(\mathbf{x}), \dots, g_R(\mathbf{x})$, computed for the input pattern under consideration. It is convenient to assume that the discriminant functions $g_i(\mathbf{x})$ are scalar values and that the pattern \mathbf{x} belongs to the i 'th category if and only if

$$g_i(\mathbf{x}) > g_j(\mathbf{x}), \quad \text{for } i, j = 1, 2, \dots, R, i \neq j \quad (3.2)$$

Thus, within the region \mathcal{X}_i , the i 'th discriminant function will have the largest value. This maximum property of the discriminant function $g_i(\mathbf{x})$ for the pattern of class i is fundamental, and it will be subsequently used to choose, or assume, specific forms of the $g_i(\mathbf{x})$ functions.

The discriminant functions $g_i(\mathbf{x})$ and $g_j(\mathbf{x})$ for contiguous decision regions \mathcal{X}_i and \mathcal{X}_j define the decision surface between patterns of classes i and j in E^n space. Since the decision surface itself obviously contains patterns \mathbf{x} without membership in any category, it is characterized by $g_i(\mathbf{x})$ equal to $g_j(\mathbf{x})$. Thus, the decision surface equation is

$$g_i(\mathbf{x}) - g_j(\mathbf{x}) = 0 \quad (3.3)$$

Figure 3.4(a) displays six example patterns belonging to one of the two classes, with the simplest example of a decision surface in pattern space x_1, x_2 being here a straight line. The case illustrated here is for $n = R = 2$ and exemplifies the concept of the linear discriminant function. Inspection of the figure indicates that there is an infinite number of discriminant functions yielding correct classification. Let us look at the particulars of the depicted classification task in the following example. This example will also allow the reader to gain better insight into the more formal discussion of classification issues that follows.

EXAMPLE 3.1

Six patterns in two-dimensional pattern space shown in Figure 3.4(a) need to be classified according to their membership in sets as follows

$$\begin{aligned} \left\{ [0 \ 0]^t, [-0.5 \ -1]^t, [-1 \ -2]^t \right\} &: \text{class 1} \\ \left\{ [2 \ 0]^t, [1.5 \ -1]^t, [1 \ -2]^t \right\} &: \text{class 2} \end{aligned}$$

Inspection of the patterns indicates that the equation for the decision surface can be arbitrarily chosen as shown in the figure

$$g(\mathbf{x}) = -2x_1 + x_2 + 2 \quad (3.4)$$

Let us note that in the case discussed here we first arbitrarily select the Eq. (3.4) of the decision surface rather than determine the two discriminant functions $g_i(\mathbf{x})$, for $i = 1, 2$. Equation (3.4) represents the straight line dividing the pattern space that is plane x_1, x_2 into the contiguous decision regions $\mathcal{X}_1, \mathcal{X}_2$. It is obvious that $g(\mathbf{x}) > 0$ and $g(\mathbf{x}) < 0$ in each of the half-planes containing patterns of class 1 and 2, respectively, and $g(\mathbf{x}) = 0$ for all points on the line. Therefore, the evaluation of the sign of $g(\mathbf{x}) = g_1(\mathbf{x}) - g_2(\mathbf{x})$ can in this case replace the evaluation of the general maximum condition as in (3.2). Specifically, the functions $g_1(\mathbf{x})$ and $g_2(\mathbf{x})$ have not even been searched for in this case. Let us note that following explicitly the maximum condition expressed by (3.2) we would have to find and compare two specific discriminant functions.

The following discussion analyzes classification using original condition (3.2) with two suitably chosen discriminant functions: $g_1(\mathbf{x})$ and $g_2(\mathbf{x})$. Note that according to Equation (3.3), the projection of the intersection of two discriminant functions on the plane x_1, x_2 is the decision surface given by (3.4). The example discriminant functions have been arbitrarily chosen as planes of $g_1(\mathbf{x})$ and $g_2(\mathbf{x})$ shown in Figure 3.4(b). The reader can see that they fulfill the correct classification requirements. Their contour maps are illustrated in Figure 3.4(c). The plane equations are:

$$\begin{aligned} 2x_1 - x_2 + 2g_1(\mathbf{x}) - 4 &= 0 \\ -2x_1 + x_2 + 2g_2(\mathbf{x}) &= 0 \end{aligned} \quad (3.5a)$$

Note how appropriate plane equations have been produced. We first observe that the decision line given by the equation $-2x_1 + x_2 + 2 = 0$ has two normal vectors. They are planar vectors $[2 \ -1]^t$ and $[-2 \ 1]^t$. The discriminant function $g_1(\mathbf{x})$ can be built by appropriately selecting its 3-tuple unit normal vector \mathbf{r}_1 . This can be done by augmenting the vector $[2 \ -1]^t$ by a third component of positive value, say equal to 2. The details of the procedure of building this vector are shown in Figure 3.4(d). The resulting normal vector

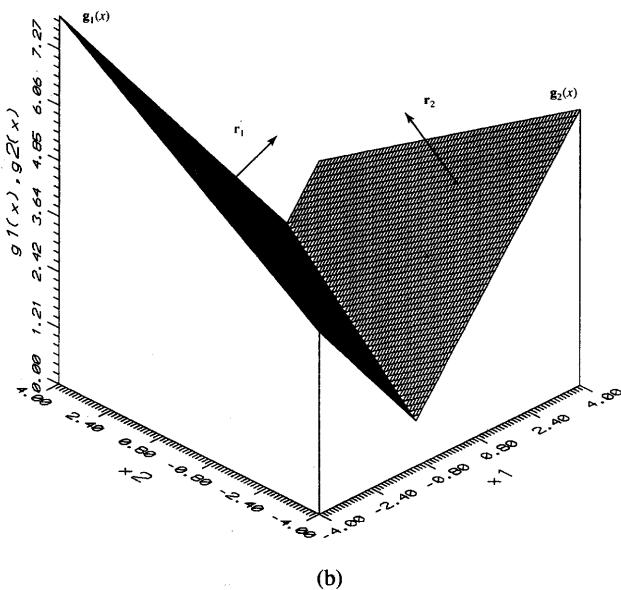
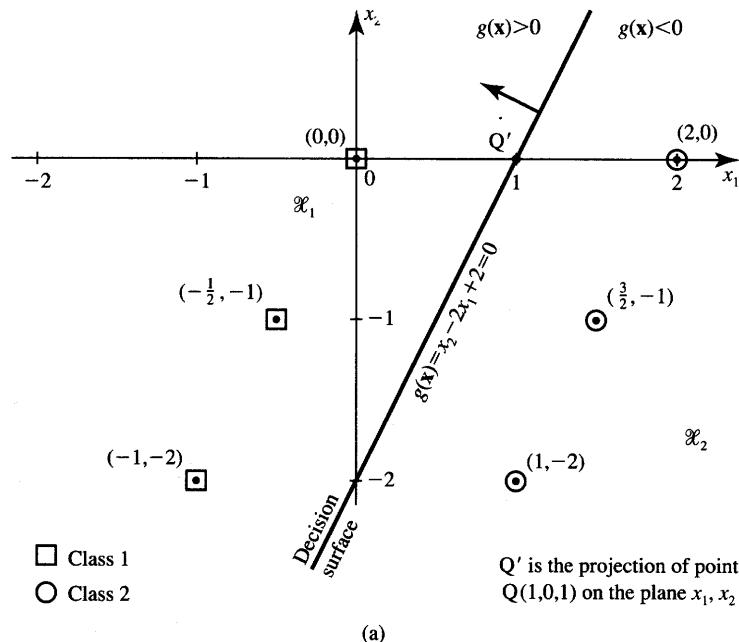
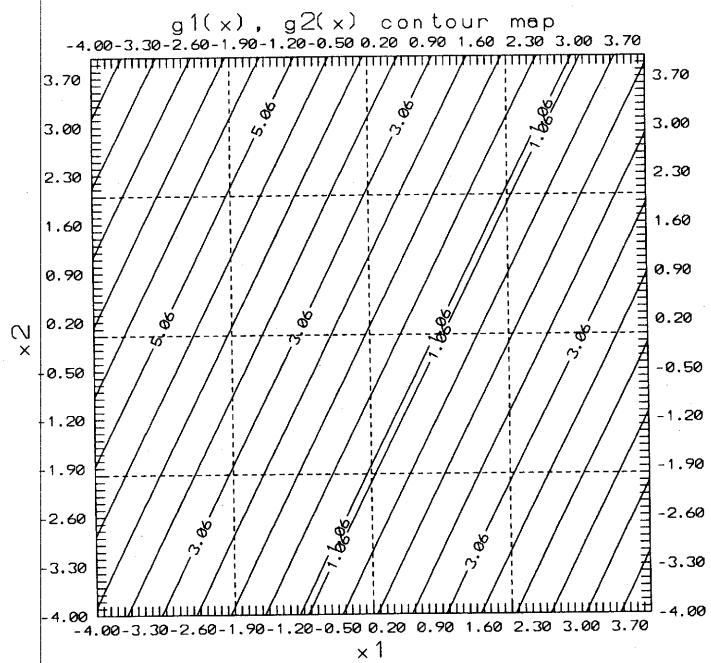
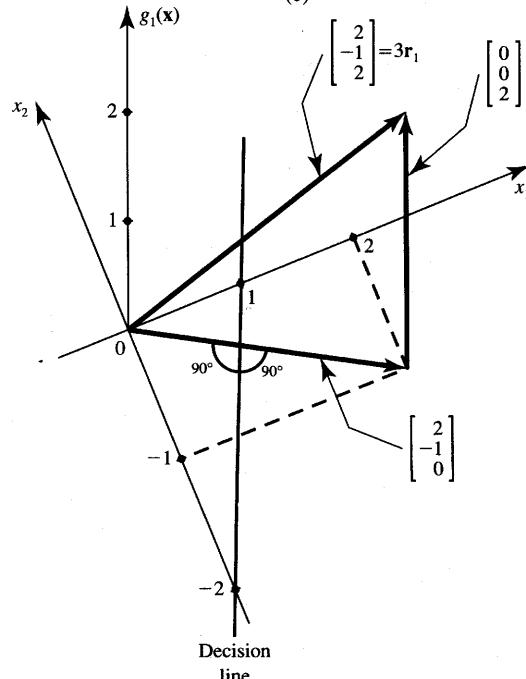


Figure 3.4a,b Illustration for Example 3.1: (a) pattern display and decision surface, (b) discriminant functions.



(c)



(d)

Figure 3.4c,d Illustration for Example 3.1 (continued): (c) contour map of discriminant functions, and (d) construction of the normal vector for $g_1(x)$.

is $[2 \ -1 \ 2]^t$; it thus needs normalization. Similar augmentation can also be carried out for the other planar normal vector $[-2 \ 1]^t$ to yield \mathbf{r}_2 .

The unit normal vectors \mathbf{r}_1 and \mathbf{r}_2 , which provide appropriate plane orientations as shown in Figure 3.4(b) and correct discrimination by the classifier are thus equal (after normalization)

$$\mathbf{r}_1 = \begin{bmatrix} 2 \\ 3 \\ -\frac{1}{3} \\ 2 \end{bmatrix}, \quad \mathbf{r}_2 = \begin{bmatrix} -\frac{2}{3} \\ 3 \\ \frac{1}{3} \\ 2 \end{bmatrix} \quad (3.5b)$$

Let us arbitrarily assume that the discriminant function planes, $g_1(\mathbf{x})$, intersects the point $x_1 = 1, x_2 = 0, g_1 = 1$. The discriminant function plane $g_2(\mathbf{x})$ has to intersect therefore the same point $x_1 = 1, x_2 = 0, g_2 = 1$. For each of the planes we have produced their normal vector and the coordinates of the intersecting point represented by a vector $\mathbf{Q} = [1 \ 0 \ 1]^t$. Using the normal vector-point equation from the Appendix we obtain for the first and second discriminant functions, respectively

$$\begin{aligned} \mathbf{r}_1^t \left(\begin{bmatrix} x_1 \\ x_2 \\ g_1 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \right) &= 0 \\ \mathbf{r}_2^t \left(\begin{bmatrix} x_1 \\ x_2 \\ g_2 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \right) &= 0 \end{aligned} \quad (3.5c)$$

The reader can notice that equations (3.5c) are identical with the equations (3.5a).

From (3.5a) we see that the explicit equations yielding the discriminant functions are:

$$\begin{aligned} g_1(\mathbf{x}) &= \begin{bmatrix} -1 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + 2 \\ g_2(\mathbf{x}) &= \begin{bmatrix} 1 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \end{aligned} \quad (3.6)$$

The reader can easily verify that equating $g_1(\mathbf{x})$ to $g_2(\mathbf{x})$ given by Equation (3.6) leads to the decision surface equation (3.4). Let us point out that the decision surface does not uniquely specify the discriminant functions. Vectors \mathbf{r}_1 and \mathbf{r}_2 as chosen in (3.5b) are also not unique and there are an infinite number of vectors that could be used here. Also, the same arbitrary constant can be added to both $g_1(\mathbf{x})$ and $g_2(\mathbf{x})$ without changing the projection of the $g_1(\mathbf{x}), g_2(\mathbf{x})$ intersection on the plane x_1, x_2 .

To conclude the example, let us compare the computed values of discriminant functions $g_i(\mathbf{x})$ for selected patterns. The comparison should yield the complete class membership information. Indeed, pattern $[2 \ 0]^t$ belongs to \mathcal{X}_2 since $g_2(2 \ 0) = 2 > g_1(2 \ 0) = 0$, pattern $[0 \ 0]^t$ belongs to \mathcal{X}_1 since $g_1(0 \ 0) = 2 > g_2(0 \ 0) = 0$, etc. In this example, the decision surface has been determined by inspection of patterns, and the example discriminant functions have been produced and discussed based on inspection of the geometrical conditions in the pattern space. ■

Assuming that the discriminant functions are known, the block diagram of a basic pattern classifier can now be adopted as in Figure 3.5(a). For a given pattern, the i 'th discriminator computes the value of the function $g_i(\mathbf{x})$ called briefly the *discriminant*. The maximum selector implements condition (3.2) and selects the largest of all inputs, thus yielding the response equal to the category number i_o .

The discussion above and the associated example of classification has highlighted a special case of the classifier into R classes for $R = 2$. Such a classifier is called the *dichotomizer*. Although the ancient Greek civilization is rather famous for other interests than decision-making machines, the word *dichotomizer* is of Greek origin. The two separate greek language roots are *dicha* and *tomia* and they mean *in two* and *cut*, respectively. It has been noted that the general classification condition (3.2) for the case of a dichotomizer can now be reduced to the inspection of the sign of the following discriminant function

$$g(\mathbf{x}) \stackrel{\Delta}{=} g_1(\mathbf{x}) - g_2(\mathbf{x}) \quad (3.7a)$$

Thus, the general classification rule (3.2) can be rewritten for a dichotomizer as follows

$$\begin{aligned} g(\mathbf{x}) > 0 &: \text{class 1} \\ g(\mathbf{x}) < 0 &: \text{class 2} \end{aligned} \quad (3.7b)$$

The evaluation of conditions in (3.7b) is easier to implement in practice than the selection of maximum. Subtraction and sign examination has replaced the maximum value evaluation. A single threshold logic unit (TLU) can be used to build such a simple dichotomizer as shown in Figure 3.5(b). As discussed in the previous chapter, the TLU can be considered as a binary (discrete) version of a neuron. The TLU with weights has been introduced in Chapter 2 as the discrete binary perceptron. The responses 1, -1, of the TLU should be interpreted as indicative of categories 1 and 2, respectively. The TLU element simply implements the sign function defined as

$$i_o = \text{sgn}[g(\mathbf{x})] = \begin{cases} -1 & \text{for } g(\mathbf{x}) < 0 \\ \text{undefined} & \text{for } g(\mathbf{x}) = 0 \\ 1 & \text{for } g(\mathbf{x}) > 0 \end{cases} \quad (3.8)$$

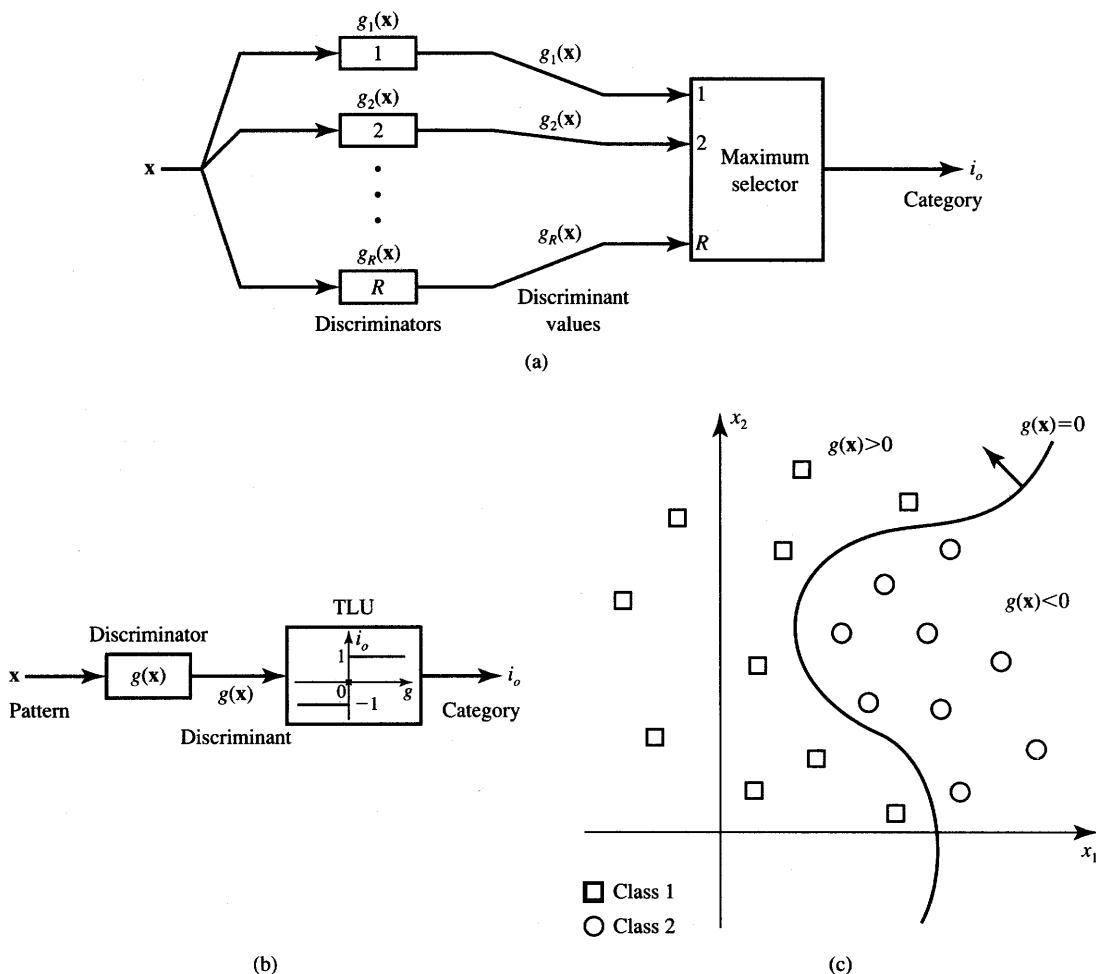


Figure 3.5 Computing the classification: (a) into R categories, (b) dichotomizer ($R = 2$), and (c) decision surface for $n = 2$ and $R = 2$.

Another example of a decision surface for $d = R = 2$ is shown in Figure 3.5(c). Although we can see that the TLU element would probably provide decision about the class membership of the depicted set of patterns, the design of the discriminator for this dichotomizer does not seem as straightforward as in Example 3.1. Let us only notice that the discriminant functions may result as nonlinear functions of x_1 , x_2 , and let us postpone more detailed discussion of such a case until later in this chapter.

Once a general functional form of the discriminant functions has been suitably chosen, discriminants can be computed using *a priori* information about

the classification of patterns, provided that such information is available. In such an approach, the design of a classifier can be based entirely on the computation of decision boundaries as derived from patterns and their membership in classes. Throughout this chapter and most portions of this book, however, we will focus mainly on classifiers whose decision capabilities are generated from training patterns by means of an iterative learning, or training, algorithm. Once a type of discriminant function has been assumed, the algorithm of learning should result in a solution for the initially unknown coefficients of discriminant functions, provided the training pattern sets are separable by the assumed type of decision function. For study of such adaptive, or trainable, classifiers, the following assumptions are made:

1. The training pattern set and classification of all its members are known, thus the training is supervised.
2. The discriminant functions have a linear form and only their coefficients are adjusted in the training procedure.

Under these assumptions, a trainable classifier can be implemented that learns by examples. In this context, we will be interested in input data vectors for which we have *a priori* knowledge of their correct classification. These vectors will be referred to as class prototypes or exemplars. The classification problem will then be one of finding decision surfaces, in n -dimensional space, that will enable correct classification of the prototypes and will afford some degree of confidence in correctly recognizing and classifying unknown patterns that have not been used for training. The only limitation regarding the unknown patterns to be recognized is that they are drawn from the same underlying distributions that have been used for classifier's training.

3.3

LINEAR MACHINE AND MINIMUM DISTANCE CLASSIFICATION

The efficient classifier having the block diagram as shown in Figure 3.5(a) must be described, in general, by discriminant functions that are not linear functions of the inputs x_1, x_2, \dots, x_n . An example of such classification is provided in Figure 3.5(c). As will be shown later, the use of nonlinear discriminant functions can be avoided by changing the classifier's feedforward architecture to the multilayer form. Such an architecture is comprised of more layers of elementary classifiers such as the discussed dichotomizer or the "dichotomizer" providing continuous response between -1 and $+1$. The elementary decision-making discrete, or continuous dichotomizers, will then again be described with the argument $g(x)$ being the basic linear discriminant function.

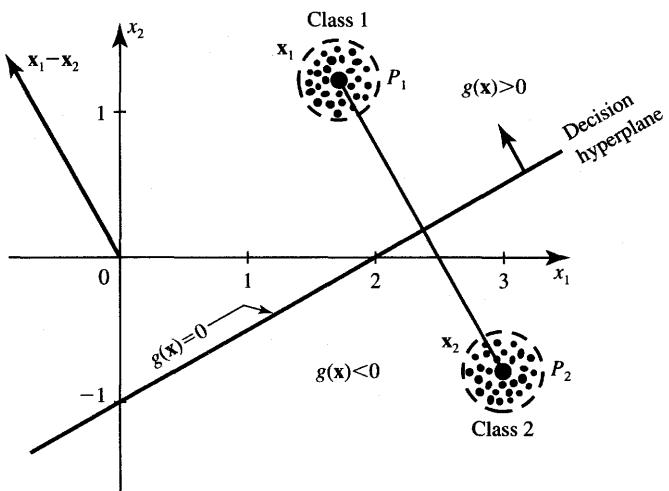


Figure 3.6 Illustration to n -dimensional linear discriminant function ($R = 2$).

Since the linear discriminant function is of special importance, it will be discussed below in detail. It will be assumed throughout that E^n is the n -dimensional Euclidean pattern space. Also, without any loss of generality, we will initially assume that $R = 2$. In the linear classification case, the decision surface is a hyperplane and its equation can be derived based on discussion and generalization of Figure 3.6.

Figure 3.6 depicts two clusters of patterns, each cluster belonging to one known category. The center points of the clusters shown of classes 1 and 2 are vectors \mathbf{x}_1 and \mathbf{x}_2 , respectively. The center, or prototype, points can be interpreted here as centers of gravity for each cluster. We prefer that the decision hyperplane contain the midpoint of the line segment connecting prototype points P_1 and P_2 , and it should be normal to the vector $\mathbf{x}_1 - \mathbf{x}_2$, which is directed toward P_1 . The decision hyperplane equation can thus be written in the following form (see Appendix):

$$\star \quad (\mathbf{x}_1 - \mathbf{x}_2)^T \mathbf{x} + \frac{1}{2} (\|\mathbf{x}_2\|^2 - \|\mathbf{x}_1\|^2) = 0 \quad (3.9)$$

The left side of Equation (3.9) is obviously the dichotomizer's discriminant function $g(\mathbf{x})$. It can also be seen that $g(\mathbf{x})$ implied here constitutes a hyperplane described by the equation

$$w_1 x_1 + w_2 x_2 + \dots + w_n x_n + w_{n+1} = 0, \text{ or} \quad (3.10a)$$

$$\mathbf{w}^T \mathbf{x} + w_{n+1} = 0 \quad (3.10b)$$

or, briefly,

$$\begin{bmatrix} \mathbf{w} \\ w_{n+1} \end{bmatrix}^T \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = 0 \quad (3.10c)$$

where \mathbf{w} denotes the weight vector defined as follows:

$$\mathbf{w} \triangleq \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

The weighting coefficients w_1, w_2, \dots, w_{n+1} of the dichotomizer can now be obtained easily from comparing (3.9) and (3.10) as follows:

$$\begin{aligned} \mathbf{w} &= \mathbf{x}_1 - \mathbf{x}_2 \\ w_{n+1} &= \frac{1}{2} (\|\mathbf{x}_2\|^2 - \|\mathbf{x}_1\|^2) \end{aligned} \quad (3.11)$$

It can be seen from Equation (3.11) that the discriminant function becomes explicitly known if prototype points P_1 and P_2 are known. We can also note that unless the cluster center coordinates $\mathbf{x}_1, \mathbf{x}_2$ are known, $g(\mathbf{x})$ cannot be determined *a priori* using the method just presented.

The linear form of discriminant functions can also be used for classifications between more than two categories. In the case of R pairwise separable classes, there will be up to $R(R - 1)/2$ decision hyperplanes like the one computed in (3.11) for $R = 2$. For $R = 3$, there are up to three decision hyperplanes. For a larger number of classes, some decision regions $\mathcal{X}_i, \mathcal{X}_j$ may not be contiguous, thus eliminating some decision hyperplanes. In such cases, the equation $g_i(\mathbf{x}) = g_j(\mathbf{x})$ has no solution. Still, the dichotomizer example just discussed can be considered as a simple case of a multiclass minimum-distance classifier. Such classifiers will be discussed in more detail later in this chapter.

Let us assume that a minimum-distance classification is required to classify patterns into one of the R categories. Each of the R classes is represented by prototype points P_1, P_2, \dots, P_R being vectors $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_R$, respectively. The Euclidean distance between input pattern \mathbf{x} and the prototype pattern vector \mathbf{x}_i is expressed by the norm of the vector $\mathbf{x} - \mathbf{x}_i$ as follows:

$$\|\mathbf{x} - \mathbf{x}_i\| = \sqrt{(\mathbf{x} - \mathbf{x}_i)^t(\mathbf{x} - \mathbf{x}_i)} \quad (3.12)$$

A minimum-distance classifier computes the distance from pattern \mathbf{x} of unknown classification to each prototype. Then, the category number of that closest, or smallest distance, prototype is assigned to the unknown pattern. Calculating the squared distances from Equation (3.12) yields

$$\|\mathbf{x} - \mathbf{x}_i\|^2 = \mathbf{x}^t \mathbf{x} - 2\mathbf{x}_i^t \mathbf{x} + \mathbf{x}_i^t \mathbf{x}_i, \quad \text{for } i = 1, 2, \dots, R \quad (3.13)$$

Obviously, the term $\mathbf{x}^t \mathbf{x}$ is independent of i and shows up in each of the R distances under evaluation in Equation (3.13). Thus, it will suffice to compute only R terms, $2\mathbf{x}_i^t \mathbf{x} - \mathbf{x}_i^t \mathbf{x}_i$, for $i = 1, \dots, R$, in (3.13), and to determine for which \mathbf{x}_i this term takes the largest of all R values. It can also be seen that choosing the largest of the terms $\mathbf{x}_i^t \mathbf{x} - 0.5\mathbf{x}_i^t \mathbf{x}_i$ is equivalent to choosing the smallest of

the distances $\|\mathbf{x} - \mathbf{x}_i\|$. This property can now be used to equate the highlighted term with a discriminant function $g_i(\mathbf{x})$:

$$g_i(\mathbf{x}) = \mathbf{x}_i^t \mathbf{x} - \frac{1}{2} \mathbf{x}_i^t \mathbf{x}_i, \quad \text{for } i = 1, 2, \dots, R \quad (3.14)$$

It now becomes clear that the discriminant function (3.14) is of the general linear form, which can be expressed as:

$$g_i(\mathbf{x}) = \mathbf{w}_i^t \mathbf{x} + w_{i,n+1}, \quad \text{for } i = 1, 2, \dots, R \quad (3.15)$$

The discriminant function coefficients that are weights \mathbf{w}_i can be determined by comparing (3.14) and (3.15) as follows:

$$\begin{aligned} \mathbf{w}_i &= \mathbf{x}_i \\ w_{i,n+1} &= -\frac{1}{2} \mathbf{x}_i^t \mathbf{x}_i, \quad \text{for } i = 1, 2, \dots, R \end{aligned} \quad (3.16)$$

At this point, note that minimum-distance classifiers can be considered as linear classifiers, sometimes called *linear machines*. Since minimum-distance classifiers assign category membership based on the closest match between each prototype and the current input pattern, the approach is also called *correlation classification*. The block diagram of a linear machine employing linear discriminant functions as in Equation (3.15) is shown in Figure 3.7. It can be viewed as a special case of the more general classifier depicted in Figure 3.5. The machine consists of R scalar product computing nodes and of a single maximum selector. During classification, after simultaneously computing all of the R discriminants $g_i(\mathbf{x})$ for a submitted pattern, the output stage of the classifier selects the maximum discriminant and responds with the number of the discriminant having the largest value.

Let us finally notice that the decision surface S_{ij} for the contiguous decision regions $\mathcal{X}_i, \mathcal{X}_j$ is a hyperplane given by the equation

$$g_i(\mathbf{x}) - g_j(\mathbf{x}) = 0, \quad \text{or} \quad (3.17a)$$

$$\mathbf{w}_i^t \mathbf{x} + w_{i,n+1} - \mathbf{w}_j^t \mathbf{x} - w_{j,n+1} = 0 \quad (3.17b)$$

It is a widely accepted convention to append formally a 1 as the $n + 1$ 'th component of each pattern vector. The augmented pattern vector is now denoted by \mathbf{y} , it consists of $n + 1$ rows, and is defined as follows:

$$\mathbf{y} \triangleq \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} \quad (3.18)$$

Using the notation of the augmented pattern vector allows for rewriting expression (3.15) for the linear discriminant function to the more compact form of

$$g_i(\mathbf{y}) = \mathbf{w}_i^t \mathbf{y} \quad (3.19)$$

Note, however, that whenever the augmented pattern vector is used, the associated weight vector \mathbf{w} contains $n + 1$ components. The augmenting weight component is $w_{i,n+1}$, for $i = 1, 2, \dots, R$. For the sake of notational simplicity, the notation

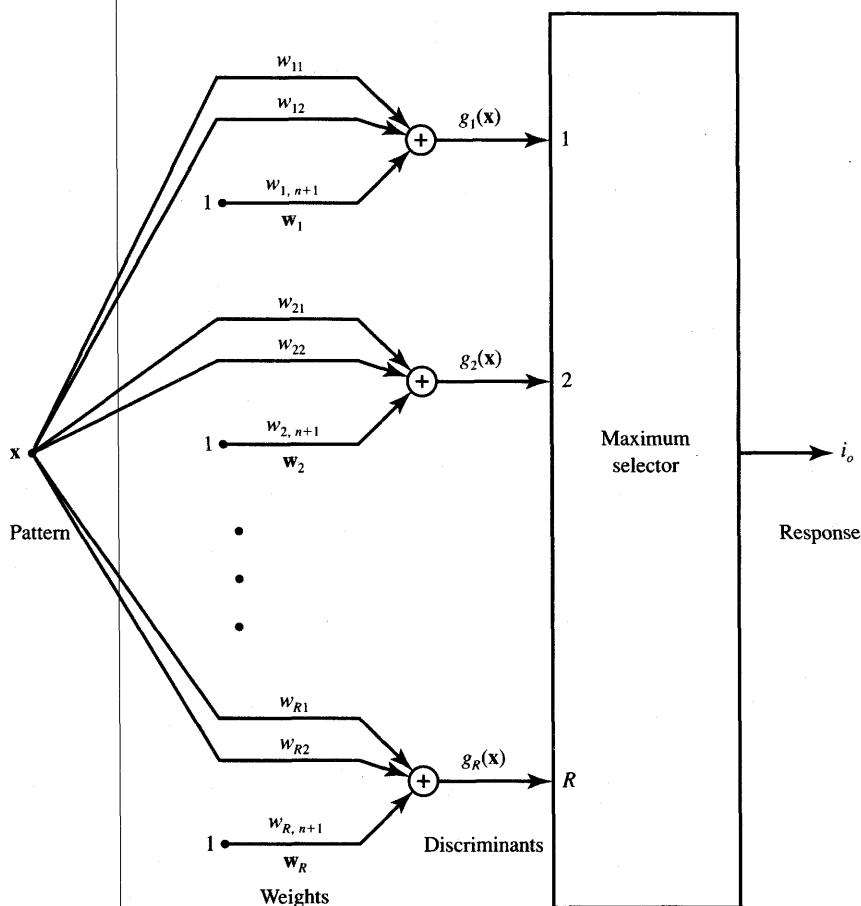


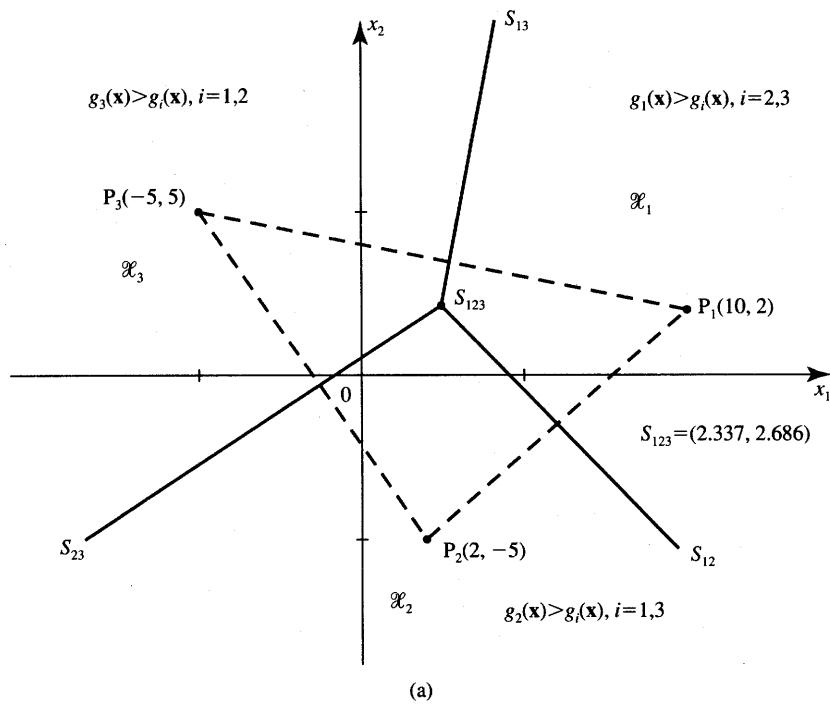
Figure 3.7 A linear classifier.

for both weight vectors and augmented weight vectors are the same throughout the text. Whether or not a pattern or weight vector has been augmented can usually be determined from the context.

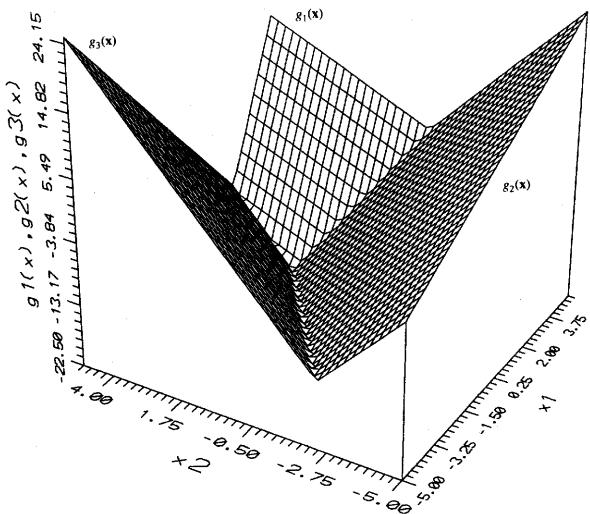
EXAMPLE 3.2

In this example a linear (minimum-distance) classifier is designed. Decision lines are generated using *a priori* knowledge about the center of gravity of the prototype points. The assumed prototype points are as shown in Figure 3.8(a) and their coordinates are

$$\mathbf{x}_1 = \begin{bmatrix} 10 \\ 2 \end{bmatrix}, \mathbf{x}_2 = \begin{bmatrix} 2 \\ -5 \end{bmatrix}, \mathbf{x}_3 = \begin{bmatrix} -5 \\ 5 \end{bmatrix}$$



(a)



(b)

Figure 3.8a,b Illustration to linear classifier with $n = 2$ and $R = 3$ in Example 3.2: (a) geometrical interpretation, (b) discriminant functions.

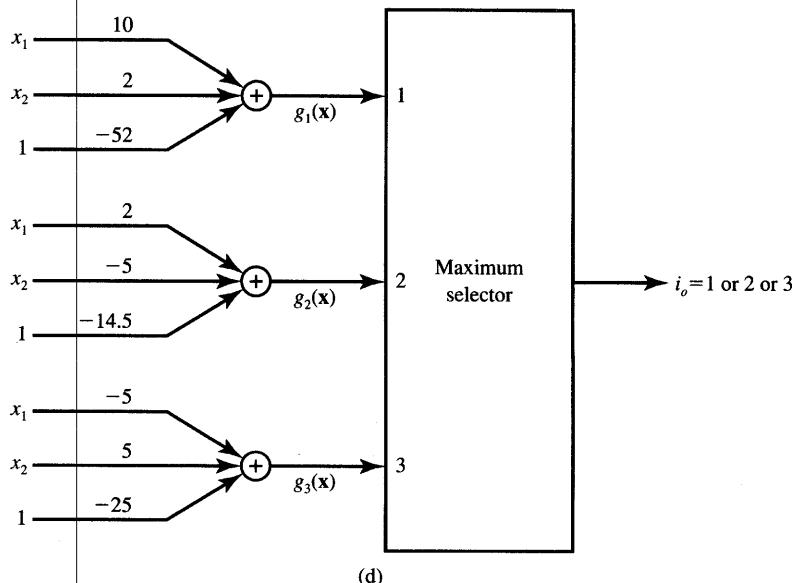
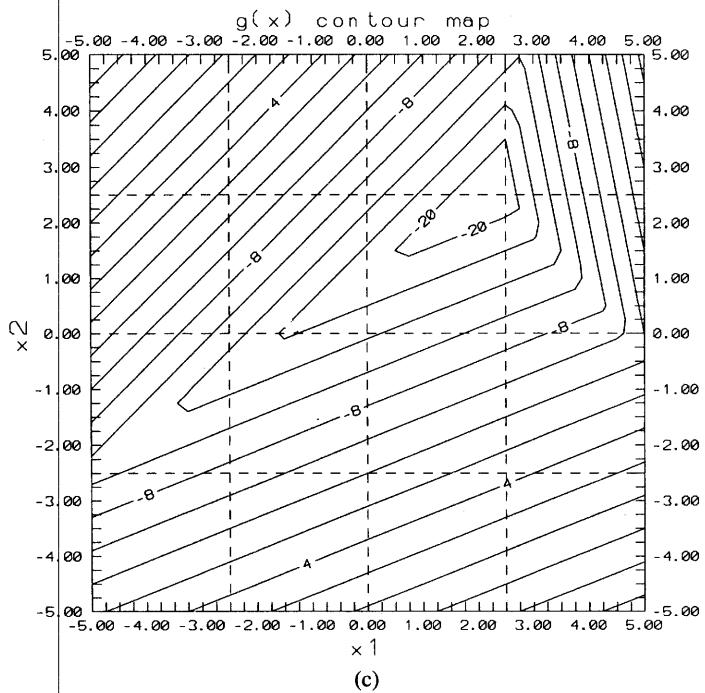


Figure 3.8c,d Illustration to linear classifier with $n = 2$ and $R = 3$ in Example 3.2 (continued): (c) contour map of discriminant functions, and (d) classifier using the maximum selector.

It is also assumed that each prototype point index corresponds to its class number. Using formula (3.16) for $R = 3$, the weight vectors can be obtained as

$$\mathbf{w}_1 = \begin{bmatrix} 10 \\ 2 \\ -52 \end{bmatrix}, \mathbf{w}_2 = \begin{bmatrix} 2 \\ -5 \\ -14.5 \end{bmatrix}, \mathbf{w}_3 = \begin{bmatrix} -5 \\ 5 \\ -25 \end{bmatrix} \quad (3.20a)$$

Note that the weight vectors in (3.20a) are augmented. The corresponding linear discriminant functions are

$$\begin{aligned} g_1(\mathbf{x}) &= 10x_1 + 2x_2 - 52 \\ g_2(\mathbf{x}) &= 2x_1 - 5x_2 - 14.5 \\ g_3(\mathbf{x}) &= -5x_1 + 5x_2 - 25 \end{aligned} \quad (3.20b)$$

The three discriminant functions are shown in Figure 3.8(b), and their contour map in the pattern space x_1, x_2 is illustrated in Figure 3.8(c). Based on the computed discriminant functions, a minimum-distance classifier using the maximum selector can be completed for $R = 3$. The resulting classifier is shown in Figure 3.8(d). Inspecting Figure 3.8(a) reveals that there are three decision lines S_{12} , S_{13} , and S_{23} separating the contiguous decision regions $\mathcal{X}_1, \mathcal{X}_2; \mathcal{X}_1, \mathcal{X}_3$; and $\mathcal{X}_2, \mathcal{X}_3$, respectively. These lines are given by the projections on the pattern plane x_1, x_2 of the intersections of the discriminant functions of Figures 3.8(b) and (c). The decision lines can be calculated by using the condition (3.17) and the discriminant functions (3.20b) as

$$\begin{aligned} S_{12}: 8x_1 + 7x_2 - 37.5 &= 0 \\ S_{13}: -15x_1 + 3x_2 + 27 &= 0 \\ S_{23}: -7x_1 + 10x_2 - 10.5 &= 0 \end{aligned} \quad (3.20c)$$

Substituting the weight vectors (3.20a) directly into the decision function formula (3.17b) also results in the decision lines (3.20c). In this example we have derived the weight vectors and equations for decision surfaces for a three-class classifier. We have also provided insight into the geometrical relationships in the pattern space. ■

The notion of linearly separable patterns is now introduced. Assume that there is a pattern set \mathcal{X} . This set is divided into subsets $\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_R$, respectively. If a linear machine can classify the patterns from \mathcal{X}_i as belonging to class i , for $i = 1, 2, \dots, R$, then the pattern sets are *linearly separable*. Using this property of the linear discriminant functions, the linear separability can be formulated more formally. If R linear functions of \mathbf{x} as in (3.10) exist such that

$$g_i(\mathbf{x}) > g_j(\mathbf{x}) \quad \text{for all } \mathbf{x} \in \mathcal{X}_i, \quad i = 1, 2, \dots, R; j = 1, 2, \dots, R, i \neq j$$

then the pattern sets \mathcal{X}_i are linearly separable.

Figure 3.9 shows the example sets of linearly nonseparable patterns in two- and three-dimensional pattern space. It depicts the parity function $f(x_1, x_2, \dots, x_n)$

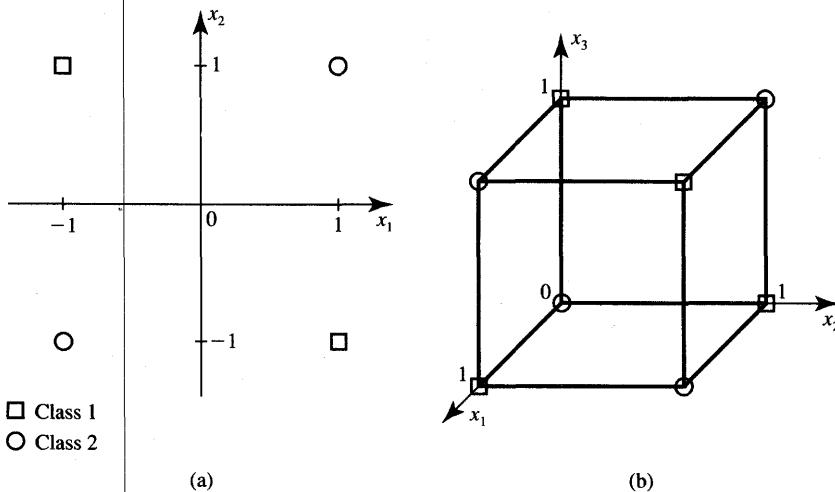


Figure 3.9 Example of parity function resulting in linearly nonseparable patterns ($R = 2$):
 (a) $x_1 \oplus x_2$ and (b) $x_1 \oplus x_2 \oplus x_3$.

for two and three variables (see Appendix). Figure 3.9(a) exemplifies the bipolar and Figure 3.9(b) the unipolar version of the parity function spanned at the corners of an n -dimensional hypercube. The parity function as defined in Equation (3.21) is often called the *XOR* function:

$$XOR(x_1, x_2, \dots, x_n) = x_1 \oplus x_2 \oplus \dots \oplus x_n \quad (3.21)$$

where the symbol \oplus denotes the Exclusive OR Boolean function operator. This function is often convenient to exemplify classification of patterns that are linearly nonseparable.

Geometrically, it can easily be seen that if hyperplanes exist that divide the sets $\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_R$, then the patterns are linearly separable. In contrast to Figures 3.9, 3.3, and 3.5(c) showing examples of linearly nonseparable patterns, Figures 3.4, 3.6, 3.8(a) show classification of patterns that are linearly separable. It can be noticed that decision surfaces for linearly separable patterns define convex decision regions in the pattern space.

3.4

NONPARAMETRIC TRAINING CONCEPT

Thus far our approach to the design of pattern classifiers has been analytical and based on computations of decision boundaries derived from inspection of sample patterns, prototypes, or their clusters. In theoretical considerations and

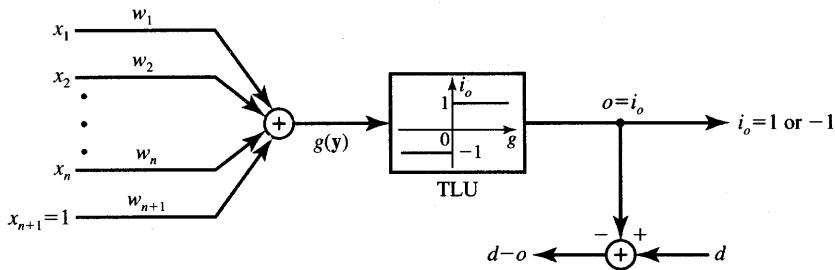


Figure 3.10 Linear dichotomizer using hard-limiting threshold element, or the TLU-based perceptron.

Examples 3.1 and 3.2, we have shown that coefficients of linear discriminant functions called weights can be determined based on *a priori* information about sets of patterns and their class membership.

In this section we will begin to examine neural network classifiers that derive their weights during the learning cycle. The sample pattern vectors $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_P$, called the *training sequence*, are presented to the machine along with the correct response. The response is provided by the teacher and specifies the classification information for each input vector. The classifier modifies its parameters by means of iterative, supervised learning. The network learns from experience by comparing the targeted correct response with the actual response. The classifier structure is usually adjusted after each incorrect response based on the error value generated.

Let us now look again at the dichotomizer introduced and defined in Section 3.2. We will develop a supervised training procedure for this two-class linear classifier. The expanded diagram of the dichotomizer introduced originally in Figure 3.5(b) is now redrawn in Figure 3.10. The dichotomizer shown consists of $n + 1$ weights and the TLU performing as a binary decision element. It is identical to the binary bipolar perceptron from Figure 2.7(a). The TLU itself can be considered a binary response neuron. The input to the binary response neuron is the weighted sum of components of the augmented input vector \mathbf{y} .

In the next part of this section, we discuss the adaptive linear binary classifier and derive the perceptron training algorithm based on the originally nonadaptive dichotomizer. Assuming that the desired response is provided, the error signal is computed. The error information can be used to adapt the weights of the discrete perceptron from Figure 3.10. First we examine the geometrical conditions in the augmented weight space. This will make it possible to devise a meaningful training procedure for the dichotomizer under consideration.

From previous considerations we know that the decision surface equation in n -dimensional pattern space is

$$\mathbf{w}'\mathbf{x} + w_{n+1} = 0 \quad (3.22a)$$

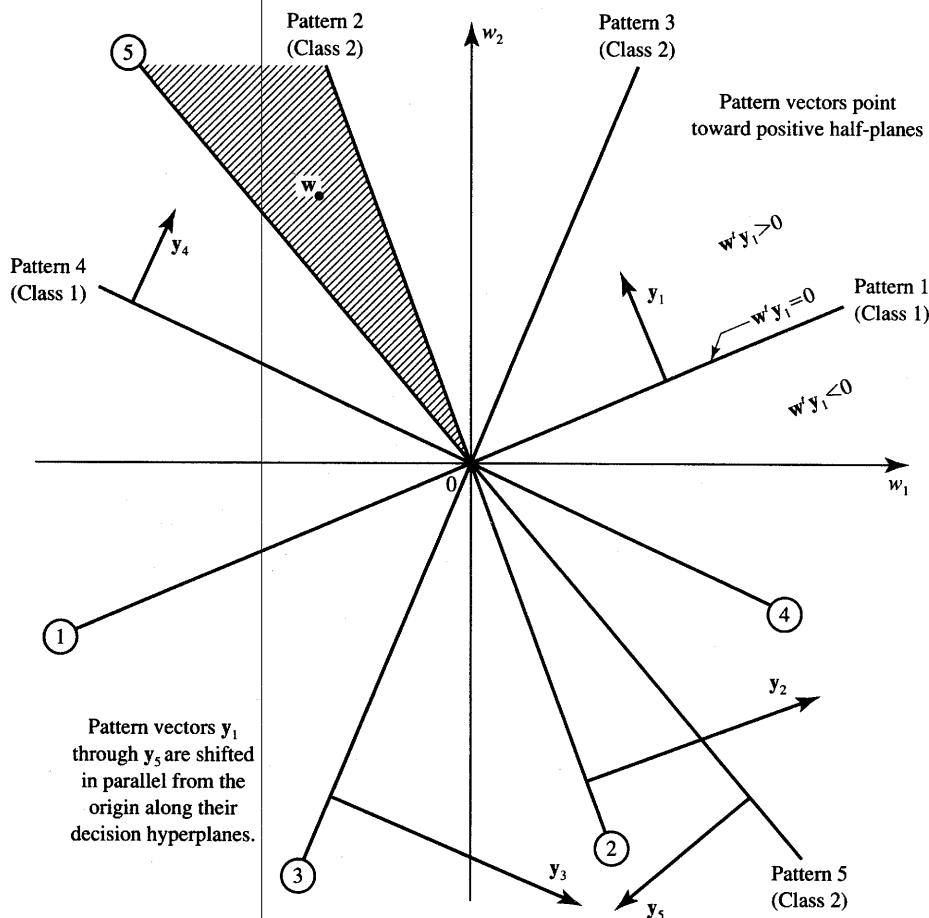


Figure 3.11 Decision hyperplane in augmented weight space for a five pattern set from two classes.

and Equation (3.22a) can be rewritten in the augmented weight space E^{n+1} as

$$\mathbf{w}' \mathbf{y} = 0 \quad (3.22b)$$

This is a normal vector-point equation and is the first equation in this text considered in the weight space as compared to the earlier equations evaluated in the pattern space. Equation (3.22b) describes a decision hyperplane in augmented weight space. In contrast to the decision hyperplane of Equation (3.22a) in n -dimensional space, this hyperplane always intersects the origin, point $\mathbf{w} = \mathbf{0}$. Its normal vector, which is perpendicular to the plane, is the pattern \mathbf{y} . To visualize this, five example decision hyperplanes in the augmented weight space for five prototype patterns of two classes are shown in Figure 3.11.

In further discussion it will be understood that the normal vector will always point toward the side of the space for which $\mathbf{w}'\mathbf{y} > 0$, called the positive side, or semispace, of the hyperplane. Accordingly, the vector \mathbf{y}_1 points toward the positive side of the decision hyperplane $\mathbf{w}'\mathbf{y}_1 = 0$ in Figure 3.11. The pattern vector orientation is thus fixed toward class 1 decision half-plane. The same remarks also apply to the remaining four patterns $\mathbf{y}_2, \dots, \mathbf{y}_5$, for which decision hyperplanes and normal vectors have also been shown in the figure. By labeling each decision boundary in the augmented weight space with an arrow pointing into the positive half-plane, we can easily find a region in the weight space that satisfies the linearly separable classification. (This notation using arrows or pattern vectors for pointing positive half-space is used throughout the text.) To find the solution for weights, we will look for the intersection of the positive decision regions due to the prototypes of class 1 and of the negative decision regions due to the prototypes of class 2.

Inspection of the figure reveals that the intersection of the sets of weights yielding all five correct classifications of depicted patterns is in the shaded region of the second quadrant as shown in Figure 3.11. Let us now attempt to arrive iteratively at the weight vector \mathbf{w} located in the shaded weight solution area. To accomplish this, the weights need to be adjusted from the initial value located anywhere in the weight space. This assumption is due to our ignorance of the weight solution region as well as weight initialization. The adjustment discussed, or network training, is based on an error-correction scheme.

As shown below, analysis of geometrical conditions can provide useful guidelines for developing the weight vector adjustment procedure. However, if the dimensionality of the augmented pattern vector is higher than three, our powers of visualization are no longer of assistance in determining the decision or adjustment conditions. Under these circumstances, the only reasonable recourse would be to follow the analytical approach. Fortunately, such an analytical approach can be devised based on geometrical conditions and visualization of decision boundaries for two or three dimensions only of pattern vectors.

Figure 3.12(a) shows a decision surface for the training pattern \mathbf{y}_1 in the augmented weight space of the discrete perceptron from Figure 3.10. Assume that the initial weights are \mathbf{w}^1 (case A) and that pattern \mathbf{y}_1 of class 1 is now input. In such a case, weights \mathbf{w}^1 are located in the negative half-plane. Obviously, the pattern is misclassified since we have for the present weights \mathbf{w}^1 and pattern \mathbf{y}_1 a negative decision due to $\mathbf{w}^{1t}\mathbf{y}_1 < 0$. To increase the discriminant function $\mathbf{w}^{1t}\mathbf{y}_1$, we need to adjust the weight vector, preferably in the direction of steepest increase, which is that of a gradient. The gradient $\nabla_{\mathbf{w}}$ can be expressed as follows:

$$\nabla_{\mathbf{w}}(\mathbf{w}'\mathbf{y}_1) = \mathbf{y}_1 \quad (3.23)$$

Thus, when the pattern of class 1 is misclassified, the adjusted weights should preferably become

$$\mathbf{w}' = \mathbf{w}^1 + c\mathbf{y}_1 \quad (3.24)$$

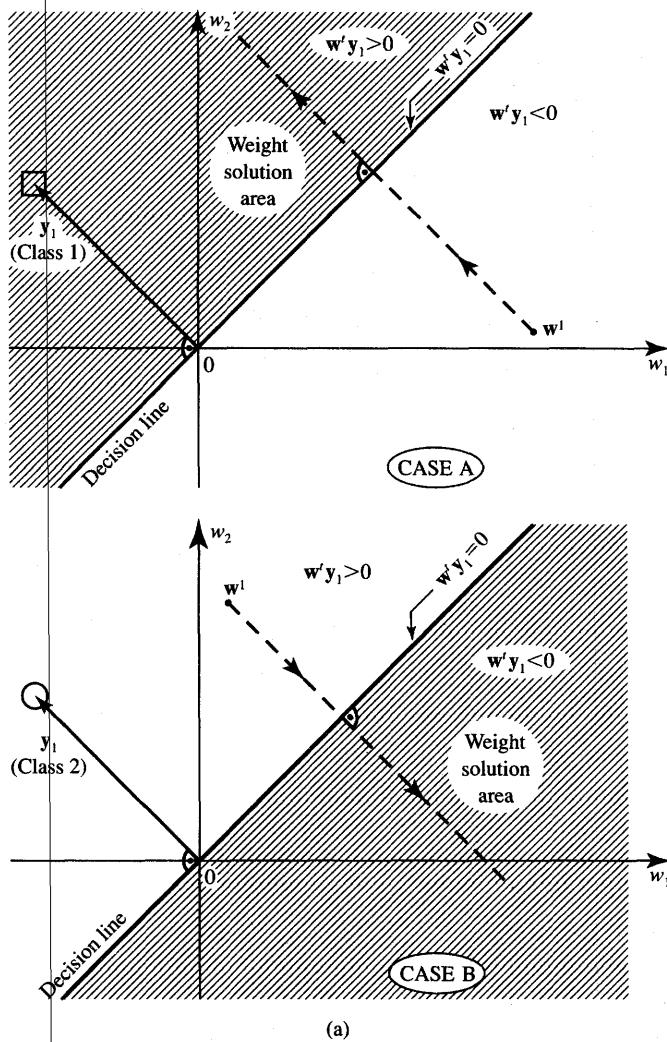


Figure 3.12a Weight adjustments of learning dichotomizer: (a) steepest descent.

where constant $c > 0$ is called the *correction increment*, and a prime is used to denote weights after correction. It can be seen that c controls the size of the adjustment. Repeating the weight adjustment a number of times would bring the weight into the shaded solution region, independent of the chosen correction increment.

Case B in Figure 3.12(a) illustrates a similar misclassification with the initial weights now at w^1 and y_1 of class 2 being input. Since $w^1y_1 > 0$ and misclassification occurs also in this case, the discriminant function needs to be changed through weight adjustment, preferably in the steepest decrease, or negative gradient, direction. We thus obtain from Equation (3.23) the following expression

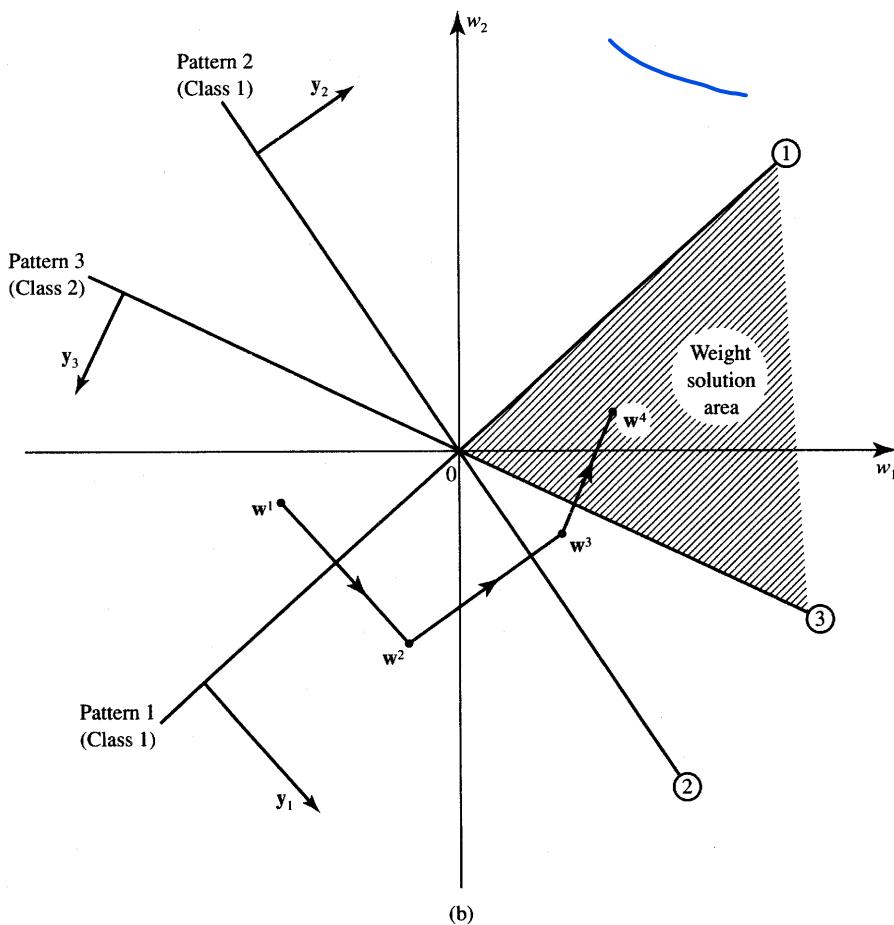


Figure 3.12b Weight adjustments of learning dichotomizer (continued): (b) example.

for the adjusted weights:

$$\mathbf{w}' = \mathbf{w}^1 - c\mathbf{y}_1 \quad (3.25)$$

An illustration of a series of weight adjustments in the augmented weight space is shown in Figure 3.12(b). It depicts the weight corrections for an ordered training set of inputs consisting of three augmented pattern vectors \mathbf{y}_1 , \mathbf{y}_2 , and \mathbf{y}_3 . The patterns are numbered here in the sequence as they appear at the input of the classifier. As shown in previous discussion, weights in the shaded region will provide the solution for the augmented weight vector. Three lines labeled ①, ②, and ③ are obviously fixed for variable weight values that result during the training process shown. This is because their normal vectors, patterns \mathbf{y}_1 , \mathbf{y}_2 , and \mathbf{y}_3 , are fixed.

Consider training starting at the initial weight of \mathbf{w}^1 . During the first step of classification, the first prototype pattern of class 1 is presented and misclassified.

The new augmented weights after correction become

$$\mathbf{w}^2 = \mathbf{w}^1 + c\mathbf{y}_1$$

Similarly, when the second pattern is presented for the current weight vector \mathbf{w}^2 , it is misclassified, and the corrected weight vector \mathbf{w}^2 takes the value

$$\mathbf{w}^3 = \mathbf{w}^2 + c\mathbf{y}_2$$

Since the third pattern belongs to class 2, the misclassification happens again in this step and the new weight vector results as

$$\mathbf{w}^4 = \mathbf{w}^3 - c\mathbf{y}_3$$

Notice that the discussed training procedure with input \mathbf{y} is equivalent to

- increasing the discriminant function $g(\mathbf{y})$ as in (3.19) by $c\|\mathbf{y}\|^2$ if the pattern of class 1 is undetected, and
- decreasing the discriminant function by $c\|\mathbf{y}\|^2$ if the pattern of class 2 is undetected.

Both the increment and decrement of $g(\mathbf{y})$ take place in such a way that the weight vector is displaced in the direction normal to the decision hyperplane. Since this change is in the direction of the shortest path to the correct classification weight area, it is the most appropriate direction.

The supervised training procedure can now be summarized using the following expression for the augmented weight vector \mathbf{w}' after adjustment

$$\mathbf{w}' = \mathbf{w} \pm c\mathbf{y} \quad (3.26)$$

where the positive sign in (3.26) applies for undetected pattern of class 1, and the negative sign applies for undetected pattern of class 2. If a correct classification takes place under this rule, no adjustment of weights is made.

The reader can notice that weight correction formula (3.26) is the same as the perceptron learning rule (2.34–5). Accurate correspondence of both training methods requires that the learning constant in (2.34–5) is a half of the correction increment c in (3.26), and that \mathbf{x} used in (2.34–5) is the augmented pattern vector.

3.5

TRAINING AND CLASSIFICATION USING THE DISCRETE PERCEPTRON: ALGORITHM AND EXAMPLE

Let us look now in more detail at the weight adjustment aspects. Again using the geometrical relationship, we choose the correction increment c so that

the weight adjustment step size is meaningfully controlled. The distance p of a point \mathbf{w}^1 from the plane $\mathbf{w}^t \mathbf{y} = 0$ in $(n + 1)$ -dimensional Euclidean space is computed according to the formula (see Appendix):

$$p = \pm \frac{\mathbf{w}^{1t} \mathbf{y}}{\|\mathbf{y}\|} \quad (3.27a)$$

where the sign in front of the fraction is chosen to be the opposite of the sign of the value of w_{n+1} . A simpler rule is that the sign must be chosen to be identical to the sign of $\mathbf{w}^{1t} \mathbf{y}$. Since p is always a nonnegative scalar by definition of the distance, expression (3.27a) can be rewritten using the absolute value notation as follows:

$$p = \frac{|\mathbf{w}^{1t} \mathbf{y}|}{\|\mathbf{y}\|} \quad (3.27b)$$

Let us now require that the correction increment constant c be selected such that the corrected weight vector \mathbf{w}^2 based on (3.26) dislocates on the decision hyperplane $\mathbf{w}^{2t} \mathbf{y} = 0$, which is the decision hyperplane used for this particular correction step. This implies that

$$\begin{aligned} \mathbf{w}^{2t} \mathbf{y} &= 0, \text{ or} \\ (\mathbf{w}^1 \pm c \mathbf{y})^t \mathbf{y} &= 0 \end{aligned} \quad (3.28)$$

and the required correction increment results for this training step as

$$c = \mp \frac{\mathbf{w}^{1t} \mathbf{y}}{\mathbf{y}^t \mathbf{y}} \quad (3.29a)$$

Since the correction increment c is positive, (3.29a) can be briefly rewritten as

$$c = \frac{|\mathbf{w}^{1t} \mathbf{y}|}{\mathbf{y}^t \mathbf{y}} \quad (3.29b)$$

The length of the weight adjustment vector $c \mathbf{y}$ can now be expressed as

$$\|c \mathbf{y}\| = \frac{|\mathbf{w}^{1t} \mathbf{y}|}{\mathbf{y}^t \mathbf{y}} \|\mathbf{y}\| \quad (3.30)$$

Noting that $\mathbf{y}^t \mathbf{y} = \|\mathbf{y}\|^2$ leads to the conclusion that, as required, the distance p from the point \mathbf{w}^1 to the decision plane and expressed by (3.27) is identical to the length of the weight increment vector (3.30). For this case the correction increment c is therefore not constant and depends on the current training pattern as expressed by (3.29).

The basic correction rule (3.26) for $c = 1$ leads to a very simple adjustment of the weight vector. Such adjustment alters the weight vector exactly by the pattern vector \mathbf{y} . Using the value of the correction increment calculated in (3.29) as a reference, several different adjustment techniques can be devised depending on the length of the weight correction vector $\mathbf{w}^2 - \mathbf{w}^1$. This length is proportional

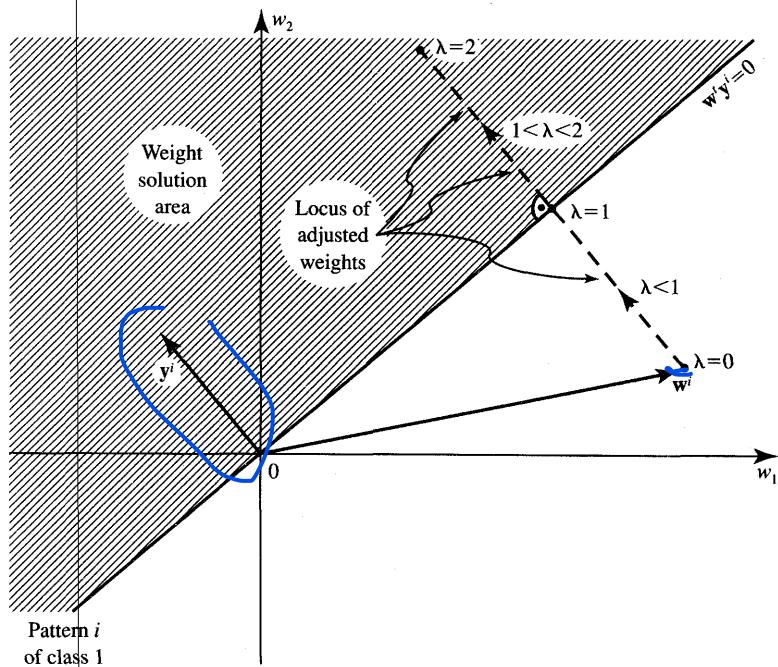


Figure 3.13 Illustration of correction increment value, i 'th step.

to a coefficient $\lambda > 0$, where

$$c = \lambda \frac{|\mathbf{w}^{1T}\mathbf{y}|}{\mathbf{y}^T\mathbf{y}} \quad (3.31)$$

It follows from the preceding discussion of geometrical terms that λ is the ratio of the distance between the old weight vector \mathbf{w}^1 and the new weight vector \mathbf{w}^2 , to the distance from \mathbf{w}^1 to the pattern hyperplane in the weight space.

The different choices for correction increment values c are depicted in Figure 3.13. No weight adjustment in the i 'th step takes place for $\lambda = 0$. For $\lambda = 1$ the corrected weights displace exactly onto the decision plane $\mathbf{w}^T\mathbf{y}^i = 0$ as stated before in case (3.30). For $\lambda = 2$, the corrected weights are reflected symmetrically with respect to the decision plane. In the two intermediate cases we have

$0 < \lambda < 1$ fractional correction rule

$1 < \lambda < 2$ absolute correction rule

The reader may notice that the fractional correction rule, although it moves \mathbf{w}^1 in the right direction, results in another misclassification during the repeated demonstration of the same training pattern, unless the weights have moved to the

appropriate side of its decision plane due to the other corrections that might have taken place in the meantime. For the *fixed correction rule* introduced originally with $c = \text{const}$, correction of weights is always the same fixed portion of the current training vector \mathbf{y} . Let us also note that the training algorithm based on the computation of distance $\|\mathbf{y}\|$ requires that the initial weight vector be different from $\mathbf{0}$. This is needed to make the denominator of (3.31) of nonzero value.

EXAMPLE 3.3

Let us look at a simple, but nonetheless, instructive example of nonparametric training of a discrete perceptron. The trained classifier should provide the following classification of four patterns x with known class membership d

$$\mathbf{y} = \begin{bmatrix} x \\ d \end{bmatrix}$$

$$x_1 = 1, x_3 = 3, d_1 = d_3 = 1 : \text{class 1}$$

$$x_2 = -0.5, x_4 = -2, d_2 = d_4 = -1 : \text{class 2}$$

The single discrete perceptron with unknown weights w_1 and w_2 as shown in Figure 3.14(a) needs to be trained using the augmented input vectors as below

$$\mathbf{y}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{y}_2 = \begin{bmatrix} -0.5 \\ 1 \end{bmatrix}, \mathbf{y}_3 = \begin{bmatrix} 3 \\ 1 \end{bmatrix}, \mathbf{y}_4 = \begin{bmatrix} -2 \\ 1 \end{bmatrix}$$

Decision lines $\mathbf{w}'\mathbf{y}_i = 0$, for $i = 1, 2, 3, 4$, have been sketched in the augmented weight space as illustrated in Figure 3.14(b), along with their corresponding normal vectors, which are simply patterns \mathbf{y}_i , for $i = 1, 2, 3, 4$. The shaded area in the weight plane represents the anticipated solution region for the given supervised training task. The shaded area of weight solutions is known to us based on the similar considerations as in Figure 3.11. The classifier weights, however, must yet be trained to relocate possibly to the shaded area in this example.

Let us review the training with an arbitrary selection of $c = 1$, and with the initial weights chosen arbitrarily as $\mathbf{w}^1 = [-2.5 \quad 1.75]^T$. Using (3.26), the weight training with each step 1 through 10 and illustrated in Figure 3.14(b) can be summarized as follows

$$\Delta \mathbf{w}^k = \frac{c}{2} [d_k - \text{sgn}(\mathbf{w}^{kt} \mathbf{y}_k)] \mathbf{y}^k$$

During the training we obtain the following outputs and weight updates.

Step 1 Pattern \mathbf{y}_1 is input

$$o_1 = \text{sgn} \left([-2.5 \quad 1.75] \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) = -1$$

$$d_1 - o_1 = 2$$

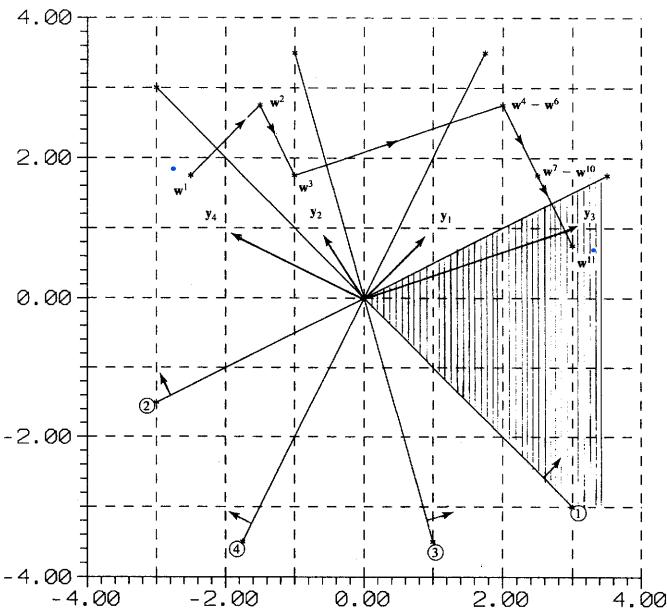
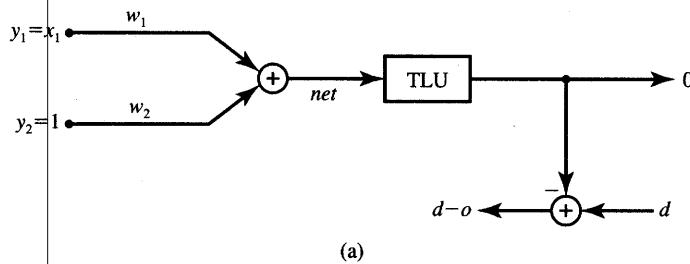


Figure 3.14a,b Discrete perceptron classifier training in Example 3.3: (a) network diagram, (b) fixed correction rule training.

Pattern vector $\mathbf{y}^1 = \mathbf{y}_1$ is added to the present weight vector \mathbf{w}^1

$$\mathbf{w}^2 = \mathbf{w}^1 + \mathbf{y}^1 = \begin{bmatrix} -1.5 \\ 2.75 \end{bmatrix}$$

Step 2 Pattern \mathbf{y}_2 is input

$$o_2 = \text{sgn} \left(\begin{bmatrix} -1.5 & 2.75 \end{bmatrix} \begin{bmatrix} -0.5 \\ 1 \end{bmatrix} \right) = 1$$

$$d_2 - o_2 = -2$$

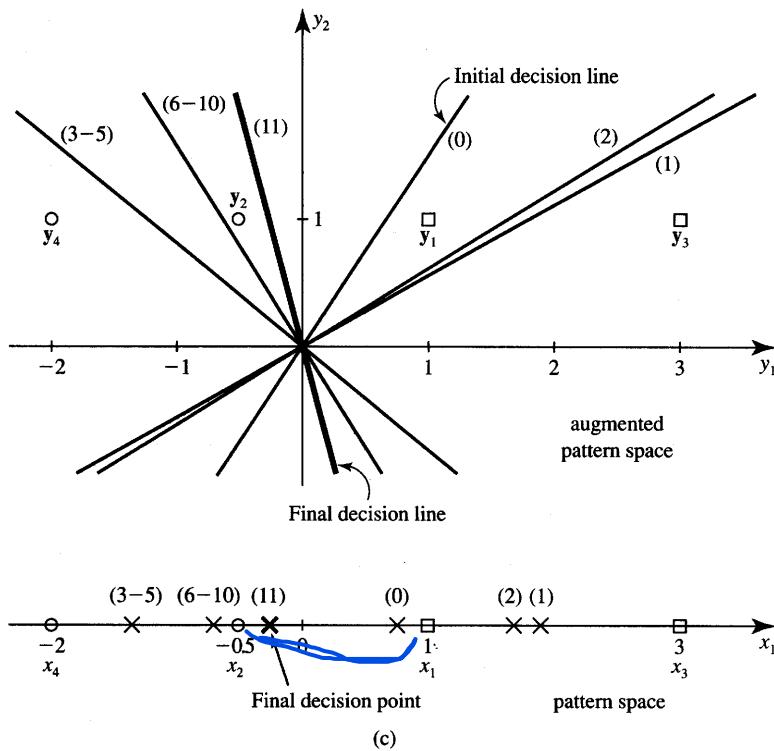


Figure 3.14c Discrete perceptron classifier training in Example 3.3 (continued): (c) decision surfaces.

Pattern vector $\mathbf{y}^2 = \mathbf{y}_2$ is subtracted from the present weight vector \mathbf{w}^2

$$\mathbf{w}^3 = \mathbf{w}^2 - \mathbf{y}^2 = \begin{bmatrix} -1 \\ 1.75 \end{bmatrix}$$

Step 3 Pattern \mathbf{y}_3 is input

$$o_3 = \text{sgn} \left(\begin{bmatrix} -1 & 1.75 \end{bmatrix} \begin{bmatrix} 3 \\ 1 \end{bmatrix} \right) = -1$$

$$d_3 - o_3 = 2$$

Pattern vector $\mathbf{y}^3 = \mathbf{y}_3$ is added to the present weight vector \mathbf{w}^3

$$\mathbf{w}^4 = \mathbf{w}^3 + \mathbf{y}^3 = \begin{bmatrix} 2 \\ 2.75 \end{bmatrix}$$

Since we have no evidence of correct classification for weights \mathbf{w}^4 , the training set consisting of an ordered sequence of patterns \mathbf{y}_1 , \mathbf{y}_2 , and \mathbf{y}_3 needs to be recycled. We thus have $\mathbf{y}^4 = \mathbf{y}_1$, $\mathbf{y}^5 = \mathbf{y}_2$, etc. The superscript is used to denote the following training step number.

Steps 4, 5 $w^6 = w^5 = w^4$. (No misclassification, thus no weight adjustment.)

The reader can easily verify that the adjustments following in Steps 6 through 10 are as follows:

$$\begin{aligned}w^7 &= [2.5 \quad 1.75]^t \\w^{10} &= w^9 = w^8 = w^7 \\w^{11} &= [3 \quad 0.75]^t\end{aligned}$$

As indicated, only Steps 6 and 10 have resulted in weight adjustment during continued training with recycled inputs. Since after ten training steps the weight vector w^{11} ended up in the solution area, $w_1 = 3$, and $w_2 = 0.75$ represent the final weights of the classifier from Figure 3.14(a) and the training terminates. The reader can verify that w^{11} represents the solution weight vector and provides correct classification for the entire training set.

The training has been so far performed in the augmented weight space as shown in Figure 3.14(b). It should be realized, however, that the original decision surfaces used by the TLU element are generated in the pattern space. Figure 3.14(c) shows the decision surfaces produced for the example training for each of Steps 1 through 10. In the augmented pattern space y_1, y_2 , the decision surfaces are the straight lines shown; in the pattern space, the decision surfaces reduce to points on the x_1 axis.

We now simulate a fixed-correction training rule for a case of higher dimension. An example with $n = 3$ and $R = 2$ is shown in Figure 3.15. As illustrated in the first eight rows of Figure 3.15(a), eight binary patterns spanned in three-dimensional space and representing one of the two classes are submitted to the input of the classifier in an ascending binary sequence. As listed in the rightmost column, four patterns located on the plane $x_3 = 0$, which is the bottom of the unity side cube in E^3 space, belong to class 2; the remaining four patterns belong to class 1. In the calculations shown, we also assume that the perceptron can respond with zero. Such "no decisive response" is interpreted as a mistake and is followed by a correction.

Accordingly, starting from the initial weight vector $w = 0$ as shown in the first row of iteration 1, the first corrected weight vector in the second row becomes the difference of the current weight vector $[0 \quad 0 \quad 0 \quad 0]^t$ and the misclassified pattern vector $[0 \quad 0 \quad 0 \quad 1]^t$. The *net* values at the input of the TLU expressing scalar products $w'y$ are shown in the rightmost column of the figure. As shown, the training takes only three full sweeps through the set of prototypes, since the patterns are apparently easily separable. The resulting decision plane shown in 3.15(c) is $4x_3 - 1 = 0$. The equation has been obtained by inspection of the last row of the iteration 3 of Figure 3.15(a).

Input data and classes					
	x_1	x_2	x_3	x_4	d
	0	0	0	1	2
	0	0	1	1	1
	0	1	0	1	2
	0	1	1	1	1
	1	0	0	1	2
	1	0	1	1	1
	1	1	0	1	2
	1	1	1	1	1
Training Steps		w_1	w_2	w_3	w_4
Iteration = 1					$net = \mathbf{w}^T \mathbf{y}$
	0	0	0	1	0.00
	0	0	1	1	0.00
	0	1	0	1	0.00
	0	1	1	1	0.00
	1	0	0	1	0.00
	1	0	1	1	-1.00
	1	1	0	1	0.00
	1	1	1	1	-1.00
Iteration = 2					
	0	0	0	1	0.00
	0	0	1	1	0.00
	0	1	0	1	0.00
	0	1	1	1	0.00
	1	0	0	1	0.00
	1	0	1	1	-1.00
	1	1	0	1	0.00
	1	1	1	1	-1.00
Iteration = 3					
	0	0	0	1	0.00
	0	0	1	1	0.00
	0	1	0	1	0.00
	0	1	1	1	0.00
	1	0	0	1	0.00
	1	0	1	1	0.00
	1	1	0	1	0.00
	1	1	1	1	0.00

The training took 2 iterations.

The final weights are: 0.00 0.00 4.00 -1.00.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \text{ is pattern vector.}$$

$x_4 = 1$ is augmented pattern component, d is desired class membership.

(a)

Figure 3.15a Example dichotomizer training, $n = 3$: (a) fixed correction rule ($c = 1$).

Input		data and classes				
x_1	x_2	x_3	x_4	d		
0	0	0	1	2		
0	0	1	1	1		
0	1	0	1	2		
0	1	1	1	1		
1	0	0	1	2		
1	0	1	1	1		
1	1	0	1	2		
1	1	1	1	1		
Training Steps		w_1	w_2	w_3	w_4	$net = w^t y$
Iteration = 1						
0	0	0	1	0.00	0.00	0.00
0	0	1	1	0.00	0.00	-1.50
0	1	0	1	0.00	0.00	-0.38
0	1	1	1	0.00	0.00	0.75
1	0	0	1	0.00	0.00	-0.38
1	0	1	1	0.00	0.00	0.75
1	1	0	1	0.00	0.00	-0.38
1	1	1	1	0.00	0.00	0.75
Iteration = 2						
0	0	0	1	0.00	0.00	-0.38
0	0	1	1	0.00	0.00	0.75
0	1	0	1	0.00	0.00	-0.38
0	1	1	1	0.00	0.00	0.75
1	0	0	1	0.00	0.00	-0.38
1	0	1	1	0.00	0.00	0.75
1	1	0	1	0.00	0.00	-0.38
1	1	1	1	0.00	0.00	0.75

The training took 1 iteration.

The final weights are: 0.00 0.00 1.13 -0.38.

$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ is pattern vector.

$x_4 = 1$ is augmented pattern component, d is desired class membership.

(b)

Figure 3.15b Example dichotomizer training, $n = 3$ (continued): (b) absolute correction rule.

Similar results, but produced within only two sweeps through the training set, have been obtained for the absolute correction rule. The resulting plane is $1.13x_3 - 0.38 = 0$ for the weight computation using this rule. The simulated training is shown in Figure 3.15(b), and the resulting decision plane is shown in Figure 3.15(c).

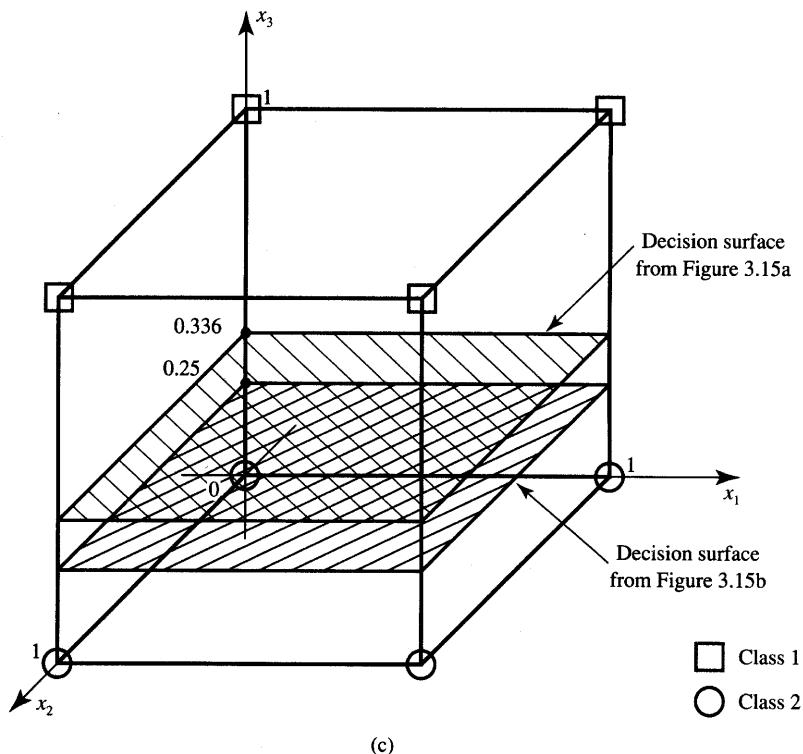


Figure 3.15c Example dichotomizer training, $n = 3$ (continued): (c) decision planes for parts (a) and (b).

Let us summarize the classification of linearly separable patterns belonging to two classes only. The training task for the classifier was to find the weight vector \mathbf{w} such that

$$\begin{aligned} \mathbf{w}'\mathbf{y} &> 0 & \text{for each } \mathbf{x} \text{ of } \mathcal{X}_1 \\ \mathbf{w}'\mathbf{y} &< 0 & \text{for each } \mathbf{x} \text{ of } \mathcal{X}_2 \end{aligned} \quad (3.32)$$

Completion of the training with the fixed correction training rule for any initial weight vector and any correction increment constant leads to the following weights

$$\mathbf{w}^* = \mathbf{w}^{k_o} = \mathbf{w}^{k_o+1} = \mathbf{w}^{k_o+2} \dots \quad (3.33)$$

with \mathbf{w}^* as a solution vector for (3.32). The integer k_o is the training step number starting at which no more misclassification occurs, and thus no weight adjustments take place for $k_o \geq 0$. This theorem, called the *perceptron convergence theorem*, states that a classifier for two linearly separable classes of patterns is always trainable in a finite number of training steps. Below we provide the proof of the theorem.

Assume that the solution weight vector \mathbf{w}^* exists, and is normalized such that $\|\mathbf{w}^*\| = 1$. The existence of hypothetical solution weights \mathbf{w}^* allows rewriting of (3.32) for a small arbitrarily selected constant $0 < \delta < 1$ as follows

$$\begin{aligned}\mathbf{w}^{*t}\mathbf{y} &> \delta > 0 \quad \text{for each } \mathbf{x} \in \mathcal{X}_1 \\ \mathbf{w}^{*t}\mathbf{y} &< -\delta < 0 \quad \text{for each } \mathbf{x} \in \mathcal{X}_2\end{aligned}\quad (3.34)$$

Let us evaluate the following function:

$$\phi(\mathbf{w}) \triangleq \frac{\mathbf{w}^{*t}\mathbf{w}}{\|\mathbf{w}\|} \quad (3.35)$$

Since $\phi(\mathbf{w})$ is a scalar product of normalized vectors, $\phi(\mathbf{w}) = \cos \measuredangle(\mathbf{w}^*, \mathbf{w}) \leq 1$. Rearranging the numerator and denominator of (3.35) we obtain for the k 'th learning step, respectively,

$$\mathbf{w}^{*t}\mathbf{w}^{k+1} = \mathbf{w}^{*t}\mathbf{w}^k + \mathbf{w}^{*t}\mathbf{y} > \mathbf{w}^{*t}\mathbf{w}^k + \delta \quad (3.36a)$$

$$\|\mathbf{w}^{k+1}\|^2 = (\mathbf{w}^{kt} + \mathbf{y}^t)(\mathbf{w}^k + \mathbf{y}) < \|\mathbf{w}^k\|^2 + 1 \quad (3.36b)$$

for the normalized pattern used for training and for $c = 1$.

Upon performing the total of k_o training steps (3.36a) and (3.36b) can be rewritten, respectively, as

$$\mathbf{w}^{*t}\mathbf{w}^{k_o+1} > k_o \delta \quad (3.36c)$$

This condition can be rewritten as follows:

$$\|\mathbf{w}^{k_o+1}\|^2 < k_o \quad (3.36d)$$

The function (3.35) now becomes for $\mathbf{w} = \mathbf{w}^{k_o+1}$

$$\phi(\mathbf{w}^{k_o+1}) = \frac{\mathbf{w}^{*t}\mathbf{w}^{k_o+1}}{\|\mathbf{w}^{k_o+1}\|} \quad (3.37a)$$

and from (3.36c) and (3.36d) note that

$$\phi(\mathbf{w}^{k_o+1}) > \sqrt{k_o} \delta \quad (3.37b)$$

Since $\phi(\mathbf{w}^{k_o+1})$ has an upper bound of 1, the inequality (3.37b) would be violated if a solution were not found for $\sqrt{k_o} \delta < 1$. Thus, this concludes the proof that the discrete perceptron training is convergent. The maximum number of learning steps is $k_o = 1/\delta^2$ (Amit 1989). If the value $1/\delta^2$ is not an integer, it should be rounded up to the closest integer.

The number of steps k_o needed to achieve correct classification depends strongly on the c value and on the sequence of training patterns submitted. The theorem can also be extended and proven for R category machines classifying patterns that are linearly separable and trained through submission of such patterns in any sequence (Nilsson 1965).

The perceptron training procedure expressed in (3.26) may be rewritten as

$$\mathbf{w}^{k+1} = \mathbf{w}^k + \frac{c}{2}(d^k - o^k)\mathbf{y}^k \quad (3.38)$$

where k denotes the number of the current training step, o^k is the actual output of the classifier being trained, and d^k is the teacher-submitted desired output for the vector \mathbf{y}^k applied at the input. When $d^k = o^k$, no adjustment of weights is undertaken. When $d^k = 1$ and $o^k = -1$, signifying jointly that the pattern of class 1 was mistakenly not detected, the weight correction becomes

$$\mathbf{w}^{k+1} - \mathbf{w}^k = c\mathbf{y}^k \quad (3.39a)$$

Whenever a pattern of class 2 is not detected, and the classification is resulting in $o^k = 1$ while $d^k = -1$, the weight correction is

$$\mathbf{w}^{k+1} - \mathbf{w}^k = -c\mathbf{y}^k \quad (3.39b)$$

In summary, the training of a single discrete perceptron two-class classifier requires a change of \mathbf{w} if and only if a misclassification occurs. If the reason for misclassification is $\mathbf{w}'\mathbf{y} < 0$, then all weights w_i are increased in proportion to y_i ; if $\mathbf{w}'\mathbf{y} > 0$, then all weights w_i are decreased in proportion to y_i . The weights are then punished in an appropriate way. If the classification is correct, no punishment takes place, but its absence can be taken as a reward. We can formulate the single discrete perceptron training algorithm as given below.

The error value E used in the algorithm is computed to detect the series of correct classifications within the training cycle and to terminate the training. Left-pointing arrows below indicate the operation of an assignment. By the operation of assignment, the argument on the left side is assigned the value indicated on the right side.

■ Summary of the Single Discrete Perceptron Training Algorithm (SDPTA)

Given are P training pairs

$$\{\mathbf{x}_1, d_1, \mathbf{x}_2, d_2, \dots, \mathbf{x}_P, d_P\}, \text{ where } \\ \mathbf{x}_i \text{ is } (n \times 1), d_i \text{ is } (1 \times 1), i = 1, 2, \dots, P.$$

Note that augmented input vectors are used:

$$\mathbf{y}_i = \begin{bmatrix} \mathbf{x}_i \\ 1 \end{bmatrix}, \quad \text{for } i = 1, 2, \dots, P$$

In the following, k denotes the training step and p denotes the step counter within the training cycle.

Step 1: $c > 0$ is chosen.

Step 2: Weights are initialized at \mathbf{w} at small random values, \mathbf{w} is $(n + 1) \times 1$. Counters and error are initialized:

$$k \leftarrow 1, p \leftarrow 1, E \leftarrow 0$$

Step 3: The training cycle begins here. Input is presented and output computed:

$$\mathbf{y} \leftarrow \mathbf{y}_p, d \leftarrow d_p \\ o \leftarrow \text{sgn}(\mathbf{w}'\mathbf{y})$$

Step 4: Weights are updated:

$$\mathbf{w} \leftarrow \mathbf{w} + \frac{1}{2}c(d - o)\mathbf{y}$$

Step 5: Cycle error is computed:

$$E \leftarrow \frac{1}{2}(d - o)^2 + E$$

Step 6: If $p < P$ then $p \leftarrow p + 1$, $k \leftarrow k + 1$, and go to Step 3; otherwise, go to Step 7.

Step 7: The training cycle is completed. For $E = 0$, terminate the training session. Output weights and k .

If $E > 0$, then $E \leftarrow 0$, $p \leftarrow 1$, and enter the new training cycle by going to Step 3.

The training rule expressed by the Equations (3.26) and (3.38) will be extended in the next section to cases where the TLU or the threshold device is no longer the most favored decision computing element. Instead, a continuous perceptron element with sigmoidal activation function will be introduced to facilitate the training of multilayer feedforward networks used for classification and recognition.

3.6

SINGLE-LAYER CONTINUOUS PERCEPTRON NETWORKS FOR LINEARLY SEPARABLE CLASSIFICATIONS

In this section we introduce the concept of an error function in multidimensional weight space. Also, the TLU element with weights will be replaced by the continuous perceptron. This replacement has two main direct objectives. The first one is to gain finer control over the training procedure. The other is to facilitate working with differentiable characteristics of the threshold element, thus enabling computation of the error gradient. Such a continuous characteristic, or sigmoidal activation function, describes the neuron in lieu of the sgn function as discussed in Chapter 2.

According to the training theorem discussed in the last section, the TLU weights are converging to some solution \mathbf{w}^{k_0} for any positive correction increment constant. The weight modification problem could be better solved, however, by minimizing the scalar criterion function. Such a task can possibly be attempted by again using the gradient, or steepest descent, procedure.

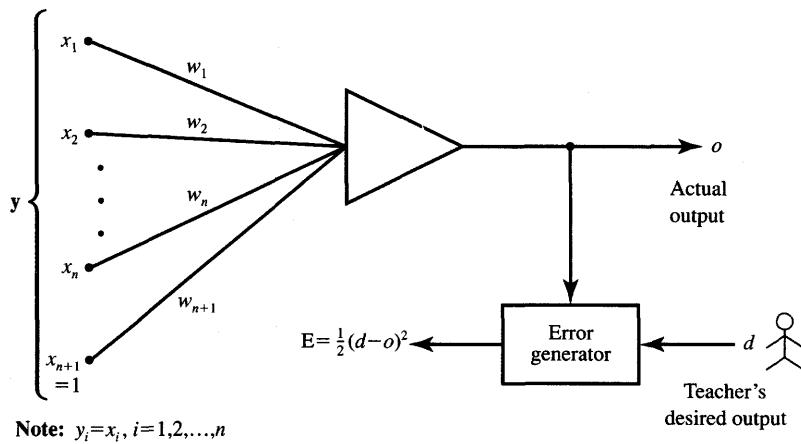


Figure 3.16 Continuous perceptron training using square error minimization.

The basic procedure of descent is quite simple. Starting from an arbitrary chosen weight vector \mathbf{w} , the gradient $\nabla E(\mathbf{w})$ of the current error function is computed. The next value of \mathbf{w} is obtained by moving in the direction of the negative gradient along the multidimensional error surface. The direction of negative gradient is the one of steepest descent. The algorithm can be summarized as below:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \eta \nabla E(\mathbf{w}^k) \quad (3.40)$$

where η is the positive constant called the *learning constant* and the superscript denotes the step number.

Let us define the error E_k in the k 'th training step as the squared difference between the desired value d^k at the output of the continuous perceptron and its actual output value o^k computed. As shown in Figure 3.16, the desired value d^k is provided by the teacher.

The expression for classification error to be minimized is

$$E_k = \frac{1}{2}(d^k - o^k)^2, \text{ or} \quad (3.41a)$$

$$E_k = \frac{1}{2} [d^k - f(\mathbf{w}^k \mathbf{y}^k)]^2 \quad (3.41b)$$

where the coefficient $1/2$ in front of the error expression is intended for convenience in simplifying the expression of the gradient value, and it does not affect the location of the error minimum or the error minimization itself.

Our intention is to achieve minimization of the error function $E(\mathbf{w})$ in $(n+1)$ -dimensional weight space. An example of a well-behaving error function is shown in Figure 3.17. The error function has a single minimum at $\mathbf{w} = \mathbf{w}_f$, which can be achieved using the negative-gradient descent starting at the initial weight

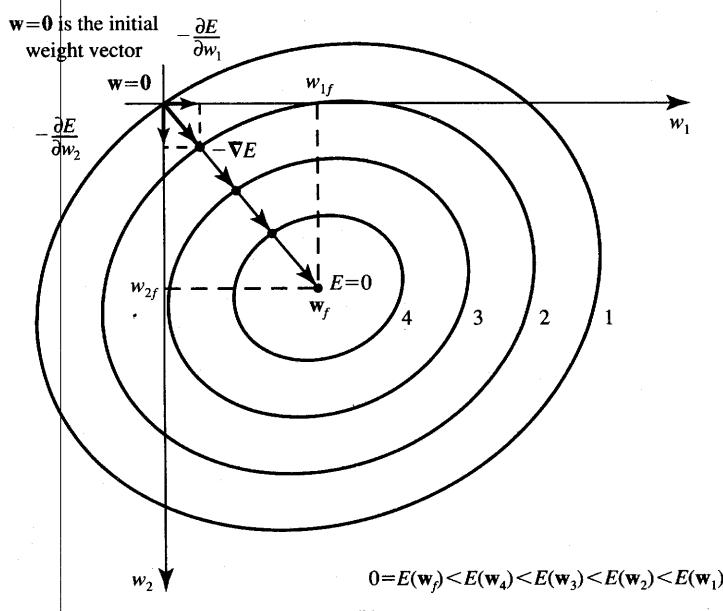
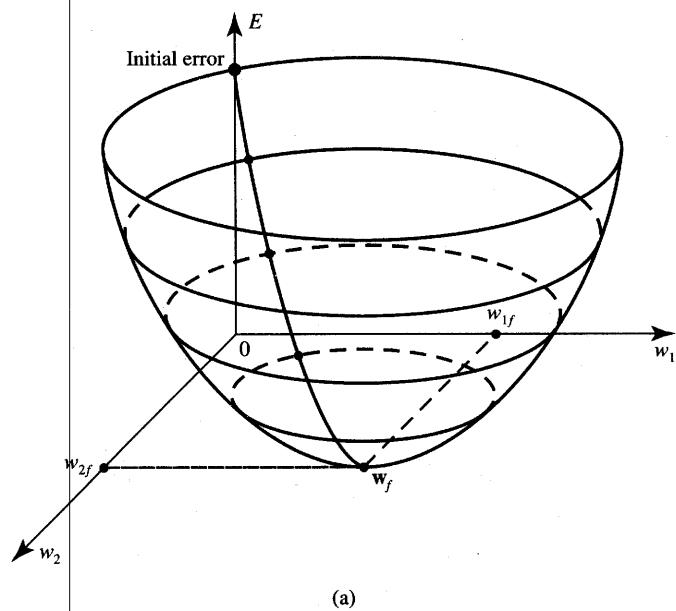


Figure 3.17 Minimization of the error function: (a) error surface ($n = 1$) and (b) error contours for determination of the gradient vector ∇E .

vector $\mathbf{0}$ shown in the figure. The vicinity of the point $E(\mathbf{w}_f) = 0$ is shown to be reached within a finite number of steps. The movement of the weight vector can be observed both on the error surface in Figure 3.17(a), or across the error contour lines shown in Figure 3.17(b). In the depicted case, the weights displace from contour 1 through 4 ideally toward the point $E = 0$. By definition of the steepest descent concept, each elementary move should be perpendicular to the current error contour.

Unfortunately, neither error functions for a TLU-based dichotomizer nor those for more complex multiclass classifiers using TLU units result in a form suitable for the gradient descent-based training. The reason for this is that the error function has zero slope in the entire weight space where the error function exists, and it is nondifferentiable on the decision surface itself. This property directly results from calculating any gradient vector component $\nabla E(\mathbf{w})$ using (3.41) with $\text{sgn}(\text{net})$ replacing $f(\text{net})$. Indeed, the derivative of the internal function $\text{sgn}(\cdot)$ is nonexistent in zero, and of zero value elsewhere.

The error minimization algorithm (3.40) requires computation of the gradient of the error (3.41) as follows:

$$\nabla E(\mathbf{w}) = \frac{1}{2} \nabla [d - f(\text{net})]^2 \quad (3.42)$$

The training step superscript k in (3.42) has been temporarily skipped for simplicity, but it should be understood that the error gradient computation refers strictly to the k 'th training step. In the remaining part of this section, the gradient descent training rule for the single continuous perceptron will be derived in detail.

The $n + 1$ -dimensional gradient vector (3.43) is defined as follows:

$$\nabla E(\mathbf{w}) \triangleq \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_{n+1}} \end{bmatrix} \quad (3.43)$$

Using (3.42) we obtain for the gradient vector

$$\nabla E(\mathbf{w}) = -(d - o)f'(\text{net}) \begin{bmatrix} \frac{\partial(\text{net})}{\partial w_1} \\ \frac{\partial(\text{net})}{\partial w_2} \\ \vdots \\ \frac{\partial(\text{net})}{\partial w_{n+1}} \end{bmatrix} \quad (3.44)$$

Since $net = \mathbf{w}'\mathbf{y}$ we have

$$\frac{\partial(\text{net})}{\partial w_i} = y_i, \quad \text{for } i = 1, 2, \dots, n+1 \quad (3.45)$$

and (3.44) can be rewritten as

$$\nabla E(\mathbf{w}) = -(d - o)f'(\text{net})\mathbf{y} \quad (3.46a)$$

or

$$\frac{\partial E}{\partial w_i} = -(d - o)f'(\text{net})y_i \quad (3.46b)$$

which is the training rule of the continuous perceptron. It can be seen that the rule is equivalent to the delta training rule (2.38). The computation of adjusted weights as in (3.40) requires the assumption of η and the specification for the activation function used.

Note that in further considerations we assume trainable weights. Therefore, we will no longer need to use the steepness coefficient λ of the activation function as a variable. The assumption that $\lambda = 1$ is thus as valid as the assumption of any other constant λ value used to scale all the weights in the same proportion.

Let us express $f'(\text{net})$ in terms of continuous perceptron output. Using the bipolar continuous activation function $f(\text{net})$ of the form

$$f(\text{net}) = \frac{2}{1 + \exp(-\text{net})} - 1 \quad (3.47)$$

we obtain

$$f'(\text{net}) = \frac{2 \exp(-\text{net})}{[1 + \exp(-\text{net})]^2} \quad (3.48)$$

A useful identity (3.49) can be applied here:

$$\frac{2 \exp(-\text{net})}{[1 + \exp(-\text{net})]^2} = \frac{1}{2}(1 - o^2) \quad (3.49)$$

The identity (3.49) is verified below. Letting $o = f(\text{net})$ as in (3.47) on the right side of (3.49) leads to

$$\frac{1}{2}(1 - o^2) = \frac{1}{2} \left[1 - \left(\frac{1 - \exp(-\text{net})}{1 + \exp(-\text{net})} \right)^2 \right] \quad (3.50)$$

The right side of (3.50) reduces after rearrangements to

$$\frac{1}{2} \left[1 - \left(\frac{1 - \exp(-\text{net})}{1 + \exp(-\text{net})} \right)^2 \right] = \frac{2 \exp(-\text{net})}{[1 + \exp(-\text{net})]^2} \quad (3.51)$$

which completes the verification of (3.49).

The gradient (3.46a) becomes

$$\nabla E(\mathbf{w}) = -\frac{1}{2}(d - o)(1 - o^2)\mathbf{y} \quad (3.52)$$

and the complete delta training rule for the bipolar continuous activation function results from (3.40) as

$$\mathbf{w}^{k+1} = \mathbf{w}^k + \frac{1}{2} \eta (d^k - o^k)(1 - o^{k2})\mathbf{y}^k \quad (3.53)$$

where k denotes the reinstated number of the training step.

It may be noted that the weight adjustment rule (3.53) corrects the weights in the same direction as the discrete perceptron learning rule originally formulated in (3.38). The size, or simply length, of the weight correction vector is the main difference between the rules of Equations (3.38) and (3.53). Both these rules involve adding or subtracting a fraction of the pattern vector \mathbf{y} . The essential difference is the presence of the moderating factor $1 - o^{k2}$. This scaling factor is obviously always positive and smaller than 1. For erroneous responses and *net* close to 0, or a weakly committed perceptron, the correction scaling factor will be larger than for those responses generated by a *net* of large magnitude.

Another significant difference between the discrete and continuous perceptron training is that the discrete perceptron training algorithm always leads to a solution for linearly separable problems. In contrast to this property, the negative gradient-based training does not guarantee solutions for linearly separable patterns (Wittner and Denker 1988).

At this point the TLU-based classifier has been modified and consists of a continuous perceptron element. In the past, the term perceptron has been used to describe the TLU-based discrete decision element with synaptic weights and a summing node. Here, we have extended the perceptron concept by replacing the TLU decision element with a neuron characterized by a continuous activation function. As in the case of a discrete perceptron, the pattern components arrive through synaptic weight connections yielding the *net* signal. The *net* signal excites the neuron, which responds according to its continuous activation function.

EXAMPLE 3.4

This example revisits the training of Example 3.3, but it discusses a continuous perceptron rather than a discrete perceptron classifier. The training pattern set is identical to that of Example 3.3. The example demonstrates supervised training using the delta rule for a continuous perceptron, which is shown in Figure 3.16. As has been shown, the training rule (3.53) minimizes the error (3.41) in each step by following the negative gradient direction. The error value in step k is

$$E_k = \frac{1}{2} \left\{ d^k - \left[\frac{2}{1 + \exp(-\lambda net^k)} - 1 \right] \right\}^2$$

where $\text{net}^k = \mathbf{w}^{kt} \mathbf{y}^k$. Note that for fixed patterns $\mathbf{y}^k = \mathbf{y}_k$, for $k = 1, 2, 3, 4$, the error can be expressed as a function of two weights only. Thus, we can easily track the error minimization procedure during training in the two-dimensional weight space. The error function in the first training step can be obtained using numerical values of Example 3.3 plugged into the error expression above. For the first pattern being input we have

$$E_1(\mathbf{w}) = \frac{1}{2} \left\{ 1 - \left[\frac{2}{1 + \exp[-\lambda(w_1 + w_2)]} - 1 \right] \right\}^2$$

Plugging $\lambda = 1$ and reducing the terms simplifies the above expression for error to the form as below

$$E_1(\mathbf{w}) = \frac{2}{[1 + \exp(w_1 + w_2)]^2}$$

for the first training step. Subsequent Steps 2, 3, and 4 need to reduce the training step error values, respectively, as follows:

$$E_2(\mathbf{w}) = \frac{2}{[1 + \exp(0.5w_1 - w_2)]^2}$$

$$E_3(\mathbf{w}) = \frac{2}{[1 + \exp(3w_1 + w_2)]^2}$$

$$E_4(\mathbf{w}) = \frac{2}{[1 + \exp(2w_1 - w_2)]^2}$$

During training, the error E_k in k 'th training step is minimized as a function of weights at $\mathbf{w} = \mathbf{w}^k$. The error surfaces E_1 through E_4 are shown in Figure 3.18. It can be seen from the figure that each training step error is monotonically decreasing and it asymptotically reaches 0 at decision regions yielding correct classification.

Although the error surfaces clearly visualize the anticipated displacement of \mathbf{w}^k toward the lower error values using the steepest descent technique, reconstruction of the trajectories of trained weights for the sequence of steps is somewhat complex. Adding errors E_1 through E_4 provides better overall insight into the error minimization process. The total error surface is shown in Figure 3.19(a). The total error monotonically decreases and reaches values near 0 in the shaded region of Figure 3.14(b). A contour map of the total error is depicted in Figure 3.19(b).

The classifier training has been simulated for $\eta = 0.5$ for four arbitrarily chosen initial weight vectors, including the one taken from Example 3.3. The resulting trajectories of 150 simulated training steps are shown in Figure 3.19(c) (each tenth step is shown). In each case the weights converge during training toward the center of the solution region obtained for the discrete perceptron case from Figure 3.14(b). ■

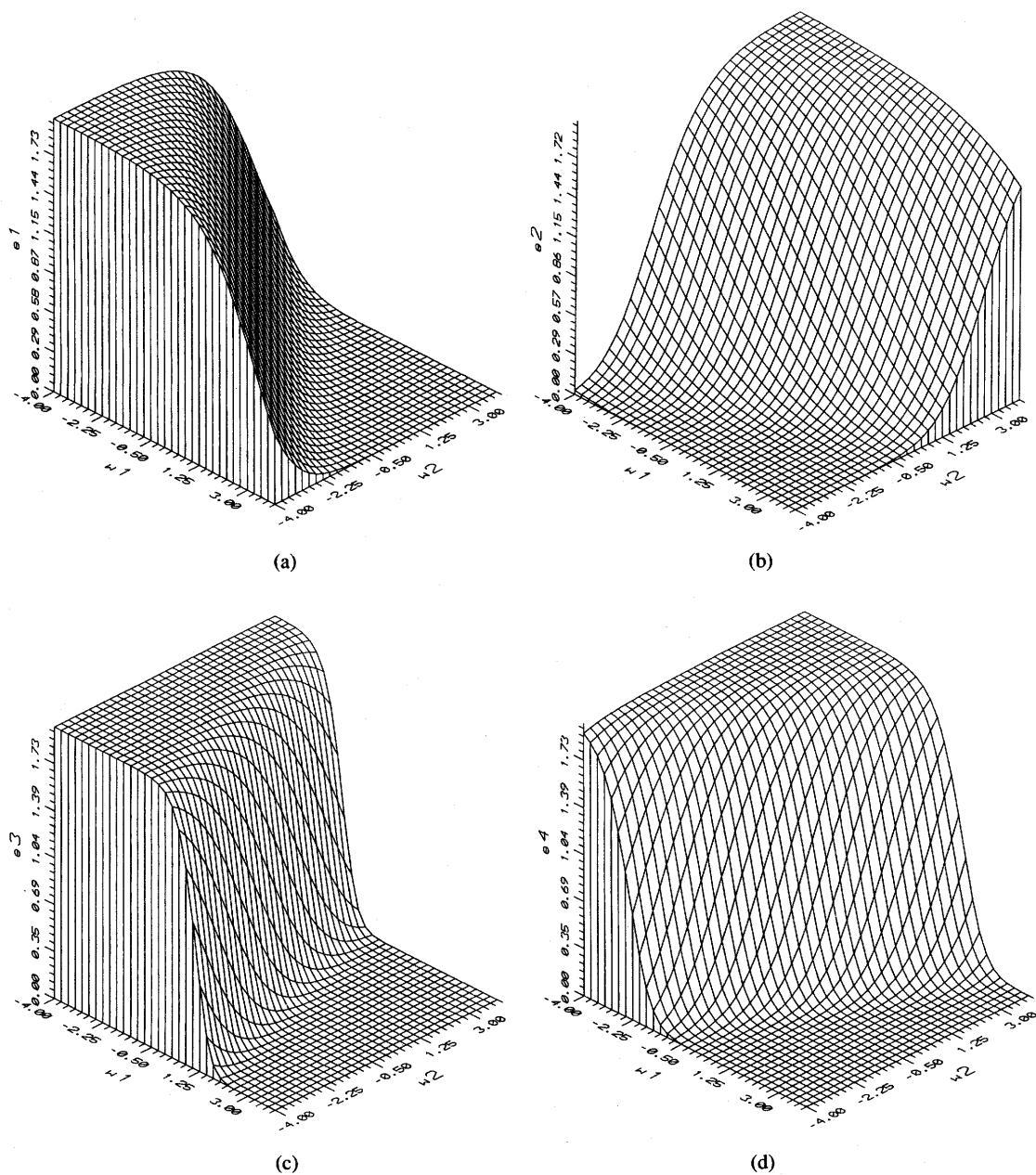
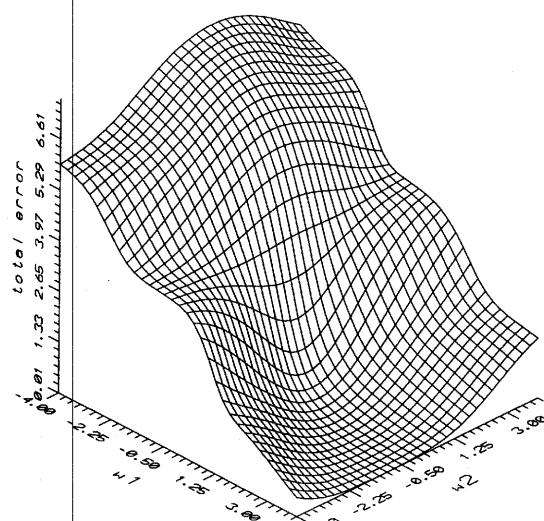
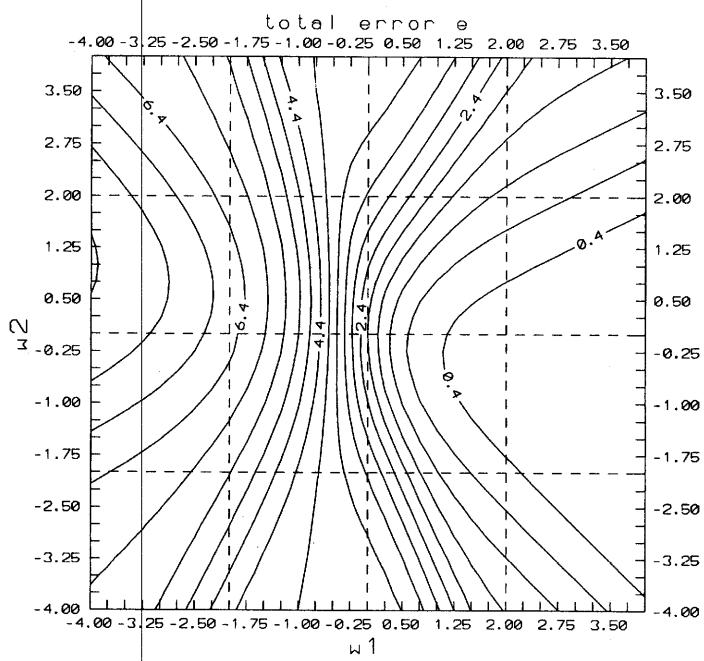


Figure 3.18 Error functions for individual patterns, Example 3.4: (a) first pattern, E_1 , (b) second pattern, E_2 , (c) third pattern, and E_3 , and (d) fourth pattern, E_4 .



(a)



(b)

Figure 3.19a,b Delta rule training illustration for training in Example 3.4: (a) total error surface, (b) total error contour map.

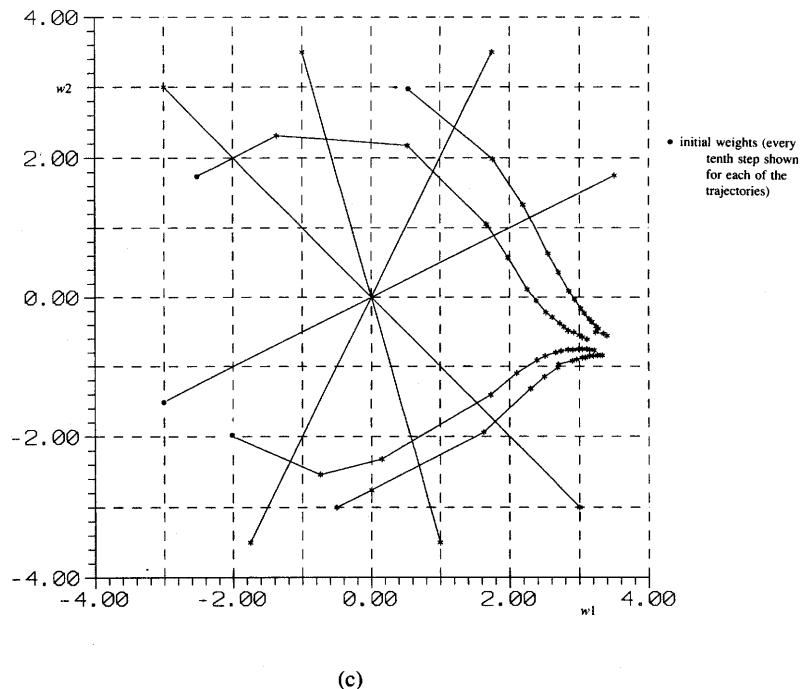


Figure 3.19c Delta rule training illustration for training in Example 3.4 (continued): (c) trajectories of weight adjustments during training (each tenth step shown).

We can formulate the single continuous perceptron training algorithm as given below.

■ *Summary of the Single Continuous Perceptron Training Algorithm (SCPTA)*

Given the same data as for SDPTA in Section 3.5:

Step 1: $\eta > 0$, $\lambda = 1$, $E_{\max} > 0$ chosen.

Step 2: Same as SDPTA in Section 3.5.

Step 3: The training cycle begins here. Input y is presented and output computed:

$$\begin{aligned} \mathbf{y} &\leftarrow \mathbf{y}_p, d \leftarrow d_p \\ o &\leftarrow f(\mathbf{w}' \mathbf{y}), \text{ with } f(\text{net}) \text{ as in (2.3a)} \end{aligned}$$

Step 4: Weights are updated:

$$\mathbf{w} \leftarrow \mathbf{w} + \frac{1}{2} \eta(d - o)(1 - o^2)\mathbf{y}$$

Steps 5,6: Same as SDPTA in Section 3.5

Step 7: The training cycle is completed. For $E < E_{\max}$ terminate the training session. Output weights, k and E .

If $E \geq E_{\max}$, then $E \leftarrow 0$, $p \leftarrow 1$, and enter the new training cycle by going to Step 3.

3.7

MULTICATEGORY SINGLE-LAYER PERCEPTRON NETWORKS

Our approach so far has been limited to training of dichotomizers using both discrete and continuous perceptron elements. In this section we will attempt to apply the error-correcting algorithm to the task of multiclass classification. The assumption needed is that classes are linearly pairwise separable, or that each class is linearly separable from each other class. This assumption is equivalent to the fact that there exist R linear discriminant functions such that

$$g_i(\mathbf{x}) > g_j(\mathbf{x}), \quad \text{for } i, j = 1, 2, \dots, R, i \neq j$$

Let us devise a suitable training procedure for such an R -category classifier. To begin, we need to define the augmented weight vector \mathbf{w}_q as

$$\mathbf{w}_q \triangleq [w_{q1} \quad w_{q2} \quad \dots \quad w_{qn+1}]^t$$

Assume that an augmented pattern \mathbf{y} of class i is presented to the maximum selector-based classifier as in Figure 3.7. The R decision functions $\mathbf{w}_1' \mathbf{y}, \mathbf{w}_2' \mathbf{y}, \dots, \mathbf{w}_R' \mathbf{y}$ are evaluated. If $\mathbf{w}_i' \mathbf{y}$ is larger than any of the remaining $R - 1$ discriminant functions, no adjustment of weight vectors is needed, since the classification is correct. This indicates that

$$\begin{aligned} \mathbf{w}'_1 &= \mathbf{w}_1 \\ \mathbf{w}'_2 &= \mathbf{w}_2 \\ &\vdots \\ \mathbf{w}'_R &= \mathbf{w}_R \end{aligned} \tag{3.54a}$$

The prime is again used to denote the weights after correction. If, however, for some m value we have $\mathbf{w}_i' \mathbf{y} \leq \mathbf{w}_m' \mathbf{y}$, then the updated weight vectors become

$$\begin{aligned} \mathbf{w}'_i &= \mathbf{w}_i + c\mathbf{y} \\ \mathbf{w}'_m &= \mathbf{w}_m - c\mathbf{y} \\ \mathbf{w}'_k &= \mathbf{w}_k, \quad \text{for } k = 1, 2, \dots, R, \quad k \neq i, m \end{aligned} \tag{3.54b}$$

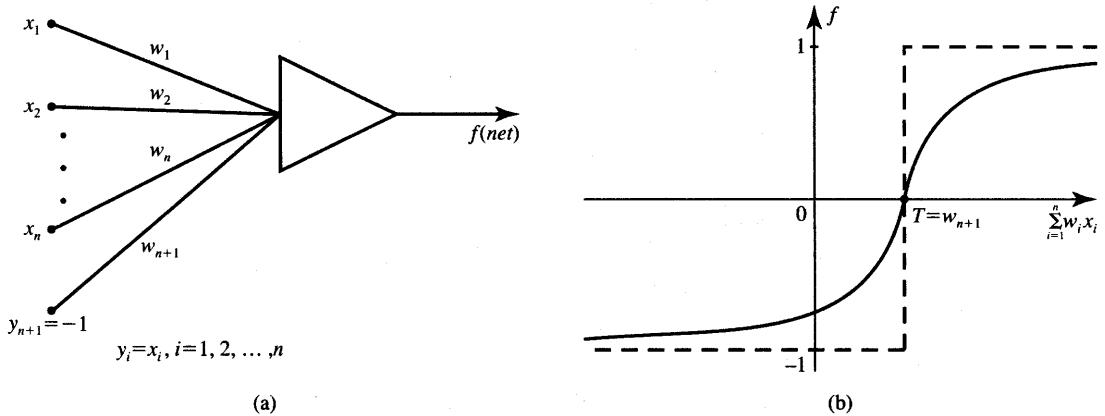


Figure 3.20 The biased neuron: (a) simplified diagram and (b) neuron's response for nonzero threshold.

The matrix formula (3.54b) can be rewritten using the double-subscript notation for the weights used in Figure 3.7 as follows:

$$\begin{aligned} w'_{ij} &= w_{ij} + cy_j \quad \text{for } j = 1, 2, \dots, n+1 \\ w'_{mj} &= w_{mj} - cy_j \quad \text{for } j = 1, 2, \dots, n+1 \\ w'_{kj} &= w_{kj} \quad \text{for } k = 1, 2, \dots, R, \quad k \neq i, m, j = 1, 2, \dots, n+1 \end{aligned} \quad (3.54c)$$

It can be seen from Eq. (3.54c) that the weight value of the connection between the i 'th output and the j 'th component of the input is supplemented with cy_j if the i 'th output is too small. The weight value toward the m 'th output from the j 'th component of the input is reduced by cy_j if the m 'th output is excessive.

So far in this chapter we have been dealing with the augmented pattern vector \mathbf{y} with the $(n+1)$ 'th component always of fixed value +1. It is somewhat instructive to take a different look at this fixed component of the pattern vector to gain better insight into the perceptron's thresholding operation. Figure 3.20(a) shows the perceptron-based dichotomizer in which the augmented pattern component is $y_{n+1} = -1$. Compared with the diagram from Figure 3.16, the weight values w_{n+1} on both figures have to be of opposite values to yield identical decision functions. Since the weights are iteratively chosen during the training process anyway, it is not relevant at all which of the two perceptron diagrams is adopted for the weight training. It becomes thus irrelevant whether y_{n+1} is assumed of value +1 or -1.

Let us assume now that the neuron is excited as in Figure 3.20(a). We can now write that

$$net = \mathbf{w}^T \mathbf{x} - w_{n+1} \quad (3.55)$$

Denoting w_{n+1} as the neuron activation threshold, or bias, value equals T

$$T \stackrel{\Delta}{=} w_{n+1} \quad (3.56)$$

it becomes obvious that the output of the neuron can be expressed as

$$f(\text{net}) = \begin{cases} > 0 & \text{for } \mathbf{w}'\mathbf{x} > T \\ < 0 & \text{for } \mathbf{w}'\mathbf{x} < T \end{cases} \quad (3.57)$$

The activation function expressed now in the form of $f(\mathbf{w}'\mathbf{x})$ with $\mathbf{w}'\mathbf{x}$ as an argument is shown in Figure 3.20(b). The figure shows an activation function of a neuron with the positive threshold value, $T > 0$, and sketched versus nonaugmented activation $\mathbf{w}'\mathbf{x}$. It is instructive to notice that now the $(n + 1)$ 'th weight value is equal to the threshold T , and the neuron is excited if the weighted sum of the original, unaugmented pattern exceeds the threshold value T .

Otherwise, the neuron remains inhibited. The nonzero threshold value causes the neuron to behave as a biased device with T being its bias level. It is important to stress again that from the training viewpoint, any value $y_{n+1} = \text{const}$ is an appropriate choice. When $y_{n+1} = -1$, however, the w_{n+1} value becomes equal to the actual firing threshold of the neuron with input being the original pattern \mathbf{x} . In further considerations we will use $y_{n+1} = -1$ and $w_{n+1} = T$ unless otherwise stated.

So far our discussion has focused on a continuous perceptron performing the dichotomizer-like function. We now show that the continuous perceptron is useful in building any linear classifier. We begin by modifying the minimum distance classifier so that it uses discrete perceptrons and no maximum selector is needed.

Let us now try to eliminate the maximum selector in an R -class linear classifier as in Figure 3.7 and replace it first with R discrete perceptrons. The network generated this way is comprised of R discrete perceptrons as shown in Figure 3.21. The properly trained classifier from Figure 3.7 should respond with $i_o = 1$ when $g_1(\mathbf{x})$ is larger than any of the remaining discriminant functions $g_i(\mathbf{x})$, for $i = 2, 3, \dots, R$, for all patterns belonging to \mathcal{X}_1 . Instead of signaling the event by the single-output maximum selector responding with $i_o = 1$, a TLU #1 as in Figure 3.21 may be used. Then, outputs $o_1 = 1$ and $o_2 = o_3 = \dots = o_R = -1$ should indicate category 1 input.

The adjustment of threshold can be suitably done by altering the value of $w_{1,n+1}$ through adding additional threshold value T_1 . Such an adjustment is clearly feasible since $g_1(\mathbf{x})$ exceeds all remaining discriminant values for all \mathbf{x} in \mathcal{X}_1 . The adjustment is done by changing only the single weight value $w_{1,n+1}$ at the input of the first TLU by T_1 . None of the remaining weights is affected during the T_1 setting step with the input of class 1. Applying a similar procedure to the remaining inputs of the maximum selector from Figure 3.7, the classifier using R individual TLU elements as shown in Figure 3.21 can be obtained. For

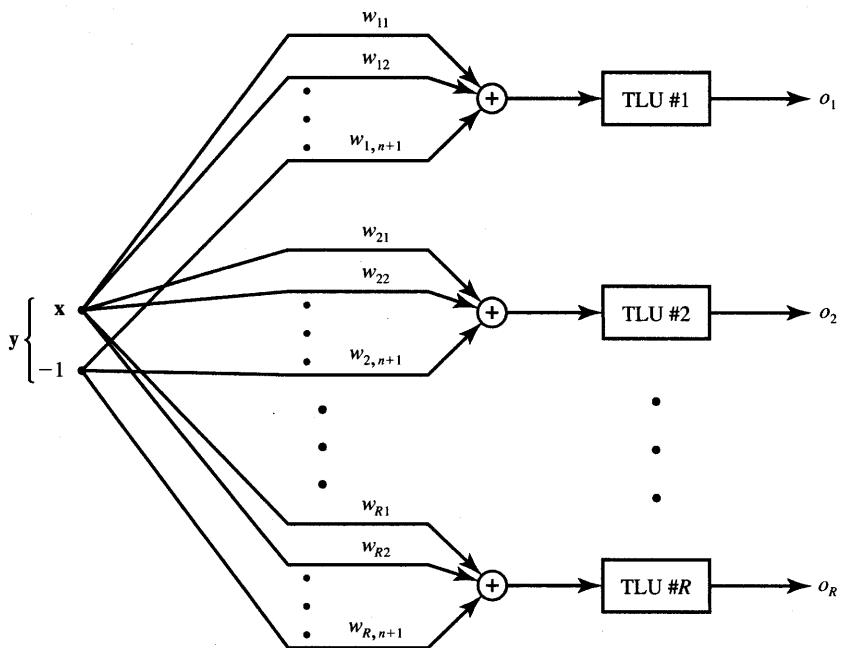


Figure 3.21 *R*-category linear classifier using *R* discrete perceptrons.

this classifier, the k 'th TLU response of +1 is indicative of class k and all other TLUs respond with -1.

The training procedure outlined in (3.54) uses the maximum selector network which will be covered in Chapter 7 in more detail. For our neural network studies however, architectures with the TLU elements are much more important than those with maximum selectors. As shown in the previous paragraphs, the approach can be used to devise the multiclass perceptron-based classifier by replacing the maximum selector through R threshold logic units. Alternatively, direct supervised training of the network of Figure 3.21 can be performed. The weight adjustment during the k 'th step for this network is as follows:

$$\mathbf{w}_i^{k+1} = \mathbf{w}_i^k + \frac{c^k}{2}(d_i^k - o_i^k)\mathbf{y}^k, \quad \text{for } i = 1, 2, \dots, R \quad (3.58)$$

where d_i and o_i are the desired and actual responses, respectively, of the i 'th discrete perceptron. The formula (3.58) expresses the *R*-category discrete perceptron classifier training. For *R*-category classifiers with so-called *local representation*, the desired response for the training pattern of the i 'th category is

$$d_i = 1, \quad d_j = -1, \quad \text{for } j = 1, 2, \dots, R, \quad j \neq i \quad (3.59)$$

For *R*-category classifiers with so-called *distributed representation*, condition

(3.59) is not required, because as more than a single neuron is allowed to respond +1 in this mode.

We can formulate the R -category discrete perceptron training algorithm as given below. Local representation is assumed, thus indicating that R individual TLU elements are used for this R -category classifier.

■ *Summary of the R-Category Discrete Perceptron Training Algorithm (RDPTA)*

Given are P training pairs

$$\{\mathbf{x}_1, \mathbf{d}_1, \mathbf{x}_2, \mathbf{d}_2, \dots, \mathbf{x}_P, \mathbf{d}_P\},$$

where \mathbf{x}_i is $(n \times 1)$, \mathbf{d}_i is $(R \times 1)$, $i = 1, 2, \dots, P$. Note that augmented input vectors are used:

$$\mathbf{y}_i = \begin{bmatrix} \mathbf{x}_i \\ -1 \end{bmatrix}, \quad \text{for } i = 1, 2, \dots, P$$

and k denotes the training step; p denotes the step counter within the training cycle.

Step 1: Same as SDPTA in Section 3.5.

Step 2: Weights are initialized at \mathbf{W} at small random values, $\mathbf{W} = [\mathbf{w}_{ij}]$ is $R \times (n + 1)$.

$$k \leftarrow 1, p \leftarrow 1, E \leftarrow 0$$

Step 3: The training cycle begins here. Input is presented and output computed:

$$\begin{aligned} \mathbf{y} &\leftarrow \mathbf{y}_p, \mathbf{d} \leftarrow \mathbf{d}_p \\ o_i &\leftarrow \text{sgn}(\mathbf{w}_i^T \mathbf{y}), \quad \text{for } i = 1, 2, \dots, R \end{aligned}$$

where \mathbf{w}_i is the i 'th row of \mathbf{W}

Step 4: Weights are updated:

$$\mathbf{w}_i \leftarrow \mathbf{w}_i + \frac{1}{2}c(d_i - o_i)\mathbf{y}, \quad \text{for } i = 1, 2, \dots, R$$

Step 5: Cycle error is computed:

$$E \leftarrow \frac{1}{2}(d_i - o_i)^2 + E, \quad \text{for } i = 1, 2, \dots, R$$

Steps 6, 7: Same as SDPTA in Section 3.5.

The example below demonstrates how a multiclassifier using a maximum selector is converted so that it uses only discrete perceptrons. Also, the multiclassifier training procedure for a multi-perceptron classifier is illustrated.

EXAMPLE 3.5

Let us revisit the three-class classifier design problem of Example 3.2. We will modify the classifier from Figure 3.8(d) so that it uses three discrete perceptrons. This will require changing the threshold values of each unit as described earlier in this section.

It is easy to verify from (3.20a) that the discriminant values $g_1(\mathbf{x})$, $g_2(\mathbf{x})$, and $g_3(\mathbf{x})$ for each of the three prototype patterns are as shown below:

Discriminant	Input		
	Class 1 $[10 \ 2]^t$	Class 2 $[2 \ -5]^t$	Class 3 $[-5 \ 5]^t$
$g_1(\mathbf{x})$	52	-42	-92
$g_2(\mathbf{x})$	-4.5	14.5	-49.5
$g_3(\mathbf{x})$	-65	-60	25

As required by the definition of the discriminant function, the responses on the diagonal are the largest in each column. Inspection of the discriminant functions (3.20b) yields the thresholds $w_{1,3}$, $w_{2,3}$, and $w_{3,3}$ of values 52, 14.5, and 25, respectively. Let us choose additional threshold $T_1 = -2$. This lowers the weight $w_{1,3}$ from 52 to 50. This will ensure that the TLU #1 only will be responding with +1 for class 1 input. Choosing similarly T_2 and T_3 values of -2 allows for TLU #2 and TLU #3 to respond properly. The resulting three-perceptron classifier is shown in Figure 3.22(a).

Instead of using classifiers which are designed based on the minimum-distance classification for known prototypes and deriving the network with three perceptrons from the form of the discriminant functions, our objective in the following part of the example is to train the network of three perceptrons using expression (3.58). Randomly chosen initial weight vectors are

$$\mathbf{w}_1^1 = \begin{bmatrix} 1 \\ -2 \\ 0 \end{bmatrix}, \quad \mathbf{w}_2^1 = \begin{bmatrix} 0 \\ -1 \\ 2 \end{bmatrix}, \quad \mathbf{w}_3^1 = \begin{bmatrix} 1 \\ 3 \\ -1 \end{bmatrix}$$

Assuming that augmented patterns are presented in the sequence \mathbf{y}_1 , \mathbf{y}_2 , \mathbf{y}_3 , \mathbf{y}_1 , \mathbf{y}_2 , ..., and local representation condition (3.59) holds, we obtain (incorrect answers and thus requiring corrections are marked by asterisks)

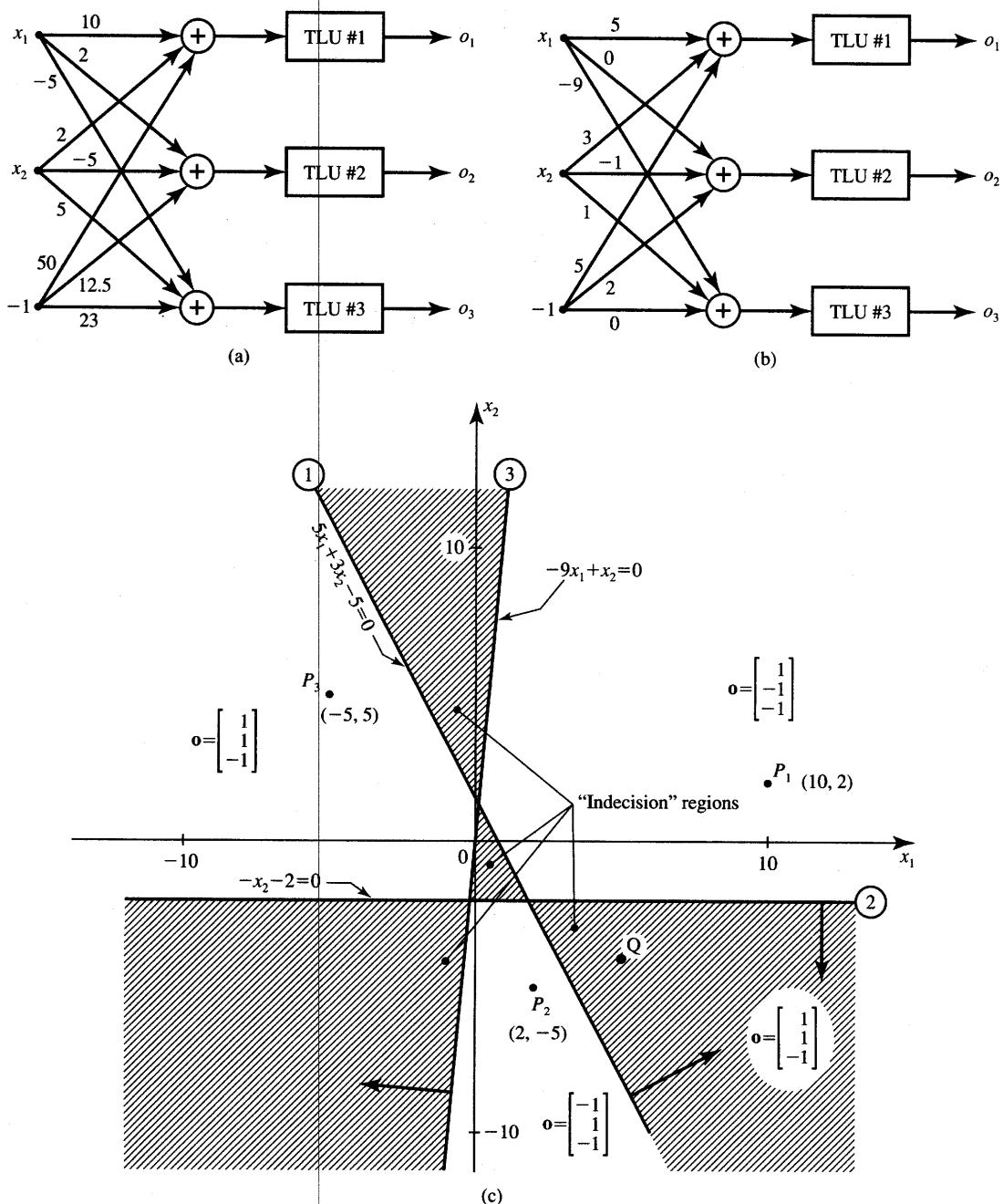


Figure 3.22 Three-class classifier for Example 3.5: (a) three-perceptron classifier from maximum selector, (b) three-perceptron trained classifier, and (c) decision regions for classifier from part (b).

Step 1: y_1 is input:

$$\operatorname{sgn} \left([\begin{array}{ccc} 1 & -2 & 0 \end{array}] \begin{bmatrix} 10 \\ 2 \\ -1 \end{bmatrix} \right) = 1$$

$$\operatorname{sgn} \left([\begin{array}{ccc} 0 & -1 & 2 \end{array}] \begin{bmatrix} 10 \\ 2 \\ -1 \end{bmatrix} \right) = -1$$

$$\operatorname{sgn} \left([\begin{array}{ccc} 1 & 3 & -1 \end{array}] \begin{bmatrix} 10 \\ 2 \\ -1 \end{bmatrix} \right) = 1^*$$

Since the only incorrect response is provided by TLU #3, we have for $c = 1$:

$$\mathbf{w}_1^2 = \mathbf{w}_1^1, \quad \mathbf{w}_2^2 = \mathbf{w}_2^1, \quad \mathbf{w}_3^2 = \begin{bmatrix} 1 \\ 3 \\ -1 \end{bmatrix} - \begin{bmatrix} 10 \\ 2 \\ -1 \end{bmatrix} = \begin{bmatrix} -9 \\ 1 \\ 0 \end{bmatrix}$$

Step 2: y_2 is input:

$$\operatorname{sgn} \left([\begin{array}{ccc} 1 & -2 & 0 \end{array}] \begin{bmatrix} 2 \\ -5 \\ -1 \end{bmatrix} \right) = 1^*$$

$$\operatorname{sgn} \left([\begin{array}{ccc} 0 & -1 & 2 \end{array}] \begin{bmatrix} 2 \\ -5 \\ -1 \end{bmatrix} \right) = 1$$

$$\operatorname{sgn} \left([\begin{array}{ccc} -9 & 1 & 0 \end{array}] \begin{bmatrix} 2 \\ -5 \\ -1 \end{bmatrix} \right) = -1$$

The weight updates are:

$$\mathbf{w}_1^3 = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 \\ -5 \\ -1 \end{bmatrix} = \begin{bmatrix} -1 \\ 3 \\ 1 \end{bmatrix}, \quad \mathbf{w}_2^3 = \mathbf{w}_2^2, \quad \mathbf{w}_3^3 = \mathbf{w}_3^2$$

Step 3: y_3 is input:

$$\operatorname{sgn}(\mathbf{w}_1^{3t} \mathbf{y}_3) = 1^*$$

$$\operatorname{sgn}(\mathbf{w}_2^{3t} \mathbf{y}_3) = -1$$

$$\operatorname{sgn}(\mathbf{w}_3^{3t} \mathbf{y}_3) = 1$$

The weight updates are:

$$\mathbf{w}_1^4 = \begin{bmatrix} 4 \\ -2 \\ 2 \end{bmatrix}, \quad \mathbf{w}_2^4 = \mathbf{w}_2^3, \quad \mathbf{w}_3^4 = \mathbf{w}_3^3$$

This terminates the first learning cycle and the third step of weight adjustment. The reader may verify easily that the only adjusted weights from now on are those of the first perceptron. The outcome of the subsequent training steps is:

$$\mathbf{w}_1^5 = \mathbf{w}_1^4$$

$$\mathbf{w}_1^6 = \begin{bmatrix} 2 \\ 3 \\ 3 \end{bmatrix}$$

$$\mathbf{w}_1^7 = \begin{bmatrix} 7 \\ -2 \\ 4 \end{bmatrix}$$

$$\mathbf{w}_1^8 = \mathbf{w}_1^7$$

$$\mathbf{w}_1^9 = \begin{bmatrix} 5 \\ 3 \\ 5 \end{bmatrix}$$

This outcome terminates the training procedure since none of the last three pattern presentations require the adjustment of weights. The three-perceptron network obtained as a result of the training is shown in Figure 3.22(b). It performs the following classification:

$$o_1 = \text{sgn}(5x_1 + 3x_2 - 5)$$

$$o_2 = \text{sgn}(-x_2 - 2)$$

$$o_3 = \text{sgn}(-9x_1 + x_2)$$

The resulting decision surfaces are shown in Figure 3.22(c). We may notice that the three-perceptron classifier produces, in fact, triple dichotomization of the plane x_1, x_2 . Three decision surfaces produced, which are lines in this case, are

$$5x_1 + 3x_2 - 5 = 0$$

$$-x_2 - 2 = 0$$

$$-9x_1 + x_2 = 0$$

The lines are shown in Figure 3.22(c) along with their corresponding normal vectors directed toward their positive sides. Note that in contrast to the minimum-distance classifier, this method has produced several indecision regions. Indecision regions are regions where no class membership of an input pattern can be uniquely determined based on the response of the classifier. Patterns in shaded areas are not assigned any reasonable classification. One of such patterns may be Q , arbitrarily selected as an example input. The reader can easily verify that the classifier's output for input Q is $\mathbf{o} = [1 \ 1 \ -1]'$. It thus indicates an indecisive response. However, no patterns such as Q have been used for training in the example.

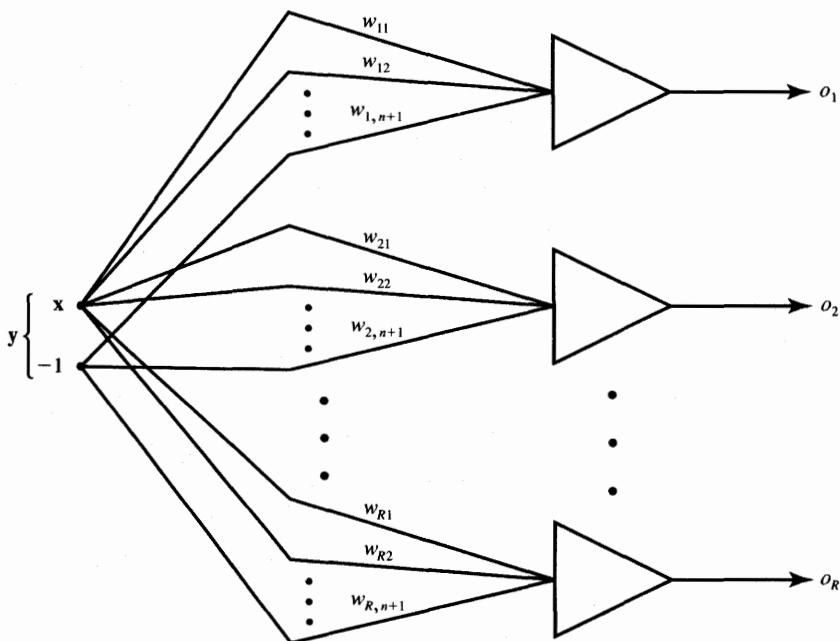


Figure 3.23 R -category linear classifier using continuous perceptrons.

The gradient descent training rule derived for $R = 2$ is also applicable to the single-layer continuous perceptron network as shown in Figure 3.23. The figure displays the R class perceptron classifier derived from the TLU-based classifier in Figure 3.21. We can now produce the generalized gradient descent training rule for a single-layer network of any size. To do this, we combine training rule (3.53) with (3.58) as follows:

$$\mathbf{w}_i^{k+1} = \mathbf{w}_i^k + \frac{1}{2} \eta (d_i^k - o_i^k)(1 - o_i^{k2}) \mathbf{y}^k \quad \text{for } i = 1, 2, \dots, R \quad (3.60)$$

The training rule of Eq. (3.60) is equivalent to individual weight adjustment

$$w_{ij}^{k+1} = w_{ij}^k + \frac{1}{2} \eta (d_i^k - o_i^k)(1 - o_i^{k2}) y_j^k \quad (3.61)$$

for $j = 1, 2, \dots, n+1$, and $i = 1, 2, \dots, R$

Expressions (3.60) and (3.61) exemplify the delta training rule for a bipolar continuous activation function as in (2.3a). Since the formula is of paramount importance for the multilayer perceptron network training, it will be covered in the next chapter in more detail. At this point we will only mention that the delta training rule can be used both for networks with single-layer and for multilayer networks, provided that continuous perceptrons are used.

Let us finish with the general diagram of a supervised learning system as shown in Figure 3.24. The discussed training mode can be represented in block

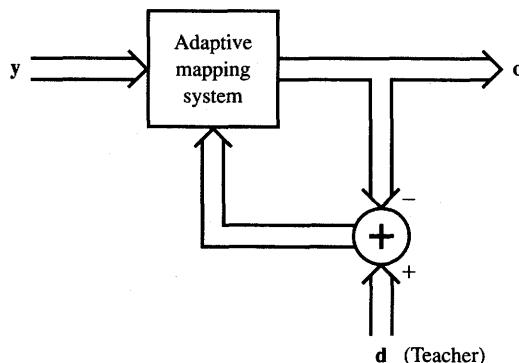


Figure 3.24 Supervised learning adaptive mapping system.

diagram form as illustrated. Both TLU-based and continuous perceptron classifiers discussed in this chapter fall into the diagram of Figure 3.24. The single recall step of the system maps the set in the input space y onto sets in the output space o . The error function $E(d, o)$ is being used as a criterion to accomplish the supervised learning. Although the definition of error function E is not limited to any particular form, it is widely accepted that it should be dependent on the difference vector $d - o$. The error function has been used to modify the internal parameters (called weights) of the adaptive mapping system.

3.8

CONCLUDING REMARKS

The exposition in this chapter has been organized around the assumptions of traditional pattern recognition principles. Although the coverage does not fully take advantage of the potential that is manifested by more complex neural network architectures, it provides a good overall base for the discussion of basic neural network concepts. A list of such concepts includes classification, trainable classifiers, design of dichotomizers and of R category classifiers, linear separability of patterns, perceptron training theorem, negative gradient descent-based training, and others.

In this chapter we have reviewed basic concepts of deterministic pattern classification. Trainable classifiers have been discussed that use gradient descent minimization of the error between the desired and actual response. The gradient descent-based supervised training rules for single-layer neural classifiers have been derived. Both discrete and continuous perceptron single-layer neural classifiers have been discussed and have been designed based on the weight space considerations. The training procedures have been illustrated using the

geometrical visualization in the weight space. It has also been shown that replacing the $\text{sgn}(\cdot)$ function of the TLU element in a discrete perceptron with the continuous activation function modifies the learning by the factor $f'(\text{net})$.

Material covered in this chapter has been largely based on Nilsson (1965) and other sources, such as Tou and Gonzalez (1974), Sklansky and Wassel (1981), and Young and Calvert (1974). The chapter covers the results of early efforts in machine learning, and it emphasizes the perceptron model of learning. The mathematical concepts that resulted from development of perceptrons continue to play a central role in most feedforward neural network learning and recall. Intuitive geometric explanations and mathematical foundations presented in this chapter can be viewed as fundamental for multilayer neural network learning.

To bridge the traditional learning concept using discrete perceptrons in single-layer networks with multilayer networks, continuous perceptron network learning has been introduced. By replacing the discontinuous threshold operation with a continuous activation function, it is possible to compute the mapping error gradient in multilayer feedforward networks. The multilayer network training algorithm has opened new areas of solutions and applications. It also opened new questions. They will all be introduced in the following chapter.

PROBLEMS

Please note that problems highlighted with an asterisk (*) are typically computationally intensive and the use of programs is advisable to solve them.

- P3.1** Write down equations along with the constraint inequalities for the decision planes subdividing the cube into four decision regions $\mathcal{X}_1, \dots, \mathcal{X}_4$ as shown in Figure P3.1. The pattern vectors' \mathbf{x} memberships in classes are:

$$\begin{aligned} i_o = 1 & \text{ for } \mathbf{x} = [1 \ 0 \ 0]^t, \quad [1 \ 1 \ 0]^t \\ i_o = 2 & \text{ for } \mathbf{x} = [1 \ 0 \ 1]^t, \quad [1 \ 1 \ 1]^t \\ i_o = 3 & \text{ for } \mathbf{x} = [0 \ 1 \ 0]^t, \quad [0 \ 1 \ 1]^t \\ i_o = 4 & \text{ for } \mathbf{x} = [0 \ 0 \ 0]^t, \quad [0 \ 0 \ 1]^t \end{aligned}$$

Note that the faces, edges, and vertices here belong to the pattern space. Whenever the decision surface intersects the edge of the cube, the intersection point should halve that edge (see Appendix for appropriate form of plane equations).

- P3.2** A commonly used method for classification is template matching. Templates in this problem are unipolar binary vectors $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_8$, and \mathbf{v}_9 corresponding to the digit bit maps shown in Figure P3.2. These vectors are stored as weights within the network. Both template and input vectors contain 42 rows with a 0 entry for white pixels and a 1 entry for black

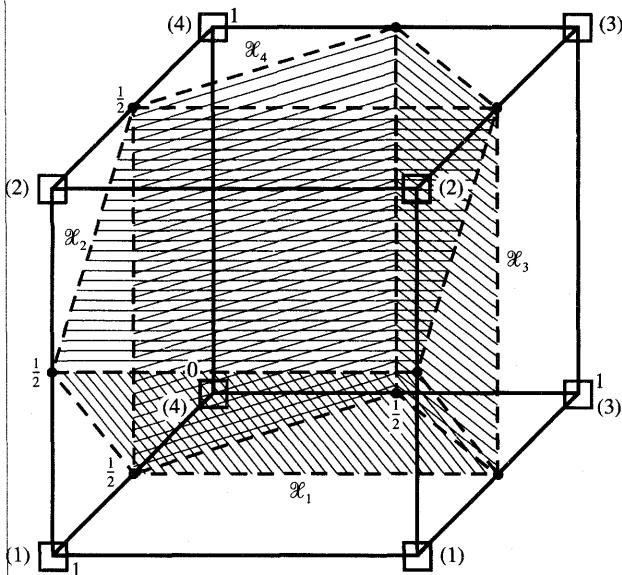


Figure P3.1 Geometric interpretation for classification of cube vertices in three-dimensional space (class numbers in parentheses).

pixels. The measure of matching of an input x to a template v is the scalar product $v'x$. The product value x_j is proportional to the similarity of input and template.

Since the templates have different numbers of black pixels, each x_j should be multiplied by a coefficient w_j inversely proportional to the number of black pixels in each template to equalize measures of similarity. This simple classification procedure reduces now to the selection of the maximum of the four responses being normalized matching scores $x_j w_j$ and computing i_o denoting the number of the largest input.

Compute and tabulate the normalized matching scores between each pair of the digits 0, 1, 8, 9 shown at the input in Figure P3.2. Inspect the matching scores and for each digit being input find

- (a) its nearest neighbor digit with the strongest overlap of the two vectors,
- (b) its furthest neighbor digit with the weakest overlap of the two vectors.

Assume now that the input bit maps of digits are randomly distorted. Based on the matching scores calculated determine which is

- (c) the most likely misclassification pair among the pairs of digits
- (d) the least likely misclassification pair among the pairs of digits.

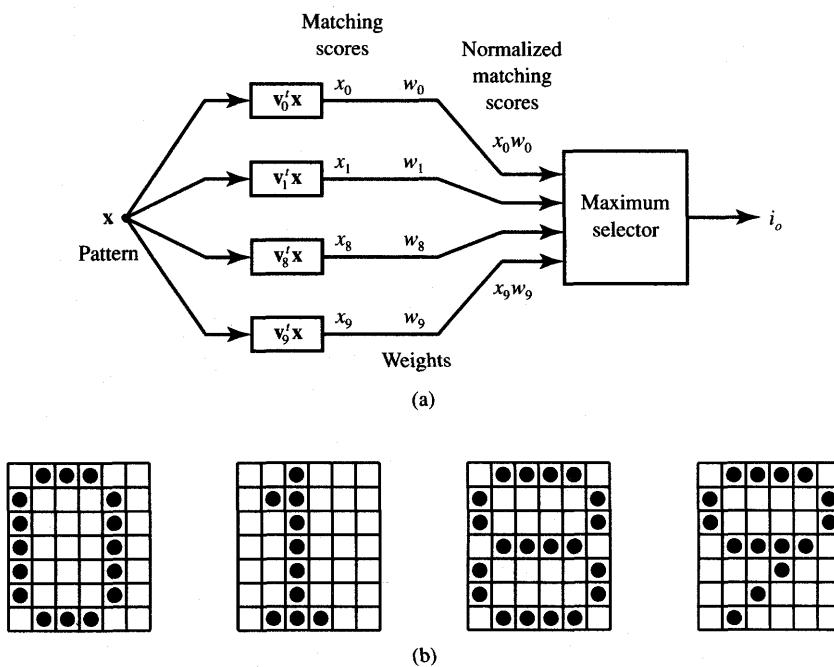


Figure P3.2 Template matching-based digit recognition system for Problem P3.2: (a) classifier and (b) character bit maps used for classification.

P3.3 For the minimum-distance (linear) dichotomizer, the weight and augmented pattern vectors are

$$\mathbf{w} = \begin{bmatrix} 2 \\ -1 \\ 2 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

- (a) Find the equation of the decision surface in the pattern space.
- (b) Find the equation of the decision surface in the augmented pattern space.
- (c) Compute the new solution weight vector if the two class prototype points are

$$\mathbf{x}_1 = [2 \ 5]^t \text{ and } \mathbf{x}_2 = [-1 \ -3]^t.$$

- (d) Sketch the decision surfaces for each case in parts (a), (b), and (c).

P3.4 Compute the solution weight vectors \mathbf{w}_1 , \mathbf{w}_2 , \mathbf{w}_3 , and \mathbf{w}_4 needed for the linear machine to classify patterns as indicated in Figure P3.1 and using the decision planes shown. The decision plane equations should be developed in this problem based on inspection of conditions in the pattern space, not on the iterative training procedure.

- P3.5 Sketch the example discriminant functions $g_1(\mathbf{x})$ and $g_2(\mathbf{x})$ as in Example 3.1 for the dichotomizer with the following augmented weight and input vectors, respectively:

$$\mathbf{w} = \begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

knowing that the pattern $\mathbf{y} = [0 \ 0 \ 1]^t$ belongs to class 1.

- P3.6 A dichotomizer has to be trained to assign $x = 0$ and $x = 2$ to class 1 and 2, respectively. Display the movement of the weight vector on the weight plane starting from the initial weights of $[1 \ 1]^t$ values and follow intermediate steps until weights fall to the solution region

- (a) Use $c = 1$.
- (b) Use $\lambda = 1$, as in (3.31).
- (c) Use $\lambda = 2$, as in (3.31).

(Hint: In the case of $\lambda = 1$, adjusted weights fall on the decision plane. This can also happen for $\lambda \neq 1$ in other training modes. Thus, to achieve the correct classification, the final adjustment must move weights from the decision plane into the decision region.)

- P3.7 Prototype points are given as

$$\begin{aligned} \mathbf{x}_1 &= [5 \ 1]^t, \mathbf{x}_2 = [7 \ 3]^t, \mathbf{x}_3 = [3 \ 2]^t, \mathbf{x}_4 = [5 \ 4]^t : \text{class 1} \\ \mathbf{x}_5 &= [0 \ 0]^t, \mathbf{x}_6 = [-1 \ -3]^t, \mathbf{x}_7 = [-2 \ 3]^t, \mathbf{x}_8 = [-3 \ 0]^t : \text{class 2} \end{aligned}$$

- (a) Determine if the two classes of patterns are linearly separable.
- (b) Determine the center of gravity for patterns of each class, and find and draw the decision surface in pattern space.
- (c) Using (3.9) or (3.11) design the dichotomizer for the given prototype points and determine how it would recognize the following input patterns of unknown class membership:

$$\mathbf{x} = [4 \ 2]^t, \quad \mathbf{x} = [0 \ 5]^t, \quad \mathbf{x} = \begin{bmatrix} 36 \\ 13 \end{bmatrix}^t$$

- P3.8 Class prototype vectors are known as

$$\begin{aligned} \mathbf{x}_1 &= [-2], \mathbf{x}_2 = \left[-\frac{2}{3} \right], \mathbf{x}_3 = [3] : \text{class 1} \\ \mathbf{x}_4 &= [1], \mathbf{x}_5 = [2] : \text{class 2} \end{aligned}$$

- (a) Draw patterns in the augmented pattern space.
- (b) Draw separating lines in the augmented weight space for each pattern.

- (c) Find the set of weights for the linear dichotomizer, or conclude that this is not a linearly separable classification problem and go to part (d).
- (d) Design the dichotomizer using a single discrete perceptron and non-linear discriminant function of quadratic type.

[Hint: $g(\mathbf{x})$ needs to be chosen as a quadratic function of \mathbf{x} .]

- P3.9* Implement the Single Discrete Perceptron Training Algorithm for $c = 1$ for the discrete perceptron dichotomizer, which provides the following classification of six patterns:

$$\mathbf{x} = [0.8 \ 0.5 \ 0]^t, [0.9 \ 0.7 \ 0.3]^t, [1 \ 0.8 \ 0.5]^t : \text{class 1}$$

$$\mathbf{x} = [0 \ 0.2 \ 0.3]^t, [0.2 \ 0.1 \ 1.3]^t, [0.2 \ 0.7 \ 0.8]^t : \text{class 2}$$

Perform the training task starting from initial weight vector $\mathbf{w} = \mathbf{0}$ and obtain the solution weight vector.

- P3.10* Repeat Problem P3.9 for the user-selectable coefficient λ . Solve the training task for $\lambda = 1$, then for $\lambda = 2$ as in (3.31). Note that the initial weight vector $\mathbf{w} \neq \mathbf{0}$ should be assumed.

- P3.11 Derive the continuous perceptron training rule (delta training rule) (3.40) and obtain a formula for \mathbf{w}^{k+1} similar to (3.53). Assume the unipolar activation function of the neuron

$$f(\text{net}) = \frac{1}{1 + \exp(-\text{net})}$$

Note that $f'(\text{net})$ should be expressed as function of the neuron's output.

- P3.12* Implement the delta training rule as in (3.53) for a single continuous perceptron suitable for the classification task specified in Problem P3.9. Train the network using data from Problem P3.9. Try several different learning constant values and note the speed of weight convergence in each learning simulation.

- P3.13 The error function to be minimized has the contour map shown in Figure P3.13 and is given by

$$E(\mathbf{w}) = \frac{1}{2} [(w_2 - w_1)^2 + (1 - w_1)^2]$$

- (a) Find analytically the gradient vector

$$\nabla E(\mathbf{w}) = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \end{bmatrix}$$

- (b) Find analytically the weight vector \mathbf{w}^* that minimizes the error function such that $\nabla E(\mathbf{w}) = \mathbf{0}$.

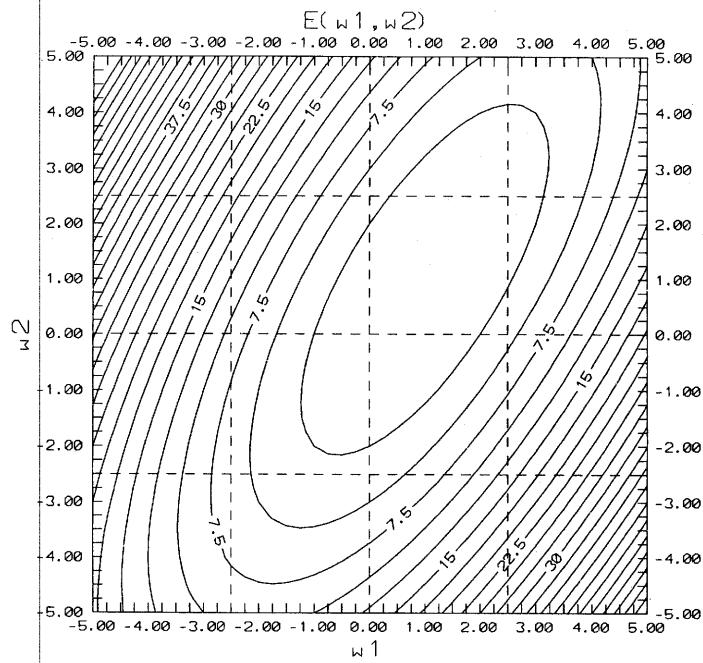


Figure P3.13 Contour map of the error function for Problem P3.13.

- (c) Based on part (a) draw the computed gradient vector at three selected weight values. Superimpose the three vectors on Figure P3.13 and check for the consistency of the gradient vectors with the contours' direction and steepness.

P3.14 The multiclass trainable classifier using the maximum selector requires weight adjustment as expressed in (3.54). Implement the sequence of training steps that is needed to classify correctly the three classes as in Example 3.2 and Figure 3.8. Present the patterns in the sequence P_1, P_2, P_3 , and start from initial weights $\mathbf{w} = \mathbf{0}$. Assume $c = 1$. The augmented pattern component equals -1 .

P3.15* Implement the training procedure (3.54) for a four-class classifier having the prototype points as in Problem P3.1. Use $c = 1$ and obtain weights $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$, and \mathbf{w}_4 , connecting each of the four inputs to the maximum selector as shown in Figure 3.7 for $R = 4$.

Then convert the classifier obtained using the maximum selector with four inputs to the form as in Figure 3.21 so that it uses four discrete perceptrons. Determine the additional threshold values T_1, \dots, T_4 if the augmented pattern component is $y_4 = -1$.

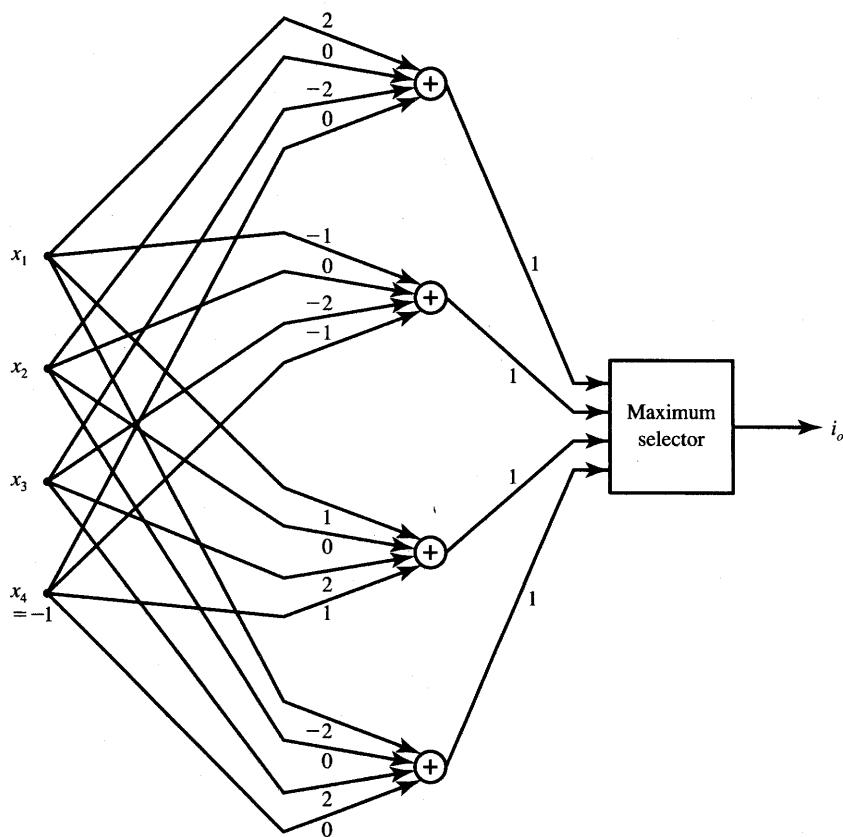


Figure P3.16 A pretrained four-discrete perceptron classifier for Problems P3.16 and P3.17.

P3.16 After the completion of training of the linear machine, its weights have been obtained as in Figure P3.16. The patterns that have been submitted during the supervised training cycle are eight vertices of the three-dimensional cube spanned across $0 \leq x_i \leq 1$. The augmented pattern component is assumed to be $y_4 = -1$ on the figure. Find the class membership of each of the cube vertices.

P3.17 Convert the classifier shown in Figure P3.16 to the form found in Figure 3.21 and determine the additional threshold values T_1, \dots, T_4 necessary for correct classification.

P3.18 The initial weight vectors for a three-class classifier with discrete bipolar perceptrons performing the classification as in Example 3.5 are selected as $\mathbf{w}_1^1 = \mathbf{w}_2^1 = \mathbf{w}_3^1 = \mathbf{0}$. For patterns presented in the sequence $P_1, P_2, P_3, P_4, P_5, \dots$, and trained as in (3.58) find

- (a) final weights for $c = 1$. Assume $\text{net} = 0$ as an incorrect response, thus requiring weight adjustment.
- (b) any “indecision” regions of the trained classifier that may be of significance on the x_1, x_2 plane in case of recognition of points that are not identical to the prototypes.
- P3.19 Training the weight adjustments as in (3.58) (RDPTA) of a three-class classifier using a three discrete bipolar perceptron network has been completed in three steps with the augmented pattern component of -1 and $c = 1$.

Step 1: presentation of y_1 resulted in adjustment of all weights initialized as

$$\mathbf{w}_1^1 = \mathbf{w}_2^1 = \mathbf{w}_3^1 = \mathbf{0}.$$

Step 2: presentation of y_2 resulted in adjustment of weight \mathbf{w}_3^2 only.

Step 3: presentation of y_3 resulted in adjustment of \mathbf{w}_2^3 only.

The final weights are

$$\mathbf{w}_1^4 = \begin{bmatrix} 1 \\ 3 \\ -1 \end{bmatrix}, \mathbf{w}_2^4 = \begin{bmatrix} 5 \\ -1 \\ -2 \end{bmatrix}, \mathbf{w}_3^4 = \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}$$

Find patterns \mathbf{x}_1 , \mathbf{x}_2 , and \mathbf{x}_3 used for the training.

- P3.20 Assume that the training rule given in (3.38) has been modified to the form

$$\mathbf{w}^{k+1} = \mathbf{w}^k + c_k e^k \frac{\mathbf{y}^k}{\|\mathbf{y}^k\|^2}$$

where c_k is a constant, $0 < c_k < 1$, and the error e^k is defined as

$$e^k = d^k - \mathbf{w}^{kt} \mathbf{y}^k$$

and is equal to the difference between the current desired response d^k and the current value of the neuron's activation net^k [see (2.43) for the learning rule]. Assume that the $(k+1)$ 'th step requires weight adaptation. Prove that after the weight correction is made in the k 'th step, and again the same input pattern $\mathbf{y}^{k+1} = \mathbf{y}^k$ is presented, the error is reduced $(1 - c_k)$ times. [This is the property of Widrow-Hoff learning rule (Widrow 1962).]

- P3.21* Implement the RDPTA training algorithm for the four-category classifier using four bipolar discrete perceptrons and using eight prototype points in three-dimensional input pattern space. Use $c = 1$ and compute weights \mathbf{w}_1 , \mathbf{w}_2 , \mathbf{w}_3 , and \mathbf{w}_4 connecting each of the four inputs to the perceptrons 1, 2, 3, and 4, respectively, as a result of the training. Carry out the training using the prototype points as specified in Problem P3.1. Sketch

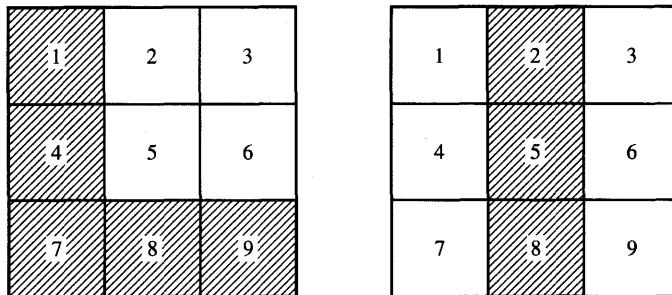


Figure P3.22 Bit maps of characters *L* and *I* for Problem P3.22.

the decision planes produced as a result of the training and verify that the classification is in agreement with specifications.

- P3.22* Design and train the classifier of printed characters *L* and *I* shown in the bit map form in Figure P3.22. Assign input vector entries 0 and 1 to white and black pixels, respectively. Use a single discrete bipolar perceptron with ten input weights, including the threshold.
- P3.23 The three-class classifier from Example 3.5 produces the decision regions as illustrated in Figure 3.22. In addition to three correct classification regions for points used during training, it also produces four “indecision” regions, shaded on the figure. Compute the classifier’s response in each of the four “indecision” regions by inspecting the decision surfaces shown in the figure.

REFERENCES

- Amit, D. J. 1989. *Modeling Brain Function: The World of Attractor Neural Networks*. Cambridge: Cambridge University Press.
- Andrews, H. C. 1972. *Introduction to Mathematical Techniques in Pattern Recognition*. New York: Wiley Interscience.
- Nilsson, N. J. 1965. *Learning Machines: Foundations of Trainable Pattern Classifiers*. New York: McGraw Hill Book Co; also republished as *The Mathematical Foundations of Learning Machines*. San Mateo, Calif.: Morgan Kaufmann Publishers.
- Rektorys, K. 1969. *Survey of Applicable Mathematics*. Cambridge: The MIT Press.
- Rosenblatt, F. 1961. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Washington D.C.: Spartan Books.

- Sklansky, J. and G. N. Wassel. *Pattern Classifiers and Trainable Machines*. Berlin: Springer-Verlag.
- Tou, J. T. and R. C. Gonzalez. 1974. *Pattern Recognition Principles*. Reading, Mass.: Addison-Wesley Publishing Co.
- Widrow, B. 1962. "Generalization and Information Storage in Networks of Adaline 'Neurons,'" in *Self-Organizing Systems 1962*, ed. M. C. Jovitz, G. T. Jacobi, G. Goldstein. Washington, D.C.: Spartan Books, 435-461.
- Wittner, B. S. and S. Denker. 1988. "Strategies for Teaching Layered Networks Classification Tasks," in *Neural Information Processing Systems*, ed. D. L. Anderson. New York: American Institute of Physics, 850-857.
- Young, T. Y. and T. W. Calvert. 1974. *Classification, Estimation and Pattern Recognition*. New York: Elsevier Publishing Co.

MULTILAYER FEEDFORWARD NETWORKS

A certain lady claims that, after tasting a cup of tea with milk, she can say which is first poured in the cup—milk or tea. This lady states that even if she sometimes makes mistakes, she is more often right than wrong.

R. FISHER

- 4.1 Linearly Nonseparable Pattern Classification
 - 4.2 Delta Learning Rule for Multiperceptron Layer
 - 4.3 Generalized Delta Learning Rule
 - 4.4 Feedforward Recall and Error Back-propagation Training
 - 4.5 Learning Factors
 - 4.6 Classifying and Expert Layered Networks
 - 4.7 Functional Link Networks
 - 4.8 Concluding Remarks
-

Our study thus far has focused on using neural networks for classification with input patterns that are linearly separable. Networks discussed in the previous chapter have been able to acquire experiential knowledge during the supervised training process. Both two- and multiclassification learning procedures have been formulated for single-layer networks. The experiential knowledge acquisition has been based on the convergent training of single-layer discrete perceptron networks, which can adjust their weights incrementally in order to achieve correct classification of linearly separable sets of patterns.

To this point, we have discussed networks that use a linear combination of inputs with weights being proportionality coefficients. Such networks work with the argument of the nonlinear element simply computed as a scalar product of the weight and input vectors. We may call networks of this type *linear* keeping in mind, however, that the nonlinear operation $f(\text{net})$ is still performed by the decision element on the argument net at every computing node.

For training patterns that are linearly separable, the linear neural network introduced in previous chapters must be modified if it is to perform the correct classification. The modification could involve either a departure from the concept of the linear discriminant function or a major change in network architecture. Typical nonlinear discriminant functions are chosen to be quadratic or piecewise linear. The piecewise linear discriminant functions can be implemented by a single-layer linear network employing perceptrons (Nilsson 1965). Our chosen architecture, however, will be that of the multilayer network. Each layer of a multilayer network is composed of a linear network, i.e., it is based on the original concept of the linear discriminant function. As discussed in Section 2.2, multilayer networks are of the feedforward type and are functionally similar to the networks covered in Chapter 3.

Although multilayer learning machines have been known of for more than a quarter of a century, the lack of appropriate training algorithms has prevented their successful applications for practical tasks. Recent developments in the supervised training algorithms of multilayer networks with each layer employing the linear discriminant function have led to their widespread and successful present use. Multilayer networks are often called *layered networks*. They can implement arbitrary complex input/output mappings or decision surfaces separating pattern classes.

The most important attribute of a multilayer feedforward network is that it can learn a mapping of any complexity. The network learning is based on repeated presentations of the training samples, as has been the case for single-layer networks. The trained network often produces surprising results and generalizations in applications where explicit derivation of mappings and discovery of relationships is almost impossible.

In addition to the classification tasks studied in Chapters 3 and 4 to gain insight into the single- and multilayer neural networks, many other tasks can be performed by such networks. Examples include function approximation, handwritten character recognition, speech recognition, dynamical plant control, robot kinematics and trajectory generation, expert systems, and many other applications. We postpone the discussion of applications until Chapter 8 and focus on principles and training methods for layered networks.

Multilayer feedforward network theory and applications have been dominating in the neural network literature for several years now. Researchers have looked for the properties of layered networks and their training methods. Industrial entrepreneurs found a wealth of successful applications, others developed accelerator boards to speed up network simulations on personal computers, and graduate students began to write and sell inexpensive training software. Layered networks seem to be the most widespread neural network architecture at present.

To understand the nature of mapping performed by multilayer feedforward networks and the training of such networks, we must return to the fundamentals. In this chapter we will study trainable layered neural networks employing the

input pattern mapping principles. In the case of layered network training, we will see that the mapping error can be propagated into hidden layers so that the output error information passes backward. This mechanism of backward error transmission is used to modify the synaptic weights of internal and input layers. The delta learning rule and its generalization are used throughout the chapter for supervised training of multilayer continuous perceptron networks.

4.1

LINEARLY NONSEPARABLE PATTERN CLASSIFICATION

Expressions (3.32) formulated the condition for linear separability of patterns. This condition can now be briefly restated for the case of linearly nonseparable dichotomization. Assume the two training sets \mathcal{Y}_1 and \mathcal{Y}_2 of augmented patterns are available for training. If no weight vector w exists such that

$$\begin{aligned} y'w &> 0 \text{ for each } y \in \mathcal{Y}_1, \quad \text{and} \\ y'w &< 0 \text{ for each } y \in \mathcal{Y}_2 \end{aligned}$$

then the pattern sets \mathcal{Y}_1 and \mathcal{Y}_2 are *linearly nonseparable*.

Let us now see how the original *pattern space* can be mapped into the so-called *image space* so that a two-layer network can eventually classify the patterns that are linearly nonseparable in the original pattern space.

Assume initially that the two sets of patterns \mathcal{X}_1 and \mathcal{X}_2 should be classified into two categories. The example patterns are shown in Figure 4.1(a). Three arbitrary selected partitioning surfaces 1, 2, and 3 have been shown in the pattern space x . The partitioning has been done in such a way that the pattern space now has compartments containing only patterns of a single category. Moreover, the partitioning surfaces are hyperplanes in pattern space E^n . The partitioning shown in Figure 4.1(a) is also nonredundant, i.e., implemented with minimum number of lines. It corresponds to mapping the n -dimensional original pattern space x into the three-dimensional image space o .

Recognizing that each of the decision hyperplanes 1, 2, or 3 is implemented by a single discrete perceptron with suitable weights, the transformation of the pattern space to the image space can be performed by the network as in Figure 4.1(b). As can be seen from the figure, only the first layer of discrete perceptrons responding with o_1 , o_2 , and o_3 is involved in the discussed space transformation. Let us look at some of the interesting details of the proposed transformation.

The discussion below shows how a set of patterns originally linearly nonseparable in the pattern space can be mapped into the image space where it becomes linearly separable. Realizing that the arrows point toward the positive side of the decision hyperplane in the pattern space, each of the seven compartments from Figure 4.1(a) is mapped into one of the vertices of the $[-1, 1]$

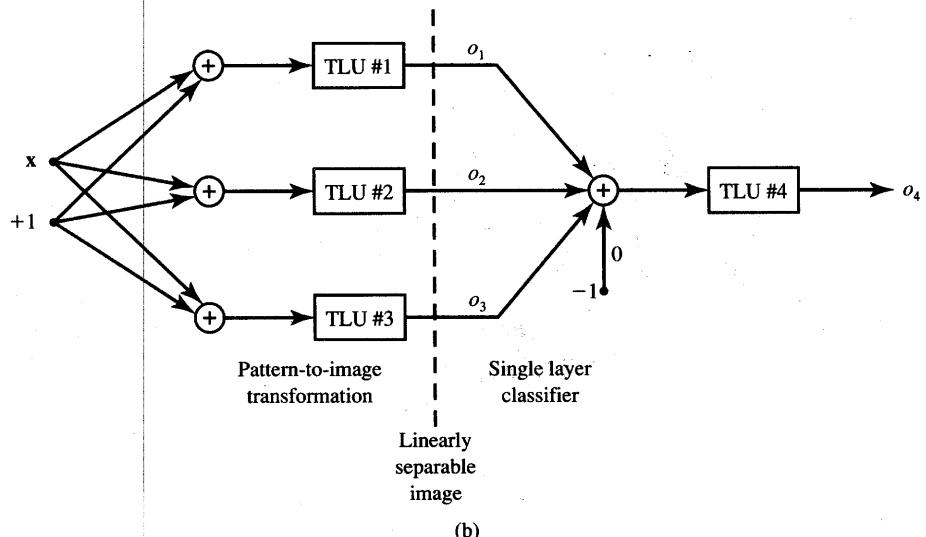
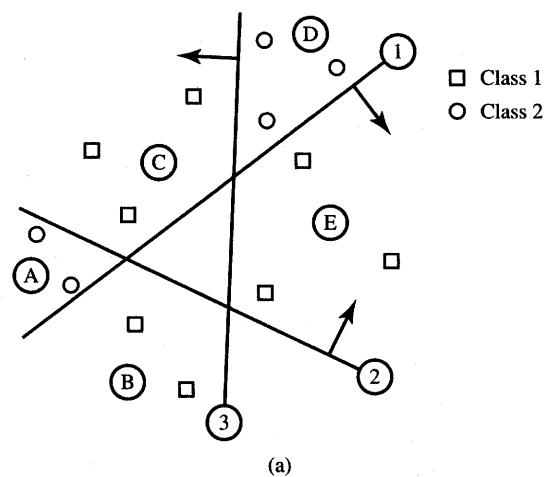


Figure 4.1a,b Classification of linearly nonseparable patterns: (a) partitioning in the pattern space, (b) layered network implementing the classification from part (a).

cube. The result of the mapping for the patterns from the figure is depicted in Figure 4.1(c) showing the cube in image space o_1 , o_2 , and o_3 with corresponding compartment labels at corners.

The patterns of class 1 from the original compartments B, C, and E are mapped into vertices $(1, -1, 1)$, $(-1, 1, 1)$, and $(1, 1, -1)$, respectively. In turn,

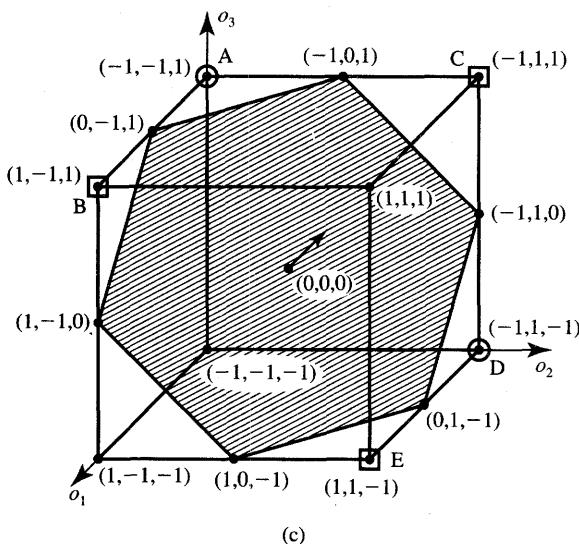


Figure 4.1c Classification of linearly nonseparable patterns (*continued*): (c) classification in the image space by the output perceptron from part (b).

patterns of class 2 from compartments A and D are mapped into vertices $(-1, -1, 1)$ and $(-1, 1, -1)$, respectively. This shows that in the image space o , the patterns of class 1 and 2 are easily separable by a plane arbitrarily selected, such as the one shown in Figure 4.1(c) having the equation $o_1 + o_2 + o_3 = 0$. The single discrete perceptron in the output layer with the inputs o_1 , o_2 , and o_3 , zero bias, and the output o_4 is now able to provide the correct final mapping of patterns into classes as follows:

$$o_4 = \begin{cases} \text{sgn}(o_1 + o_2 + o_3) > 0 : & \text{class 1} \\ \text{sgn}(o_1 + o_2 + o_3) < 0 : & \text{class 2} \end{cases}$$

From the discussion above, we see that the procedure of training, in the case of the layered network, has to produce linearly separable images in the image space. If the pattern parameters and desired responses were completely known, the weight calculation of the two-layer linear machine could be accomplished, as just described, without an experiential knowledge acquisition. In another approach, a single-layer-only machine with the piecewise linear discriminant functions can be employed. Such a classifier with a nonlinear discriminant function would be able to produce appropriate partitioning of the input space and to classify patterns that are linearly nonseparable.

Although both these approaches are feasible, they possess a rather significant disadvantage. Since we favor acquiring experiential knowledge by the layered

learning network over the calculation of weights from pattern parameters, we should adopt and use its stepwise supervised learning algorithms. We will also favor employing linear discriminant functions instead of piecewise, quadratic, or any other type of nonlinear function. This preference is based on the premise that the continuous perceptron with *net* in the form of a scalar product is trainable and it offers both mapping and representation advantages. However, no efficient and systematic learning algorithms were known for layered linear machines employing perceptrons until recently (Werbos 1974; McClelland and Rumelhart 1986).

EXAMPLE 4.1

In this example we design a simple layered classifier that is able to classify four linearly nonseparable patterns. The discussion will emphasize geometrical considerations in order to visualize the input-to-image-to-output space mapping during the learning cycle. The classifier is required to implement the XOR function as defined in (3.21) for two variables. The decision function to be implemented by the classifier is:

	x_1	x_2	Output
	0	0	1
	0	1	-1
	1	0	-1
	1	1	1

The patterns, along with the proposed pattern space partitioning implemented by the first layer consisting of two bipolar discrete perceptrons, are shown in Figure 4.2(a). The arbitrary selected partitioning shown is provided by the two decision lines having equations

$$\begin{aligned} -2x_1 + x_2 - \frac{1}{2} &= 0 \\ x_1 - x_2 - \frac{1}{2} &= 0 \end{aligned} \quad (4.1a)$$

For the lines shown, the corresponding unit normal vectors pointing toward the positive side of each line are equal to

$$\mathbf{r}_1 = \frac{1}{\sqrt{5}} \begin{bmatrix} -2 \\ 1 \end{bmatrix}, \quad \mathbf{r}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad (4.1b)$$

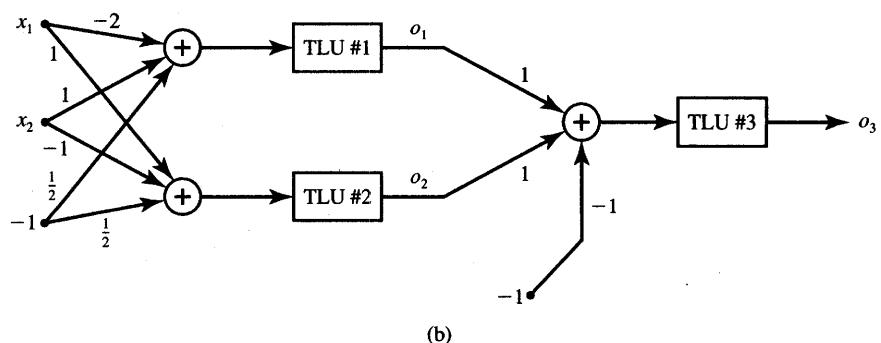
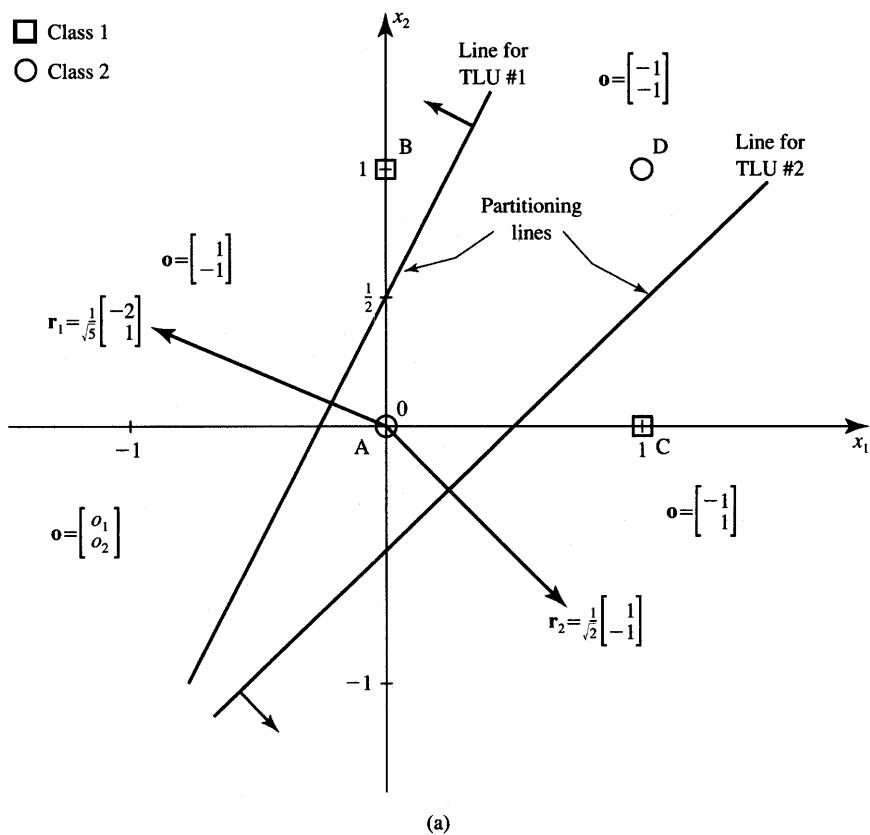


Figure 4.2 Figure for Example 4.1: (a) pattern to image space transformation and (b) classifier diagram.

Figure 4.2(a) shows that the three compartments produced in the x_1, x_2 plane are mapped through the two first-layer discrete perceptrons with TLU #1 and TLU #2 shown in Figure 4.2(b). The mapping of patterns into the image space is:

$$\begin{aligned} o_1 &= \text{sgn}\left(-2x_1 + x_2 - \frac{1}{2}\right) \\ o_2 &= \text{sgn}\left(x_1 - x_2 - \frac{1}{2}\right) \end{aligned} \quad (4.2a)$$

The performed mapping is explained in more detail in Figure 4.3. Figure 4.3(a) provides the mapping and classification summary table. As before, the first layer provides an appropriate mapping of patterns into images in the image space. The second layer implements the classification of the images rather than of the original patterns. Note that both input patterns A and D collapse in the image space into a single image $(-1, -1)$. The image space with images of original patterns is shown in Figure 4.3(b). An arbitrary decision line providing the desired classification and separating A,D and B,C in the image space has been selected as

$$o_1 + o_2 + 1 = 0 \quad (4.2b)$$

The TLU #3 implements the decision o_3 as

$$o_3 = \text{sgn}(o_1 + o_2 + 1) \quad (4.2c)$$

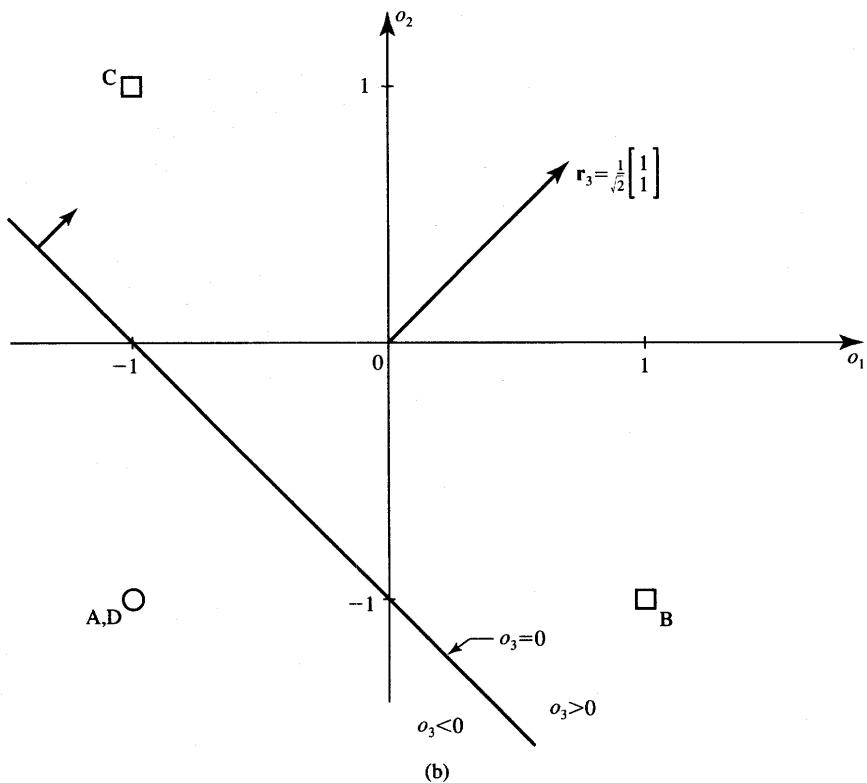
The complete diagram of the classifier is shown in Figure 4.2(b). The output perceptron has weights $w_1 = 1$, $w_2 = 1$, and $w_3 = -1$ to satisfy Eq. (4.2c). Figure 4.4 illustrates the joint mapping between the pattern and output space. Discrete perceptron mapping of plane x_1, x_2 into o_3 is shown in Figure 4.4(a). Figure 4.4(b) presents the contour map $o_3(x_1, x_2)$ for continuous bipolar perceptrons with $\lambda = 6$ used rather than discrete perceptrons. The weight values for the mapping depicted are identical to those in Figure 4.2(b), only the $\text{sgn}(\cdot)$ functions have been replaced with functions defined as in (2.3a).

This example provides insight into the parametric design of a layered classifier for linearly nonseparable patterns. The design has used arbitrary but meaningful mapping of inputs into their images and then into outputs, and decision lines suitable for classification of patterns with known parameters have been produced by inspection. The example, however, has not yet involved an experiential training approach needed to design neural network classifiers.

Before we derive a formal training algorithm for layered perceptron networks, one additional aspect is worth highlighting. The layered networks develop

Symbol	Pattern Space		Image Space		TLU #3 Input $o_1 + o_2 + 1$	Output Space o_3	Class Number
	x_1	x_2	o_1	o_2			
A	0	0	-1	-1	-	-1	2
B	0	1	1	-1	+	+1	1
C	1	0	-1	1	+	+1	1
D	1	1	-1	-1	-	-1	2

(a)



(b)

Figure 4.3 Mapping performed by the output perceptron: (a) classification summary table and (b) decision line.

significant, if not powerful, self-organization and mapping properties during training. The self-organization and mapping do not have to be engineered in advance. Instead, the acquisition of knowledge takes place gradually during the training phase by inspection of mapping examples.

Figure 4.5(a) illustrates the specific case of classification of more involved planar patterns. Both shaded disjointed areas A and B on the figure belong to

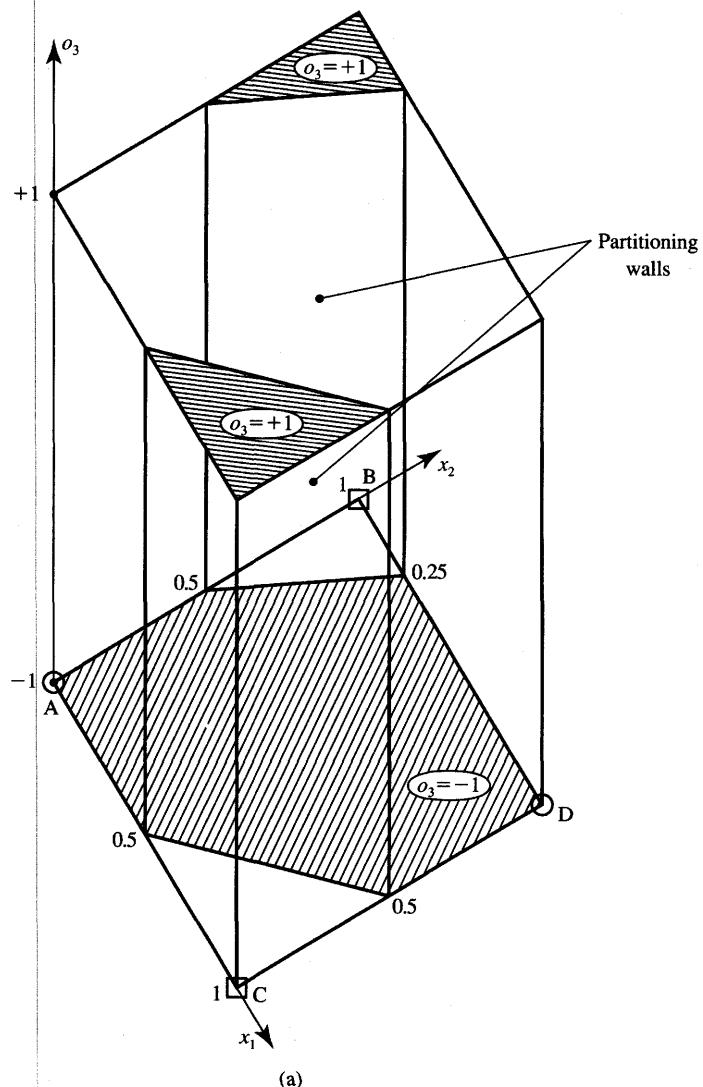


Figure 4.4a Linearly nonseparable classification of patterns in Example 4.1: (a) input to output mapping using discrete perceptrons.

the category 1, and the rest of the pattern space of the plane belongs to the category 2. The classification of the input patterns, or mapping of the input space to classes, can be provided by the network shown in Figure 4.5(b). If the weighted sum of the nonaugmented patterns exceeds the threshold value T , the TLU element responds with $+1$, otherwise with -1 .

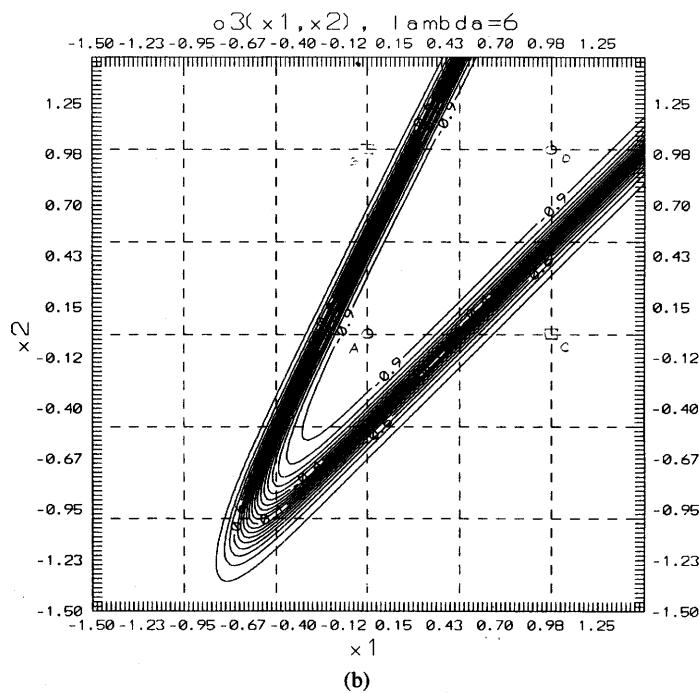


Figure 4.4b Linearly nonseparable classification of patterns in Example 4.1 (continued):
(b) input to output mapping using continuous perceptrons ($\lambda = 6$).

It is easy to verify that the input layer provides the following mapping: units #1 and #2 select the vertical strip $a < x_1 < b$, and units #3 and #4 select the strip $c < x_1 < d$. Units #5 with #6 select the horizontal strip $e < x_2 < f$. The output layer unit has the threshold value of 3.5. It responds with +1 signifying membership in class 1 when either its weighted input activations from units 1 and 2 at the summing node are both 1, or its weighted input activations from units 3 and 4 are both 1, and, at the same time, its weighted input activations from units 5 and 6 are also both 1. The summed excitatory input value to TLU #7 is thus of value 4 in the case of inputs in either area A or B, which results in a TLU #7 output of +1, thus signifying class 1.

The layered networks consisting of discrete perceptrons described in this section are also called “committee” networks (Nilsson 1965). The term committee is used because it takes a number of votes of the first layer to determine the input pattern classification. The inputs in the pattern space are first mapped into the images consisting of vertices of the cube $[-1, 1]$. Up to 2^n images, each image being a vertex of the cube, are available as binary images of n -tuple input vectors. Then, the image space mapping is investigated by the output perceptrons as to its separability and as to the mapped vectors class membership.

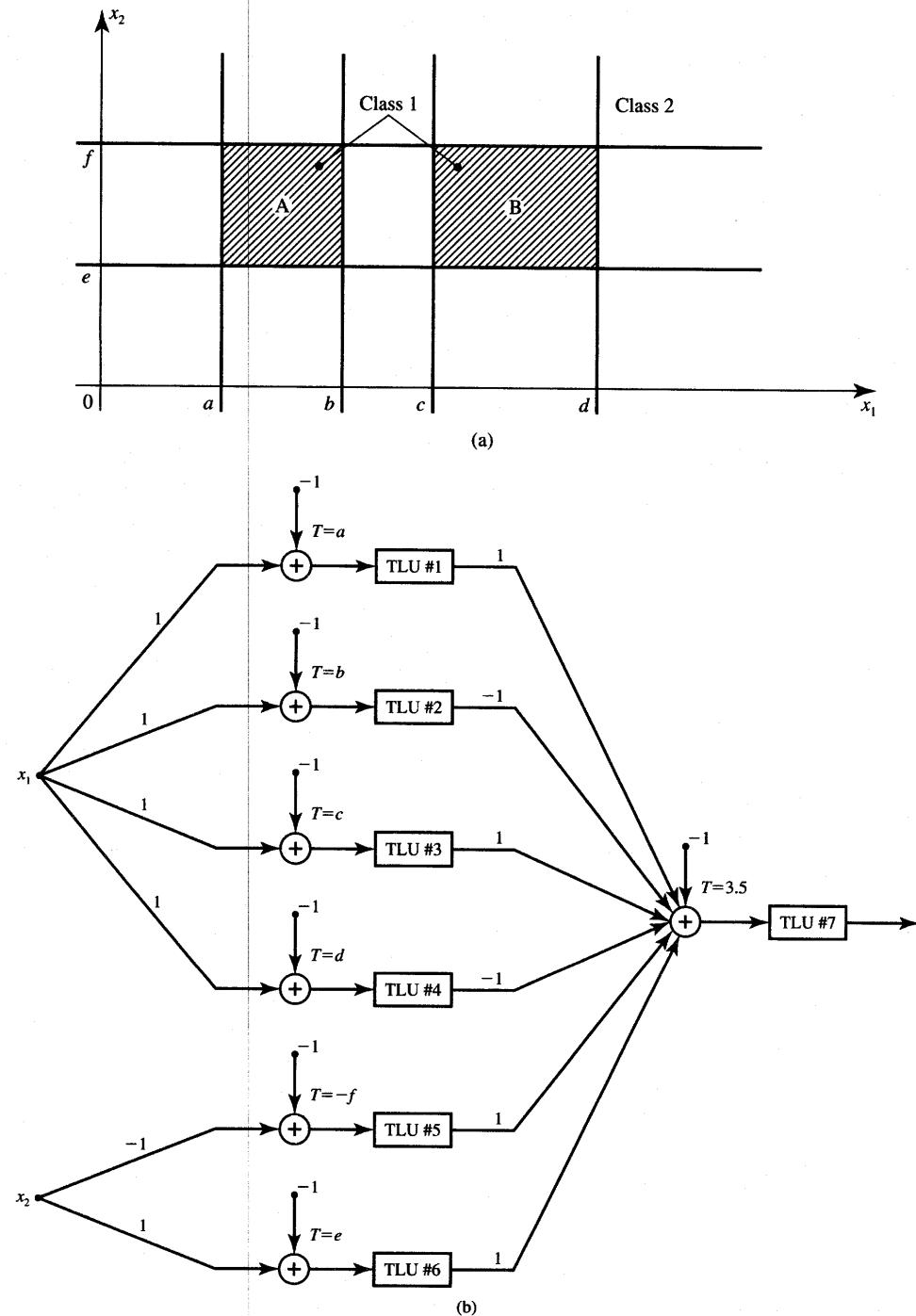


Figure 4.5 Planar pattern classification example: (a) pattern space and (b) discrete perceptron classifier network.

4.2

DELTA LEARNING RULE FOR MULTIPERCEPTRON LAYER

The following discussion is focused on a training algorithm applied to the multilayer feedforward networks. The algorithm is called the *error back-propagation training* algorithm. As mentioned earlier in this chapter, the algorithm has reawakened the scientific and engineering community to the modeling of many quantitative phenomena using neural networks.

The back-propagation training algorithm allows experiential acquisition of input/output mapping knowledge within multilayer networks. Similarly, as in simple cases of the delta learning rule training studied before, input patterns are submitted during the back-propagation training sequentially. If a pattern is submitted and its classification or association is determined to be erroneous, the synaptic weights as well as the thresholds are adjusted so that the current least mean-square classification error is reduced. The input/output mapping, comparison of target and actual values, and adjustment, if needed, continue until all mapping examples from the training set are learned within an acceptable overall error. Usually, mapping error is cumulative and computed over the full training set.

During the association or classification phase, the trained neural network itself operates in a feedforward manner. However, the weight adjustments enforced by the learning rules propagate exactly backward from the output layer through the so-called "hidden layers" toward the input layer. To formulate the learning algorithm, the simple continuous perceptron network involving K neurons will be revisited first. Let us take another look at the network shown in Figure 3.23. It is redrawn again in Figure 4.6 with a slightly different connection form and notation, but both networks are identical.

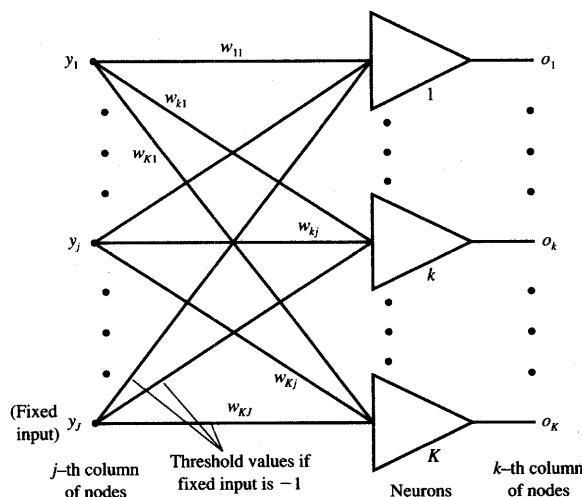


Figure 4.6 Single-layer network with continuous perceptrons.

The input and output values of the network are denoted y_j and o_k , respectively. We thus denote y_j , for $j = 1, 2, \dots, J$, and o_k , for $k = 1, 2, \dots, K$, as signal values at the j 'th column of nodes, and k 'th column of nodes, respectively. As before, the weight w_{kj} connects the output of the j 'th neuron with the input to the k 'th neuron.

Using the vector notation, the forward pass in the network from Figure 4.6 can be expressed as follows

$$\mathbf{o} = \Gamma[\mathbf{W}\mathbf{y}] \quad (4.3a)$$

where the input and output vector and the weight matrix are, respectively

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_J \end{bmatrix}, \quad \mathbf{o} = \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_K \end{bmatrix}$$

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1J} \\ w_{21} & w_{22} & \cdots & w_{2J} \\ \vdots & \vdots & \ddots & \vdots \\ w_{K1} & w_{K2} & \cdots & w_{KJ} \end{bmatrix}$$

and the nonlinear diagonal operator $\Gamma[\cdot]$ is

$$\Gamma[\cdot] = \begin{bmatrix} f(\cdot) & 0 & \cdots & 0 \\ 0 & f(\cdot) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f(\cdot) \end{bmatrix}$$

The desired (target) output vector is

$$\mathbf{d} \triangleq \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_K \end{bmatrix}$$

Observe that the activation vector net_k of the layer k is contained in the brackets in relationship (4.3a) and it can be expressed as

$$\text{net}_k = \mathbf{W}\mathbf{y} \quad (4.3b)$$

The error expression introduced in (2.37) for a single perceptron is now generalized to include all squared errors at the outputs $k = 1, 2, \dots, K$

$$E_p = \frac{1}{2} \sum_{k=1}^K (d_{pk} - o_{pk})^2 = \frac{1}{2} \|\mathbf{d}_p - \mathbf{o}_p\|^2 \quad (4.4)$$

for a specific pattern p , where $p = 1, 2, \dots, P$. Let us note that the subscript p in (4.4) refers to a specific pattern that is at the input and produces the output error.

At this point, the delta training rule introduced in (2.36), and later intuitively obtained in (3.55), can be formally derived for a multiperceptron layer. Let us assume that the gradient descent search is performed to reduce the error E_p through the adjustment of weights. For simplicity, it is assumed that the threshold values T_k , for $k = 1, 2, \dots, K$, are adjustable along with the other weights, and no distinction is made between the weights and thresholds during learning. Now the thresholds T_k are learned exactly in the same manner as the remaining weights. This, of course, implies that

$$w_{kj} = T_k, \quad \text{for } k = 1, 2, \dots, K$$

and the fixed input is of value

$$y_J = -1$$

during both the training and feedforward recall phases. Requiring the weight adjustment as in (2.39) we compute individual weight adjustment as follows:

$$\Delta w_{kj} = -\eta \frac{\partial E}{\partial w_{kj}} \quad (4.5a)$$

where the error E is defined in (4.4) with subscript p skipped for brevity. For each node in layer k , $k = 1, 2, \dots, K$, we can write using (4.3b)

$$net_k = \sum_{j=1}^J w_{kj} y_j \quad (4.5b)$$

and further, using (4.3a) the neuron's output is

$$o_k = f(net_k)$$

The *error signal term* δ called *delta* produced by the k 'th neuron is defined for this layer as follows

$$\delta_{ok} \stackrel{\Delta}{=} -\frac{\partial E}{\partial (net_k)} \quad (4.6)$$

It is obvious that the gradient component $\partial E / \partial w_{kj}$ depends only on the net_k of a single neuron, since the error at the output of the k 'th neuron is contributed to only by the weights w_{kj} , for $j = 1, 2, \dots, J$, for the fixed k value. Thus, using the chain rule we may write

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial (net_k)} \cdot \frac{\partial (net_k)}{\partial w_{kj}} \quad (4.7)$$

The second term of the product of Eq. (4.7) is the derivative of the sum of products of weights and patterns $w_{k1}y_1 + w_{k2}y_2 + \dots + w_{kJ}y_J$ as in (4.5b). Since the values y_j , for $j = 1, 2, \dots, J$, are constant for a fixed pattern at the input, we obtain

$$\frac{\partial (net_k)}{\partial w_{kj}} = y_j \quad (4.8)$$

Combining (4.6) and (4.8) leads to the following form for (4.7):

$$\frac{\partial E}{\partial w_{kj}} = -\delta_{ok}y_j \quad (4.9)$$

The weight adjustment formula (4.5a) can now be rewritten using the error signal δ_{ok} term as below

$$\Delta w_{kj} = \eta \delta_{ok}y_j, \quad \text{for } k = 1, 2, \dots, K \text{ and } j = 1, 2, \dots, J \quad (4.10)$$

Expression (4.10) represents the general formula for *delta training/learning weight adjustments* for a single-layer network. It can be noted that Δw_{kj} in (4.10) does not depend on the form of an activation function. As mentioned in Chapter 2, the delta value needs to be explicitly computed for specifically chosen activation functions. It also follows from (4.10) that the adjustment of weight w_{kj} is proportional to the input activation y_j , and to the error signal value δ_{ok} at the k 'th neuron's output.

To adapt the weights, the error signal term δ_{ok} introduced in (4.6) needs to be computed for the k 'th continuous perceptron. Note that E is a composite function of net_k , therefore it can be expressed for $k = 1, 2, \dots, K$, as follows:

$$E(net_k) = E[o_k(net_k)] \quad (4.11)$$

Thus, we have from (4.6)

$$\delta_{ok} = -\frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial (net_k)} \quad (4.12)$$

Denoting the second term in (4.12) as a derivative of the activation function

$$f'_k(net_k) \triangleq \frac{\partial o_k}{\partial (net_k)} \quad (4.13a)$$

and noting that

$$\frac{\partial E}{\partial o_k} = -(d_k - o_k) \quad (4.13b)$$

allows rewriting formula (4.12) as follows

$$\delta_{ok} = (d_k - o_k)f'_k(net_k), \quad \text{for } k = 1, 2, \dots, K \quad (4.14)$$

Equation (4.14) shows that the error signal term δ_{ok} depicts the local error $(d_k - o_k)$ at the output of the k 'th neuron scaled by the multiplicative factor $f'_k(net_k)$, which is the slope of the activation function computed at the following activation value

$$net_k = f^{-1}(o_k)$$

The final formula for the weight adjustment of the single-layer network can now be obtained from (4.10) as

$$\Delta w_{kj} = \eta(d_k - o_k)f'_k(net_k)y_j \quad (4.15a)$$

and it is identical to the delta training rule (2.40). The updated weight values become

$$w'_{kj} = w_{kj} + \Delta w_{kj} \quad \text{for } k = 1, 2, \dots, K, \text{ and } j = 1, 2, \dots, J \quad (4.15b)$$

Formula (4.15) refers to any form of the nonlinear and differentiable activation function $f(\text{net})$ of the neuron. Let us examine the two commonly used delta training rules for the two selected typical activation functions $f(\text{net})$.

For the unipolar continuous activation function defined in (2.4a), $f'(\text{net})$ can be obtained as

$$f'(\text{net}) = \frac{\exp(-\text{net})}{[1 + \exp(-\text{net})]^2} \quad (4.16a)$$

This can be rewritten as

$$f'(\text{net}) = \frac{1}{1 + \exp(-\text{net})} \cdot \frac{1 + \exp(-\text{net}) - 1}{1 + \exp(-\text{net})} \quad (4.16b)$$

or, if we use (2.4a) again, it can be rearranged to a more useful form involving output values only

$$f'(\text{net}) = o(1 - o) \quad (4.16c)$$

Let us also observe that the delta value of (4.14) for this choice of the activation function becomes

$$\delta_{ok} = (d_k - o_k)o_k(1 - o_k) \quad (4.17)$$

The delta value for the bipolar continuous activation function as in (2.3a) can be expressed as

$$\delta_{ok} = \frac{1}{2}(d_k - o_k)(1 - o_k^2) \quad (4.18a)$$

which uses the following identity for $f'(\text{net})$

$$f'(\text{net}) = \frac{1}{2}(1 - o^2) \quad (4.18b)$$

Expression (4.18b) was derived in Section 3.6, see formulas (3.48) through (3.51).

Summarizing the discussion above, the updated individual weights under the delta training rule can be expressed for $k = 1, 2, \dots, K$, and $j = 1, 2, \dots, J$, as follows

$$w'_{kj} = w_{kj} + \eta(d_k - o_k)o_k(1 - o_k)y_j \quad (4.19a)$$

for

$$o_k = \frac{1}{1 + \exp(-\text{net}_k)}$$

and

$$w'_{kj} = w_{kj} + \frac{1}{2}\eta(d_k - o_k)(1 - o_k^2)y_j \quad (4.19b)$$

for

$$o_k = 2 \left(\frac{1}{1 + \exp(-net_k)} - \frac{1}{2} \right)$$

The updated weights under the delta training rule for the single-layer network shown in Figure 4.6 can be succinctly expressed using the vector notation

$$\mathbf{W}' = \mathbf{W} + \eta \delta_o \mathbf{y}^t \quad (4.20)$$

where the error signal vector δ_o is defined as a column vector consisting of the individual error signal terms:

$$\delta_o \triangleq \begin{bmatrix} \delta_{o1} \\ \delta_{o2} \\ \vdots \\ \delta_{oK} \end{bmatrix}$$

Error signal vector entries δ_{ok} are given by (4.14) in the general case, or by (4.17) or (4.18a) depending on the choice of the activation function. Noticeably, entries δ_{ok} are local error signals dependent only on o_k and d_k of the k 'th neuron. It should be noted that the nonaugmented pattern vector in input space is $J - 1$ dimensional. Input values y_j , for $j = 1, 2, \dots, J - 1$, are not limited, while y_J is fixed at -1 .

The training rule for this single-layer neural network does provide a more diversified response than the network of K discrete perceptrons discussed in Chapter 3. The properly trained network also provides a continuous degree of associations that would have been of the binary form if TLUs alone were used. As such, a continuous perceptron network will be able to provide a more diversified set of responses rather than only binary-valued vectors, which are the natural responses at the classifier's output. These novel aspects of a continuous perceptron network will be discussed later along with introductory examples of its application.

The algorithm for multicategory perceptron training is given below.

■ *Summary of the Multicategory Continuous Perceptron Training Algorithm (MCPTA)*

Given are P training pairs arranged in the training set

$$\{\mathbf{y}_1, \mathbf{d}_1, \mathbf{y}_2, \mathbf{d}_2, \dots, \mathbf{y}_P, \mathbf{d}_P\}$$

where \mathbf{y}_i is $(J \times 1)$, \mathbf{d}_i is $(K \times 1)$, and $i = 1, 2, \dots, P$. Note that the J 'th component of each \mathbf{y}_i has the value -1 since input vectors have been augmented. Integer q denotes the training step and p denotes the counter within the training cycle.

Step 1: $\eta > 0$, $E_{\max} > 0$ chosen.

Step 2: Weights \mathbf{W} are initialized at small random values; \mathbf{W} is $(K \times J)$:

$$q \leftarrow 1, p \leftarrow 1, E \leftarrow 0.$$

Step 3: Training step starts here. Input is presented and output computed:

$$\begin{aligned} \mathbf{y} &\leftarrow \mathbf{y}_p, \mathbf{d} \leftarrow \mathbf{d}_p \\ o_k &\leftarrow f(\mathbf{w}_k^T \mathbf{y}), \quad \text{for } k = 1, 2, \dots, K \end{aligned}$$

where \mathbf{w}_k is the k 'th row of \mathbf{W} and $f(\text{net})$ is as defined in (2.3a).

Step 4: Weights are updated:

$$\mathbf{w}_k \leftarrow \mathbf{w}_k + \frac{1}{2} \eta (d_k - o_k) (1 - o_k^2) \mathbf{y}, \quad \text{for } k = 1, 2, \dots, K$$

where \mathbf{w}_k is the k 'th row of \mathbf{W} (See Note at end of list.)

Step 5: Cumulative cycle error is computed by adding the present error to E :

$$E \leftarrow \frac{1}{2} (d_k - o_k)^2 + E, \quad \text{for } k = 1, 2, \dots, K.$$

Step 6: If $p < P$, then $p \leftarrow p + 1$, $q \leftarrow q + 1$, and go to Step 3; otherwise, go to Step 7.

Step 7: The training cycle is completed.

For $E < E_{\max}$ terminate the training session.

Output weights \mathbf{W} , q , and E . If $E > E_{\max}$, then $E \leftarrow 0$, $p \leftarrow 1$, and initiate a new training cycle by going to Step 3.

■ **NOTE:** If formula (2.4a) is used in Step 3, then the weights are updated in Step 4 as follows:

$$\mathbf{w}_k \leftarrow \mathbf{w}_k + \eta (d_k - o_k) o_k (1 - o_k) \mathbf{y}, \quad \text{for } k = 1, 2, \dots, K$$

4.3

GENERALIZED DELTA LEARNING RULE

We now focus on generalizing the delta training rule for feedforward layered neural networks. The architecture of the two-layer network considered below is shown in Figure 4.7. It has, strictly speaking, two layers of processing neurons. If, however, the layers of nodes are counted, then the network can also be labeled as a three-layer network. The i 'th column of signals would be understood in such

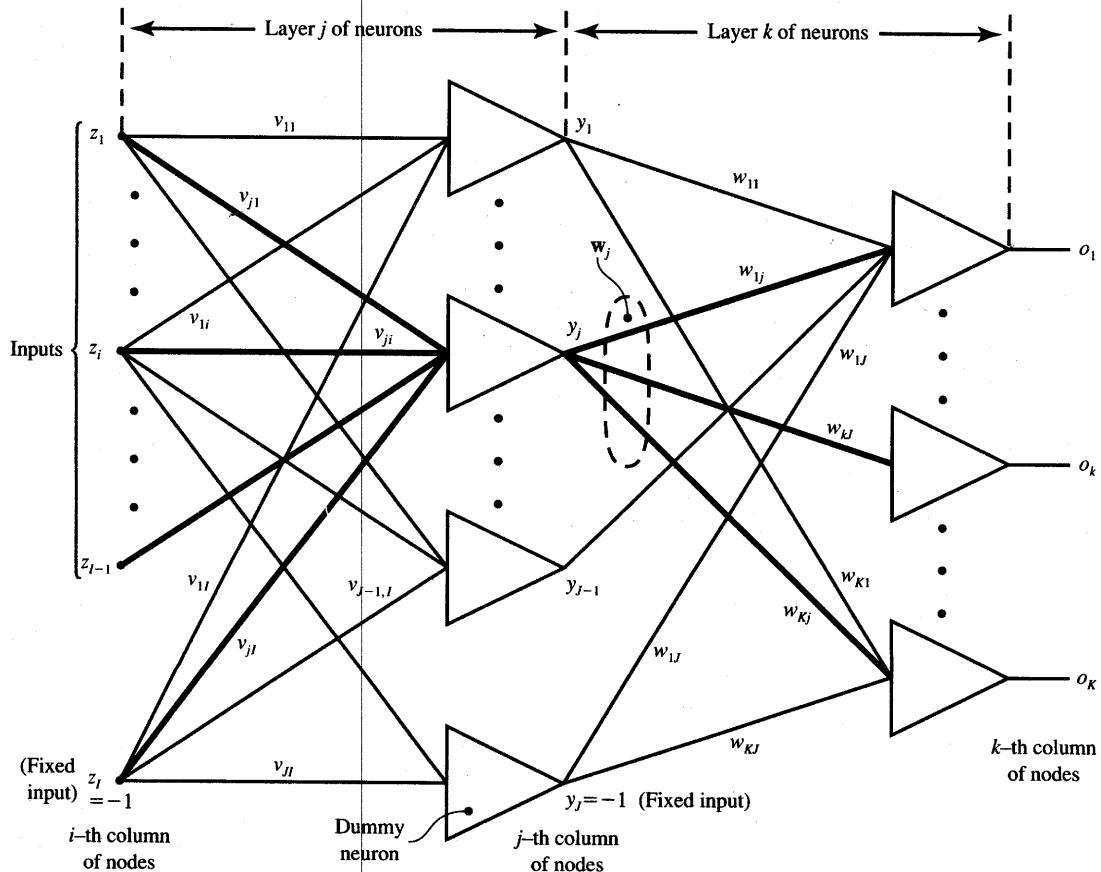


Figure 4.7 Layered feedforward neural network with two continuous perceptron layers.

a case to be an output layer response of a nonexistent (input) layer of neurons. We can thus refer to the architecture in Figure 4.7 as a two-neuron layer network, or three-node layer network.

There is no agreement in technical literature as to which approach is to be used to describe network architectures. In this text we will use the term "layer" in reference to the actual number of existing and processing neuron layers. Therefore, we will not count input terminals as layers. This convention is becoming more frequently adopted and seems more logical since the input nodes play no significant role in processing. Thus, the network of Figure 4.7 is a two-layer network. Let us also note that an N -layer network has $N - 1$ layers of neurons whose outputs are not accessible.

Layers with neurons whose outputs are not directly accessible are called *internal* or *hidden layers*. Thus, all but the output layer are hidden layers. Since the output of layer j of the neurons is not accessible from input and output, the

network from Figure 4.7 can be called a single hidden-layer network. In subsequent considerations, we will derive a general expression for the weight increment Δv_{ji} for any layer of neurons that is not an output layer. The reader may notice the change of subscripts. The computed weights lead now from node i toward node j as shown in the figure.

The negative gradient descent formula (4.5a) for the hidden layer now reads

$$\Delta v_{ji} = -\eta \frac{\partial E}{\partial v_{ji}}, \quad \text{for } j = 1, 2, \dots, J \text{ and } i = 1, 2, \dots, I \quad (4.21a)$$

and formula (4.7) becomes

$$\frac{\partial E}{\partial v_{ji}} = \frac{\partial E}{\partial (\text{net}_j)} \cdot \frac{\partial (\text{net}_j)}{\partial v_{ji}} \quad (4.21b)$$

Let us notice that the inputs to the layer are z_i , for $i = 1, 2, \dots, I$. Based on relation (4.8), the second term in the product (4.21b) is equal to z_i , and we may express the weight adjustment similarly to (4.10) as

$$\Delta v_{ji} = \eta \delta_{yj} z_i \quad (4.21c)$$

where δ_{yj} is the error signal term of the hidden layer having output y . This error signal term is produced by the j 'th neuron of the hidden layer, where $j = 1, 2, \dots, J$. The error signal term is equal to

$$\delta_{yj} \triangleq -\frac{\partial E}{\partial (\text{net}_j)}, \quad \text{for } j = 1, 2, \dots, J \quad (4.22)$$

In contrast to the output layer neurons' excitation net_k , which affected the k 'th neuron output only, the net_j contributes now to *every* error component in the error sum containing K terms specified in expression (4.4). The error signal term δ_{yj} at the node j can be computed as follows:

$$\delta_{yj} = -\frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial (\text{net}_j)} \quad (4.23a)$$

where

$$\frac{\partial E}{\partial y_j} = \frac{\partial}{\partial y_j} \left(\frac{1}{2} \sum_{k=1}^K \{d_k - f[\text{net}_k(y)]\}^2 \right) \quad (4.23b)$$

and, obviously, the second term of (4.23a) is equal to

$$\frac{\partial y_j}{\partial (\text{net}_j)} = f'_j(\text{net}_j) \quad (4.23c)$$

Routine calculations of (4.23b) result in

$$\frac{\partial E}{\partial y_j} = -\sum_{k=1}^K (d_k - o_k) \frac{\partial}{\partial y_j} \{f[\text{net}_k(y)]\} \quad (4.24a)$$

Calculation of the derivative in braces of expression (4.24a) yields

$$\frac{\partial E}{\partial y_j} = -\sum_{k=1}^K (d_k - o_k) f'(\text{net}_k) \frac{\partial (\text{net}_k)}{\partial y_j} \quad (4.24b)$$

We can simplify the above expression to the compact form as below by using expression (4.14) for δ_{ok} and (4.5b) for net_k .

$$\frac{\partial E}{\partial y_j} = - \sum_{k=1}^K \delta_{ok} w_{kj} \quad (4.24c)$$

Combining (4.23c) and (4.24c) results in rearranging δ_{yj} expressed in (4.23a) to the form

$$\delta_{yj} = f'_j(net_j) \sum_{k=1}^K \delta_{ok} w_{kj}, \quad \text{for } j = 1, 2, \dots, J \quad (4.25)$$

The weight adjustment (4.21c) in the hidden layer now becomes

$$\Delta v_{ji} = \eta f'_j(net_j) z_i \sum_{k=1}^K \delta_{ok} w_{kj}, \quad \text{for } j = 1, 2, \dots, J \text{ and} \\ i = 1, 2, \dots, I \quad (4.26a)$$

where the $f'_j(net_j)$ terms are to be computed either from (4.16d) or from (4.18b) as in case of simple delta rule training. Formula (4.26a) expresses the so-called *generalized delta learning rule*. The adjustment of weights leading to neuron j in the hidden layer is proportional to the weighted sum of all δ values at the adjacent following layer of nodes connecting neuron j with the output. The weights that fan out from node j are themselves the weighting factors. The weights affecting δ_{yj} of the j 'th hidden neuron have been highlighted in the output layer in Figure 4.7. All output layer errors $\delta_{ok} w_{kj}$, for $k = 1, 2, \dots, K$, contribute to the adjustment of highlighted weights v_{ji} , for $i = 1, 2, \dots, I$, of the hidden layer. The modified weights of the hidden layer can be expressed now as

$$v'_{ji} = v_{ji} + \eta f'_j(net_j) z_i \sum_{k=1}^K \delta_{ok} w_{kj}, \quad \text{for } j = 1, 2, \dots, J \text{ and} \\ i = 1, 2, \dots, I \quad (4.26b)$$

The hidden layer weight adjustment based on the generalized delta training rule for the network in Figure 4.7 can be succinctly stated in vector notation as

$$\mathbf{V}' = \mathbf{V} + \eta \mathbf{\delta}_y \mathbf{z}^t \quad (4.27)$$

where

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_I \end{bmatrix}, \\ \mathbf{V} = \begin{bmatrix} v_{11} & v_{12} & \cdots & v_{1I} \\ v_{21} & v_{22} & \cdots & v_{2I} \\ \vdots & \vdots & \cdots & \vdots \\ v_{J1} & v_{J2} & \cdots & v_{JI} \end{bmatrix}$$

and $\mathbf{\delta}_y$ is the column vector with entries δ_{yj} given by (4.25).

Defining now the j 'th column of matrix \mathbf{W} as \mathbf{w}_j , vector $\boldsymbol{\delta}_y$ can be expressed compactly as follows

$$\boldsymbol{\delta}_y = \mathbf{w}_j^T \boldsymbol{\delta}_o \mathbf{f}'_y \quad (4.28)$$

where \mathbf{f}'_y is the column vector with entries f'_{yj} expressed for each hidden layer neuron $1, 2, \dots, J$, for unipolar and bipolar activation functions, respectively, as

$$f'_{yj} = y_j(1 - y_j) \quad (4.29a)$$

$$f'_{yj} = \frac{1}{2}(1 - y_j^2) \quad (4.29b)$$

Vector $\boldsymbol{\delta}_o$ used in (4.28) is defined as in (4.20).

Comparison of the delta training rule (4.20) for adjusting the output layer weights and the generalized delta training rule (4.27) for adjusting the hidden layer weights indicate that both formulas are fairly uniform. The significant difference is in subscripts referring to the location of weights and input signals, and in the way the error signal vector $\boldsymbol{\delta}$ is computed. The vector $\boldsymbol{\delta}_o$ contains scalar entries (4.17) or (4.18a). Each component of vector $\boldsymbol{\delta}_o$ is simply the difference between the desired and actual output values times the derivative of the activation function. The vector $\boldsymbol{\delta}_y$, however, contains entries that are scalar products $\mathbf{w}_j^T \boldsymbol{\delta}_o f'_{yj}$ expressing the weighted sum of contributing error signals $\boldsymbol{\delta}_o$ produced by the following layer. The generalized delta learning rule propagates the error back by one layer, allowing the same process to be repeated for every layer preceding the discussed layer j . In the following section we will formalize the training method for layered neural networks.

4.4

FEEDFORWARD RECALL AND ERROR BACK- PROPAGATION TRAINING

As discussed in the previous section, the network shown in Figure 4.7 needs to be trained in a supervised mode. The training pattern vectors \mathbf{z} should be arranged in pairs with desired response vectors \mathbf{d} provided by the teacher. Let us look at a network feedforward operation, or recall. As a result of this operation the network computes the output vector \mathbf{o} .

Feedforward Recall

In general, the layered network is mapping the input vector \mathbf{z} into the output vector \mathbf{o} as follows

$$\mathbf{o} = N[\mathbf{z}] \quad (4.30)$$

where N denotes a composite nonlinear matrix operator. For the two-layer network shown, the mapping $\mathbf{z} \rightarrow \mathbf{o}$ as in (4.30) can be represented as a mapping within a mapping, or

$$\mathbf{o} = \Gamma[\mathbf{W}\Gamma[\mathbf{V}\mathbf{z}]], \quad (4.31a)$$

where the internal mapping is

$$\Gamma[\mathbf{V}\mathbf{z}] = \mathbf{y} \quad (4.31b)$$

and it relates to the hidden layer mapping $\mathbf{z} \rightarrow \mathbf{y}$. Note that the right arrows denote mapping of one space into another. Each of the mappings is performed by a single-layer of the layered network. The operator Γ is a nonlinear diagonal operator with diagonal elements being identical activation functions defined as in (4.3a). The diagonal elements of Γ operate on *net* values produced at inputs of each neuron. It follows from (4.31) that the $f(\cdot)$ arguments here are elements of vectors net_j and net_k for the hidden and output layers, respectively.

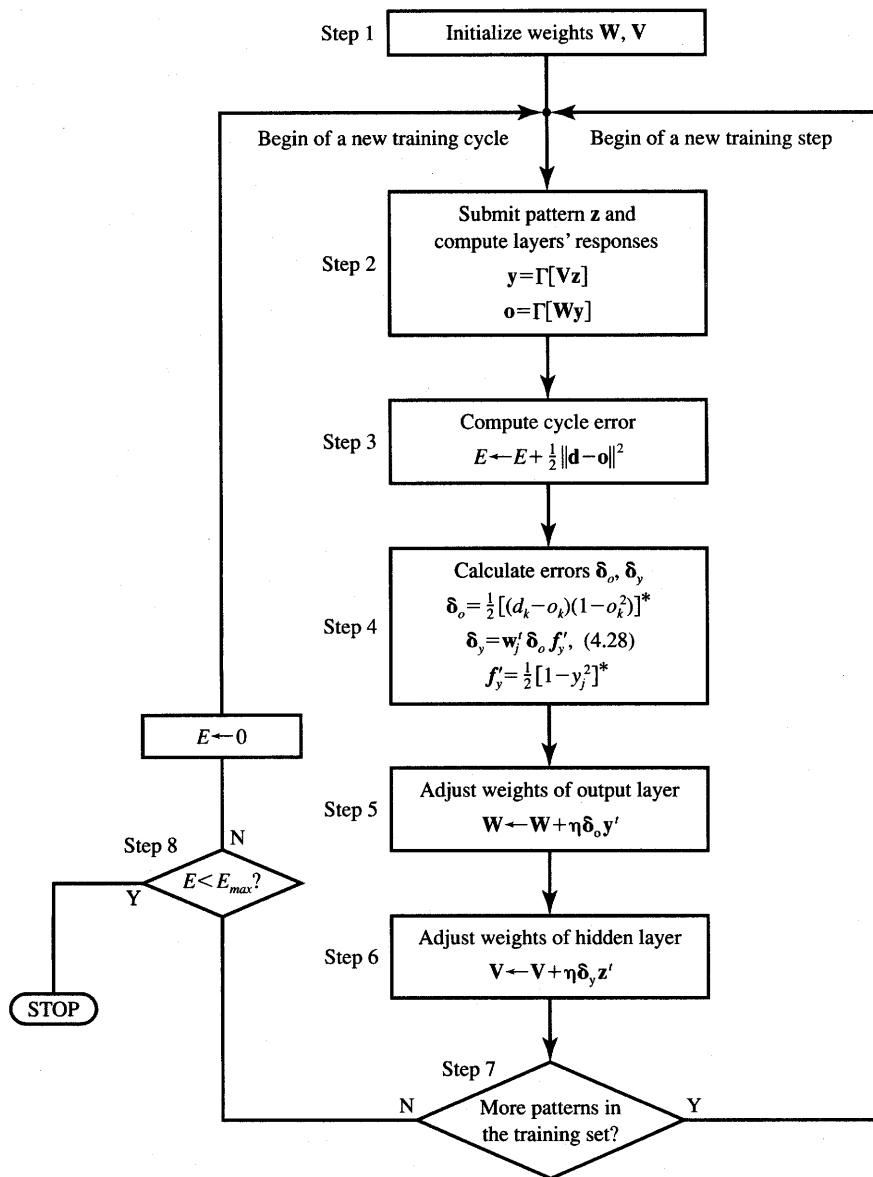
As can be seen from (4.31b), the assumption of identical and fixed activation functions $f(\text{net})$ leads to the conclusion that the only parameters for mapping $\mathbf{z} \rightarrow \mathbf{o}$ so that \mathbf{o} matches \mathbf{d} are weights. Specifically, we have two matrices \mathbf{V} and \mathbf{W} to be adjusted so that the error value proportional to $\|\mathbf{d} - \mathbf{o}\|^2$ is minimized. Thus, we can look at layered neural networks as versatile nonlinear mapping systems with weights serving as parameters (Narendra 1990).

Error Back-propagation Training

Figure 4.8(a) illustrates the flowchart of the error back-propagation training algorithm for a basic two-layer network as in Figure 4.7. The learning begins with the feedforward recall phase (Step 2). After a single pattern vector \mathbf{z} is submitted at the input, the layers' responses \mathbf{y} and \mathbf{o} are computed in this phase. Then, the error signal computation phase (Step 4) follows. Note that the error signal vector must be determined in the output layer first, and then it is propagated toward the network input nodes. The $K \times J$ weights are subsequently adjusted within the matrix \mathbf{W} in Step 5. Finally, $J \times I$ weights are adjusted within the matrix \mathbf{V} in Step 6.

Note that the cumulative cycle error of input to output mapping is computed in Step 3 as a sum over all continuous output errors in the entire training set. The final error value for the entire training cycle is calculated after each completed pass through the training set $\{\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_P\}$. The learning procedure stops when the final error value below the upper bound, E_{\max} , is obtained as shown in Step 8.

Figure 4.8(b) depicts the block diagram of the error back-propagation trained network operation and explains both the flow of signal, and the flow of error within the network. The feedforward phase is self-explanatory. The shaded portion of the diagram refers to the feedforward recall. The blank portion of the diagram refers to the training mode of the network. The back-propagation of



*If $f(\text{net})$ given by (2.4a) is used in Step 2, then in Step 4 use
 $\delta_o = [(d_k - o_k)(1 - o_k)o_k], f'_j = [(1 - y_j)y_j]$

(a)

Figure 4.8a Error back-propagation training (EBPT algorithm): (a) algorithm flowchart.

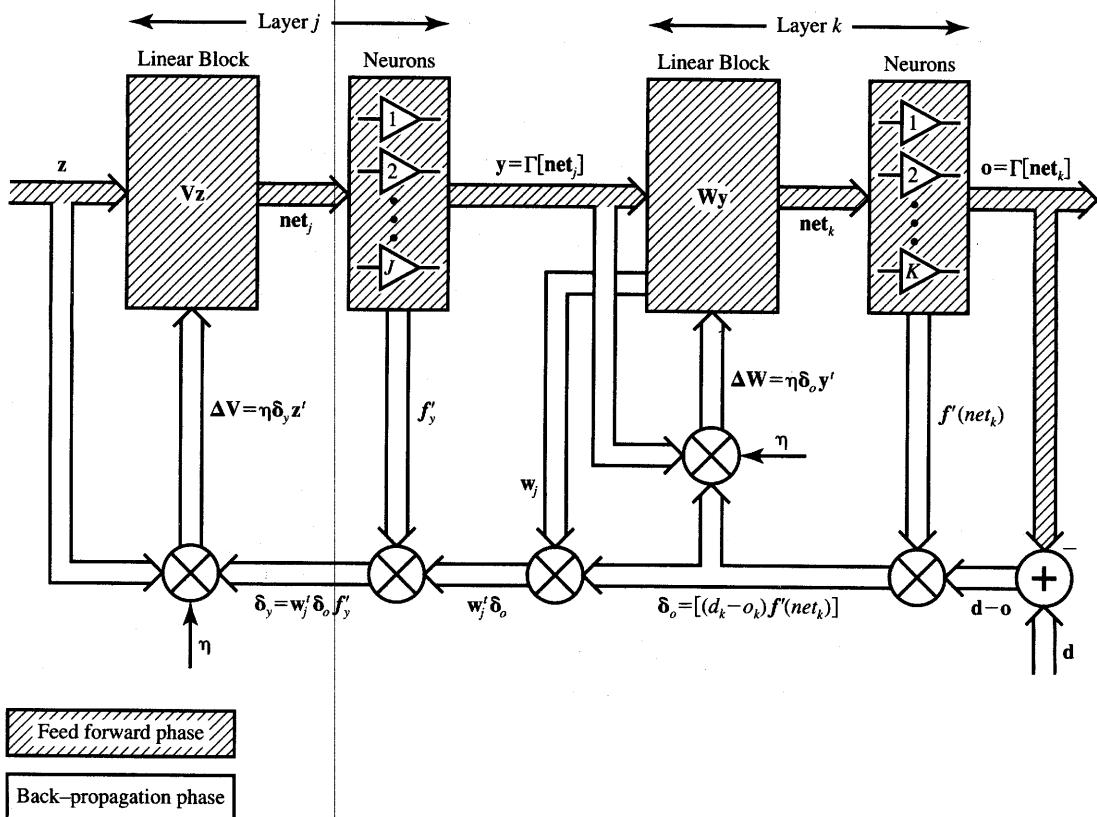


Figure 4.8b Error back-propagation training (EBPT algorithm) (continued): (b) block diagram illustrating forward and backward signal flow.

error $\mathbf{d} - \mathbf{o}$ from each output, for $k = 1, 2, \dots, K$, using the negative gradient descent technique is divided into functional steps such as calculation of the error signal vector δ_o and calculation of the weight matrix adjustment ΔW of the output layer. The diagram also illustrates the calculation of internal error signal vector δ_y and of the resulting weight adjustment ΔV of the input layer.

The algorithm of error back-propagation training is given below.

Summary of the Error Back-Propagation Training Algorithm (EBPTA)

Given are P training pairs

$$\{\mathbf{z}_1, \mathbf{d}_1, \mathbf{z}_2, \mathbf{d}_2, \dots, \mathbf{z}_P, \mathbf{d}_P\},$$

where \mathbf{z}_i is $(I \times 1)$, \mathbf{d}_i is $(K \times 1)$, and $i = 1, 2, \dots, P$. Note that the I 'th component of each \mathbf{z}_i is of value -1 since input vectors have been augmented. Size $J - 1$ of the hidden layer having outputs \mathbf{y} is

selected. Note that the J 'th component of \mathbf{y} is of value -1 , since hidden layer outputs have also been augmented; \mathbf{y} is $(J \times 1)$ and \mathbf{o} is $(K \times 1)$.

Step 1: $\eta > 0$, E_{\max} chosen.

Weights \mathbf{W} and \mathbf{V} are initialized at small random values; \mathbf{W} is $(K \times J)$, \mathbf{V} is $(J \times I)$.

$$q \leftarrow 1, p \leftarrow 1, E \leftarrow 0$$

Step 2: Training step starts here (See Note 1 at end of list.)

Input is presented and the layers' outputs computed [$f(\text{net})$ as in (2.3a) is used]:

$$\mathbf{z} \leftarrow \mathbf{z}_p, \mathbf{d} \leftarrow \mathbf{d}_p$$

$$y_j \leftarrow f(\mathbf{v}_j^t \mathbf{z}), \quad \text{for } j = 1, 2, \dots, J$$

where \mathbf{v}_j , a column vector, is the j 'th row of \mathbf{V} , and

$$o_k \leftarrow f(\mathbf{w}_k^t \mathbf{y}), \quad \text{for } k = 1, 2, \dots, K$$

where \mathbf{w}_k , a column vector, is the k 'th row of \mathbf{W} .

Step 3: Error value is computed:

$$E \leftarrow \frac{1}{2}(d_k - o_k)^2 + E, \quad \text{for } k = 1, 2, \dots, K$$

Step 4: Error signal vectors δ_o and δ_y of both layers are computed.

Vector δ_o is $(K \times 1)$, δ_y is $(J \times 1)$. (See Note 2 at end of list.)

The error signal terms of the output layer in this step are

$$\delta_{ok} = \frac{1}{2}(d_k - o_k)(1 - o_k^2), \quad \text{for } k = 1, 2, \dots, K$$

The error signal terms of the hidden layer in this step are

$$\delta_{yj} = \frac{1}{2}(1 - y_j^2) \sum_{k=1}^K \delta_{ok} w_{kj}, \quad \text{for } j = 1, 2, \dots, J$$

Step 5: Output layer weights are adjusted:

$$w_{kj} \leftarrow w_{kj} + \eta \delta_{ok} y_j, \quad \text{for } k = 1, 2, \dots, K \text{ and} \\ j = 1, 2, \dots, J$$

Step 6: Hidden layer weights are adjusted:

$$v_{ji} \leftarrow v_{ji} + \eta \delta_{yj} z_i, \quad \text{for } j = 1, 2, \dots, J \text{ and} \\ i = 1, 2, \dots, I$$

Step 7: If $p < P$ then $p \leftarrow p + 1$, $q \leftarrow q + 1$, and go to Step 2; otherwise, go to Step 8.

Step 8: The training cycle is completed.

For $E < E_{\max}$ terminate the training session. Output weights \mathbf{W} , \mathbf{V} , q , and E .

If $E > E_{\max}$, then $E \leftarrow 0$, $p \leftarrow 1$, and initiate the new training cycle by going to Step 2.

■ **NOTE 1** For best results, patterns should be chosen at random from the training set (justification follows in Section 4.5).

■ **NOTE 2** If formula (2.4a) is used in Step 2, then the error signal terms in Step 4 are computed as follows

$$\delta_{ok} = (d_k - o_k)(1 - o_k)o_k, \quad \text{for } k = 1, 2, \dots, K$$

$$\delta_{yj} = y_j(1 - y_j) \sum_{k=1}^K \delta_{ok} w_{kj}, \quad \text{for } j = 1, 2, \dots, J$$

Several aspects of the error back-propagation training method are noteworthy. The incremental learning of the weight matrix in the output and hidden layers is obtained by the outer product rule as

$$\Delta \mathbf{W} = \eta \boldsymbol{\delta} \mathbf{y}^t$$

where $\boldsymbol{\delta}$ is the error signal vector of a layer and \mathbf{y} is the input signal vector to that layer. Noticeably, the error signal components δ_{ok} at the output layer are obtained as simple scalar products of the output error component $d_k - o_k$ and $f'(net_k)$. In contrast to this mode of error computation, hidden-layer error signal components δ_{yj} are computed with the weight matrix \mathbf{W} seen in the feedforward mode, but now using its columns \mathbf{w}_j . As we realize, the feedforward mode involves rows of matrix \mathbf{W} for the computation of the following layer's response.

Another observation can be made regarding linear versus nonlinear operation of the network during training and recall phases. Although the network is nonlinear in the feedforward mode, the error back-propagation is computed using the linearized activation function. The linearization is achieved by extracting the slope $f'(net)$ at each neuron's operating point and using it for back-transmitted error signal scaling.

Example of Error Back-Propagation Training

The following example demonstrates the main features of the error back-propagation training algorithm applied to a simple two-layer network.

EXAMPLE 4.2

In this example we will perform a single training step for the three-neuron, two-layer network shown in Figure 4.9(a). Let us note that the original network has been augmented in the figure with dummy neurons 1 and 2 to conform to the general network structure as illustrated in Figure 4.7. Obviously, the virtual inputs to the network have to be assigned values o_1 and o_2 , which are the responses of the dummy neurons. A single two-component training pattern is assumed to be present at inputs o_1 and o_2 . In addition, dummy neuron 0 with a fixed output of -1 has been added to generate the weights w_{30} , w_{40} , and w_{50} corresponding to the thresholds T_3 , T_4 , and T_5 , respectively, for actual neurons 3, 4, and 5. Its only function is to augment the input vectors to each neuron by a fixed bias component.

For the sake of clarity and to avoid identical indices for different weights in this example, the hidden layer neurons in column j have been numbered 3 and 4, and the output neuron has been numbered 5. The learning starts after all weights have been randomly initialized. Inputs o_1 and o_2 are presented and o_3 , o_4 , and o_5 are computed with $o_0 = -1$. This feedforward recall phase more than likely results in a nonzero error E defined as in (4.4).

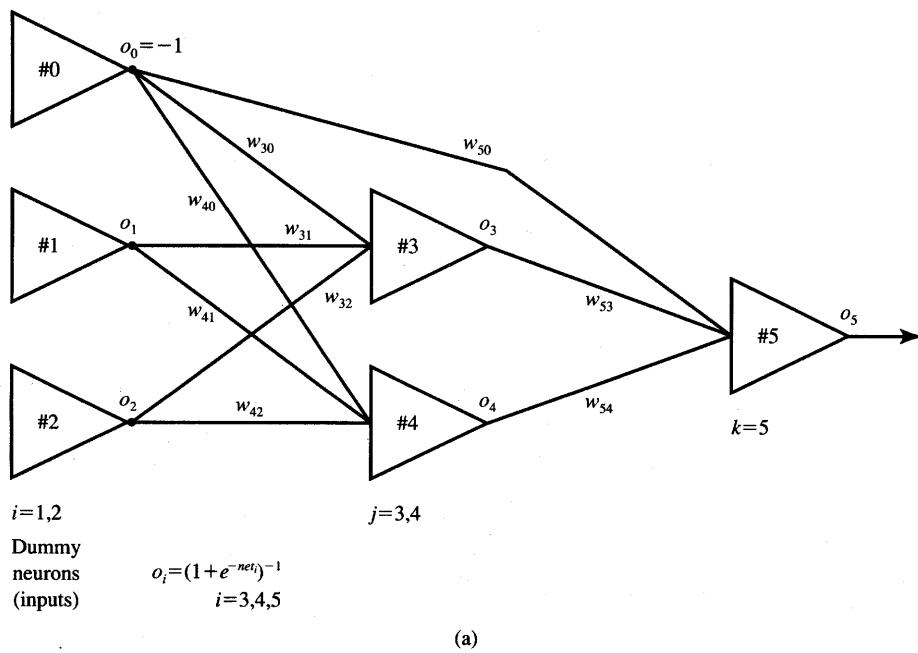


Figure 4.9a Figure for Example 4.2: (a) network diagram.

The error signal term value δ_5 can now be computed using (4.14), or for the selected activation function using formula (4.17) as follows:

$$\delta_5 = (d_5 - o_5)o_5(1 - o_5)$$

The generalized delta values δ_3 and δ_4 as in (4.25) need to be expressed for hidden layer neurons 3 and 4:

$$\begin{aligned}\delta_3 &= f'_3(\text{net}_3) \sum_{k=5}^5 \delta_k w_{k3} \\ \delta_4 &= f'_4(\text{net}_4) \sum_{k=5}^5 \delta_k w_{k4}\end{aligned}\quad (4.32a)$$

Using (4.29a) yields the final expressions for δ_3 and δ_4 as

$$\begin{aligned}\delta_3 &= o_3(1 - o_3)\delta_5 w_{53} \\ \delta_4 &= o_4(1 - o_4)\delta_5 w_{54}\end{aligned}\quad (4.32b)$$

Now the weight vector corrections Δw_{kj} and Δw_{ji} can be computed for the selected η value. Output layer weight adjustments computed from (4.10) are

$$\begin{aligned}\Delta w_{50} &= -\eta\delta_5 \\ \Delta w_{53} &= \eta\delta_5 o_3 \\ \Delta w_{54} &= \eta\delta_5 o_4\end{aligned}\quad (4.33a)$$

The adjustments of hidden-layer weights associated with the neurons 3 and 4 computed from (4.21c) or (4.26a) are, respectively,

$$\begin{aligned}\Delta w_{30} &= -\eta\delta_3 \\ \Delta w_{31} &= \eta\delta_3 o_1 \\ \Delta w_{32} &= \eta\delta_3 o_2\end{aligned}\quad (4.33b)$$

and

$$\begin{aligned}\Delta w_{40} &= -\eta\delta_4 \\ \Delta w_{41} &= \eta\delta_4 o_1 \\ \Delta w_{42} &= \eta\delta_4 o_2\end{aligned}\quad (4.33c)$$

Superposition of the computed weight adjustments (4.33) and of the present weights w_{kj} and w_{ji} results in the updated weight values. This terminates the single supervised learning step based on the single pattern vector from the training set. Subsequently, the next pattern vector is submitted and the training step repeated. The training steps proceed until all patterns in the training set are exhausted. This terminates the complete training cycle understood as the sweep through the sequence of P training steps, where P denotes the number of training patterns in the training set.

The cumulative cycle error is computed for the complete training cycle using expression (4.4). It is then compared with the maximum error allowed.

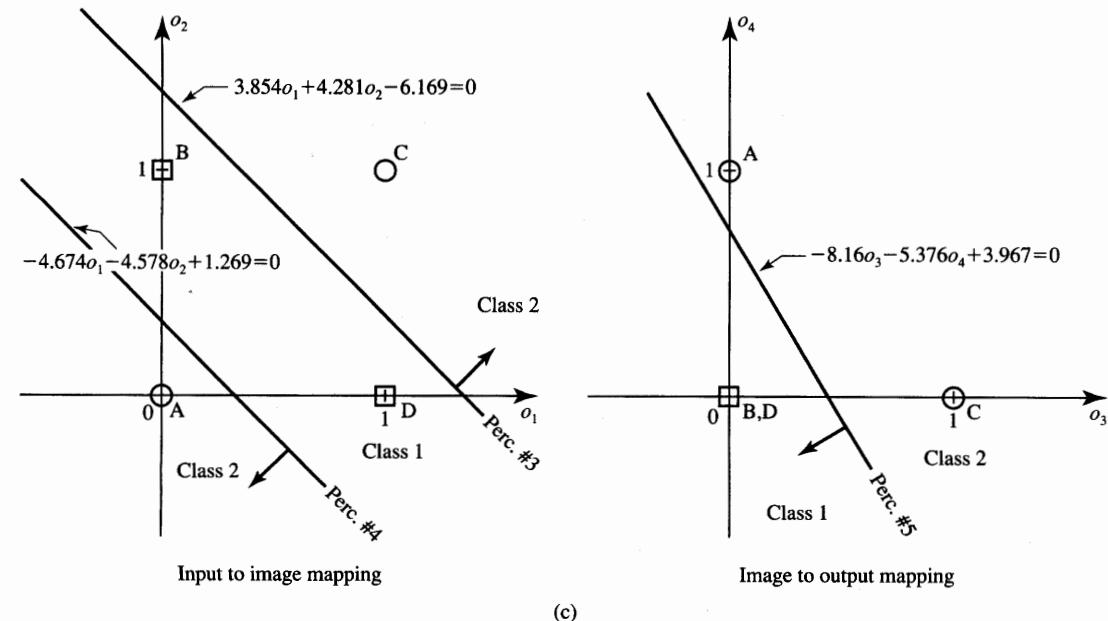
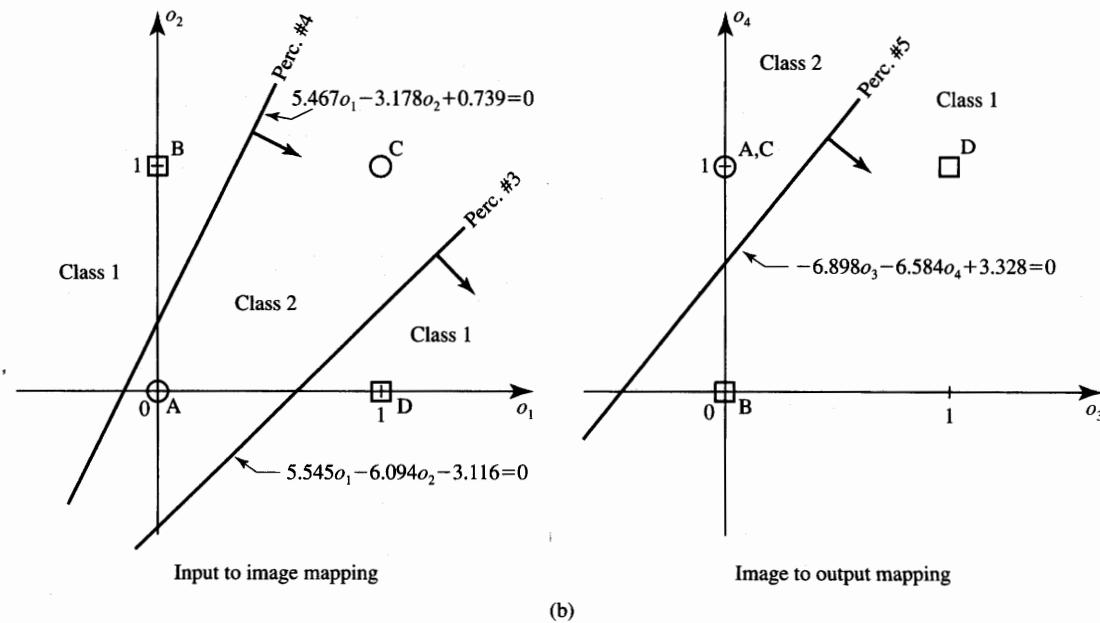


Figure 4.9b,c Figure for Example 4.2 (continued): (b) space transformations, run 1, and (c) space transformations, run 2.

The training cycles repeat until the cycle error drops below the specified maximum error value. If the network has failed to learn the training set successfully, the training should be restarted with different initial weights. If this also fails, other remedies need to be taken to meet the training requirements. Such steps are discussed in Section 4.5 in more detail.

The network covered in this example can also be used to demonstrate the classification of linearly nonseparable patterns. Due to its size, it can be used directly to solve the XOR problem for two variables. Network training has been simulated and two sample runs with random initial weight values are summarized below for $\eta = 0.1$. Matrices \mathbf{W} and \mathbf{V} are defined for this network as

$$\mathbf{W} \triangleq \begin{bmatrix} w_{50} & w_{53} & w_{54} \end{bmatrix}$$

$$\mathbf{V} \triangleq \begin{bmatrix} w_{30} & w_{31} & w_{32} \\ w_{40} & w_{41} & w_{42} \end{bmatrix}$$

The resulting weight matrices obtained for run 1 (1244 steps) with random initial weights are:

$$\mathbf{W}_1 = \begin{bmatrix} -3.328 & 6.898 & -6.584 \end{bmatrix}$$

$$\mathbf{V}_1 = \begin{bmatrix} 3.116 & 5.545 & -6.094 \\ -0.739 & 5.467 & -3.178 \end{bmatrix}$$

The resulting weight matrices obtained for run 2 (2128 steps) with another random set of initial weights are:

$$\mathbf{W}_2 = \begin{bmatrix} -3.967 & -8.160 & -5.376 \end{bmatrix}$$

$$\mathbf{V}_2 = \begin{bmatrix} 6.169 & 3.854 & 4.281 \\ -1.269 & -4.674 & -4.578 \end{bmatrix}$$

Since we require that the network from Figure 4.9(a) to function as a classifier with binary outputs, let us preserve all its weights as computed above. Continuous perceptrons used for training, however, need to be replaced with bipolar binary neurons. This will result in a binary-valued response that provides the required class numbers of values 1 or 0 for classes 1 and 2, respectively.

Based on the result obtained, we can analyze the implemented mapping of input-to-image-to-output space. As a result of run 1, perceptrons 3 and 4 implement the following mapping to image space o_1 and o_2 , respectively,

$$5.545o_1 - 6.094o_2 - 3.116 = 0$$

$$5.467o_1 - 3.178o_2 + 0.739 = 0$$

The generated decision lines are shown in Figure 4.9(b). The reader can verify that the decision lines are obtained using the rows of computed matrix \mathbf{V} , and also under the assumption that net_3 and net_4 of zero value

determine the respective partitioning lines. In addition, perceptron 5 performs the image-to-output space mapping and the equation of the class decision line is as follows:

$$6.898o_3 - 6.584o_4 + 3.328 = 0$$

The line is also shown in the figure. It can be seen that mapped patterns B,D and A,C are positioned at different sides of the decision line, which is generated by the output perceptron in the image space. This part of the example demonstrates how linearly nonseparable original patterns can be correctly classified by this classifier. Interestingly, these results have been produced as a result of error back-propagation network training.

Inspecting the results of run 2, the reader can easily verify that the decision surfaces generated by perceptrons 3, 4, and 5 are as shown in Figure 4.9(c). Markedly, it can be seen that the results of run 2 are very similar to those obtained in Example 4.1 without training the classifier network. Reviewing the mappings from Figures 4.9(b) and (c), note that they both merge patterns of the same class to the same compartment in the image space. For the decision lines as shown and an activation function with $\lambda = 1$ used for training the network, the output values close to 0 or 1 need to be thresholded to values of 0 or 1 to yield binary classification response. As an alternative, λ can be increased and outputs, in limit, will approach the 0, 1 levels.

Training Errors

Every supervised training algorithm covered so far in the text, including the error back-propagation training algorithm, involves the reduction of an error value. For the purpose of weight adjustment in a single training step, the error to be reduced is usually that computed only for a pattern currently applied at the input of the network. For the purpose of assessing the quality and success of the training, however, the joint error must be computed for the entire batch of training patterns. The discussion below addresses the main definitions of errors used for evaluation of neural network training.

Note that the *cumulative error* is computed over the error back-propagation training cycle (Step 3, EBPTA), and it is expressed as a quadratic error

$$E = \frac{1}{2} \sum_{p=1}^P \sum_{k=1}^K (d_{pk} - o_{pk})^2 \quad (4.34)$$

This error is a sum of P errors computed for single patterns using formula (4.4). It can be seen that the error of Eq. (4.34) depicts the accuracy of the neural network mapping after a number of training cycles have been implemented. Such definition of error, however, is not very useful for comparison of networks with

different numbers of training patterns P and having a different number of output neurons. Networks with the same number of outputs K when trained using large numbers of patterns in the training set will usually produce large cumulative errors (4.34) due to the large number of terms in the sum. For similar reasons, networks with large K trained using the same training set would usually also produce large cumulative errors. Thus, a more adequate error measure can be introduced as in (4.35):

$$E_{\text{rms}} = \frac{1}{PK} \sqrt{\sum_{p=1}^P \sum_{k=1}^K (d_{pk} - o_{pk})^2} \quad (4.35)$$

The value E_{rms} has the sense of a *root-mean-square normalized error*, and it seems to be more descriptive than E as in (4.34) when comparing the outcome of the training of different neural networks among each other.

In some applications, the networks' continuous responses are of significance and thus any of the discussed error measures E and E_{rms} bear useful information. The degree of association or the accuracy of mapping can be measured by these continuous error measures. In other applications, however, the neurons' responses are assigned binary values after the thresholding. These applications include classifier networks. For example, all unipolar neurons responding below 0.1 and above 0.9 can be considered as approximating binary responses 0 and 1, respectively.

Assuming that the network is trained as a classifier, usually all the desired output values can be set to zero except for the one corresponding to the class the input pattern is from. That desired output value is set to 1. In such cases, the *decision error*, rather than the continuous response errors, more adequately reflects the accuracy of neural network classifiers. The decision error can be defined as

$$E_d = \frac{N_{\text{err}}}{PK} \quad (4.36)$$

where N_{err} is the total number of bit errors resulting at K thresholded outputs over the complete training cycle. Note that networks in classification applications may perform as excellent classifiers and exhibit zero decision errors while still yielding substantial E and E_{rms} errors.

Multilayer Feedforward Networks as Universal Approximators

Although classification is a very important form of neural computation, the binary response only of neural networks would seriously limit their mapping potential. Our focus in this section is to study the performance of multilayer feed-forward networks as universal approximators. As we discussed early in Section 2.4, the problem of finding an approximation of a multivariable function $h(\mathbf{x})$ can

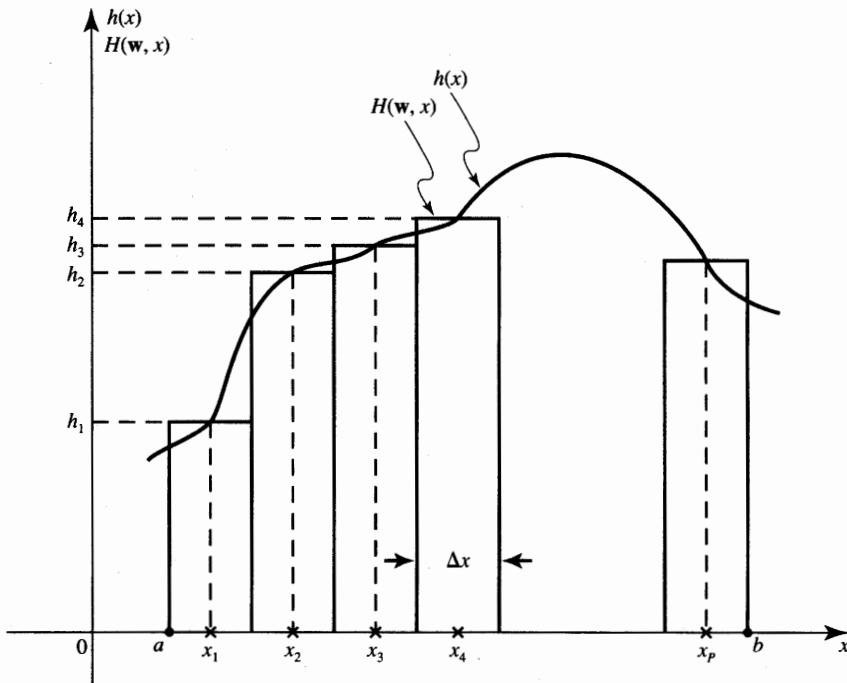


Figure 4.10 Approximation of $h(x)$ with staircase function $H(\mathbf{w}, x)$.

be approached through supervised training of an input-output mapping from a set of examples. The learning proceeds as a sequence of iterative weight adjustments until the solution weight vector \mathbf{w}^* is found that satisfies the minimum distance criterion (2.26).

An example function $h(x)$ to be approximated is illustrated in Figure 4.10. Assume that P samples of the function are known on a set of arguments $\{x_1, x_2, \dots, x_P\}$. The samples are simply examples of function values in the interval (a, b) . It is assumed for simplicity that the example arguments are uniformly distributed between a and b and are Δx apart, i.e.

$$x_{i+1} - x_i = \Delta x = \frac{b - a}{P}, \quad \text{for } i = 1, 2, \dots, P$$

We note that the interval (a, b) is divided into P equal intervals of length Δx defined as

$$\left(x_i - \frac{\Delta x}{2}, x_i + \frac{\Delta x}{2} \right), \quad \text{for } i = 1, 2, \dots, P$$

where

$$x_1 - \frac{\Delta x}{2} = a, \quad x_P + \frac{\Delta x}{2} = b$$

The function values $h_i = h(x_i)$ at argument values determine the height of each of the P rectangles depicted on the figure. This way we obtain a staircase approximation $H(\mathbf{w}, x)$ of the continuous-valued function $h(x)$. Let us define a function $\zeta(x)$ such that

$$\zeta(x) = \frac{1}{2} \operatorname{sgn} x + \frac{1}{2} = \begin{cases} 0 & \text{for } x < 0 \\ \text{undefined} & \text{for } x = 0 \\ 1 & \text{for } x > 0 \end{cases} \quad (4.37)$$

The function $\zeta(x)$ is called the *unit step function*. The staircase approximation can be expressed using the unit step functions as follows

$$H(\mathbf{w}, x) = h_1 \left[\zeta\left(x - x_1 + \frac{\Delta x}{2}\right) - \zeta\left(x - x_1 - \frac{\Delta x}{2}\right) \right] + \dots + h_P \left[\zeta\left(x - x_P + \frac{\Delta x}{2}\right) - \zeta\left(x - x_P - \frac{\Delta x}{2}\right) \right] \quad (4.38a)$$

or, briefly

$$H(\mathbf{w}, x) = \sum_{i=1}^P h_i \left[\zeta\left(x - x_i + \frac{\Delta x}{2}\right) - \zeta\left(x - x_i - \frac{\Delta x}{2}\right) \right] \quad (4.38b)$$

We note that each term in brackets in (4.38b) is a unity height window of width Δx centered at x_i as shown by the continuous line in Figure 4.11(a). This window expressed in terms of the $\operatorname{sgn}(\cdot)$ function using (4.37) can be rewritten as

$$\zeta\left(x - x_i + \frac{\Delta x}{2}\right) - \zeta\left(x - x_i - \frac{\Delta x}{2}\right) = \frac{1}{2} \operatorname{sgn}\left(x - x_i + \frac{\Delta x}{2}\right) - \frac{1}{2} \operatorname{sgn}\left(x - x_i - \frac{\Delta x}{2}\right) \quad (4.39)$$

Figure 4.11(b) illustrates how to implement Equation (4.39) by using two TLU elements with appropriate thresholds and summing their properly weighted responses. It can thus be seen that two binary perceptrons are needed to produce a single window. We may notice that a network with $2P$ individual TLU elements similar to the one shown in Figure 4.11(b) can be used to implement the staircase approximation (4.38). An example of such a network is illustrated in Figure 2.9(a).

Note that the network of Figure 4.11(b) has two constant inputs x_i and -1 . Merging them would result in a functionally identical network, which is shown in Figure 4.12(a). Binary response units TLU #1 and TLU #2 are now replaced with the bipolar continuous perceptrons as shown in Figure 4.12(b). As a result, the mapping of input x into output o takes the form of a single bump centered at x_i rather than of the rectangular window. The bump is shown in Figure 4.11(a) by the dashed line. Let us also note that with the increasing steepness factor λ , the dashed line is getting closer to the continuous one, and the bump approaches the rectangular window. In fact, even the increase of λ is not necessary because it can be replaced with scaling the weight up by using the multiplicative factor λ .

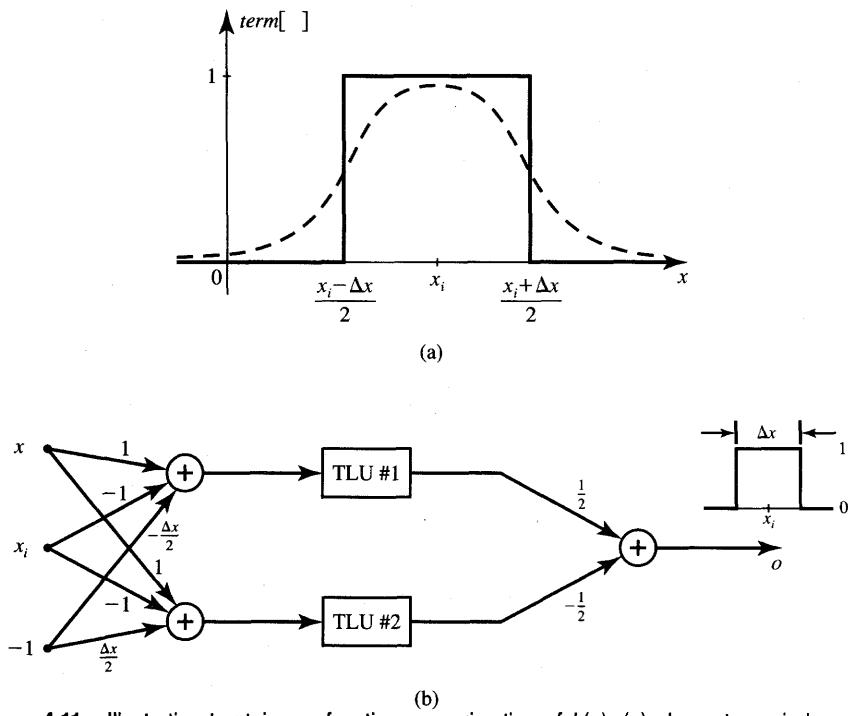


Figure 4.11 Illustration to staircase function approximation of $h(x)$: (a) elementary window and (b) window implementation using two TLUs.

for the weights shown in Figure 4.12(b). In two or more dimensions, we could add one function producing a bump for each dimension. Figure 2.9(c) provides here an appropriate illustration. We could also combine peaks as in (4.38) for a staircase approximation of a multivariable function $H(\mathbf{w}, \mathbf{x})$. Although the bump approach may not be the best for a particular problem, it is intended only as proof of existence.

The preliminary and rather nonrigorous considerations above indicate that a sufficient number of continuous neurons can implement a finite sum of localized bumps spread over the multidimensional domain of the argument space \mathbf{x} . The construction presented indicates that multivariable functions $h(\mathbf{x})$ can be modeled by two-layer networks. Whether or not the approximations are learnable and the approximating weights \mathbf{w}^* can be found by training remains an open question. We also do not know exactly what may be the best network architectures and whether or not the number of $2P$ units in the first layer is adequate or if the number is excessive.

In a more formal approach, networks of the type shown in Figure 4.12 can be used to map R^n into R by using P examples of the function $h(\mathbf{x})$ to be

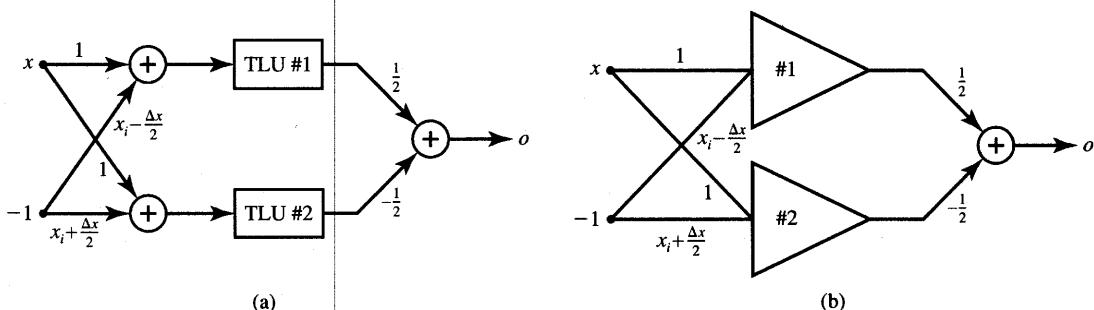


Figure 4.12 Networks implementing the window function of Eq. (4.39): (a) with binary term (4.39) and (b) with continuous neurons [dashed line in Figure 4.11(b)].

approximated by performing the nonlinear mapping with continuous neurons in the first layer as follows

$$\mathbf{y} = \Gamma[\mathbf{V}\mathbf{x}] \quad (4.40a)$$

and by computing the linear combination by the single node of the output layer

$$o = H(\mathbf{x}) = \mathbf{w}'\mathbf{y} \quad (4.40b)$$

Although the concept of nonlinear mapping (4.40a) followed by linear mapping (4.40b) pervasively demonstrates the approximating potential of neural networks, most theoretical and practical technical reports deal with the second layer also providing the nonlinear mapping (Poggio and Girosi 1990; Funanashi 1989; Hornik, Stinchcombe, and White 1989).

The general network architecture performing the nested nonlinear scheme was shown earlier in Figure 4.7. It consists of a single hidden layer and it implements K mappings, each mapping being the component of \mathbf{o}_k

$$\mathbf{o}_k = \Gamma[\mathbf{W}\Gamma[\mathbf{V}\mathbf{o}]] \quad (4.41)$$

This standard class of neural networks architecture can approximate virtually any multivariable function of interest, provided sufficiently many hidden neurons are available. The studies of Funanashi (1989), Hornik, Stinchcombe, and White (1989) prove that multilayer feedforward networks perform as a class of universal approximators. The results also provide a fundamental basis for establishing the ability of multilayer feedforward networks to learn the connection strengths that achieve the desired accuracy of the approximation. The reader is referred to the specialized literature for more details (Hornik, Stinchcombe, and White 1989; Funanashi 1989; Poggio and Girosi 1990).

Failures in approximation application can, in general, be attributed to inadequate learning. The reasons for failure quoted in the literature are an inadequate

(too low or too high) number of hidden neurons, or the lack of a deterministic relationship between the input and target output values used for training. If functions to be approximated are not bounded and $h(\mathbf{x})$ cannot be properly scaled, the use of the linear mapping by the second layer as in (4.40b) may offer the solution. Rather than assuming the ordinary 'sigmoidal' activation functions, simple identity activation functions of the output neurons can be used in such cases.

The function values that need to be used as desired output values during training of the network from Figure 4.7 with neurons' response bounded must also be bounded. The range of function values is $(-1, 1)$ for bipolar continuous activation functions and $(0, 1)$ for unipolar continuous activation functions. Several practical demonstrations of neural network approximation capability are presented in Chapter 8 in relation to their applications to robotics and control systems. The example below discusses the main features of the function approximation issues.

EXAMPLE 4.3

In this example we will review an application of a three-neuron network for approximation of a two-variable function. The network to be trained has the architecture shown in Figure 4.9(a). Bipolar continuous activation functions are used for the network training. Weight matrices \mathbf{W} and \mathbf{V} are defined as in Example 4.2.

Let us attempt to train the network to compute the length of a planar vector with components o_1 and o_2 . This yields the desired value for o_5 as follows:

$$d = \sqrt{o_1^2 + o_2^2} \quad (4.42)$$

Using (4.41) we can write

$$o_5 = \Gamma[\mathbf{W}\Gamma[\mathbf{V}\mathbf{o}]] \quad (4.43)$$

where

$$\mathbf{o} = [-1 \quad o_1 \quad o_2]^t$$

The weights w_{kj} and v_{ji} need to be adjusted during training. The input domain for training has been chosen to be the first quadrant of the plane o_1, o_2 with $0 < o_i < 0.7$, for $i = 1, 2$.

In the first experiment only 10 training points have been selected. They have been uniformly spread only in the lower half of the first quadrant. The network has reached an acceptable error level established at 0.01 after 2080 training steps with $\eta = 0.2$. Matrices \mathbf{W} and \mathbf{V} obtained from the

simulation are

$$\mathbf{W} = \begin{bmatrix} 0.03 \\ 3.66 \\ 2.73 \end{bmatrix} \quad (4.44)$$

$$\mathbf{V} = \begin{bmatrix} -1.29 & -3.04 & -1.54 \\ 0.97 & 2.61 & 0.52 \end{bmatrix}$$

To check how the network has learned the examples from the training set, the complete surface constructed by the trained network has been generated. The magnitude of the mapping error computed as $|d - o_5|$ is illustrated in Figure 4.13(a). Magnitudes of error associated with each training pattern are also shown on the surface.

The results produced by the network indicate that the mapping of the function has been learned much better within the training domain than outside it. Thus, any generalization provided by the trained network beyond the training domain remains questionable. This is particularly vivid for the left upper corner of the quadrant where the discrepancy between the accurate function value and its approximation reaches 0.45. Also, patterns that are clustered in the training domain are mapped more accurately. Therefore, the recall of the training patterns that are at the borders of the training area does not yield as low an error as recalled patterns, which are located inside it.

Figure 4.13(b) provides more detailed insight into the mapping property of this network. It displays the error $d - o_5$ in a form of the contour map. The map indicates that an ideal approximation contour for $d = o_5$ would be located somewhere between the contours -0.02 and 0.01 . Note that the 10 training points are marked as circles on the graph.

The training experiment was then repeated for the same architecture but with 64 training points now densely covering the entire quadrant domain. The network has reached an error level of value 0.02 after 1200 training steps with $\eta = 0.4$. The weights obtained as a result of training are

$$\mathbf{W} = \begin{bmatrix} -3.47 & -1.8 & 2.07 \end{bmatrix}$$

$$\mathbf{V} = \begin{bmatrix} -2.54 & -3.68 & 0.61 \\ 2.76 & 0.07 & 3.83 \end{bmatrix} \quad (4.45)$$

The mapping performed by the network is shown in Figure 4.14(a). This figure shows the result of the surface reconstruction and has been obtained as a series of numerous recalls for points o_1 and o_2 densely covering the first quadrant, which is our domain of interest. Figure 4.14(b) provides the detailed contour error map of the error $d - o_5$ for the designed network. The training points for this experiment have been selected at the intersection of the mesh. The mapping is reasonably accurate in the entire domain; however, it tends to get worse for points approaching the boundaries of the training domain.

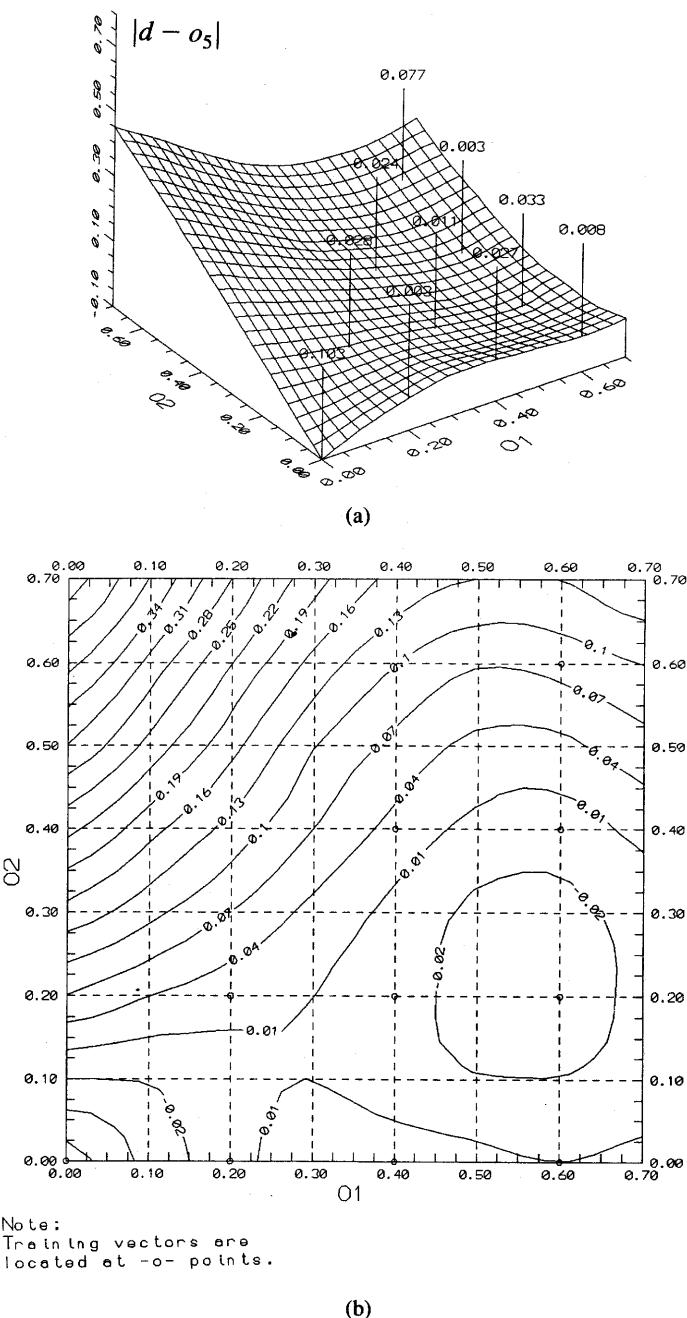


Figure 4.13 Continuous function approximation error, 10 training points, network with three neurons: (a) $|d - o_5|$ training points shown and (b) contour map of $d - o_5$.

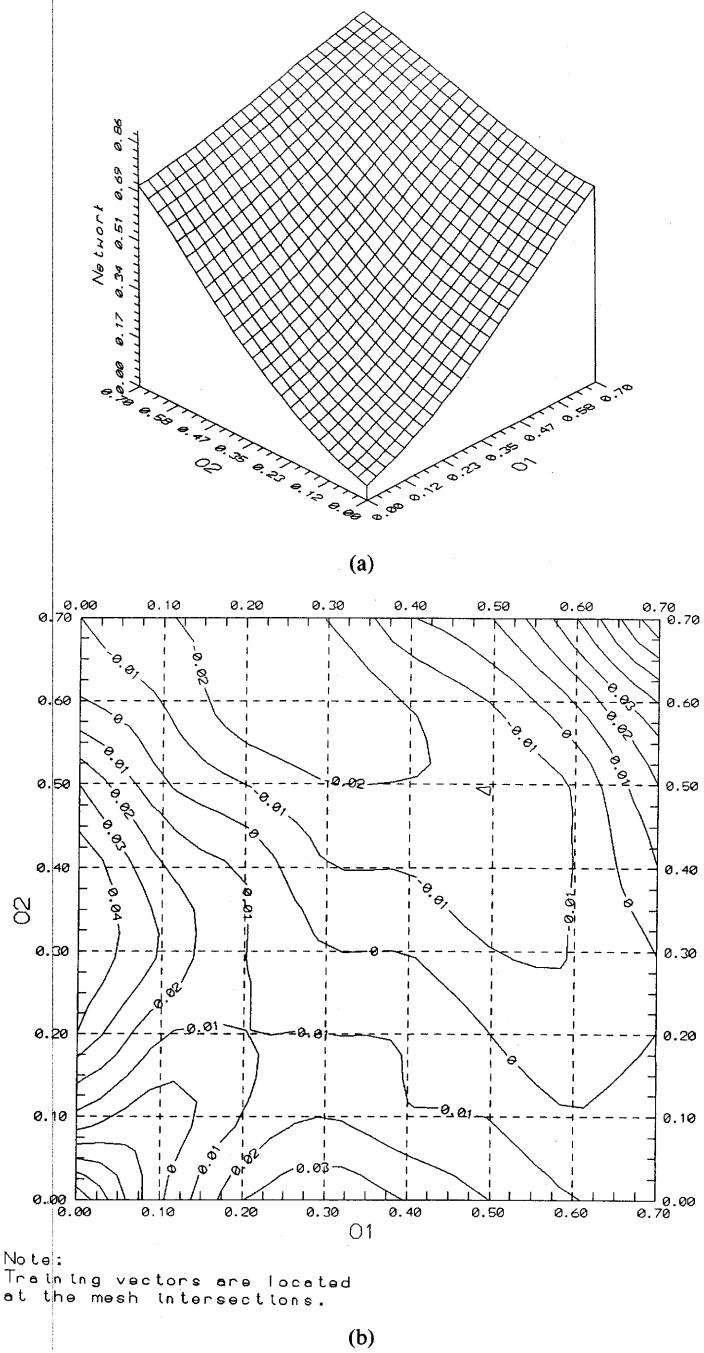


Figure 4.14 Continuous function network approximation, 64 training points, network with three neurons: (a) resulting mapping and (b) contour map of $d - o_5$.

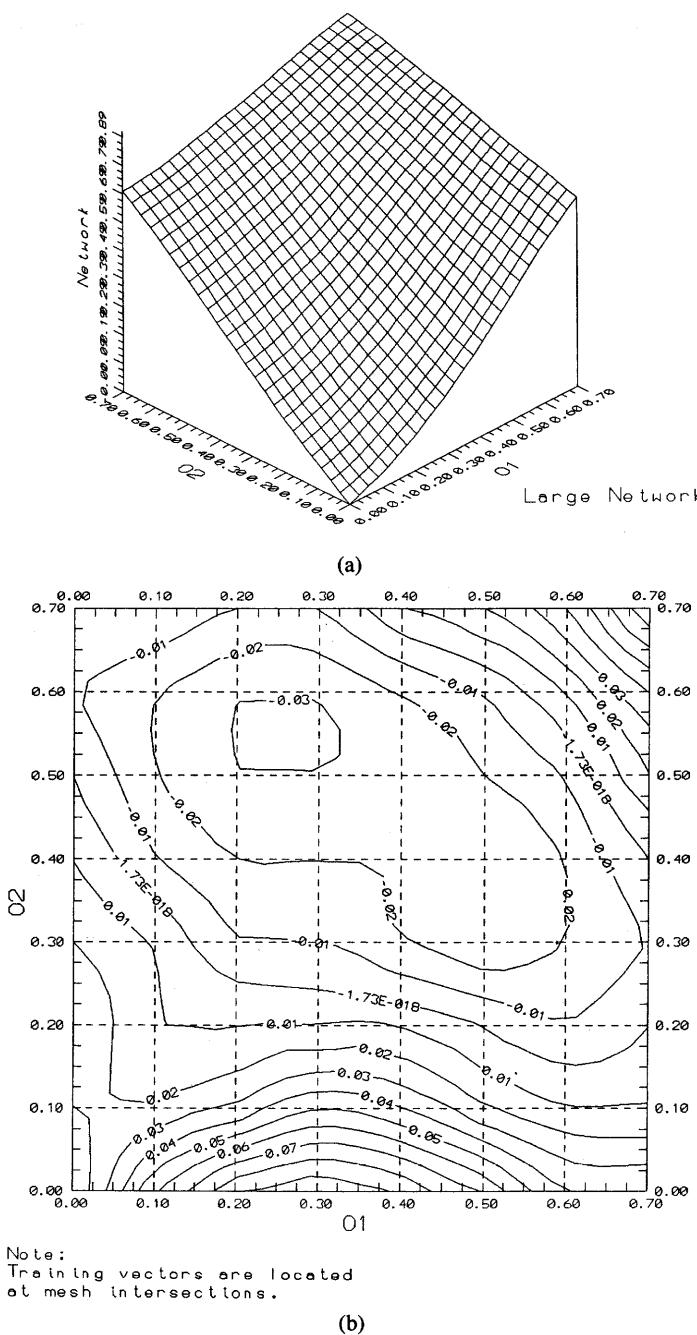


Figure 4.15 Continuous function network approximation, 64 training points, network with 11 neurons: (a) resulting mapping and (b) contour map of $d - o_5$.

As the last in a series of computational experiments, the architecture of the network has been changed to contain 10 hidden neurons ($J = 10$). The same set of 64 training points has been used. The network has reached a desired error level of value 0.015 after 1418 training steps with $\eta = 0.4$. Weights obtained are

$$\mathbf{W} = [-2.22 \quad -0.30 \quad -0.30 \quad -0.47 \quad 1.49 \quad -0.23 \quad 1.85 \\ -2.07 \quad -0.24 \quad 0.79 \quad -0.15] \quad (4.46)$$

$$\mathbf{V} = \begin{bmatrix} 0.57 & 0.66 & -0.10 & -0.53 & 0.14 & 1.06 & -0.64 & -3.51 & -0.03 & 0.01 \\ 0.64 & -0.57 & -1.13 & -0.11 & -0.12 & -0.51 & 2.94 & 0.11 & -0.58 & -0.89 \end{bmatrix}$$

The mapping obtained in this case is displayed in Figure 4.15(a), and the respective error contour map is provided in Figure 4.15(b). The overall quality of the mapping is comparable to that shown in Figure 4.14(b). However, the network training required much more CPU time than for the case of two neurons only in the hidden layer.

4.5

LEARNING FACTORS

The back-propagation learning algorithm in which synaptic strengths are systematically modified so that the response of the network increasingly approximates the desired response can be interpreted as an optimization problem. The generic criterion function optimization algorithm is simply negative gradient descent with a fixed step size. The output error function (4.4), which serves as an objective function, is defined in the overall weight space, which has $J(I + K)$ dimensions. The learning algorithm modifies the weight matrices so that the error value decreases.

The essence of the error back-propagation algorithm is the evaluation of the contribution of each particular weight to the output error. This is often referred to as the problem of credit assignment. Since the objective function of a neural network contains continuously differentiable functions of the weights, the evaluation of credit assignment can be easily accomplished numerically. As a reminder, note that this evaluation would not have been possible without replacing the discrete perceptrons with continuous perceptrons.

It might appear that the error back-propagation algorithm has made a breakthrough in supervised learning of layered neural networks. In practice, however, implementation of the algorithm may encounter different difficulties. The difficulties are typical of those arising in other multidimensional optimization approaches. One of the problems is that the error minimization procedure may produce only a *local minimum* of the error function. Figure 4.16 shows a typical cross section of an error space in a single weight dimension. It can be seen that the error is a nonnegative function of the weight variable. The ideal mapping would reduce E_{rms} to zero.

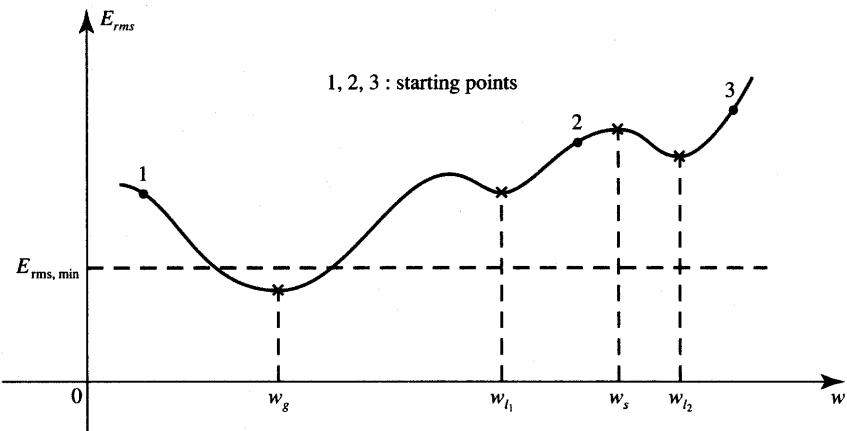


Figure 4.16 Minimization of the error E_{rms} as a function of single weight.

In practice, however, the learning would be considered successful for E_{rms} below an acceptable minimum E_{rms} value. The error function shown in Figure 4.16 possesses one global minimum below the min E_{rms} value, but it also has two local minima at w_{l1} and w_{l2} , and one stationary point at w_s . The learning procedure will stop prematurely if it starts at point 2 or 3; thus the trained network will be unable to produce the desired performance in terms of its acceptable terminal error. To ensure convergence to a satisfactory minimum the starting point should be changed to 1. Moreover, there is a question of how long it might take a network to learn. An appropriate choice of the learning parameters should guarantee that a good quality solution is found within a reasonable period of computing time.

Although the negative gradient descent scheme and all other optimization techniques can become stuck in local minima of the error function, local minima have not been much of a problem in many of the training cases studied. Since these minima are not very deep, inserting some form of *randomness to the training* may be sufficient to get out.

However, a more convincing explanation for reasons why the local minima are not a major problem in this training procedure has its background in the stochastic nature of the algorithm. The square error surfaces produced are random. The larger the neural network, the better should be the training outcome. In fact, the error back-propagation technique has been found to be equivalent to a form of stochastic approximation explored in the early 1960s (Tsypkin 1973). The major novel aspect of the algorithm is that it is computationally efficient for the empirical study of stochastic approximation techniques (White 1989).

One of the factors that usually improves the convergence of training is the statistical nature of inputs and outputs, which may be realizations of two somewhat related random processes. Also, even when inputs are purely deterministic,

superposition of noise with the zero mean value can increase the efficiency of the training process. However, in absence of solid theories, the neural network modeler should often rely on experimentation and on understanding of basic network principles to achieve satisfactory results.

The discussion below continues to address the issue of convergence of the back-propagation algorithm. We will also focus on several main aspects and some practical properties of the algorithm. The most important of these are initial weights, cumulative weight adjustment, the form of the neuron's activation function, and selection of the learning constant and momentum term. We will also discuss selected aspects of the network architecture that are relevant for successful training.

Initial Weights

The weights of the network to be trained are typically initialized at small random values. The initialization strongly affects the ultimate solution. If all weights start out with equal weight values, and if the solution requires that unequal weights be developed, the network may not train properly. Unless the network is disturbed by random factors or the random character of input patterns during training, the internal representation may continuously result in symmetric weights.

Also, the network may fail to learn the set of training examples with the error stabilizing or even increasing as the learning continues. In fact, many empirical studies of the algorithm point out that continuing training beyond a certain low-error plateau results in the undesirable drift of weights. This causes the error to increase and the quality of mapping implemented by the network decreases. To counteract the drift problem, network learning should be restarted with other random weights. The choice of initial weights is, however, only one of several factors affecting the training of the network toward an acceptable error minimum.

Cumulative Weight Adjustment versus Incremental Updating

As stated before, the error back-propagation learning based on the single pattern error reduction (4.4) requires a small adjustment of weights which follows each presentation of the training pattern. This scheme is called *incremental updating*. As shown by McClelland and Rumelhart (1986), the back-propagation learning also implements the gradient-like descent minimization of the overall error function as defined in (4.34) computed over the complete cycle of P presentations, provided the learning constant η is sufficiently small.

The advantage of minimization of the current and single pattern error as illustrated in the flowchart of Figure 4.8 and summarized in the EBPT algorithm of Section 4.4 is that the algorithm implements a true gradient descent downhill of the error surface. Moreover, during the computer simulation, the weight adjustments determined by the algorithm do not need to be stored and gradually compounded over the learning cycle consisting of P joint error signal and weight adjustment computation steps. The network trained this way, however, may be skewed toward the most recent patterns in the cycle. To counteract this specific problem, either a small learning constant η should be used or cumulative weight changes imposed as follows

$$\Delta w = \sum_{p=1}^P \Delta w_p$$

for both output and hidden layers. The weight adjustment in this scheme is implemented at the conclusion of the complete learning cycle. This may also have an averaging effect on the training, however. Provided that the learning constant is small enough, the cumulative weight adjustment procedure can still implement the algorithm close to the gradient descent minimization.

Although both cumulative weight adjustment after each completed training cycle or incremental weight adjustment after each single pattern presentation can bring satisfactory solutions, attention should be paid to the fact that the training works best under random conditions. It would thus seem advisable to use the incremental weight updating after each pattern presentation, but choose patterns in a random sequence from a training set. This introduces much-needed noise into the training and alleviates the problems of averaging and skewed weights which would tend to favor the most recent training patterns.

Steepness of the Activation Function

As introduced in Chapter 2, the neuron's continuous activation function $f(net, \lambda)$ is characterized by its steepness factor λ . Also, the derivative $f'(net)$ of the activation function serves as a multiplying factor in building components of the error signal vectors δ_o and δ_y . Thus, both the choice and shape of the activation function would strongly affect the speed of network learning.

The derivative of the activation function (2.3a) can be easily computed as follows:

$$f'(net) = \frac{2\lambda \exp(-\lambda net)}{[1 + \exp(-\lambda net)]^2} \quad (4.47)$$

and it reaches a maximum value of $1/\lambda$ at $net = 0$. Figure 4.17 shows the slope function of the activation function and it illustrates how the steepness λ affects the learning process. Since weights are adjusted in proportion to the value

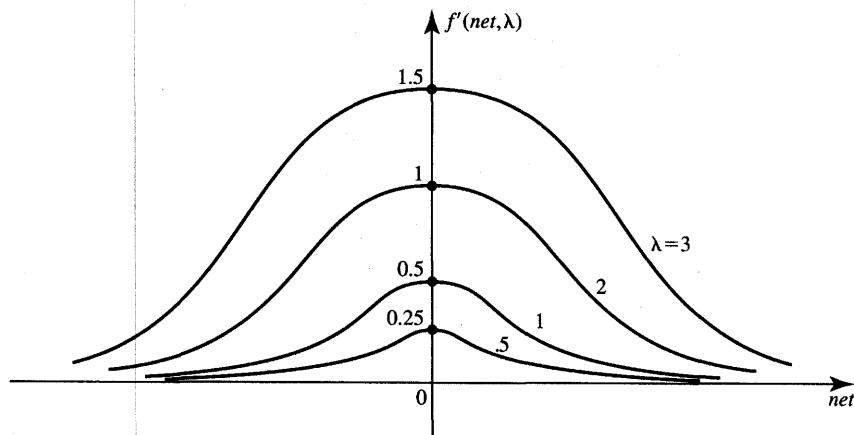


Figure 4.17 Slope of the activation function for various λ values.

$f'(net)$, the weights that are connected to units responding in their midrange are changed the most. The weights of uncommitted neurons with uncertain responses are thus affected more strongly than of those neurons that are already heavily turned on or turned off. Since the local signal errors δ_{ok} and δ_{yj} are computed with $f'(net)$ as a multiplier, the transmitted components of the back-propagating error are large only for neurons in steep thresholding mode.

The other feature apparent from Figure 4.17 is that for a fixed learning constant all adjustments of weights are in proportion to the steepness coefficient λ . This particular observation leads to the conclusion that using activation functions with large λ may yield results similar as in the case of large learning constant η . It thus seems advisable to keep λ at a standard value of 1, and to control the learning speed using solely the learning constant η , rather than controlling both η and λ .

Learning Constant

The effectiveness and convergence of the error back-propagation learning algorithm depend significantly on the value of the learning constant η . In general, however, the optimum value of η depends on the problem being solved, and there is no single learning constant value suitable for different training cases. This problem seems to be common for all gradient-based optimization schemes. While gradient descent can be an efficient method for obtaining the weight values that minimize an error, error surfaces frequently possess properties that make the procedure slow to converge.

When broad minima yield small gradient values, then a larger value of η will result in a more rapid convergence. However, for problems with steep and narrow

minima, a small value of η must be chosen to avoid overshooting the solution. This leads to the conclusion that η should indeed be chosen experimentally for each problem. One should also remember that only small learning constants guarantee a true gradient descent. The price of this guarantee is an increased total number of learning steps that need to be made to reach the satisfactory solution. It is also desirable to monitor the progress of learning so that η can be increased at appropriate stages of training to speed up the minimum seeking.

Although the choice of the learning constant depends strongly on the class of the learning problem and on the network architecture, the values ranging from 10^{-3} to 10 have been reported throughout the technical literature as successful for many computational back-propagation experiments. For large learning constants, the learning speed can be drastically increased; however, the learning may not be exact, with tendencies to overshoot, or it may never stabilize at any minimum.

Even though the simple gradient descent can be efficient, there are situations when moving the weights within a single learning step along the negative gradient vector by a fixed proportion will yield a minor reduction of error. For flat error surfaces for instance, too many steps may be required to compensate for the small gradient value. Furthermore, the error contours may not be circular and the gradient vector may not point toward the minimum. Some heuristics for improving the rate of convergence are proposed below based on the observations just discussed.

Momentum Method

The purpose of the momentum method is to accelerate the convergence of the error back-propagation learning algorithm. The method involves supplementing the current weight adjustments (4.5) and (4.21a) with a fraction of the most recent weight adjustment. This is usually done according to the formula

$$\Delta w(t) = -\eta \nabla E(t) + \alpha \Delta w(t-1) \quad (4.48a)$$

where the arguments t and $t-1$ are used to indicate the current and the most recent training step, respectively, and α is a user-selected positive momentum constant. The second term, indicating a scaled most recent adjustment of weights, is called the *momentum term*. For the total of N steps using the momentum method, the current weight change can be expressed as

$$\Delta w(t) = -\eta \sum_{n=0}^N \alpha^n \nabla E(t-n) \quad (4.48b)$$

Typically, α is chosen between 0.1 and 0.8. Figure 4.18 illustrates the momentum method heuristics and provides the justification for its use.

Let us initiate the gradient descent procedure at point A' . The consecutive derivatives $\partial E / \partial w_1$ and $\partial E / \partial w_2$ at training points A', A'', \dots , are of the same sign. Obviously, combining the gradient components of several adjacent steps

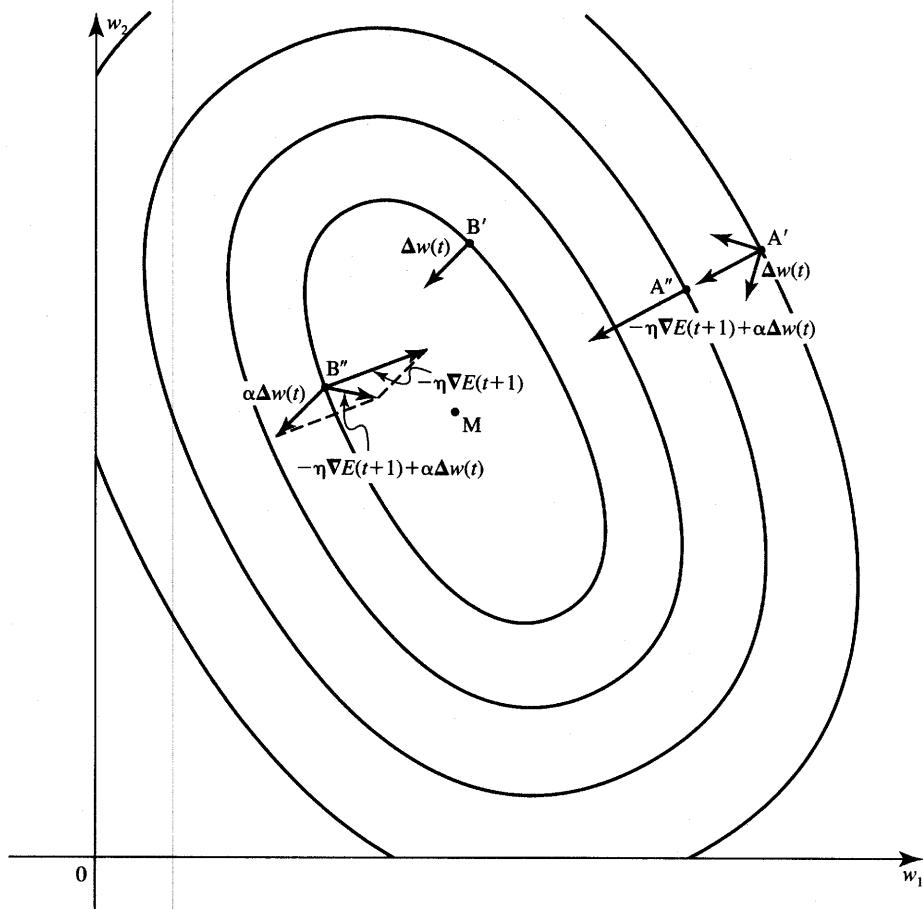


Figure 4.18 Illustration of adding the momentum term in error back-propagation training for a two-dimensional case.

would result in convergence speed-up. If the error surface has a smooth variation along a certain axis, the learning rate along this particular component should thus be increased. By adding the momentum term, the weight adjustment in A'' is enhanced by the fraction of the adjustment of weights at A' .

After starting the gradient descent procedure at B' , the two derivatives $\partial E / \partial w_1$ and $\partial E / \partial w_2$, initially negative at B' , both alter their signs at B'' . The figure indicates that the negative gradient does not provide an efficient direction of weight adjustment because the desired displacement from B'' should be more toward the minimum M , or move the weight vector along the valley rather than across it. As seen from the figure, the displacement $\eta \nabla E(t+1)$ at B'' would move the weights by a rather large magnitude almost across the valley and near the starting point B' . Moving the weights by $-\eta \nabla E(t+1) + \alpha \Delta w(t)$, however, reduces the magnitude of weight adjustment and points the resulting vector $\Delta w(t+1)$

more along the valley. One can thus infer that if the gradient components change signs in two consecutive iterations, the learning rate along this axis should be decreased. This discussion thus indicates that the momentum term typically helps to speed up convergence, and to achieve an efficient and more reliable learning profile (Jacobs 1988).

To gain some further insight into the momentum method, let us look at the comparison of the performance of the error back-propagation technique without and with the momentum term. The comparative study of both techniques has been carried out using the scalar valued function of two-dimensional weight vector as follows:

$$E(\mathbf{w}) = (aw_1)^2 + w_2^2$$

By varying a in the error expression above, it is possible to shape the error surface. Values of a that are close to unity ensure circular error contours in contrast to small values, which produce a narrow valley. For $a = 0.02$, different angles of rotation of the valley axis relative to the w_1 axis can be generated through a simple rotational transform. The basic error back-propagation technique without and with the momentum method has been used to seek the minimum of $E(\mathbf{w})$ with the final error below 10^{-3} and for initial weights of $(100, 2)$. The following table summarizes the number of computational steps for different values of η and α (Silva and Almeida 1990).

Rotation		0°	10°	20°	30°	45°
EBPT, $\eta = 0.5$		10367				
EBPT with momentum						
$\alpha = 0.5$	$\eta = 0.5$				5180	
$\alpha = 0.9$	$\eta = 0.5$				1007	
$\alpha = 0.9$	$\eta = 0.05$				10339	

We see that a twofold and even tenfold increase of training speed has been observed. The result shows that the inclusion of the momentum term can considerably speed up convergence when comparable η and α are employed compared to the regular error back-propagation technique. Although this discussion has only used w as symbol of weight adjusted using the momentum method, both weights v_{ji} and w_{kj} can be adapted using this method. Thus, the momentum term technique can be recommended for problems with convergence that occur too slowly or for cases when learning is difficult to achieve.

Network Architectures Versus Data Representation

One of the most important attributes of a layered neural network design is choosing the architecture. In this section we will study networks with a single hidden layer of J neurons, and with an output layer consisting of K neurons. In accordance with Figure 4.7, the network has I input nodes. Let us try to determine the guidelines for selection of network sizes expressed through I, J , and K . The number of input nodes is simply determined by the dimension, or size, of the input vector to be classified, generalized or associated with a certain output quantity. The input vector size usually corresponds to the number of distinct features of the input pattern. In this discussion the input vector is considered as nonaugmented.

In the case of planar images, the size of the input vector is sometimes made equal to the total number of pixels in the evaluated image. In another approach, however, it may be a vector of size two only. In such cases, a trade-off usually exists between the number of training patterns P needed to create the training set and the dimension I of the input pattern vector. To illustrate this with an example, assume three planar training images, characters C , I , and T , represented on a 3×3 grid. Using the first of the two approaches, nine pixels can be used to depict each of the characters as shown in Figure 4.19. On the figure, pixels of character C are represented using this approach. Assuming that the size of the input pattern vector is nine, we have $P = 3$ and the following three training vectors in the training pattern set $\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$:

$$\mathbf{x}_1 = [1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1]^t : \text{class C}$$

$$\mathbf{x}_2 = [0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0]^t : \text{class I}$$

$$\mathbf{x}_3 = [1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0]^t : \text{class T}$$

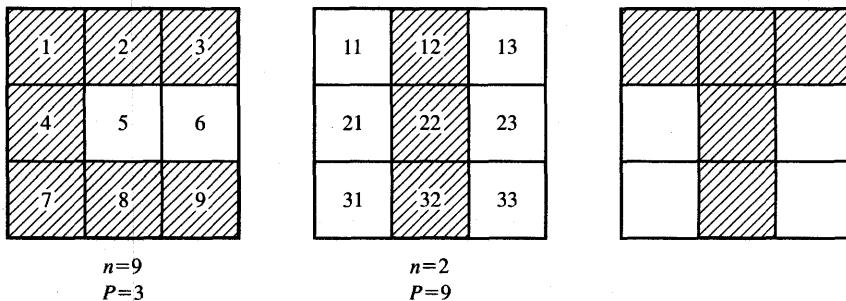


Figure 4.19 Illustration for trade-off between the size of the pattern vector and the number of necessary training patterns.

assuming that white and black pixels correspond to vector components 0, 1, respectively.

Alternatively, the dimensionality of the planar input images can be preserved as equal to only two. In such a case, however, the number of training patterns P has to be made equal to the number of pixels. Pixels of character I on the figure are labeled according to this method of pattern representation. It can now be seen that we have $P = 9$. Without any feature extraction or compression of original training data, the training pattern set can be written now as $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_9\}$, where

$$\mathbf{x}_1 = [1 \ 1]^t : \text{class C, T}$$

$$\mathbf{x}_2 = [1 \ 2]^t : \text{class C, I, T}$$

...

$$\mathbf{x}_9 = [3 \ 3]^t : \text{class C}$$

The reader may also notice that relaxation of the requirements for dimensionality of input patterns leads to further interesting conclusions. Why not, for instance, represent the three patterns from Figure 4.19 as a set of $P = 27$ patterns, each of them considered to be of single dimension? We now have arrived at three different input representations, each of them being as sensible as the other for this simple problem. Lacking the guidelines regarding the choices and trade-offs between P and n , we may have to approach the problem of input representation with flexibility and be ready to experiment.

As discussed in Chapter 3, n -dimensional pattern vectors can be handled by the network with $I = n + 1$ input nodes, where the augmentation from n to $n + 1$ input nodes is required for single- and multilayer feedforward networks trained in a supervised mode. Thus, the choice of the input pattern dimensionality uniquely imposes the size of the input layer of the network to be designed. Understandably, the choice of input representation has rather significant consequences on the properties of the input patterns. As will be shown later, patterns that are difficult to classify in low-dimensional space often become easily separable after dimensionality expansion.

Let us look at conditions for selecting the number of output neurons. The number of neurons in the output layer, K , can be made equal to the dimension of vectors to be associated if the network is employed as an associator of input to output vectors. If the network works as an auto-associator, which associates the distorted input vector with the undistorted class prototype, then we obviously have $I = K$. In the case of a network functioning as a classifier, K can be made equal to the number of classes. In such cases, described earlier in Section 3.7 as local representation, the network would also perform as a class decoder, which responds with only one output value different than all the remaining outputs. The network can be trained in this output representation to indicate the class number as equal to the active output number.

The number of output neurons K can sometimes be lowered if no class decoding is required. Binary-coded or other encoded class numbers can be postulated for such a classifier for the purpose of its training. This is called the *distributed representation* case as described in Section 3.7. For example, a four-class classifier can be trained using only two output neurons having outputs, after thresholding, of 00, 01, 10, and 11, for classes 0, 1, 2, and 3, respectively. For the same reason, alphabet character classifiers would need at least five neurons in the output layer since there are 26 classes to be identified.

We can easily note that the number of output neurons in a K -class classifier can be any integer value from $\log_2 K$ through K , including these boundaries. These numbers correspond to the local and distributed representation of classifiers' output data, respectively. It is, however, somewhat likely and intuitively plausible that shrinking the network and the compression of the output layer below the number of K neurons will affect the length of the training itself, and the robustness of the final network.

Necessary Number of Hidden Neurons

The size of a hidden layer is one of the most important considerations when solving actual problems using multilayer feedforward networks. The problem of the size choice is under intensive study with no conclusive answers available thus far for many tasks. The exact analysis of the issue is rather difficult because of the complexity of the network mapping and due to the nondeterministic nature of many successfully completed training procedures. In this section we will look at some guidelines that may assist a neural network modeler with a number of useful hints.

Single-hidden layer networks can form arbitrary decision regions in n -dimensional input pattern space. There exist certain useful solutions as to the number J of hidden neurons needed for the network to perform properly. The solutions also determine the lower bound on the number of different patterns P required in the training set. As will be shown, the number of hidden neurons depends on the dimension n of the input vector and on the number of separable regions in n -dimensional Euclidean input space.

Let us assume that the n -dimensional nonaugmented input space is linearly separable into M disjoint regions with boundaries being parts of hyperplanes. Each of the M regions in the input space can be labeled as belonging to one of the R classes, where $R \leq M$. Figure 4.20 shows an example separation for $n = 2$, $M = 7$, and $R = 3$. Intuitively, it is obvious that the number of separable regions M indicates the lower bound on the size P of the set required for meaningful training of the network to be designed. Thus, we must have $P \geq M$.

Choosing $P = M$ would indicate the coarsest possible separation of input space into M regions using parts of partitioning hyperplanes. Clearly, in such a

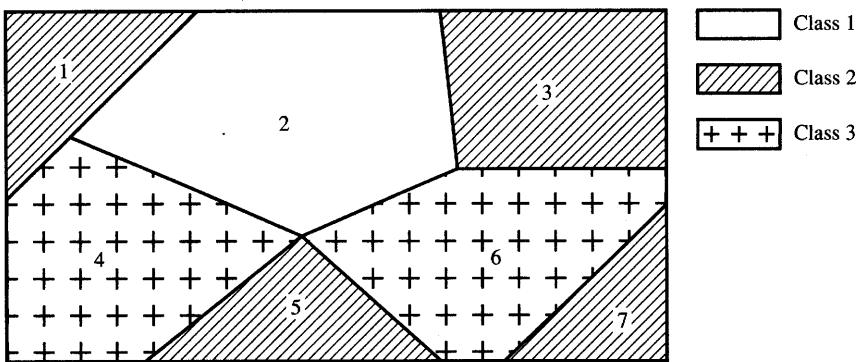


Figure 4.20 Two-dimensional input space with seven separable regions assigned to one of three classes.

case when one training pattern per region is available, granularity of the proposed classification is not fine at all. The more curved the contours of separated decision regions and involved pattern space partitioning are, the higher should be the ratio of $P/M \geq 1$. Choosing $P/M \gg 1$ would hopefully allow the network to discriminate pattern classes using fine piecewise hyperplane partitioning.

There exists a relationship between M , J , and n allowing for calculation of one of the three parameters, given the other two of them. As shown by Mirchandini and Cao (1989), the maximum number of regions linearly separable using J hidden neurons in n -dimensional input space is given by the relationship

$$M(J, n) = \sum_{k=0}^n \binom{J}{k}, \quad \text{where } \binom{J}{k} = 0 \text{ for } J < k \quad (4.49a)$$

Using formula (4.49a) allows the estimation of the hidden layer size J given n and M . A simple computation shows that the network providing the classification illustrated in Figure 4.20 should have three hidden layer neurons provided input patterns are of dimensionality two, or $n = 2$. Let us consider the case of input patterns of large dimension assuming that the expected, or estimated, size of the hidden nodes is small. For large-size input vectors compared to the number of hidden nodes, or when $n \geq J$, we have from (4.49a)

$$M = \binom{J}{0} + \binom{J}{1} + \binom{J}{2} + \dots + \binom{J}{J} = 2^J \quad (4.49b)$$

It follows from (4.49b) that the hidden neuron layer with three nodes would be capable of providing classification into up to eight classes; but since $n \geq J$, the size of the input vector has to be larger than three.

The formulas (4.49) can be inverted to find out how many hidden layer neurons J need to be used to achieve classification into M classes in n -dimensional

pattern space. This number constitutes the solution of the equation

$$M = 1 + J + \frac{J(J-1)}{2!} + \frac{J(J-1)(J-2)}{3!} + \frac{J(J-1) \cdots (J-n+1)}{n!}, \quad \text{for } J > n \quad (4.50a)$$

For the case $J \leq n$ we have for J from (4.49b) simply

$$J = \log_2 M \quad (4.50b)$$

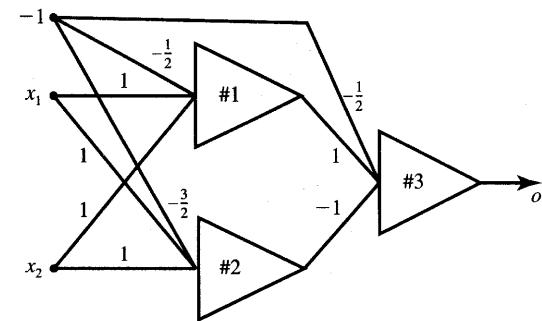
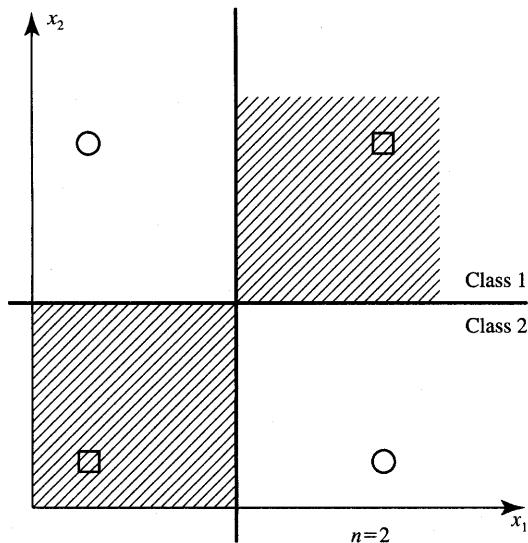
EXAMPLE 4.4

In this example we will use the discussed guidelines to propose suitable network architectures for the solution of the two-dimensional XOR problem. Assuming $J \leq n$ and $n = 2$, we obtain from (4.50b) that $M = 4$ for $J = 2$.

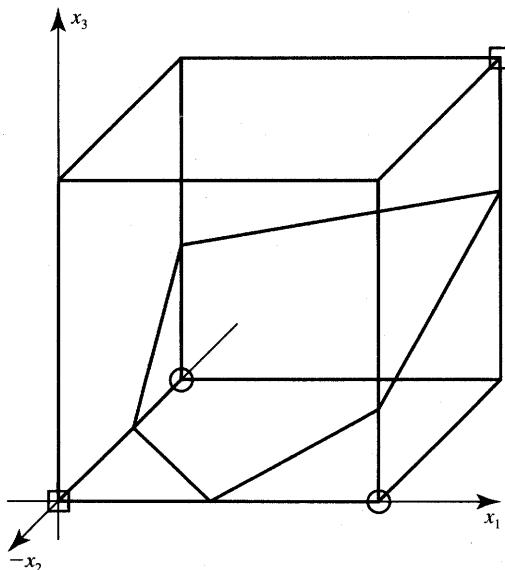
The corresponding partitioning of the two-dimensional input space is shown in Figure 4.21(a) along with an example network implementation using a single hidden layer architecture. The reader can verify that the network provides an appropriate classification by replacing continuous neurons with TLU elements and analyzing the mapping performed by each unit. Alternatively, the same two-dimensional XOR problem can be considered in three-dimensional space. It is possible to add the third dimension x_3 indicating, if equal to 1, the conjunction of both inputs x_1 and x_2 being 1; otherwise, x_3 remains zero.

Now, a single plane can partition the cube as simply as shown in Figure 4.21(b). It can be seen that neuron 1 performs the mapping of x_1 and x_2 to yield an appropriate x_3 value. Neuron 2 implements the decision plane as shown. Indeed, noting that $n = 3$ and $M = 2$, the formula (4.50b) yields $J = 1$. Thus, a single hidden layer unit as shown in Figure 4.21(b) can be used to solve alternatively the XOR problem using a total number of two neurons instead of three as in Figure 4.21a. The size of the hidden layer has been reduced by 1, or 33%, at no cost. This example has shown how to select the size of the hidden layer to solve a specific classification problem.

Another way of optimizing the architecture is related to *pruning of feed-forward multilayer network*. Pruning is done as network trimming within the assumed initial architecture. The network can be trimmed by removal of unimportant weights. This can be accomplished by estimating the sensitivity of the total error to the exclusion of each weight in the network (Karnin 1990). The weights which are insensitive to error changes can be discarded after each step of incremental training. Unimportant neurons can also be removed (Sietsma and



(a)



(b)

Figure 4.21 Basic architectures implementing the XOR function: (a) two hidden layer neurons and (b) single hidden layer neuron.

Dow 1988). In either case it is advisable to retrain the network with the modified architecture. The trimmed network is of smaller size and is likely to train faster than before its trimming.

4.6

CLASSIFYING AND EXPERT LAYERED NETWORKS

Multilayer feedforward neural networks can be successfully applied to a variety of classification and recognition problems. The application of such networks represents a major change in the traditional approach to problem solution. It is not necessary to know a formal mathematical model of the classification or recognition problem to train and then recall information from the feedforward neural systems. Instead, if a sufficient training set and a suitable network architecture are devised, then the error back-propagation algorithm can be used to adapt network parameters to obtain an acceptable solution (Cybenko 1990). The solution is obtained through experimentation and simulation rather than through rigorous and formal approach to the problem. As such, neural network computation offers techniques that are in the middle ground, somewhere between the traditional engineering and the artificial intelligence approach.

Although it is not yet clear from the technical literature what constitutes an adequate training pattern set and network architecture, a number of successful applications ranging from speech recognition to sonar signal processing have been reported (Sejnowski and Rosenberg 1987; Gorman and Sejnowski 1988). A number of technical reports describe other successful phonetic classification and speech experiments (Leung and Zue 1989; Waibel 1989; Bengio et al. 1989). More detailed exposition of multilayer feedforward network applications for practical tasks is provided in Chapter 8. The discussion below is mainly to enhance the principles, to illustrate the main features of the training, and to discuss example results.

Let us summarize the task the error back-propagation network needs to solve. Given are the training data in the form of P vector pairs $(\mathbf{z}_p, \mathbf{d}_p)$, or the training set

$$\{(\mathbf{z}_p, \mathbf{d}_p), \quad p = 1, 2, \dots, P\} \quad (4.51)$$

For a given pattern p the network maps \mathbf{z}_p into \mathbf{o}_p using the highly nonlinear operation of Equation (4.31a) as follows:

$$\mathbf{o}_p = \Gamma[\mathbf{W}\Gamma[\mathbf{V}\mathbf{z}_p]] \quad (4.52)$$

The goal of the training has been to produce \mathbf{o}_p such that it replicates \mathbf{d}_p , or $\mathbf{o}(\mathbf{z}_p) \cong \mathbf{d}_p$. The quality of approximation is determined by the error

$$E = \sum_{p=1}^P \|\mathbf{o}(\mathbf{W}, \mathbf{V}, \mathbf{z}_p) - \mathbf{d}_p\|^2 \quad (4.53)$$

Minimization of error (4.53) can be interpreted as a classical interpolation and estimation problem—given data, we seek parameters w and v that approximate data for a class of functions selected as in (4.52).

Character Recognition Application

Let us look at a typical application of the error back-propagation algorithm for handwritten character recognition, details of which are covered in Chapter 8 (Burr 1988). An input character is first normalized so that it extends to the full height and width of the bar mask. The handwritten alphabet character is then encoded into 13 line segments arranged in a template as in Figure 4.22(a). The encoding takes the form of shadow projection. A shadow projection operation is defined as simultaneously projecting a point of the character into its three closest vertical, horizontal, and diagonal bars. After all points are projected, shaded encoded bars are obtained.

Each of the 13 segments shown in Figure 4.22(b) is now represented by the shadow code ranging from 0 to 50. The shadow codes for the bars representing character S from top to bottom, left to right are: 50, 46, 4, 14, 50, 24, 14, 6, 43, 42, and 50. The shadow codes can be understood as extracted pattern features. The shadow codes need to be normalized to within the 0, 1, which is the range for unipolar neurons used in the network. This enables the use of the activation

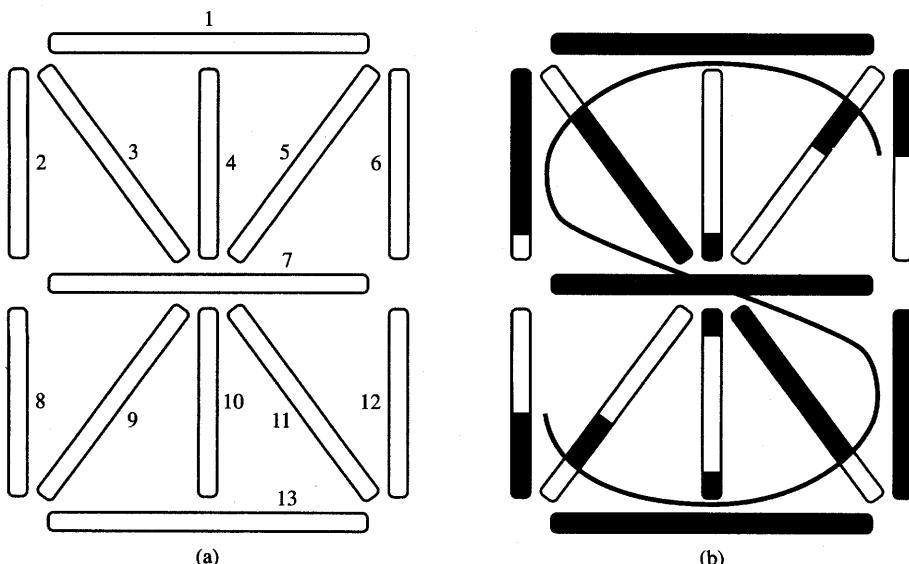


Figure 4.22 Thirteen-segment bar mask for encoding alphabetic capital letters: (a) template and (b) encoded S character. [Adapted from Burr (1988), © IEEE; reprinted with permission.]

function in its region of high slope and makes the training easier and more efficient. The network thus has 13 real valued inputs between 0 and 1, and 26 outputs, one for each alphabetic character. The convention used has been that the output corresponding to a letter should be 1 if the character is detected, and 0 otherwise.

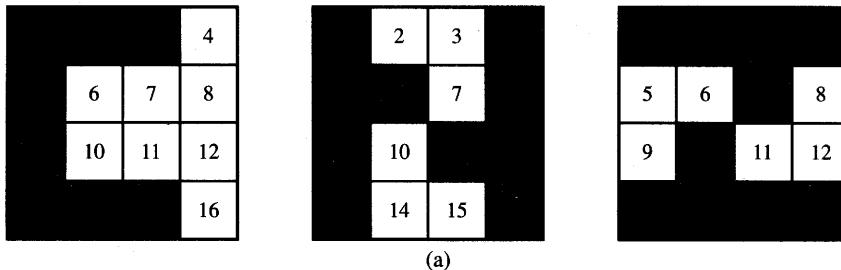
Thus, the network is of size $I = 14$ (including one fixed input) and $K = 26$. The network has been tested with 12, 16, 20, and 24 hidden-layer neurons. Networks with 20 or 24 hidden-layer neurons usually trained faster with $\eta = 0.3$ and 2.0. The training data consisted of 104 handprinted uppercase characters, 4 for each letter scaled and centered in the template as in Figure 4.22. After training, the network performed at levels of 90 to 95% accuracy for the new 104 written samples provided to test the generalization.

In a related experiment of handwritten digit recognition, a seven-segment template has been used for shadow code computation. The template has been obtained by removing bars 3-5 and 9-11 from the 13-bar template of Figure 4.22(a). A similar projection method has been employed to obtain shadow codes for 100 handwritten numerals 0 through 9. Fifty vectors have been used to train the network, 50 remaining vectors were needed for test purposes. The architecture was trained with 7 inputs and 10 output neurons. The best results have been reported by Burr (1988) for the 6 to 16 hidden-layer neurons for $\eta = 2$ and $\alpha = 0.9$ used for training. The accuracy of recognition achieved using this method was between 96 and 98% correct answers on the test set.

For the experiment with letter recognition, there was a total of $14 \times 12 + 13 \times 26 = 506$ weights for the 12 hidden nodes of the network. We may thus consider that the network had 506 degrees of freedom. As observed by Cybenko (1990), the number of degrees of freedom in many successful applications exceeds the number of training samples used to train the network. This, in turn, contradicts the common belief that the estimation problems should be overdetermined, and that there should be more data points than parameters to be estimated. This, in particular, is the case in linear regression and other conventional data fitting approaches. Neural networks of the class discussed in both the example covered as well as in other applications typically seem to solve underdetermined problems (Cybenko 1990). One of the possible explanations is that the true network dimensionality is not determined by the number of its weights, but is considerably smaller.

EXAMPLE 4.5

This example demonstrates network training using the error back-propagation technique for the bit-map classification of a single-hidden layer network. The task is to design a network that can classify the simplified bit maps of three characters as shown in Figure 4.23(a). Assuming that the



For $\eta = 1.000$									
Total iterations = 144									
Final error = 0.00995									
$V^t =$	-0.225	0.646	0.488	-0.317	0.707	-0.608	-0.954	0.896	0.540
	-0.198	0.734	-0.159	0.503	1.004	-0.497	1.033	-0.922	0.503
	-0.877	1.129	0.826	-0.485	-0.471	-0.801	-0.143	-0.591	-0.731
	0.646	0.185	-1.356	0.376	-1.003	0.596	0.214	0.509	-0.064
	0.261	0.323	0.425	0.343	1.357	-0.299	-1.521	1.405	-0.264
	0.770	-0.823	0.210	-0.855	-0.320	-0.659	-0.263	0.545	0.160
	-0.817	0.018	-0.809	-0.855	-1.350	-0.279	0.966	-0.603	-0.916
	0.693	-0.766	-0.316	-0.713	-0.038	-0.294	-1.151	0.039	0.891
	-0.507	-0.799	-0.238	0.893	0.314	-0.798	0.101	0.667	0.599
	-0.197	-0.027	-0.724	0.177	-0.161	1.044	1.544	-0.131	-0.016
	0.823	0.040	-0.371	0.229	-0.472	-0.336	-0.511	0.541	-0.293
	1.033	-0.920	-0.235	-0.225	-1.029	-0.212	-0.499	0.198	-0.564
	-0.186	-0.312	-1.030	-0.135	0.174	0.551	0.632	-0.535	-0.026
	-1.193	0.339	0.901	-0.804	0.332	0.793	-0.564	-0.931	-0.618
	0.513	0.479	0.906	-0.077	0.185	-0.152	-0.433	0.075	-0.893
	-0.303	-0.184	-0.955	0.061	-1.696	1.188	0.578	-0.544	0.102
	-0.324	-0.101	0.123	0.493	-0.745	0.002	-0.188	-0.325	0.954
$W^t =$	-1.402	1.500	-0.350						
	0.011	-1.709	1.046						
	2.202	-1.346	-0.491						
	0.156	-0.060	-0.598						
	2.659	-1.062	-2.063						
	-1.371	-0.635	0.617						
	-1.043	-1.260	2.460						
	-0.395	2.172	-2.264						
	-1.249	-0.106	-0.571						

(b)

Figure 4.23a,b Figure for Example 4.5: (a) bit maps for classification, (b) resulting weight matrices, 16 nonaugmented inputs, eight hidden layer neurons, three output neurons.

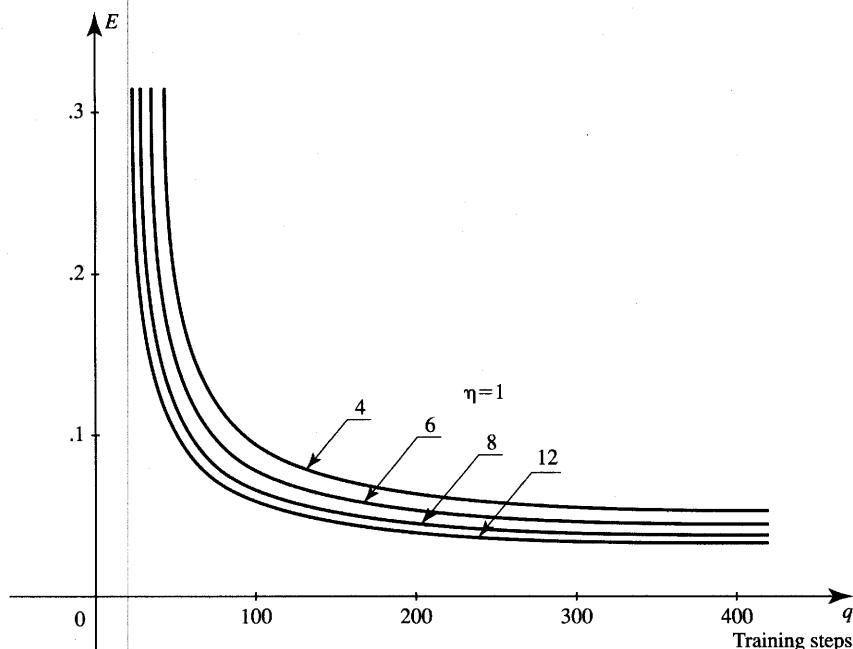


Figure 4.23c Figure for Example 4.5 (continued): (c) learning profiles for several different hidden layer sizes.

dimensionality of the nonaugmented input pattern is 16, we have $P = 3$ for the following input/desired output training data:

Character C: $\mathbf{z}_1 = [1 \ 1 \ 1 \ -1 \ 1 \ -1 \ -1 \ -1 \ 1 \ -1 \ -1 \ 1 \ 1 \ 1 \ -1]^t$

Class number: $\mathbf{d}_1 = [1 \ 0 \ 0]^t$

Character N: $\mathbf{z}_2 = [1 \ -1 \ -1 \ 1 \ 1 \ 1 \ -1 \ 1 \ 1 \ -1 \ 1 \ 1 \ 1 \ -1 \ -1 \ 1]^t$

Class number: $\mathbf{d}_2 = [0 \ 1 \ 0]^t$

Character Z: $\mathbf{z}_3 = [1 \ 1 \ 1 \ 1 \ -1 \ -1 \ 1 \ -1 \ -1 \ 1 \ -1 \ -1 \ 1 \ 1 \ 1 \ 1]^t$

Class number: $\mathbf{d}_3 = [0 \ 0 \ 1]^t$

The readers may notice that black and white pixels are coded as 1 and -1 , respectively, and that the local representation is used for detecting the class membership.

The program listed in the Appendix has been used to train the network with eight hidden neurons with $\eta = 1$ and $\lambda = 1$. Due to the necessary augmentation of inputs and of the hidden layer by one fixed input, the trained network has 17 input nodes, nine hidden neurons, and three output

neurons with an architecture like that of Figure 4.7. Error E has been defined as in (4.4) or as in the EBPT algorithm of Section 4.4. Since this is a rather simple classification task, the training takes only 144 iterations, which is an equivalent of 48 incremental training cycles consisting of three patterns each. The resulting weight matrices \mathbf{W} and \mathbf{V} are shown in Figure 4.23(b).

The typical output vectors recalled by the trained network with input vectors equal to \mathbf{z}_1 , \mathbf{z}_2 , and \mathbf{z}_3 are, respectively,

$$\begin{aligned}\mathbf{o}_1 &= [0.901 \quad 0.050 \quad 0.078]^t \\ \mathbf{o}_2 &= [0.011 \quad 0.984 \quad 0.013]^t \\ \mathbf{o}_3 &= [0.014 \quad 0.013 \quad 0.984]^t\end{aligned}\quad (4.54)$$

Simple thresholding of the computed vector entries of (4.54) provides the desired binary classification. It has been observed that using a value of 0.2 for η has increased the number of training steps to 722 for the same error $E_{\max} = 0.01$. Using values of 5 and 10 for η has reduced the number of training steps to 37 and 16, respectively. However, drastic changes of E in both directions for large η values have also been observed. Thus, to achieve moderately fast and reliable learning, the learning constant of value $\eta = 1$ has been used.

Different sizes of hidden layers have been attempted during training with $\eta = 1$. Learning profiles for 4, 6, 8, and 12 (nonaugmented) hidden nodes are shown in Figure 4.23(c). Increasing the number of hidden nodes has allowed for training to converge in fewer training steps. It takes longer, however, to train the large network. In addition, the network hardware becomes more expensive. The increase in the number of hidden nodes from 8 to 16 produced marginal reduction of the training from 144 to 106 steps. It has therefore been concluded that the suitable number of hidden nodes for this type of problem is between 4 and 8. ■

Expert Systems Applications

In the previous sections of this chapter, we introduced the supervised learning techniques of layered feedforward networks. Let us again consider such layered networks, which respond with outputs that can be represented as variables assuming one of several possible values, or continuum of values. Let us note that considering more than two output values of a neuron, or even continuous outputs, would make it possible to handle uncertainties. The degree of belief in a certain response can be determined using such fine quantized outputs.

The conventional approach to building an expert system requires a human expert to formulate the rules by which the input data can be analyzed. The

number of rules needed to drive an expert system may be large. Further, the lack of rigorous analysis, or even a lack of understanding of the knowledge domain, often makes formulation of such rules difficult. The rule formulation may become particularly complex with large sets of input data. Realizing that layered networks can be trained without encapsulating the knowledge into the rules, let us look at how they can be applied as an alternative to conventional rule-based expert systems.

Neural networks for diagnosis, fault detection, predictions, pattern recognition and association solve essentially various classification, association, and generalization problems. Such networks can acquire knowledge without extracting IF-THEN rules from a human expert provided that the number of training vector pairs is sufficient to suitably form all decision regions. Thus, neural networks would be able to ease the knowledge acquisition bottleneck that is hampering the creation and development of conventional expert systems. After training, even with a data-rich situation, neural networks will have the potential to perform like expert systems. What would often be missing, however, in neural expert system performance is their explanation function. Neural network expert systems are typically unable to provide the user with the reasons for the decisions made. The applications of neural networks for large-scale expert systems will be discussed in more detail in Chapter 8. This section covers only basic application concepts for layered diagnostic networks that are trainable using the error back-propagation technique. We will call such networks *connectionist expert systems*.

Let us take another look at the training and recall phases of error back-propagation-trained networks. Assume that a feedforward layered network is trained using the training vector pairs $(\mathbf{z}_1, \mathbf{d}_1), (\mathbf{z}_2, \mathbf{d}_2), \dots, (\mathbf{z}_P, \mathbf{d}_P)$. In the test, or recall, phase, the network described by Equation (4.52) performs the recognition task if it is tested with the input being a vector \mathbf{z}_i ($1 \leq i \leq P$) corrupted by noise. The network is expected to reproduce \mathbf{o}_i at its output in spite of the presence of noise. If no noise has been added, the network performs either a simple classification or association task.

If the trained neural network is tested with an input substantially different from any of the training set members, the expected response is supposed to solve the generalization problem. The generalization of the knowledge of the domain, which the network has learned during training, should cause it to respond correctly to any unseen before new input vector. Typical examples of generalization are diagnosis and prediction. The classification task may also be performed as a generalization operation but only in cases for which the domain models are partially defined. Examples of such classifications are medical and most technical diagnoses.

Let us consider first how a connectionist expert system for medical diagnosis can be built (Gallant 1988; Hripcsak 1988). A block diagram of an example expert system is shown in Figure 4.24. Input nodes take the information about the

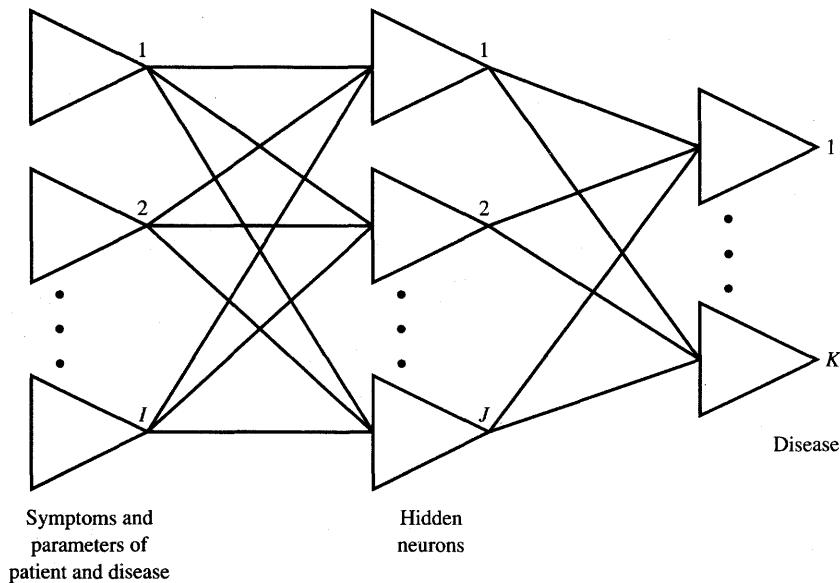


Figure 4.24 Connectionist expert system for diagnosis.

selected symptoms and their parameters. These can be numerical values assigned to symptoms, test results, or a relevant medical history for the patient. In the simplest version, input signals may be the binary value 1, if the answer to the symptom question is yes, or -1 , if the answer is no. The “unknown” answer should thus be made 0 to eliminate the effect of the missing parameter on the conclusion. With I input nodes, the network is capable of handling that many binary or numerical disease or patient data. The number of identifiable diseases can be made equal to the number of output nodes K .

In conventional automated medical diagnosis, a substantial obstacle is the formulation and entry of many rules by an expert. In contrast to a rule-driven system, a connectionist expert system can learn from currently accepted diagnoses. Diagnoses are generated from available hospital or doctor databases. As disease patterns and the art of diagnosis change, the connectionist expert system for diagnosis needs only be retrained or trained incrementally on a new set of data. In the case of major changes in diagnostic techniques, however, the expert system's numbers of input and hidden nodes and its architecture may also have to be changed.

As mentioned before, special care needs to be taken when choosing the number of hidden nodes. Too low a number of hidden nodes J can cause difficulties in mapping of I inputs into K outputs; too large a value for J will increase unnecessarily the learning and diagnosis times and/or cause uncertainty of the

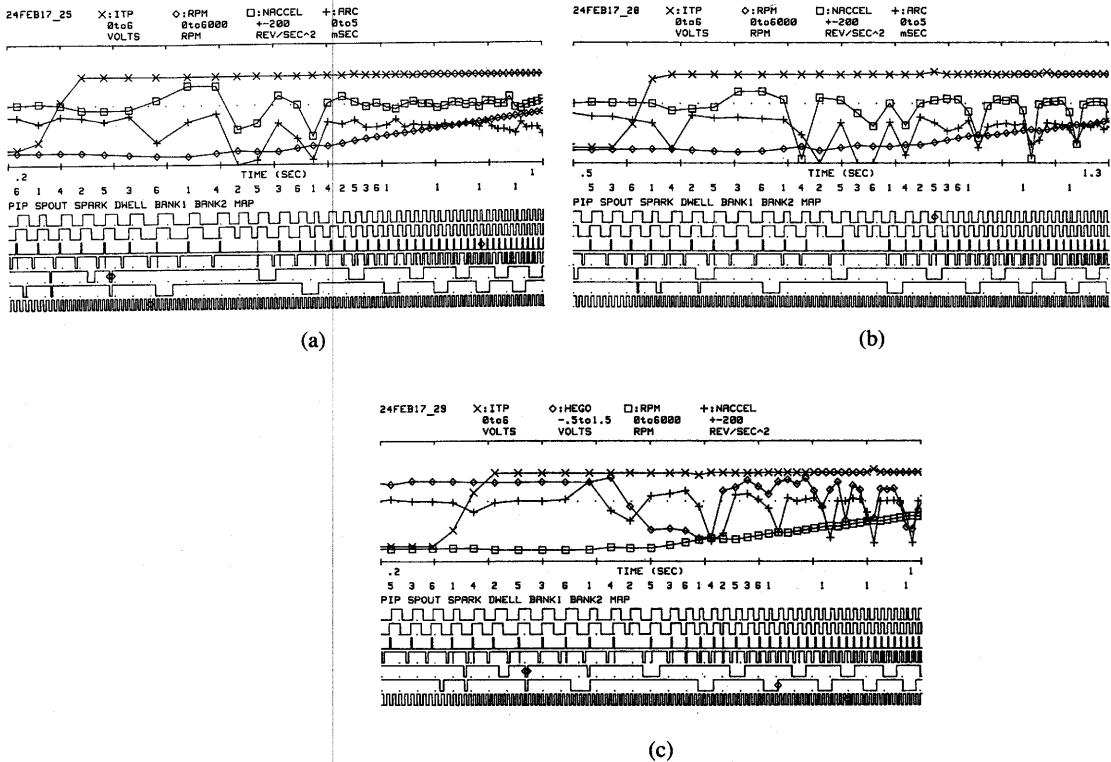


Figure 4.25 Automobile engine diagnostic data: (a) no-fault engine, (b) defective spark plug 4, and (c) defective fuel injector. [Source: Marko et al. (1989). © IEEE; reprinted with permission.]

training objective. In general, for J larger than needed, weights become more difficult to estimate reliably from the training data.

Let us review an example of a connectionist expert system for fault diagnosis of an automobile engine (Marko et al. 1989). The input and output data are controlled and monitored by an electronic engine control computer. Waveforms of analog/digital data such as those shown in Figure 4.25(a) are obtained first from an engine without faults, which is accelerated in neutral against the inertial load. These multichannel data serve as a reference and they are used for training of the expert system modeling the fault-free operation of an engine. Although a skilled engineer can analyze the complex engine output data shown in the figure, most technical personnel would find the interpretation of them rather difficult.

The practical identification of a fault becomes more complicated and time-consuming when signal anomalies are small and distributed over several signals that are time functions. This can be observed in Figures 4.25(b) and (c), which

show a set of input/output signals for a faulted engine. Figure 4.25(b) displays signals with a defective spark plug on cylinder 4, which causes a misfire at this cylinder in each engine cycle. Figure 4.25(c) displays the data from an engine with a defective fuel injector. In particular in this case, the fault can only be identified from simultaneous comparison of many of the traces of data.

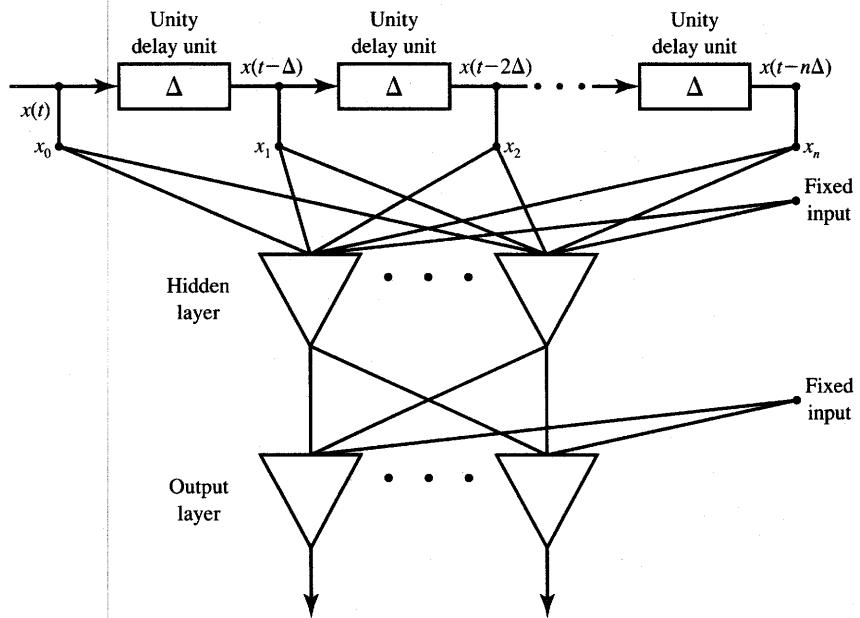
The task of defective engine diagnosis has been successfully solved by a back-propagation trained single-hidden layer network. The connectionist expert system for engine diagnosis makes it possible to identify 26 different faults such as a shorted plug, an open plug, a broken fuel injector, etc. The training set consists of 16 sets of data for each failure, each of the sets representing a single engine cycle. A total of 16×26 data vectors with 52 elements in each vector has been used for training.

The described neural network-based expert system needs 10 minutes of training time on the NESTOR NDS-100 computer. It attains 100% fault recognition accuracy on the test data set. Low learning rates, a number of hidden units equal to twice the number of inputs, and a randomized presentation order within the training set have been used in the initial development phase. To improve the learning speed, the number of hidden units was then decreased to less than the number of input units. As a result, the learning time decreased five times while maintaining 100% accuracy on the test set.

Learning Time Sequences

A number of practical applications require neural networks that respond to a sequence of patterns. Figure 3.2(b) exemplifies a case of a temporal pattern consisting of waveform samples. The neural network should produce a particular output in response to a particular sequence of inputs. Speech signals, measured waveform data, and control signals are examples of waves that are considered as discrete-time sequences rather than as unordered data.

One simple way to use the conventional training methods for mapping of training data sequences is to turn the temporal sequence into a spatial input pattern. When this is accomplished, we may consider that the given set of samples in a sequence is fed simultaneously to the network. This case is illustrated in Figure 4.26. It can be seen that a series connection of delay elements provides the solution to the problem. Inputs to the network are samples $x_i = x(t - i\Delta)$, for $i = 0, 1, \dots, n$. The network with tapped delay lines is called a *time-delay network*. The figure shows a single-channel temporal sequence. We may note that for handling multidimensional data sequences, the series connection of delay elements and tapping must be implemented through an appropriate expansion of the input node layer.



Note: Δ is equal to the sampling period

Figure 4.26 A time-delay neural network converting a data sequence into the single data vector (single variable sequence shown).

4.7

FUNCTIONAL LINK NETWORKS

Earlier discussion in this chapter focused on two-layer mapping networks and their training. The hidden layer of neurons provides an appropriate pattern to image transformation, and the output layer yields the final mapping. Instead of carrying out a two-stage transformation, input/output mapping can also be achieved through an artificially augmented single-layer network. The separating hyperplanes generated by such a network are defined in the extended input space. Since the network has only one layer, the mapping can be learned using the simple delta learning rule instead of the generalized delta rule. The concept of training an augmented and expanded network leads to the so-called *functional link network* as introduced by Pao (1989). Functional link networks are single-layer neural networks that are able to handle linearly nonseparable tasks due to the appropriately enhanced input representation.

The key idea of the method is to find a suitably enhanced representation of the input data. Additional input data that are used in the scheme incorporate

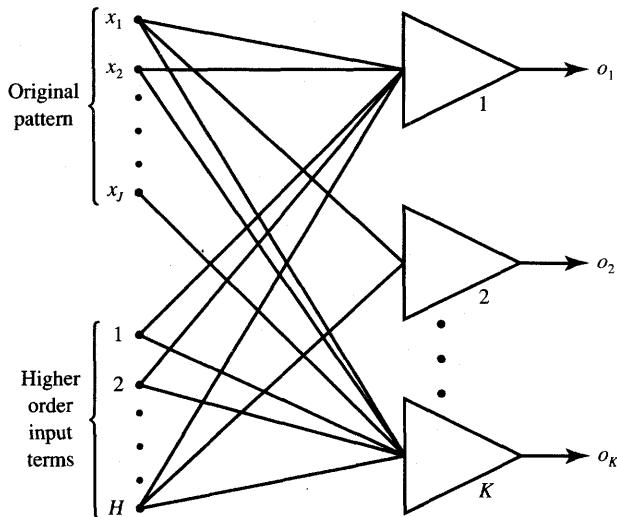


Figure 4.27 Functional link network.

higher order effects and artificially increase the dimension of the input space. The expanded input data are then used for training instead of the actual input data. The block diagram of a functional link network is shown in Figure 4.27. The network uses an enhanced representation of input patterns composed now of $J + H$ components. The higher order input terms applied at H bottom inputs of the functional link network from Figure 4.27 are to be produced in a linearly independent manner from the original pattern components. Although no new information is explicitly inserted into the process, the input representation has been enhanced and the linear separability can be achieved in the extended space. The discussion below covers two methods of extending the dimensionality of the input space for functional link network design.

Assume that the original data are represented by J -tuple vectors. In the so-called *tensor model*, suitable for handling input patterns in the form of vectors, the additional input terms are obtained for each J -dimensional input pattern as the products $x_i x_j$ for all $1 \leq i$ and $j \leq J$ such that $i < j \leq J$ (case A). A number of product terms generated is shown in Figure 4.28. Alternatively, the products can be computed as in case A and augmented with $x_i x_j x_k$ terms for all $1 \leq i, j, k \leq J$ such that $i < j < k \leq J$ (case B). This discussion shows that the number of additional inputs required for the functional link method grows very quickly. The increment H in the size of input vector is also given in Figure 4.28. The figure also lists several first new product terms generated in each of the cases A and B. The following example revisits Example 4.1 and vividly demonstrates the power of the functional link approach.

J	H		Terms generated in Case A	Additional terms generated in Case B
	Case A	Case B		
2	1	1	x_1x_2	None
3	3	4	x_1x_2, x_1x_3, x_2x_3	$x_1x_2x_3$
4	6	10	$x_1x_2, x_1x_3, x_1x_4,$ x_2x_3, x_2x_4, x_3x_4	$x_1x_2x_3, x_1x_3x_4,$ $x_2x_3x_4, x_1x_2x_4$
5	10	20	$x_1x_2, x_1x_3, x_1x_4,$ \dots	$x_1x_2x_3, x_1x_3x_4,$ \dots

Figure 4.28 Increase in input vector size and example additional terms for vector input patterns.

EXAMPLE 4.6

Let us discuss a simple solution of the two-dimensional XOR problem with the functional link network. Both cases A or B yield the single product x_1x_2 as the third input. We thus have $H = 1$. The new, extended input space representation using coordinates $\{x_1, x_2, x_1x_2\}$ of extended XOR input data is shown in Figure 4.29(a). The training set in the new input space consists of four patterns $\{-1 -1 1, -1 1 -1, 1 -1 -1, 1 1 1\}$ and the ordered set output of target values is $\{-1, 1, 1, -1\}$.

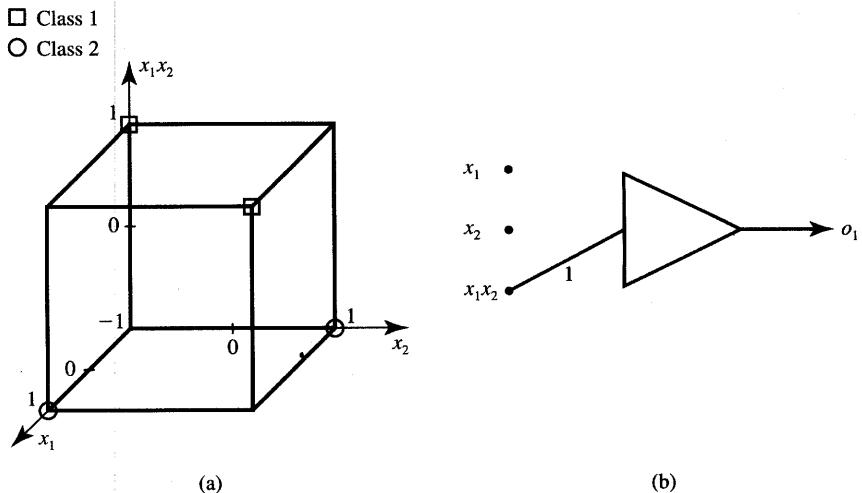


Figure 4.29 Figure for Example 4.6: (a) linear separability through enhanced input representation and (b) network diagram.

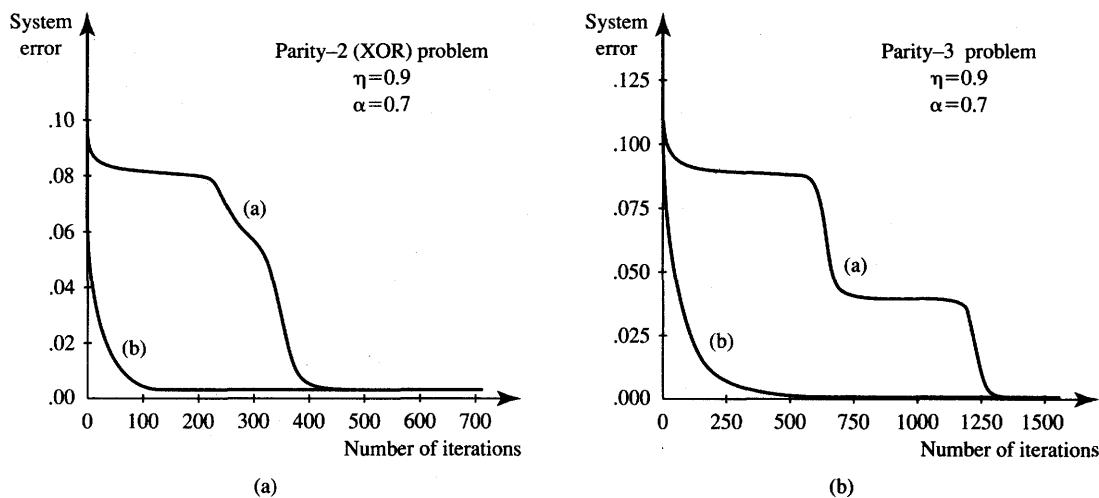


Figure 4.30 Comparison of learning rate between the hidden layer network [curves (a)] and functional link network [curves (b)]: (a) two-variable XOR problem and (b) three-variable XOR problem. [Source: Pao (1989). ©1989 Addison-Wesley; reprinted with permission.]

We now see that the enhanced input patterns are linearly separable. An example equation defining the weights is

$$x_1 x_2 = 0 \quad (4.55)$$

The horizontal plane as in (4.55) passing through the origin provides one of the infinite number of solutions. The related functional link network diagram is shown in Figure 4.29(b).

The learning rates of functional link networks are reportedly significantly better for two- and three-variable XOR problems than those of two-layer networks. Curves (b) of Figure 4.30 obtained for the functional link approach show the dramatic improvement of learning speed over the conventional back-propagation-trained two-layer networks [curves (a)]. ■

In the so-called *functional model* of a functional link network, the higher order input terms are generated using the orthogonal basis functions. Functions such as $\sin \pi x$, $\cos \pi x$, $\sin 2\pi x$, $\cos 2\pi x$, ..., can be used to enhance the representation of input x . A function $f(x)$ of a single variable can also be learned in a single-layer network. Obviously, there is a question about how many of the terms need to be retained to achieve efficient learning of function approximation. Excellent results have been reported by Pao (1989) with learning of an example function $f(x)$ using its 20 sample point pairs $x, f(x)$. The terms that needed to be generated have been x , $\sin \pi x$, $\cos \pi x$, $\sin 2\pi x$, $\cos 2\pi x$, and $\sin 4\pi x$.

The reader has certainly noticed that the network with a single-layer, or so-called "flat" neural network based on the concept of a functional link does not strictly belong to the class of layered networks discussed in this chapter. It has only one layer of neurons. However, due to its intrinsic mapping properties, the functional link network performs very similarly to the multilayer network. The linearly nonseparable patterns acquire an enhanced input representation, so that mapping can be performed through a single layer. The distinct advantage of the functional link network is apparently easier training; however, more research needs to be done to compare conclusively the functional link approach with the error back-propagation alternative.

4.8

CONCLUDING REMARKS

The material in this chapter describes the principal concepts of multilayer feedforward artificial neural networks. Appropriate input space mapping properties are developed starting with the simple prototype point space partitioning and its mapping to the linearly separable image space. The scheme, originally introduced for discrete perceptron networks and visualized in low-dimensional space, is then extended to multilayer neural networks using continuous perceptrons and formulated in terms of network training.

The delta rule and generalized delta rule training algorithms have been developed. It has been illustrated that the back-propagation algorithm solves the training task of an artificial layered neural network to perform potentially arbitrary input-output mappings defined by training examples. The algorithm uses the least mean square error minimization strategy. The gradient of error resulting for the current value of the input pattern is computed. A step in the weight space is then taken along the negative gradient to reduce the current error value. The advantage of the method is that each adjustment step is computed quickly, without presentation of all the patterns and without finding an overall direction of the descent for the training cycle. The discussion of learning parameters has been provided to enhance the understanding of error back-propagation learning performance. Due to the randomness of the minimum search during training, the error back-propagation technique is based on stochastic approximation theory. As such, it provides accurate input-output mapping in a statistical sense.

The basic concepts underlying the idea of classification applications and of intelligent layered neural networks have also been outlined in this chapter. We have seen how back-propagation-trained networks can perform like rule-based expert systems. Application examples of diagnostic layered networks show the feasibility of the neural network approach for the selected practical tasks of classification of characters and engine fault diagnosis. An alternative approach to the generalized delta rule using layered network training has also been presented.

The functional link method is able to implement arbitrary input-output mappings due to the proper expansion of input data representation. It merely requires simple delta rule training since it employs a single-layer network.

PROBLEMS

Please note that problems highlighted with an asterisk (*) are typically computationally intensive and the use of programs is advisable to solve them.

- P4.1* The linearly nonseparable patterns $\mathbf{x}_1, \dots, \mathbf{x}_{10}$ listed below have to be classified in two categories using a layered network and an appropriate pattern-image space transformation. Design a two-layer classifier with the bipolar discrete perceptrons based on the appropriate space mapping.

$$\begin{aligned}\mathbf{x}_1 &= \begin{bmatrix} 1 \\ 3 \\ 3 \end{bmatrix}, \mathbf{x}_2 = \begin{bmatrix} 3 \\ 3 \\ 3 \end{bmatrix}, \mathbf{x}_3 = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}, \mathbf{x}_4 = \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix}, \mathbf{x}_5 = \begin{bmatrix} 3 \\ 2 \\ 2 \end{bmatrix}, \\ \mathbf{x}_6 &= \begin{bmatrix} 2 \\ 1.5 \\ 1.5 \end{bmatrix}, \mathbf{x}_7 = \begin{bmatrix} -2 \\ 0 \\ 0 \end{bmatrix}, \mathbf{x}_8 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \mathbf{x}_9 = \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix}, \mathbf{x}_{10} = \begin{bmatrix} 5 \\ 0 \\ 0 \end{bmatrix}\end{aligned}$$

$\mathbf{x}_4, \mathbf{x}_6, \mathbf{x}_7, \mathbf{x}_8, \mathbf{x}_9$: class 1; remaining patterns: class 2.

- P4.2* Linearly nonseparable patterns as shown in Figure P4.2 have to be classified in two categories using a layered network. Construct the separating planes in the pattern space and draw patterns in the image space. Calculate all weights and threshold values of related TLU units. Use the minimum number of threshold units to perform the classification.

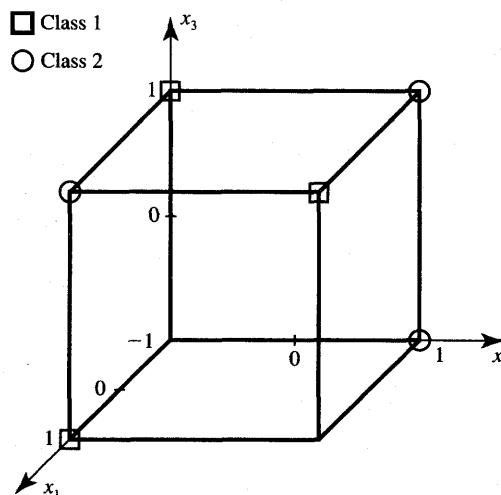
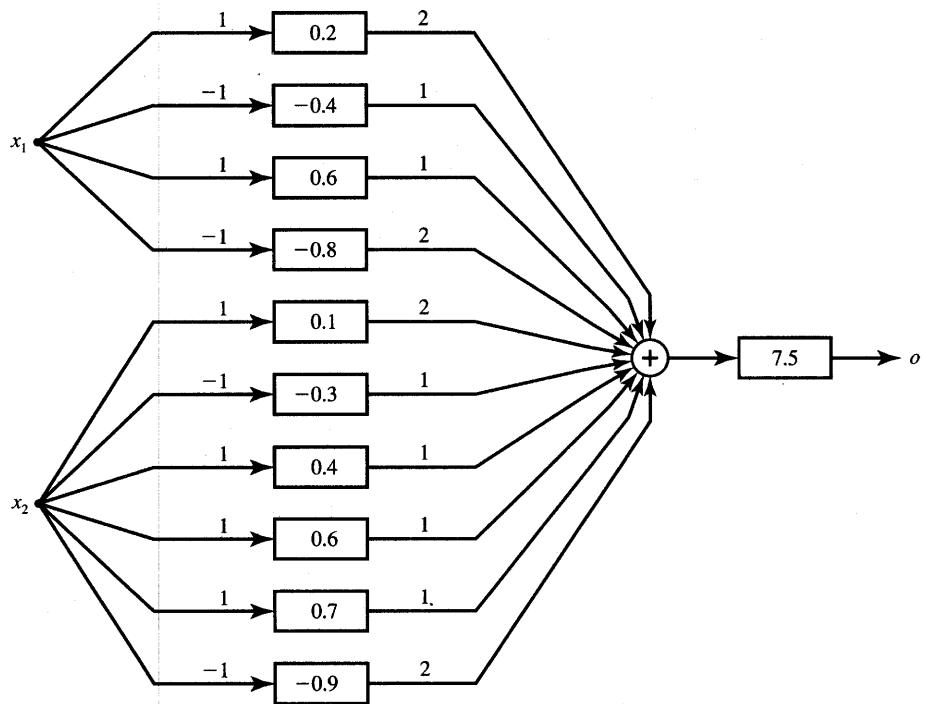


Figure P4.2 Patterns for layered network classification for Problem P4.2.



Note: threshold values marked within the TLU as below

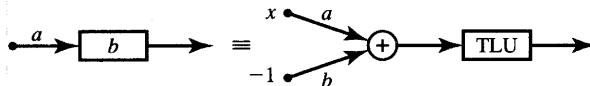


Figure P4.3 Layered discrete perceptron network for Problem P4.3.

P4.3 The layered bipolar perceptron network as shown in Figure P4.3 implements a concave partition of space x_1, x_2 . The values inside TLU elements are their respective thresholds T . Find the subset \mathcal{X} that results in the response $o = 1$ and draw it on the x_1, x_2 plane.

P4.4 Two planar input pattern regions of class 1 and 2 are shown in Figure P4.4. Note that since one decision region is concave, the dichotomization shown is linearly nonseparable. Using the layered bipolar discrete perceptron network with TLU elements, design the dichotomizer to separate pattern classes as shown. Solve the problem for Figure P4.4(a) using two hidden layers and for (b) using a single hidden layer by appropriate input to image space mapping.

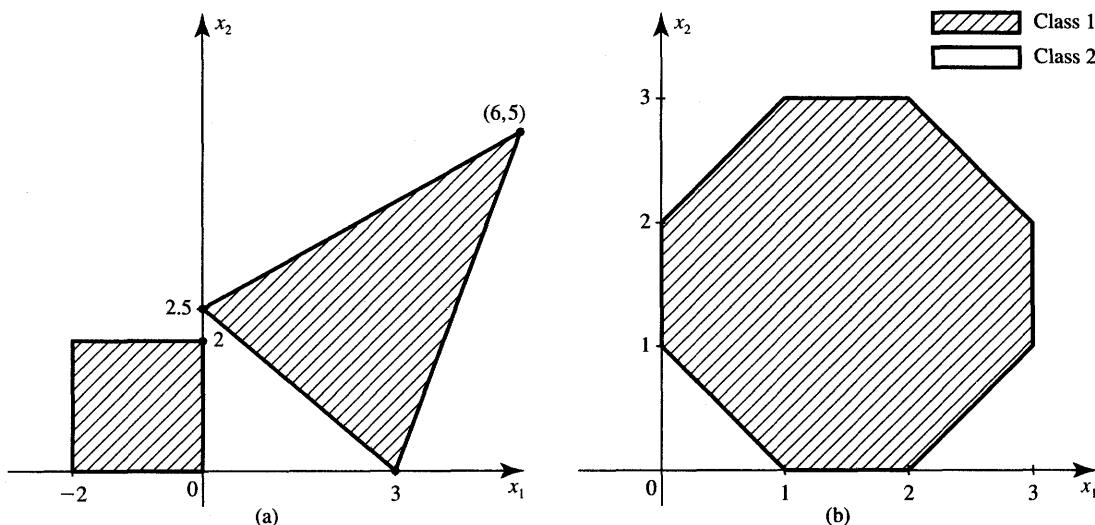


Figure P4.4 Planar images for classification using discrete perceptrons for Problem P4.4.

P4.5 Note that the TLU networks producing a unit step function defined as

$$i_0(\mathbf{w}^t \mathbf{x}) = \begin{cases} 1 & \text{for } \mathbf{w}^t \mathbf{x} > T \\ 0 & \text{for } \mathbf{w}^t \mathbf{x} < T \end{cases}$$

can implement Boolean functions. Find the diagrams of networks consisting of discrete perceptrons and appropriate weights. The networks should realize the following functions (a prime denotes the logic complement operation):

- (a) $F(x_1, x_2, x_3) = x_1 x_2 x_3$
- (b) $F(x_1, x_2, x_3) = x_1 + x_2 + x_3$
- (c) $F(x_1, x_2, x_3) = x_1(x_2 + x_3)$
- (d) $F(x_1, x_2, x_3) = x'_1 x'_2 + x'_1 x_2 x'_3 + x_1(x'_2 x'_3 + x_2 x_3)$
knowing that x_1 , x_2 , and x_3 are Boolean variables 0 and 1, and that the positive logic is used.

- P4.6** A layered TLU network employing a single hidden unipolar TLU as defined in Problem P4.5 and a single output unipolar TLU can be used to implement the XOR function defined in Example 4.1. Partial design of the network is shown in Figure P4.6. Specify all missing weights of the network including the threshold value of the output unit. Note that the image and input pattern data are combined in this case in an attempt to achieve proper response o_4 .

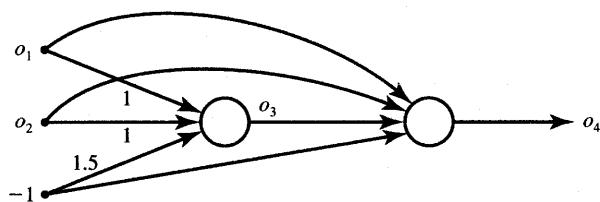


Figure P4.6 Dichotomizer using two unipolar TLU elements for solution of the XOR problem from Problem P4.6.

P4.7 The network shown in the Figure P4.7 and employing unipolar TLU elements similar to those used in Problem P4.5 has been trained to classify all eight three-bit pattern vectors that are vertices of a three-dimensional cube. The training set is $\{\mathbf{o}_i\} = \{0 \ 0 \ 0, \ \dots, \ 1 \ 1 \ 1\}$. Analyze the network and find the function it implements in terms of the inputs o_1 , o_2 , and o_3 .

P4.8 You are presented with the prototypes in augmented form:

$$\begin{aligned}\mathbf{x}_1 &= \begin{bmatrix} 3 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 4 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{x}_3 = \begin{bmatrix} 4 \\ -1 \\ 1 \end{bmatrix}, \quad \mathbf{x}_4 = \begin{bmatrix} 5 \\ 2 \\ 1 \end{bmatrix} \\ \mathbf{x}_5 &= \begin{bmatrix} 5 \\ 3 \\ 1 \end{bmatrix}, \quad \mathbf{x}_6 = \begin{bmatrix} 3 \\ 3 \\ 1 \end{bmatrix}, \quad \mathbf{x}_7 = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{x}_8 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}\end{aligned}$$

A layered machine with two discrete bipolar perceptrons in the hidden layer and a single discrete bipolar output perceptron needs to classify the prototypes so that only \mathbf{x}_1 , \mathbf{x}_2 , and \mathbf{x}_3 belong to class 1, with the remaining prototypes belonging to class 2.

(a) Check whether weight vectors

$$\mathbf{w}_1 = \begin{bmatrix} 2 \\ 1 \\ 5 \end{bmatrix}, \quad \mathbf{w}_2 = \begin{bmatrix} 0 \\ 1 \\ -2 \end{bmatrix}$$

would provide the linear separation of patterns as required.

(b) Repeat part (a) for the new weight vectors:

$$\mathbf{w}_1 = \begin{bmatrix} 0 \\ -1 \\ 1.5 \end{bmatrix}, \quad \mathbf{w}_2 = \begin{bmatrix} 1 \\ 0 \\ -2.5 \end{bmatrix}$$

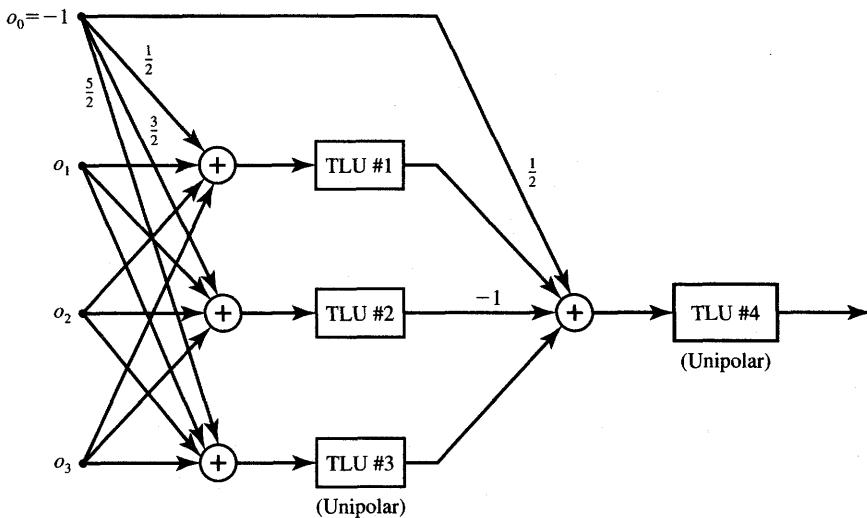


Figure P4.7 Layered discrete perceptron network for Problem P4.7. (All unlabeled weights are of unity value.)

- (c) Complete the design of the classifier by using the results from either part (a) or (b), and compute the weights of the single perceptron at the output.

P4.9 Prove that for $n = 2$, the number of hidden layer neurons J needed for hyperplane partition into M regions is

$$J = \frac{1}{2} (\sqrt{8M - 7} - 1)$$

P4.10 Assume that a two-class classification problem of a planar pattern ($n = 2$) is solved using a neural network architecture with $J = 8$ and $K = 2$. Determine the lower bound on P , which is the number of vectors in the planar training set. This number is equal to the number of separable regions M .

P4.11 Assume that a two-class classification problem for $n = 60$ needs to be solved using $J = 7$ and $K = 2$. Determine the lower bound on P , which is the number of vectors in the training set. Assume that P is equal to the number of separable regions M .

P4.12 Planar input patterns of four classes are shown in Figure P4.12. Using the layered network of bipolar discrete perceptrons, design the classifier for the linearly nonseparable classes shown. Use three perceptrons in the output layer; when none of the three perceptrons responds +1, this indicates class 4.

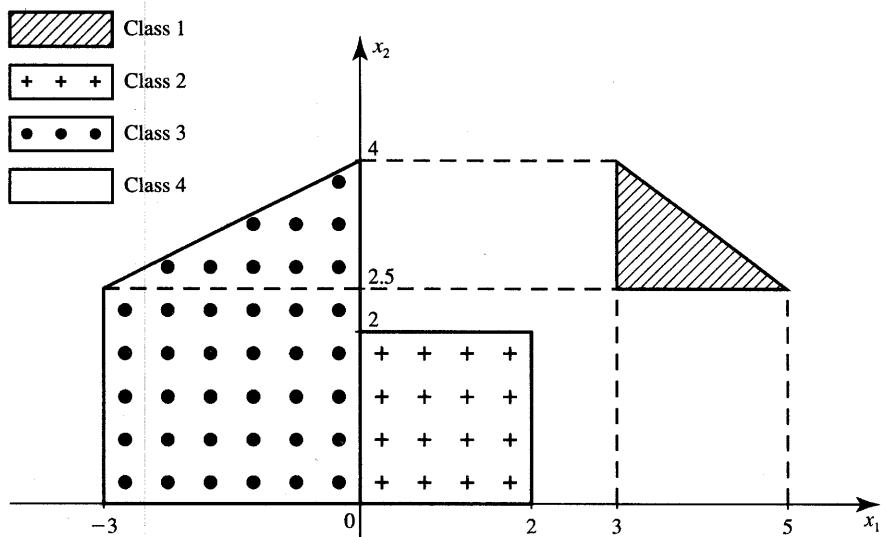


Figure P4.12 Planar images for classification using discrete perceptrons for Problem P4.12.

P4.13* Implement the delta training rule algorithm for a four continuous bipolar perceptron network and a four-dimensional augmented pattern vector. The network trained is supposed to provide classification into four categories of P patterns. Perform the training of the network for pattern data as in Problem P3.1. (The number of training patterns is $P = 8$.) Use local representation, assume $\lambda = 1$ for training.

P4.14 The network shown in Figure P4.14, when properly trained, should respond with

$$\begin{bmatrix} o_1 \\ o_2 \end{bmatrix} = \begin{bmatrix} 0.95 \\ 0.05 \end{bmatrix}$$

to the augmented input pattern

$$\begin{bmatrix} z_1 \\ z_2 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ -1 \end{bmatrix}$$

The network weights have been initialized as shown in the figure. Analyze a single feedforward and back-propagation step for the initialized network by doing the following:

- (a) Find weight matrices V and W .
- (b) Calculate net_j , y , net_k , and \mathbf{o} .

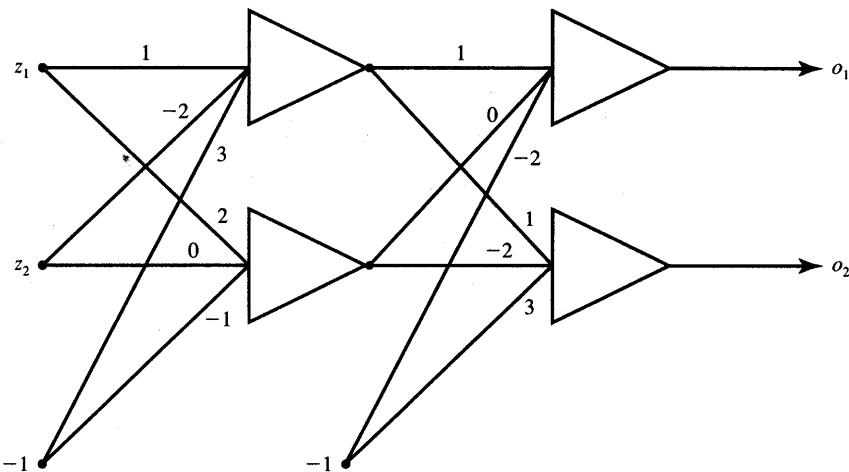


Figure P4.14 Neural network for Problem P4.14. (Initial weights shown.)

- (c) Calculate slopes $f'(net_j)$ and $f'(net_k)$.
- (d) Compute error signals δ_o and δ_y .
- (e) Compute ΔV and ΔW .
- (f) Find updated weights.

For computations assume $f(net) = [1 + \exp(-net)]^{-1}$ and $\eta = 1$.

P4.15* Write a program implementing the error back-propagation training algorithm (EBPTA) for user-selectable I , J , and K values for a single hidden layer network. The flowchart of the algorithm is outlined in Section 4.4. Learning constant η should be user-selectable; no momentum term is needed. The initial weights for the network should be selected at random. Provisions for specification of input pattern(s) and the desired response(s) should be made in order to initiate and carry out the training. Use bipolar continuous perceptrons.

P4.16* (This problem requires the use of a back-propagation training program written by the student in Problem P4.15, or available from other sources.)

Implement the classifier of three printed characters A, I, and O as shown in Figure P4.16. Set an appropriate E_{rms} value such that an error-free classification is assured. Assume no momentum term; try different η values. Evaluate the number of training cycles for comparable η values for two different architectures. The target values should be selected as $(1 \quad -1 \quad -1)$ for A, $(-1 \quad 1 \quad -1)$ for I, and $(-1 \quad -1 \quad 1)$ for O.

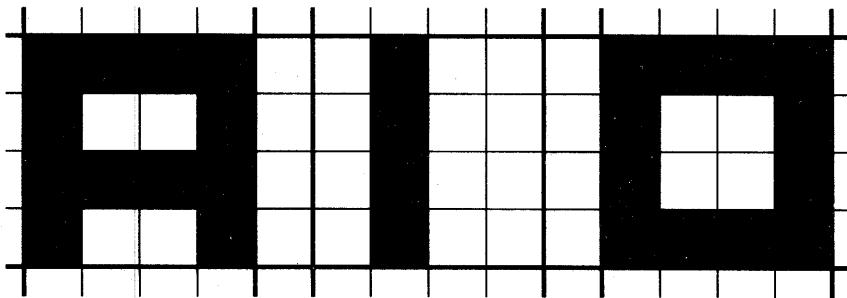


Figure P4.16 Characters A, I, and O for Problem P4.16.

Assume two different input representations according to the following guidelines:

- (a) 16 pixels, three input vectors in the training set, $n = 16$ ($I = 17$, $J = 9$, and $K = 3$)
- (b) two coordinates of each pixel, 16 input vectors in the training set, $n = 2$ ($I = 3$, $J = 9$, and $K = 3$).

P4.17 The network shown in Figure P4.17 has been trained to classify correctly a number of two-dimensional, two-class inputs.

- (a) Draw the separating lines between the two classes on the x_1, x_2 plane assuming that the neurons operate with the discrete bipolar activation function.

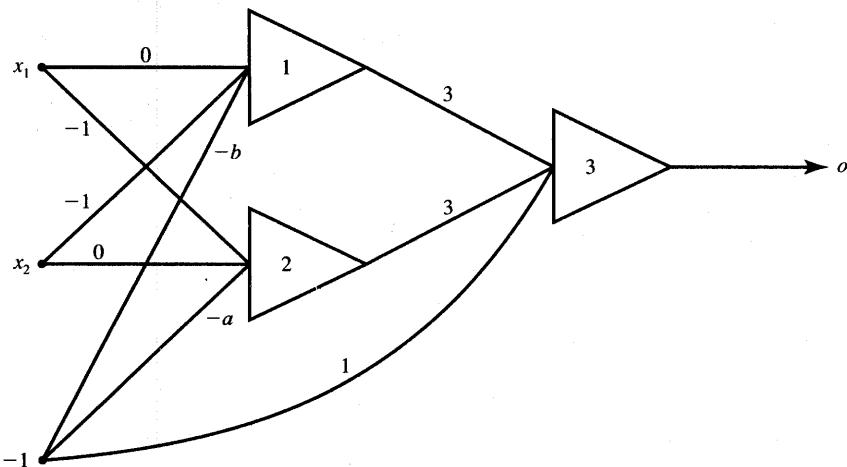


Figure P4.17 Trained neural network for Problem P4.17.

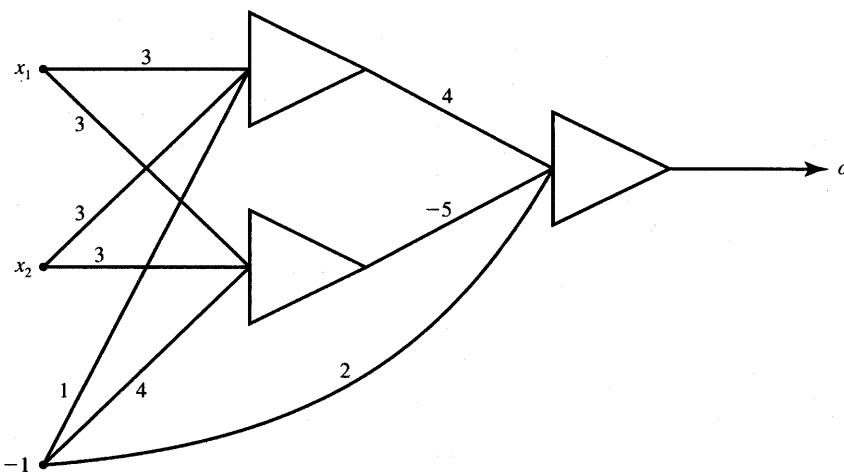


Figure P4.18 Trained neural network for Problem P4.18.

- (b) Assume the bipolar continuous activation function of neurons. Now, the thresholding above and below certain levels of output can be used to assert the membership in classes. Generate equations for the region of uncertainty in classification on both sides of the borderline analyzed in part (a) if the following criterion for indecision is used: $o > 0.9$, class 1; $o < -0.9$, class 2. Perform computations assuming for simplicity $a = b = 0$.

P4.18 The network shown in Figure P4.18 has been trained to classify correctly a set of two-dimensional, two-class patterns.

- (a) Identify the function performed by the classifier, assuming initially that the neurons have unipolar discrete activation function. Draw the resulting separating lines between the two classes on the x_1, x_2 plane.
- (b) Generate 36 points of the test inputs within the $[0, 1]$ square on the x_1, x_2 plane by incrementing each of the coordinates by 0.2. By performing recall for each of the test patterns, find the responses for the network with continuous perceptrons assuming the unipolar activation function $o = [1 + \exp(-net)]^{-1}$.

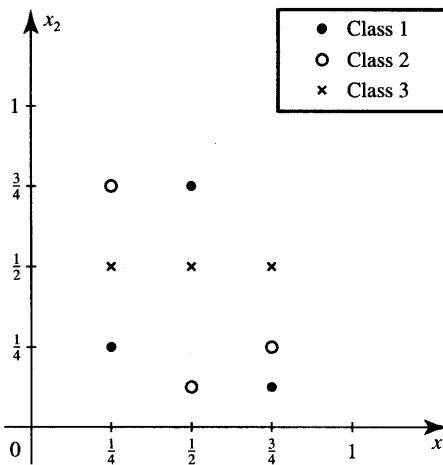


Figure P4.19 Training pattern set for neural network design in Problem P4.19.

P4.19 Design a feedforward network for the example training set containing linearly nonseparable patterns of three classes shown in Figure P4.19.

- (a) Select a suitable number of neurons in the hidden layer of the network using $n = 2$. Use $K = 3$ so that the target vectors are $[1 \ -1 \ -1]^t$, $[-1 \ 1 \ -1]^t$, and $[-1 \ -1 \ 1]^t$ for classes 1, 2, and 3, respectively.
- (b)* Train the network as selected in part (a) by using the nine vectors of the set shown. Select the E_{rms} value that assures an error-free classification. Assume no momentum term, try different η values. (This part of the problem requires the use of a back-propagation training program written by the student in Problem P4.15 or available from other sources.)

P4.20 The network consisting of unipolar discrete perceptrons responding 0 for negative input, or otherwise 1, and shown in Figure P4.20 has been designed to detect one of the aircraft fault modes (Passino, Sartori, and Antsaklis 1989). Quantities Θ (deg) and q (deg/s) denote the aircraft's pitch angle and pitch rate, respectively.

- (a) Find the decision regions in coordinates Θ, q indicating the specific failure by analyzing the network diagram.
- (b) Simplify the network architecture, if possible, by reducing the number of hidden nodes J .

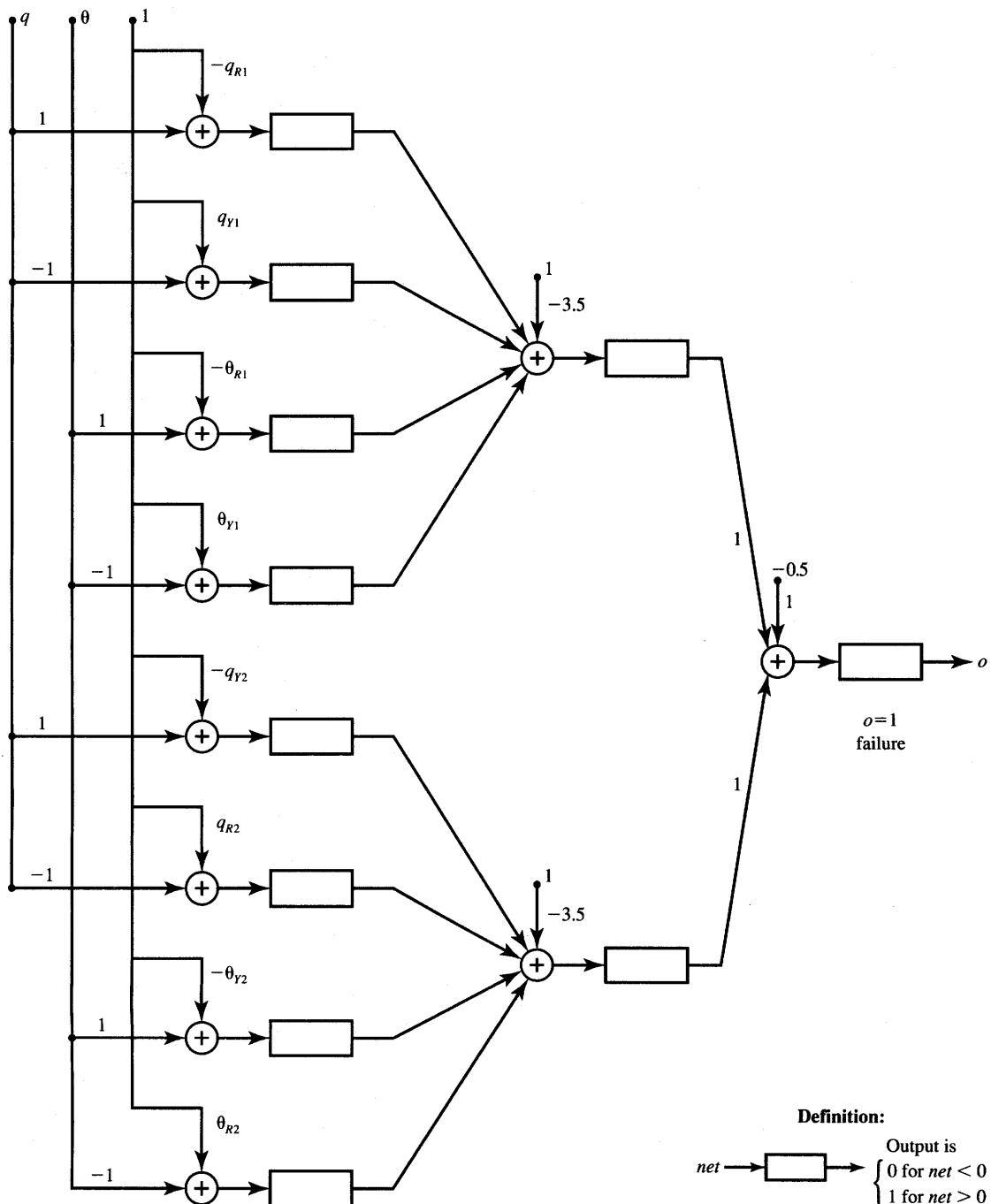


Figure P4.20 Failure detector discrete perceptron network for Problem P4.20.

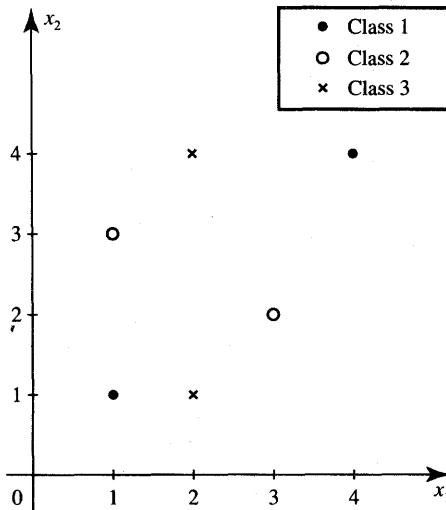


Figure P4.22 Training set for Problem P4.22.

P4.21 A functional link network based on the tensor model has been trained with the following extended input vectors

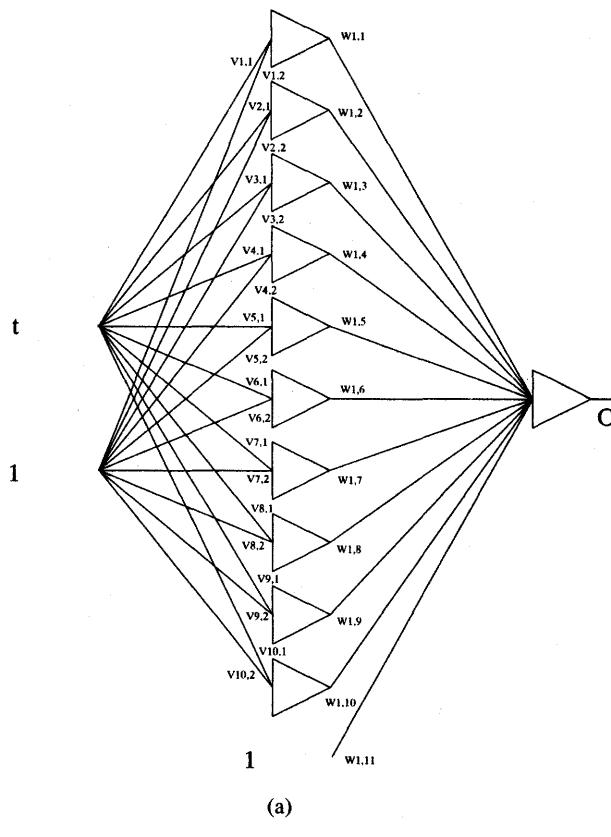
$$\begin{bmatrix} 8 \\ -8 \\ -64 \end{bmatrix}, \begin{bmatrix} -7 \\ 5 \\ -35 \end{bmatrix} \text{ of class 1, and}$$

$$\begin{bmatrix} 6 \\ 12 \\ 72 \end{bmatrix}, \begin{bmatrix} 10 \\ 8 \\ 80 \end{bmatrix}, \begin{bmatrix} 1 \\ -2 \\ -2 \end{bmatrix}, \begin{bmatrix} -3 \\ -5 \\ 15 \end{bmatrix} \text{ of class 2.}$$

- (a) Determine whether the training patterns in the original space are linearly separable.
- (b) Derive the set of weights of functional link network using the single TLU that would satisfy the required classification.

P4.22 Design a functional link classifier using the tensor model for a set of training patterns as shown in Figure P4.22. Use the delta learning rule for training of the single-layer network.

P4.23 Figure P4.23(a) shows the network that approximates a continuous function of a single variable t . The network uses bipolar continuous perceptrons, has 10 hidden layer units, and a single output unit. Knowing that the network weights are as listed in Figure P4.23(b),



$$\mathbf{V}^t = \begin{bmatrix} 1.12 & 2.46 & 6.11 & -1.08 & 0.96 & -1.03 & -0.58 & -1.11 & 1.13 & 1.05 \\ 0.36 & 0.27 & 0.09 & 0.28 & 0.24 & -0.29 & 0.12 & -0.34 & 0.05 & 0.06 \end{bmatrix}$$

$$\mathbf{W} = \begin{bmatrix} -1.35 & 0.14 & 4.26 & 1.18 & -1.02 & 1.20 & 0.55 & 1.33 & -1.27 & -1.20 & 0.45 \end{bmatrix}$$

(b)

Figure P4.23 Network for function approximation: (a) network diagram and (b) weight matrices.

find numerically the function implemented by the network by performing recall of selected test patterns in the range $-1 \leq t \leq 1$.

P4.24* Figure P4.24 illustrates an accurate function $h(x)$ (dashed line):

$$h(x) = 0.8 \sin \pi x, \quad \text{for } -1 \leq x \leq 1$$

and its neural network approximation $o(x)$ (continuous line). It can be seen that the training conditions have been $P = 11$, $\eta = 0.4$, and

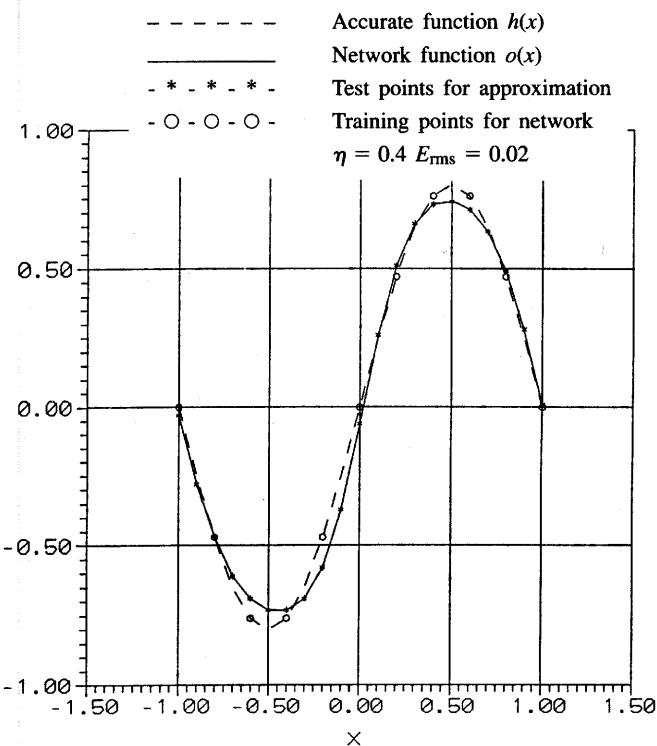


Figure P4.24 Approximating $h(x)$ with the neural network of Problem P4.24.

$E_{\text{rms}} = 0.02$. It can also be seen that 21 test points uniformly covering the range $[-1, 1]$ have been used.

Using the simulation conditions for training and testing the network as illustrated on the figure, find a neural network that performs the required approximation. Use single hidden-layer network and bipolar continuous perceptrons.

REFERENCES

- Bengio, Y., R. Cardio, R. DeMori, and P. Cosi. 1989. "Use of Multilayered Networks for Coding Speech with Multi-Phonetic Speeches," in *Advances in Neural Information Processing Systems*, vol. 1, ed. D. S. Touretzky. San Mateo, Calif.: Morgan Kaufmann Publishers, pp. 224–231.
- Burr, D. 1988. "Experiments on Neural Net Recognition of Spoken and Written Text," *IEEE Trans. Acoustics, Speech, and Signal Proc.* 36: 1162–1168.

- Cybenko, G. 1990. "Complexity Theory of Neural Networks and Classification Problems," in *Neural Networks EURASIP Workshop Proc.*, ed. L. B. Almeida, C. J. Wellekens. Sesimbra, Portugal, February 1990, pp. 24-44.
- Dietz, W. E., E. L. Kiech, and M. Ali. 1989. "Jet and Rocket Engine Fault Diagnosis in Real Time," *J. Neural Network Computing* (Summer): 5-18.
- Funanashi, K. I. 1989. "On the Approximate Realization of Continuous Mappings by Neural Networks," *Neural Networks* 2: 183-192.
- Gallant, S. I. 1988. "Connectionist Expert Systems," *Comm. ACM* 31(2): 152-169.
- Gorman, R., and T. Sejnowski. 1988. "Learned Classification of Sonar Targets Using a Massively Parallel Network," *IEEE Trans. Acoustics, Speech, and Signal Proc.* 36: 1135-1140.
- Hornik, K., M. Stinchcombe, and H. White. 1989. "Multilayer Feedforward Networks Are Universal Approximators," *Neural Networks* 2: 359-366.
- Hripcsak, G. 1988. "Problem-Solving Using Neural Networks," San Diego, Calif.: SAIC Communication.
- Jacobs, R. A. 1988. "Increased Rates of Convergence Through Learning Rate Adaptation," *Neural Networks* 1:295-307.
- Karin, E. D. 1990. "A Simple Procedure for Pruning Back-Propagation Trained Neural Networks," *IEEE Trans. on Neural Networks* 1(2): 239-242.
- Leung, H. C., and V. W. Zue. 1989. "Applications of Error Back-Propagation to Phonetic Classification," in *Advances in Neural Information Processing Systems*, vol. 1, ed. D. S. Touretzky. San Mateo, Calif.: Morgan Kaufmann Publishers, pp. 206-214.
- Marko, K. A., J. James, J. Dosdall, and J. Murphy. 1989. "Automotive Control System Diagnostics Using Neural Nets for Rapid Pattern Classification of Large Data Sets," in *Proc. 2nd Int. IEEE Joint Conf. on Neural Networks*, Washington, D.C., June 18-22, pp. 13-17.
- McClelland, J. L., D. E. Rumelhart, and the PDP Research Group. 1986. *Parallel Distributed Processing*. Cambridge: The MIT Press.
- Mirchandini, G., and W. Cao. 1989. "On Hidden Nodes in Neural Nets," *IEEE Trans. Circuits and Systems* 36(5): 661-664.
- Narendra, K. S., and K. Parthasarathy. 1990. "Identification and Control of Dynamical Systems using Neural Networks," *IEEE Trans. on Neural Networks* 1(1): 4-21.
- Nilsson, N. J. 1965. *Learning Machines: Foundations of Trainable Pattern Classifiers*. New York: McGraw Hill Book Co.; also republished as *The Mathematical Foundations of Learning Machines*. San Mateo, Calif.: Morgan Kaufmann Publishers.

- Pao, Y. H. 1989. *Adaptive Pattern Recognition and Neural Networks*. Reading, Mass.: Addison-Wesley Publishing Co.
- Passino, K. M., M. A. Sartori, and P. J. Antsaklis. 1989. "Neural Computing for Numeric-to-Symbolic Conversion in Control Systems," *IEEE Control Systems Magazine*, (April): 44-51.
- Poggio, T., and F. Girosi. 1990. "Networks for Approximation and Learning," *Proc. IEEE* 78(9): 1481-1497.
- Sejnowski, T., and C. Rosenberg. 1987. "Parallel Networks that Learn to Pronounce English Text," *Complex Systems* 1: 145-168.
- Sietsma, J., and R. J. F. Dow. 1988. "Neural Network Pruning—Why and How," in *Proc. 1988 IEEE Int. Conf. on Neural Networks*, San Diego, California, vol. I, pp. 325-333.
- Silva, F. M., and L. B. Almeida. 1990. "Acceleration Technique for the Back Propagation Algorithm," in *Neural Networks EURASIP Workshop Proc.*, ed. L. B. Almeida, C. J. Wellekens. Sesimbra, Portugal, February 1990, pp. 110-119.
- Tsyplkin, Ya. Z. 1973. *Foundations of the Theory of Learning Systems*. New York: Academic Press.
- Waibel, A. 1989. "Consonant Recognition by Modular Construction of Large Phonemic Time-Delay Networks," in *Advances in Neural Information Processing Systems*, vol. 1, ed. D. S. Touretzky. San Mateo, Calif.: Morgan Kaufmann Publishers, pp. 215-223.
- Werbos, P. 1974. "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," Ph.D. diss., Harvard University.
- White, H. 1989. "Learning in Artificial Neural Networks: A Statistical Perspective," *Neural Computation* 1(4): 425-469.
- Wieland, A., and R. Leighton. 1988. "Geometric Analysis of Neural Network Capabilities," MP-88 W 00022. McLean, Va.: Mitre Corporation.

SINGLE-LAYER FEEDBACK NETWORKS

*Who errs and mends, to
God himself commends.*

CERVANTES

- 5.1 Basic Concepts of Dynamical Systems
- 5.2 Mathematical Foundations of Discrete-Time Hopfield Networks
- 5.3 Mathematical Foundations of Gradient-Type Hopfield Networks
- 5.4 Transient Response of Continuous-Time Networks
- 5.5 Relaxation Modeling in Single-Layer Feedback Networks
- 5.6 Example Solutions of Optimization Problems
- 5.7 Concluding Remarks

The neural networks introduced in this chapter represent dynamical systems evolving in time in either a continuous, or discrete, output space. The movement of the network is usually characterized by many degrees of freedom. It is also dissipative in the sense that the movement is in the direction of lower so-called computational energies exhibited by the system. As will be shown, transition in a dynamical neural network is toward an asymptotically stable solution that is a local minimum of a dissipated energy function.

The feedforward networks presented in Chapters 3 and 4 have no feedback activity during the information recall phase. Let us look in more detail at the role and presence of feedback in feedforward networks. As discussed in earlier chapters, the feedback interactions in single- and multilayer perceptron-type networks occurred during their learning, or training, phase. The adjustment of weights during training gradually reduced the overall output error. The error reduction embedded in one of the supervised training algorithms provided that feedback-type interaction. Noticeably, that interaction was teacher-enforced rather than spontaneous, and externally imposed rather than occurring within the

system. Feedback interactions within the network ceased once the training had been completed.

The single-layer neural networks discussed in this chapter are inherently feedback-type nonlinear networks. Neurons with either a hard-limiting activation function or with a continuous activation function can be used in such systems. As discussed in Chapter 2, recall of data from within the dynamical system requires time. To recall information stored in the network, an input pattern is applied, and the network's output is initialized accordingly. Next, the initializing pattern is removed and the initial output forces the new, updated input through feedback connections. The first updated input forces the first updated output. This, in turn, produces the second updated input and the second updated response. The process of sequential updating continues, and the activity of outputs for a properly designed, or stable, network becomes less frequent. The transition process continues until no new updated responses are produced and the network has reached its equilibrium.

In the case of a *discrete-time operation*, also called recursive, *single-layer feedback networks* can be termed as *recurrent*. A recurrent network is a discrete-time dynamical system, which, at any given instant of time, is characterized by a binary output vector. Examples of such networks are illustrated in Figures 2.10, 2.11, and 2.12. The recurrent networks' sequential updating process described in the preceding paragraph can be considered either discrete synchronous or discrete asynchronous in time. A more extensive analysis of recurrent single-layer feedback networks with discrete neurons and an explanation of their update dynamics will be pursued in Chapter 6 in the framework of associative memory architectures.

The discussion of single-layer feedback networks in this chapter focuses on networks operating in *continuous time* and with continuous output vector values. Accordingly, continuous activation functions are assumed for such networks and the updating is simply continuous in time at every network output. Noticeably, network outputs are also neuron outputs since the network is with single layer.

Examples of continuous-time single-layer feedback networks are shown in Figures 2.14 and 2.15. In this chapter we will look at the relevant facets, such as dynamics, design, and updating schemes, of single-layer neural systems that store knowledge in their stable states. These networks fulfill certain assumptions that make the class of networks stable and useful, and their behavior predictable in most cases.

Remember, however, that fully coupled single-layer neural networks represent nonlinear feedback systems. Such systems are known, in general, to possess rather complex dynamics. Fortunately, single-layer feedback networks represent a class of networks that allows for great reduction of the complexity. As a result, their properties can be controlled and solutions utilized by neural network designers. This presents possibilities for solving optimization problems and applying such networks to modeling of technological and economical systems.

The networks discussed here are based on the seminal papers of Hopfield (1984, 1985, 1986). However, many years of development in the areas of continuous- and discrete-time systems have contributed to the existing state-of-the-art single-layer feedback networks. As we will see, the networks can be useful in many ways. They can provide associations or classifications, optimization problem solution, restoration of patterns, and, in general, as with perceptron networks, they can be viewed as mapping networks. Despite some unsolved problems and limitations of the fully coupled single-layer networks, their impressive performance has been documented in the technical literature. Both hardware implementations and their numerical simulations indicate that single-layer feedback networks provide a useful alternative to traditional approaches for pattern recognition, association, and optimization problems.

5.1

BASIC CONCEPTS OF DYNAMICAL SYSTEMS

The neural networks covered in this chapter are dynamical systems. As such, they process the initial condition information over time while moving through a sequence of states. Let us look at a classical problem of recognition of a distorted alphanumerical character. Characters shown in bit map form can be represented as vectors consisting of the binary variables 0 and 1. Specific example letters to be recognized are presented in Figure 3.2(a), Figure 4.19, and in Example 4.5, Figure 4.23(a). Taking a different perspective than in Chapter 4, we will now design a dynamical system so that its states of equilibrium correspond to the set of selected character vectors.

Let us consider several basic concepts before we examine their definitions more closely in Chapter 6. An *attractor* is a state toward which the system evolves in time starting from certain initial conditions. Each attractor has its set of initial conditions, which initiates the evolution terminating in that attractor. This set of conditions for an attractor is called the *basin of attraction*. If an attractor is a unique point in state space, then it is called a *fixed point*. However, an attractor may consist of a periodic sequence of states, in which case it is called the *limit cycle*, or it may have a more complicated structure.

Using the concept and properties of single-layer feedback networks, a dynamical system can be postulated and designed so that it has three vectors representing the letters C, I, and T from Figure 4.19 as fixed points. Moreover, the system can have its basins of attractions shaped in such a way that it will evolve toward an appropriate character vector even if the initializing vector does not exactly resemble the original character. Thus, a certain amount of noise or distortion of the character vector not only can be tolerated by the system, but even removed under the appropriate conditions. In our approach we are more interested in the conclusions that systems can reach than in the transient evolutions from

any initial conditions that are provided to initialize the transients. However, to enhance the understanding of dynamical network performance, a study of both aspects is necessary.

Let us note that the information about the stationary point of a dynamical system must be coded in its internal parameters and not in its initial conditions. A nontrivial recognition or association task will obviously require a large amount of information to be stored in system parameters, which are its weights. Therefore, dynamical neural networks need to have their numerous parameters selected carefully. Accordingly, their state sequences will evolve toward suitable attractors in multidimensional space. In addition, the system will have to be inherently nonlinear to be useful.

The learning of parameters of a dynamical system is dependent on the network model adopted. A variety of learning methods is applied to store information in single-layer feedback networks. Although correlation or Hebbian learning is primarily used, many applications would require a highly customized learning approach. Typically, the learning of dynamical systems is accomplished without a teacher. The adjustment of system parameters, which are functions of patterns or associations to be learned, does not depend on the difference between the desired and actual output value of the system during the learning phase.

One of the inherent drawbacks of dynamical systems is their very limited explanation capability. The solutions offered by the networks are hard to track back or to explain and are often due to random factors. This can make it difficult to accept the conclusion, but it does not reduce the networks' levels of performance. However, it should be stressed that the dynamical systems approach to cognitive tasks is still in an early development stage. The application of the dynamical models to real-size optimization or association problems beyond their present scope will require a great deal of further scientific development.

5.2

MATHEMATICAL FOUNDATIONS OF DISCRETE-TIME HOPFIELD NETWORKS

The attention given currently to this class of networks and to their mathematical model is due to their very interesting intrinsic properties. Following the development of the theory of this class of networks in the early and mid-1980s, modern microelectronic and optoelectronic technology has made it possible to fabricate microsystems based on the formulated network model (Howard, Jackel, and Graf 1988; Alspector et al. 1988). This section reviews the main properties of the continuous-time dynamical system model. A discussion of feedback network implementation issues is provided in Chapter 9.

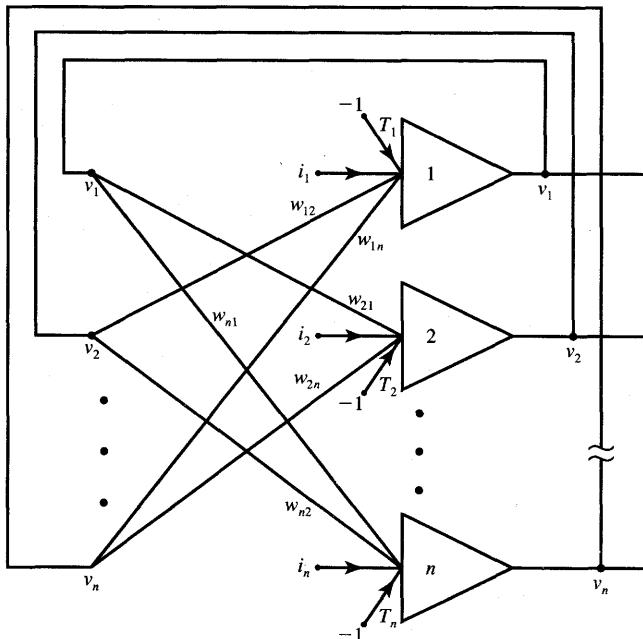


Figure 5.1 Single-layer feedback neural network.

Following the postulates of Hopfield, the single-layer feedback neural network is assumed as shown in Figure 5.1. It consists of n neurons having threshold values T_i . The feedback input to the i 'th neuron is equal to the weighted sum of neuron outputs v_j , where $j = 1, 2, \dots, n$. Denoting w_{ij} as the weight value connecting the output of the j 'th neuron with the input of the i 'th neuron, we can express the total input net_i of the i 'th neuron as

$$net_i = \sum_{\substack{j=1 \\ j \neq i}}^n w_{ij} v_j + i_i - T_i, \quad \text{for } i = 1, 2, \dots, n \quad (5.1a)$$

Note that this notation is consistent with the double subscript notation used in Chapters 2, 3, and 4. The external input to the i 'th neuron has been denoted here as i_i . Introducing the vector notation for synaptic weights and neuron output, Equation (5.1a) can be rewritten as

$$net_i = \mathbf{w}_i^T \mathbf{v} + i_i - T_i, \quad \text{for } i = 1, 2, \dots, n \quad (5.1b)$$

where

$$\mathbf{w}_i \triangleq \begin{bmatrix} w_{i1} \\ w_{i2} \\ \vdots \\ w_{in} \end{bmatrix}$$

is the weight vector containing weights connected to the input of the i 'th neuron, and

$$\mathbf{v} \triangleq \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

is the neural network output vector, also called the output vector. The complete matrix description of the linear portion of the system shown in Figure 5.1 is given by

$$\text{net} = \mathbf{W}\mathbf{v} + \mathbf{i} - \mathbf{t} \quad (5.2)$$

where

$$\text{net} \triangleq \begin{bmatrix} \text{net}_1 \\ \text{net}_2 \\ \vdots \\ \text{net}_n \end{bmatrix}, \quad \mathbf{i} \triangleq \begin{bmatrix} i_1 \\ i_2 \\ \vdots \\ i_n \end{bmatrix}$$

are vectors containing activations and external inputs to each neuron, respectively. The threshold vector \mathbf{t} has been defined here as

$$\mathbf{t} \triangleq \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_n \end{bmatrix}$$

Matrix \mathbf{W} , sometimes called the *connectivity matrix*, is an $n \times n$ matrix containing network weights arranged in rows of vectors \mathbf{w}_i^t as defined in (5.1b) and it is equal to

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^t \\ \mathbf{w}_2^t \\ \vdots \\ \mathbf{w}_n^t \end{bmatrix}$$

In the expanded form this matrix becomes

$$\mathbf{W} = \begin{bmatrix} 0 & w_{12} & w_{13} & \cdots & w_{1n} \\ w_{21} & 0 & w_{23} & \cdots & w_{2n} \\ w_{31} & w_{32} & 0 & \cdots & w_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & w_{n3} & \cdots & 0 \end{bmatrix}$$

Note that we assume the weight matrix \mathbf{W} in this model that is symmetrical, i.e., $w_{ij} = w_{ji}$, and with diagonal entries equal explicitly to zero, i.e., $w_{ii} = 0$. (See Figure 5.1; no connection exists from any neuron back to itself). Physically, this

condition is equivalent to the lack of the self-feedback in the nonlinear dynamical system of Figure 5.1. Summarizing, the j 'th neuron output is connected to each of the neurons' inputs through a multiplicative synaptic weight w_{ij} , for $i = 1, 2, \dots, n$, but it is not connected to its own input.

Let us assume momentarily that the neuron's activation function is $\text{sgn}(\cdot)$, like this of a TLU element. This causes the following response, or update, of the i 'th neuron excited as in (5.1) or (5.2):

$$\begin{aligned} v_i &\rightarrow -1 \text{ if } \text{net}_i < 0 \\ v_i &\rightarrow +1 \text{ if } \text{net}_i > 0 \end{aligned} \quad (5.3)$$

Transitions indicated by right arrows based on the update rule of (5.3) are taking place at certain times. If the total input to a particular neuron gathered additively as a weighted sum of outputs plus the external input applied is below the neuron's threshold, the neuron will have to move to, or remain in, the inhibited state. The net value as in (5.1a) exceeding zero would result in the excitatory state $+1$ immediately after the update. Let us note that the resulting state $+1$ of the i 'th neuron is either preceded by the transition or not.

The rule (5.3) of interrogating the neurons' weighted outputs and updating the output of the i 'th neuron is applied in an *asynchronous* fashion. This means that for a given time, only a single neuron is allowed to update its output, and only one entry in vector \mathbf{v} as in (5.1b) is allowed to change. The next update in a series uses the already updated vector \mathbf{v} . In other words, under asynchronous operation of the network, each element of the output vector is updated separately, while taking into account the most recent values for the elements that have already been updated and remain stable. This mode of update realistically models random propagation delays and random factors such as noise and jitter. Such phenomena would indeed be likely to occur in an artificial neural network using high-gain neurons described with an activation function close to $\text{sgn}(\text{net})$.

Formalizing the update algorithm (5.3) for a discrete-time recurrent network and using (5.1), we can obtain the following update rule:

$$v_i^{k+1} = \text{sgn}(\mathbf{w}'\mathbf{v}^k + i_i - T_i), \quad \text{for } i = 1, 2, \dots, n \text{ and } k = 0, 1, \dots \quad (5.4a)$$

where superscript k denotes the index of recursive update.

The update scheme in (5.4a) is understood to be asynchronous, thus taking place only for one value of i at a time. Note that the right arrow in (5.3) has been replaced by the delay between the right and left sides of (5.4a). The update superscript of v_i^{k+1} refers here to the discrete-time instant, and it could be replaced with $v_i[(k+1)t]$, or simply with $v_i(k+1)$, where t denotes the neurons' update interval assumed here of unity value. The recursion starts at \mathbf{v}^0 , which is the output vector corresponding to the initial pattern submitted. The first iteration for $k = 1$ results in v_i^1 , where the neuron number, i , is random. The other updates are also for random node number j , resulting in updates v_j^1 , $j \neq i$, until all updated elements of the vector \mathbf{v}^1 are obtained based on vector \mathbf{v}^0 . This

particular update algorithm is referred to as an *asynchronous stochastic recursion* of the Hopfield model network.

The matrix equation (5.2) can be used as an alternative to express the recursive update algorithm (5.3). In such a case we have

$$\mathbf{v}^{k+1} = \Gamma [\mathbf{W}\mathbf{v}^k + \mathbf{i} - \mathbf{t}], \quad \text{for } k = 0, 1, \dots, \quad (5.4b)$$

where $\text{sgn}(\cdot)$ operates on every scalar row of the bracketed matrix. However, caution should now be exercised since formula (5.4b) describes the synchronous, or parallel, update algorithm. Under this update mode, all n neurons of the layer, rather than a single one, are allowed to change their output simultaneously. Indeed, starting at vector \mathbf{v}^0 , entries of vector \mathbf{v}^1 are concurrently computed according to (5.4b) based on the originally initialized \mathbf{v}^0 value, then \mathbf{v}^2 is computed using \mathbf{v}^1 , etc.

The asynchronous update scheme requires that once an updated entry of vector \mathbf{v}^{k+1} has been computed for a particular step, this update is substituted for the current value \mathbf{v}^k and used to calculate its subsequent update. This process should continue until all entries of \mathbf{v}^{k+1} have been updated. The recursive computation continues until the output node vector remains unchanged with further iterations. It thus can be said that when using formula (5.4b), each of the recursion steps for $k = 0, 1, \dots$ should be divided into n individual, randomly sequenced single neuron updates.

It is rather illustrative to visualize the vector of neuron outputs \mathbf{v} in n -dimensional space. The output vector is one of the vertices of the n -dimensional cube $[-1, 1]$ in E^n space. The vector moves during recursions (5.4) from vertex to vertex, until it stabilizes in one of the 2^n vertices available. Note that the movement is from a vertex to an adjacent vertex since the asynchronous update mode allows for a single-component update of an n -tuple vector at a time. The final position of \mathbf{v}^k , as $k \rightarrow \infty$, is determined by weights, thresholds, inputs, and the initial vector \mathbf{v}^0 . It is also determined by the order of transitions.

Some of the open questions are whether the system has any attractors, and whether it stabilizes, which we have assumed so far without proof; and, if so, is there any link between the initial pattern \mathbf{v}^0 and its final value $\mathbf{v}^k, k \rightarrow \infty$. Further, it is interesting to see how the equilibrium is reached and how many attractors can be stored, if any, in the system discussed. Finally, we will be understandably interested to see how networks of this type can be designed and their weights computed to suit specific mapping needs.

To evaluate the stability property of the dynamical system of interest, let us study a so-called *computational energy function*. This is a function usually defined in n -dimensional output space v^n . The motivation for such choice of space is that system specifications are given most often in terms of its desired outputs and are usually available in output space as opposed to the neuron input space, which is the state space. Also, the space v^n is bounded to within the $[-1, 1]$ hypercube, including its walls, edges, and vertices. If the increments of a certain

bounded positive-valued computational energy function under the algorithm (5.3) are found to be nonpositive, then the function can be called a *Liapunov function*, and the system would be asymptotically stable (see the Appendix).

The scalar-valued energy function for the discussed system is a quadratic form (see the Appendix) and has the matrix form

$$E \triangleq -\frac{1}{2} \mathbf{v}' \mathbf{W} \mathbf{v} - \mathbf{i}' \mathbf{v} + \mathbf{t}' \mathbf{v} \quad (5.5a)$$

or, in the expanded form, it is equal to

$$E \triangleq -\frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n w_{ij} v_i v_j - \sum_{i=1}^n i_i v_i + \sum_{i=1}^n t_i v_i \quad (5.5b)$$

Let us study the changes of the energy function for the system which is allowed to update. Assume that the output node i has been updated at the k 'th instant so that $v_i^{k+1} - v_i^k = \Delta v_i$. Since only the single neuron computes, the scheme is one of asynchronous updates. Let us determine the related energy increment in this case. Computing the energy gradient vector, we obtain from (5.5a) (see the Appendix)

$$\nabla E = -\frac{1}{2} (\mathbf{W}' + \mathbf{W}) \mathbf{v} - \mathbf{i}' + \mathbf{t}' \quad (5.6a)$$

which reduces for symmetrical matrix \mathbf{W} for which $\mathbf{W}' = \mathbf{W}$ to the form

$$\nabla E = -\mathbf{W} \mathbf{v} - \mathbf{i}' + \mathbf{t}' \quad (5.6b)$$

The energy increment becomes equal:

$$\Delta E = (\nabla E)' \Delta \mathbf{v} \quad (5.6c)$$

Since only the i 'th output is updated, we have

$$\Delta \mathbf{v} \triangleq \begin{bmatrix} 0 \\ \vdots \\ \Delta v_i \\ \vdots \\ 0 \end{bmatrix}$$

and the energy increment (5.6c) reduces to the form

$$\Delta E = (-\mathbf{w}_i' \mathbf{v} - i_i + t_i) \Delta v_i \quad (5.6d)$$

This can be rewritten as:

$$\Delta E = - \left(\sum_{j=1}^n w_{ij} v_j + i_i - t_i \right) \Delta v_i, \quad \text{for } j \neq i \quad (5.6e)$$

or briefly

$$\Delta E = -\text{net}_i \Delta v_i$$

Inspecting the update rule (5.3), we see that the expression in parentheses in (5.6d) and (5.6e), which is equal to net_i , and the current update value of the output node, Δv_i , relate as follows:

- (a) when $\text{net}_i < 0$, then $\Delta v_i \leq 0$, and
 - (b) when $\text{net}_i > 0$, then $\Delta v_i \geq 0$
- (5.7)

The observations stated here mean that under the update algorithm discussed, the product term $\text{net}_i \Delta v_i$ is always nonnegative. Thus, any corresponding energy changes ΔE in (5.6d) or (5.6e) are nonpositive. We therefore can conclude that the neural network undergoing transitions will either decrease or retain its energy E as a result of each individual update.

We have shown that the nonincreasing property of the energy function E is valid only when $w_{ij} = w_{ji}$. Otherwise, the proof of nonpositive energy increments does not hold entirely. Indeed, if no symmetry of weights is imposed, the corresponding energy increments under the algorithm (5.3) take on the following value:

$$\Delta E = - \left[\sum_{j=1}^n \frac{1}{2} (w_{ij} + w_{ji}) v_j + i_i - t_i \right] \Delta v_i \quad (5.8a)$$

and the term in brackets of (5.8a) is different from net_i . Therefore, the energy increment [(5.6d) and (5.6e)] becomes nonpositive under rule (5.3) if $w_{ij} = w_{ji}$ without any further conditions on the network. The asymmetry of the weight matrix \mathbf{W} , however, may lead to the modified neural networks, which are also stable (Roska 1988). Since the performance of the modified and original symmetric networks remains similar and it is much easier to design a symmetric network, we will focus further consideration on the symmetric connection model. The additional condition $w_{ii} = 0$ postulated for the discussed network will be justified later in this chapter when the performance of the continuous-time model performance network is discussed. For the asynchronous update scheme, one of the network's stable states which acts as an attractor is described by the solution

$$\mathbf{v}^{k+1} = \lim_{k \rightarrow \infty} \text{sgn}(\mathbf{W}\mathbf{v}^k + \mathbf{i} - \mathbf{t}) \quad (5.8b)$$

We have found so far that the network computing rule (5.3) results in a nonincreasing energy function. However, we must now show that the energy function indeed has a minimum, otherwise the minimization of the energy function would not be of much use. Let us note that since the weight matrix \mathbf{W} is indefinite because of its zero diagonal, then the energy function E has neither a minimum nor maximum in unconstrained output space. This is one of the properties of quadratic forms such as (5.5). However, the function E is obviously bounded in n -dimensional space consisting of the 2^n vertices of n -dimensional cube. Thus, the energy function has to reach its minimum finally under the update algorithm (5.3).

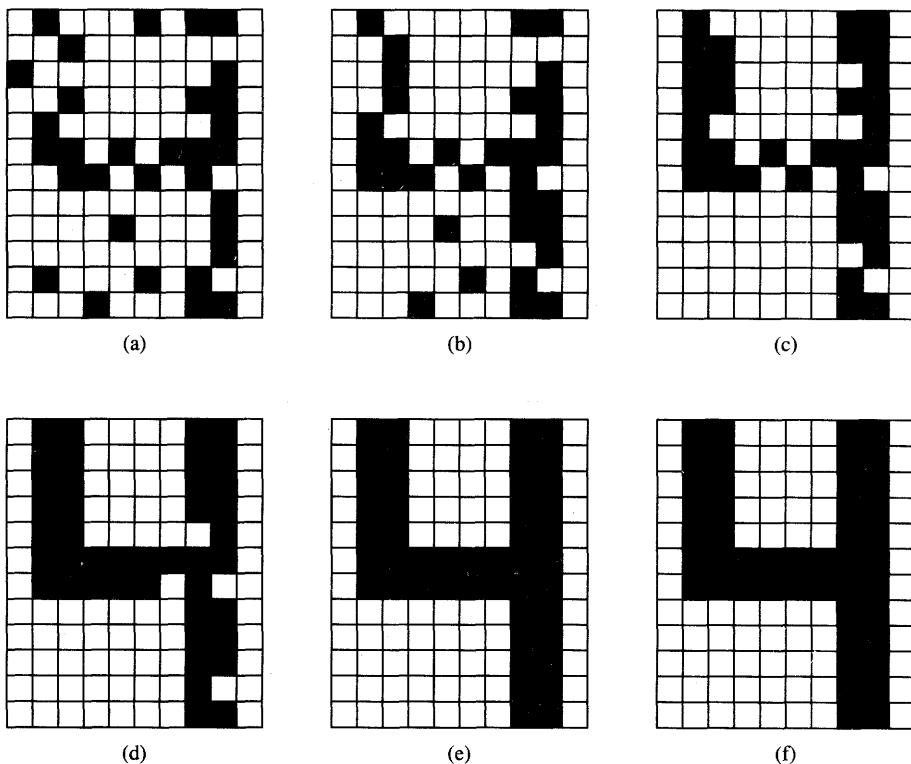


Figure 5.2 Example of recursive asynchronous update of corrupted digit 4: (a) $k = 0$, (b) $k = 1$, (c) $k = 2$, (d) $k = 3$, (e) $k = 4$, and (f) $k = 5$.

Example snapshots of neural network transitions were given in Figure 1.8. Another example of convergence of a 10×12 neuron network is illustrated in Figure 5.2. It shows the 10×12 bit map of black and white pixels representing the digit 4 during the update as defined in (5.3). Figure 5.2(a) has been made for $k = 0$, and it shows the initial, distorted digit 4 with 20% of the pixels randomly reversed. Consecutive responses are shown in Figure 5.2(b) through (f). It can be seen that the updates continue until $k = 4$ as in Figure 5.2(e), and for $k \geq 5$ no changes are produced at the network output since the system arrived into one of its stable states (5.8b).

It has been shown in the literature that the synchronous state updating algorithm can lead to persisting cyclic states that consist of two complementary patterns (Kamp and Hasler 1990). Consider the 2×2 weight matrix with zero diagonal and off-diagonal entries of -1 , and the synchronous updates of the output vector $\mathbf{v}^0 = [-1 \ -1]^t$. By processing the signal \mathbf{v}^0 once we obtain

$$\mathbf{v}^1 = \Gamma \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \end{bmatrix} = \begin{bmatrix} \text{sgn}(1) \\ \text{sgn}(1) \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

In the following update of \mathbf{v}^1 we obtain

$$\mathbf{v}^2 = \Gamma \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \text{sgn}(-1) \\ \text{sgn}(-1) \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

Thus we obtain the same vector as \mathbf{v}^0 , and the synchronous update has produced a cycle of two states rather than a single equilibrium state. The cycle consists of two complementary states. Also, we will see later that complementary patterns correspond to identical energy levels. The following example of a discrete-time neural network illustrates how the energy function and the update scheme (5.3) are related while network is undergoing asynchronous updating.

EXAMPLE 5.1

Let us look at the energy distribution and output updating process of the fully coupled single-layer network from Example 2.2. Recall that its weight matrix is

$$\mathbf{W} = \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix} \quad (5.9a)$$

For the threshold and external inputs assumed zero, the energy function (5.5) becomes

$$E(\mathbf{v}) = -\frac{1}{2} [\mathbf{v}_1 \ \mathbf{v}_2 \ \mathbf{v}_3 \ \mathbf{v}_4] \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \mathbf{v}_3 \\ \mathbf{v}_4 \end{bmatrix} \quad (5.9b)$$

and after rearrangements we obtain

$$E(\mathbf{v}) = -\mathbf{v}_1(\mathbf{v}_2 + \mathbf{v}_3 - \mathbf{v}_4) - \mathbf{v}_2(\mathbf{v}_3 - \mathbf{v}_4) + \mathbf{v}_3\mathbf{v}_4 \quad (5.9c)$$

The system produces discrete energy levels of value -6 , 0 , and 2 for bipolar binary neurons used in this example. The reader can verify this statement by analyzing the energy levels (5.9c) for each binary vector starting at $[-1 \ -1 \ -1 \ -1]^T$ and ending at $[1 \ 1 \ 1 \ 1]^T$. Figure 5.3 shows the energy levels computed for each of the 2^4 binary vectors, which are vertices of the four-dimensional cube that the four-neuron network can represent.

Each edge of the state diagram depicts the single asynchronous state transition. As can be seen from the figure, which shows energy values marked at each cube vertex, the transitions under rule (5.3) indeed displace the state vector toward a lower energy value. Since the energy value decreases at each transition, the network seeks the energy minimum of -6 ,

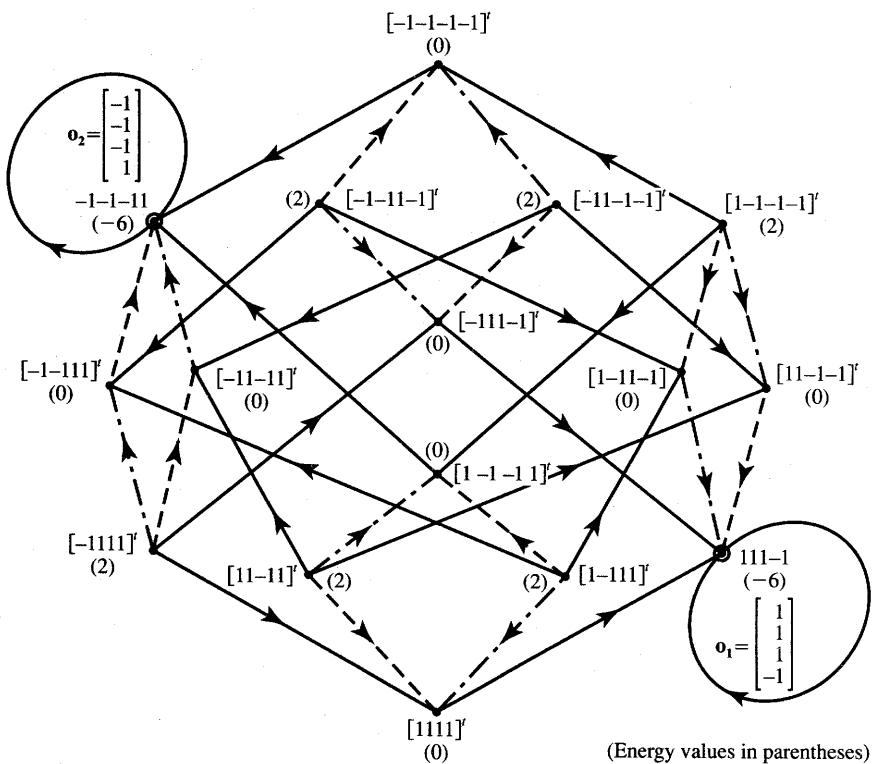


Figure 5.3 Energy levels and state transitions for the network of Example 5.1.

which is either at $\mathbf{o}_1 = [1 \ 1 \ 1 \ -1]^t$ or at $\mathbf{o}_2 = [-1 \ -1 \ -1 \ 1]^t$. Therefore, the example network always stabilizes at energy level $E = -6$ provided transitions are asynchronous.

Note that synchronous transitions, which were originally considered in Example 2.2, neither minimize the energy function nor are they stable. The reader may easily verify that starting at $\mathbf{v}^0 = [1 \ -1 \ 1 \ 1]^t$, the first synchronous transition results in

$$\mathbf{v}^1 = \begin{bmatrix} -1 \\ 1 \\ -1 \\ -1 \end{bmatrix}$$

and then the network starts oscillating between \mathbf{v}^0 and \mathbf{v}^1 since

$$\mathbf{v}^2 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ 1 \end{bmatrix} = \mathbf{v}^0$$

Thus, the energy stabilizes at the nonminimum level of 2 associated with each of the two complementary states.

In contrast to the observed phenomena, asynchronous transitions always lead to one of the energy minima. Moreover, under the asynchronous and random updating algorithm, the convergence to a minimum of E is essentially unique and nonrepetitive. Specifically, the sequence of updating bits determines to which of the equilibrium state minima the network converges. Indeed, let us look at the movement of the network starting at $v^0 = [-1 \ 1 \ 1 \ 1]^t$. Inspecting the transition map of Figure 5.3 we see that if the updates begin with the first bit, then the transitions have to end up at o_1 . If, however, the second bit of v^0 is allowed to update first, the transitions end up at o_2 .

5.3

MATHEMATICAL FOUNDATIONS OF GRADIENT-TYPE HOPFIELD NETWORKS

Gradient-type neural networks are generalized Hopfield networks in which the computational energy decreases continuously in time. Time is assumed to be a continuous variable in gradient-type networks. Such networks represent a generalization of the discrete-time networks introduced in the preceding section. As discussed in Chapter 2, the discrete-time networks introduced in the previous section can be viewed as the limit case of continuous-time networks. For a very high gain λ of the neurons, continuous-time networks perform similarly to discrete-time networks. In the limit case, $\lambda \rightarrow \infty$, they will perform identically. The *continuous-time single-layer feedback networks*, also called *gradient-type networks*, are discussed in this section.

Specifically, gradient-type networks converge to one of the stable minima in the state space. The evolution of the system is in the general direction of the negative gradient of an energy function. Typically, the network energy function is made equivalent to a certain objective (penalty) function that needs to be minimized. The search for an energy minimum performed by gradient-type networks corresponds to the search for a solution of an optimization problem. Gradient-type neural networks are examples of nonlinear, dynamical, and asymptotically stable systems. The concept, fundamentals, and examples of such networks will be discussed below. In the discussion of their properties, we will again use the scalar energy function $E[v(t)]$ in the n -dimensional output space v^n .

Although single-layer feedback networks can be completely described by a set of ordinary nonlinear differential equations with constant coefficients, modeling of such a set of equations by a physical system provides us with better

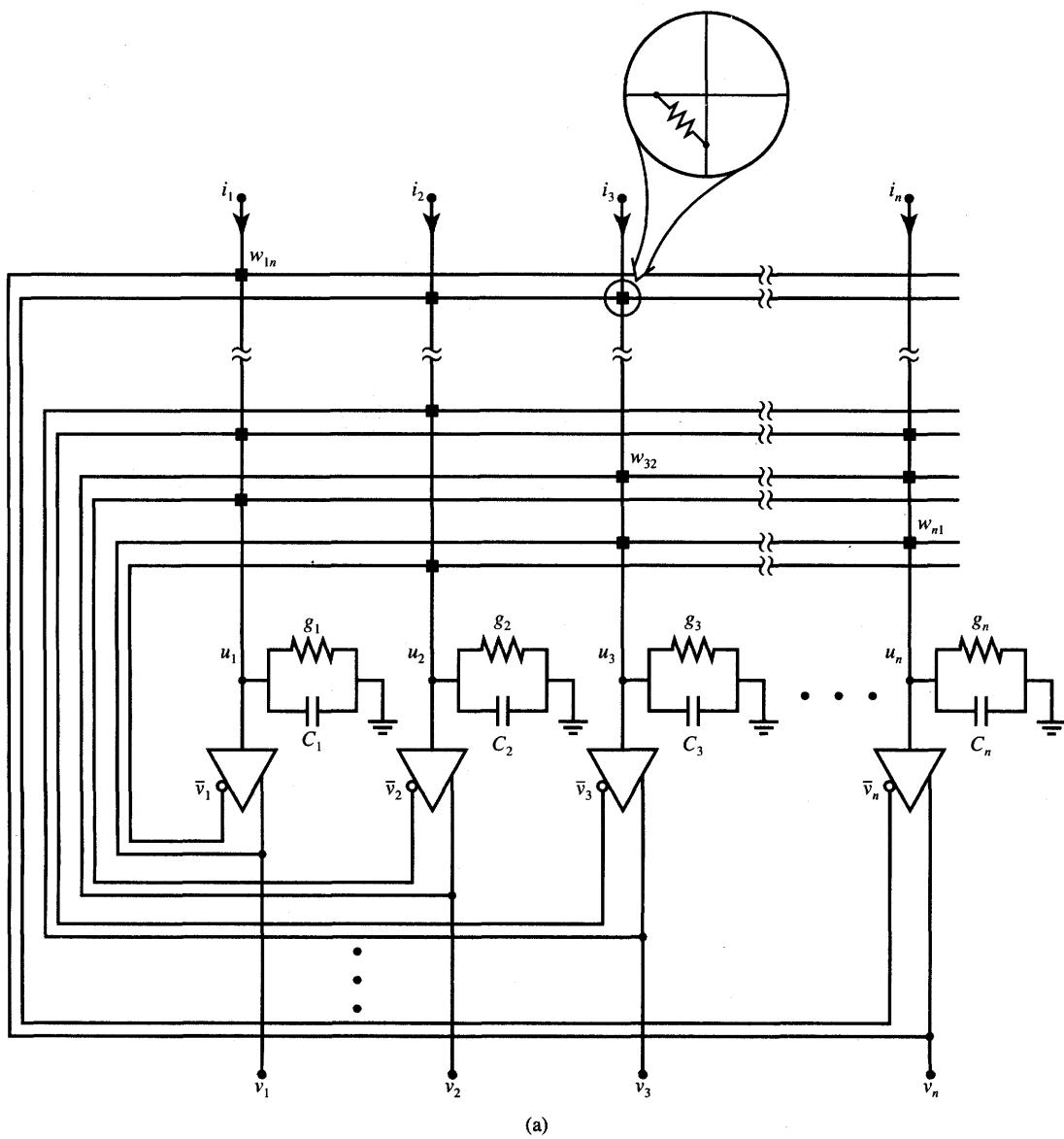


Figure 5.4a The neural network model using electrical components: (a) diagram.

insight into such networks' behavior. It also provides the link between the theory and implementation. The model of a gradient-type neural system using electrical components is shown in Figure 5.4(a). It consists of n neurons, each mapping its input voltage u_i into the output voltage v_i through the activation function $f(u_i)$, which is the common static voltage transfer characteristic (VTC) of the neuron.

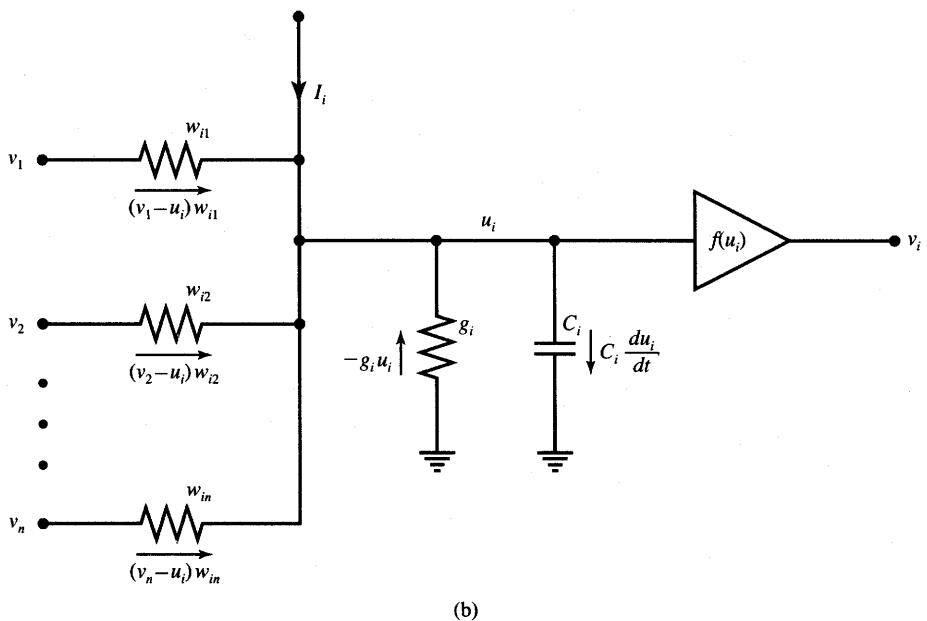


Figure 5.4b The neural network model using electrical components (continued): (b) input node of the i^{th} neuron.

To preserve the original notation of seminal Hopfield's papers, the variable u is used for this network to denote a neuron's activation; for other neural networks the symbol *net* is adopted in this text. Any high-gain voltage amplifier with saturation could be used in this model as a replacement for a neuron. Conductance w_{ij} connects the output of the j^{th} neuron to the input of the i^{th} neuron. The inverted neuron outputs \bar{v}_i are usually tapped at inverting rather than noninverting outputs to avoid negative conductance values w_{ij} connecting, in inhibitory mode, the output of the j^{th} neuron to the input of the i^{th} neuron. Note that the postulated network is required to be symmetric, i.e., $w_{ij} = w_{ji}$. Also, since it has been assumed that $w_{ii} = 0$, the outputs of neurons are not connected back to their own inputs.

Let us note that the electrical model of the Hopfield network shown in the figure has conductance g_i between the i^{th} neuron input and ground. It represents the nonzero input conductance of the i^{th} neuron. With the conductance extracted, the neuron can be considered as a parasitic-free voltage amplifier absorbing no current. Similarly, C_i represents the nonzero input capacitance of the i^{th} neuron. Capacitances C_i , for $i = 1, 2, \dots, n$, are responsible for the dynamics of the transients in this model.

Let us derive mathematical description of this network. The input of the i^{th} neuron is shown in Figure 5.4(b). The KCL equation for the input node having

potential u_i can be obtained as

$$i_i + \sum_{\substack{j=1 \\ j \neq i}}^n w_{ij}v_j - u_i \left(\sum_{\substack{j=1 \\ j \neq i}}^n w_{ij} + g_i \right) = C_i \left(\frac{du_i}{dt} \right) \quad (5.10)$$

The left side of Equation (5.10) represents the total current entering the capacitance C_i . It is a sum of $n - 1$ currents of value $(v_j - u_i)w_{ij}$, for $j = 1, 2, \dots, n$, $j \neq i$, the current i_i , and the current $-g_i u_i$. For the sake of clarity, all the branch currents involved in the summation at the left side of (5.10) have been marked in Figure 5.4(b). The reader can easily notice that the feedback current term $\sum_{j=1}^n (v_j - u_i)w_{ij}$, $j \neq i$, has been split in (5.10) to properly group terms with time variables. Since the weights w_{ij} transform the voltage $v_j - u_i$ into currents, they have dimensions of conductances and are expressed in this example in mhos. Denoting the total conductance connected to the neuron's input node i as G_i , where

$$G_i \triangleq \sum_{j=1}^n w_{ij} + g_i,$$

Equation (5.10) can be simplified to the form of a single state equation as

$$i_i + \sum_{j=1}^n w_{ij}v_j - u_i G_i = C_i \left(\frac{du_i}{dt} \right) \quad (5.11)$$

Let us now introduce the matrices \mathbf{C} and \mathbf{G} defined as

$$\mathbf{C} \triangleq \text{diag}[C_1, C_2, \dots, C_n] \text{ and } \mathbf{G} \triangleq \text{diag}[G_1, G_2, \dots, G_n]$$

and arrange $u_i(t)$, $v_i(t)$, and i_i in n -dimensional column state vector $\mathbf{u}(t)$, output vector $\mathbf{v}(t)$, and the bias current vector \mathbf{i} . The final equations of the entire model network consisting of the *state equation* (5.12a) and the *output equation* (5.12b) written in matrix form can now be expressed from (5.11) for $i = 1, 2, \dots, n$ as

$$\mathbf{C} \frac{d\mathbf{u}(t)}{dt} = \mathbf{W}\mathbf{v}(t) - \mathbf{G}\mathbf{u}(t) + \mathbf{i} \quad (5.12a)$$

$$\mathbf{v}(t) = \mathbf{f}[\mathbf{u}(t)] \quad (5.12b)$$

Let us study the stability of the system described by the ordinary nonlinear differential equations (5.12). The stability of the equations can be evaluated using a generalized computational energy function $E[\mathbf{v}(t)]$. The function $E[\mathbf{v}(t)]$ has not yet been determined in our discussion. We do not know the form of the function; however, we assume that it exists. It is obviously, as before, a scalar-valued function of output vector \mathbf{v} . The time derivative of $E[\mathbf{v}(t)]$ can easily be obtained using the chain rule as

$$\frac{dE[\mathbf{v}(t)]}{dt} = \sum_{i=1}^n \frac{\partial E(\mathbf{v})}{\partial v_i} \dot{v}_i \quad (5.13a)$$

where

$$\dot{v}_i = \frac{d}{dt} v_i$$

This may be written compactly as a scalar product of the vectors

$$\frac{dE[\mathbf{v}(t)]}{dt} = \nabla E(\mathbf{v}) \dot{\mathbf{v}} \quad (5.13b)$$

where $\nabla E(\mathbf{v})$ denotes the gradient vector

$$\nabla E(\mathbf{v}) \triangleq \begin{bmatrix} \frac{\partial E(\mathbf{v})}{\partial v_1} \\ \frac{\partial E(\mathbf{v})}{\partial v_2} \\ \vdots \\ \frac{\partial E(\mathbf{v})}{\partial v_n} \end{bmatrix}$$

Note that the computational energy function $E[\mathbf{v}(t)]$ is defined in n -dimensional output space v^n where the designed neural system specifications are usually known. The corresponding energy function in the state space u^n of an asymptotically stable system would be the system's Liapunov function. If its time derivative is found to be negative, then the energy function in the output space also has a negative derivative, since Equation (5.12b) describes a monotonic mapping of space u^n into space v^n . The formal verification of this property will be presented below.

The suitable energy function for the system in Figure 5.4 has the following form:

$$E(\mathbf{v}) = -\frac{1}{2} \mathbf{v}' \mathbf{W} \mathbf{v} - \mathbf{i}' \mathbf{v} + \sum_{i=1}^n G_i \int_0^{v_i} f_i^{-1}(z) dz \quad (5.14)$$

where $f_i^{-1}(z)$ is the inverse of the activation function f_i . Although the time argument has been omitted in (5.14), for notational convenience, it is understood that vector \mathbf{v} represents $\mathbf{v}(t)$. Let us see why and how the energy function (5.14) can provide us with insight into the system of Equation (5.11).

First, the threshold term of (5.5a) has for simplicity been absorbed into the $\mathbf{i}' \mathbf{v}$ term in (5.14). Also, the third term of the energy expression containing the integral of the inverse of the activation function has been introduced in (5.14) to account for the property of continuous activation function of neurons. We may now evaluate how the energy function (5.14) varies with time. Calculating the rate of change of energy for symmetric \mathbf{W} using the chain rule and the property

$$\frac{d}{dv_i} \left(G_i \int_0^{v_i} f_i^{-1}(z) dz \right) = G_i u_i$$

yields the following result for the time derivative of the energy (see the Appendix):

$$\frac{dE}{dt} = (-\mathbf{Wv} - \mathbf{i} + \mathbf{Gu})^t \dot{\mathbf{v}} \quad (5.15)$$

Noticeably, the expression for the gradient vector has now resulted within the parentheses of (5.15) before the transpose. The state equation (5.12a) can now be rewritten using the gradient symbol:

$$-\nabla E(\mathbf{v}) = \mathbf{C} \frac{d\mathbf{u}}{dt} \quad (5.16)$$

From (5.16) we see that the negative gradient of the energy function $E(\mathbf{v})$ is directly proportional to the speed of the state vector \mathbf{u} , with the capacitance C_i being the proportionality coefficient of the state vector's i 'th component. The appropriate changes of u_i are in the general direction of the negative gradient component $\partial E / \partial v_i$. The velocity $d\mathbf{u}/dt$ of the state vector is not strictly in the negative gradient direction, but always has a positive projection on it (see the Appendix). This is illustrated in Figure 5.5(a), which shows the equipotential, or equal energy, lines in v^2 space of an example neural network consisting of two neurons. Other examples of this class of networks and its properties will be discussed in more detail later in this chapter.

It is now interesting to see how the energy E of this system varies with time. Combining (5.13) and (5.15) yields

$$\frac{dE}{dt} = - \left(\mathbf{C} \frac{d\mathbf{u}}{dt} \right)^t \frac{d\mathbf{v}}{dt} \quad (5.17)$$

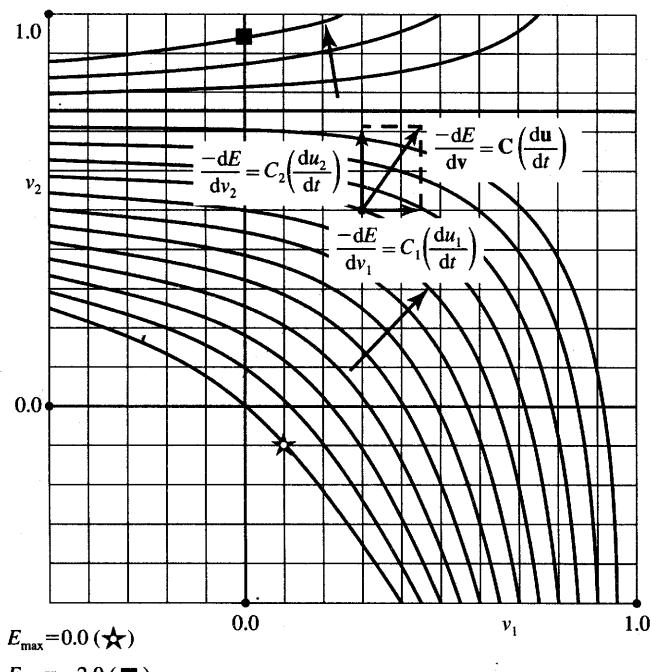
Using the inverse activation function $f^{-1}(v_i)$ of the neuron's VTC, the individual directions of changes of the output vector $\mathbf{v}(t)$, or velocities, can now be computed as follows:

$$C_i \frac{du_i}{dt} = C_i f^{-1}'(v_i) \frac{dv_i}{dt} \quad (5.18)$$

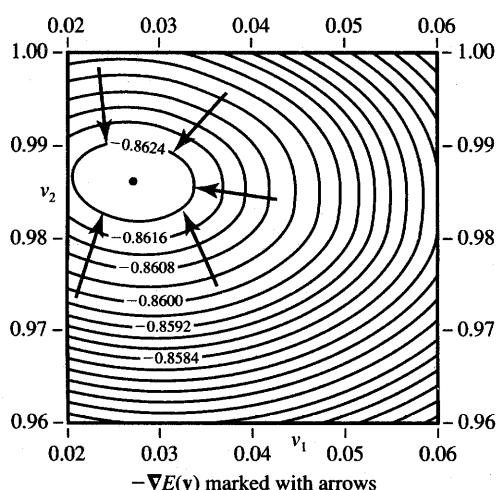
Since $f^{-1}'(v_i) > 0$, it can be noticed that du_i/dt and dv_i/dt have identical signs. But, as stated previously in (5.16), the values of du_i/dt are of the same sign as the negative gradient components $\partial E / \partial v_i$. It follows thus that the changes of E , in time, are in the general direction toward lower values of the energy function in v^n space. The changes in time, however, do not exactly follow the direction of the gradient itself, since the term $C_i f^{-1}'(v_i)$ provides here such scaling that

$$\frac{dv_i}{dt} = \frac{dE/dv_i}{C_i f^{-1}'(v_i)} \quad (5.19)$$

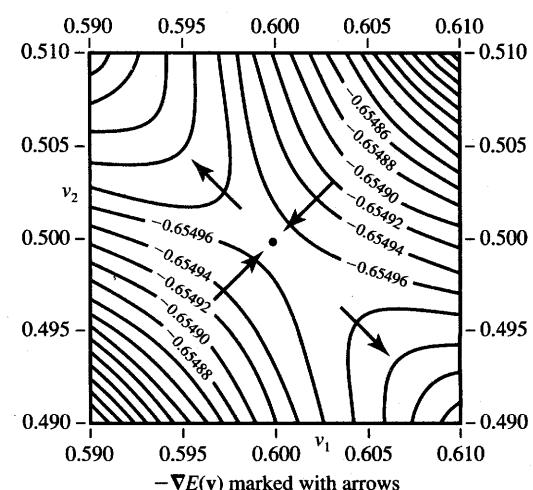
This energy minimization property of the system discussed here motivates the name of a gradient-type system. The property of energy minimization makes it possible to replace the evaluation of rather complex convergence *to the minimum of E versus time*, like it really occurs in an artificial neural system, with the



(a)



(b)



(c)

Figure 5.5 Illustration of energy gradient descent in v^2 space: (a) descent along a negative gradient, (b) minimum point, and (c) saddle point.

convergence to the minimum of $E(v)$ in the output space. It can also be noticed that such a solution for the minimum $E(v)$ results from solving the nonlinear, but purely algebraic, equations describing the nondynamic system, i.e., the system as shown in Figure 5.4(b) but with the capacitors removed.

It is interesting to note that the E function also has the meaning of the so-called cocontent function (Kennedy and Chua 1988). Remarkably, the stationary point of the system's dynamics may be examined by finding the zero value of the system's energy gradient vector $\nabla E(v)$. Since $dE/dt \leq 0$, the energy function decreases in time, unless $dE/dt = 0$. At the network's attractor, where the condition $dE/dt = 0$ holds, we also have $du_i/dt = 0$, for $i = 1, 2, \dots, n$, as can be seen from (5.17). The attractor of the network must be a point so that any oscillations or cycles are excluded, and the network is totally stable.

As the system comes to rest at the minimum of $E(v)$ after it has moved toward a minimum in the output space, it has also reached equilibrium in the neuron's input, or state, space u . This can be seen from expression (5.19). Because the system seeks out one of its energy minima, it is instructive to see the relationship between the weight matrix W and the minima of the system's energy function. Below, we discuss the conditions of existence of such minima.

As derived in previous paragraphs, finding an equilibrium point for the dynamic network with continuous activation function neurons corresponds to finding the solution v^* of the following algebraic equation

$$\nabla E(v) = \mathbf{0} \quad (5.20a)$$

Speaking more precisely, an equilibrium point v^* is called the *stationary point*. The stationary point can be a maximum, minimum, or saddle. The case of a minimum is illustrated on the contour map of the energy function of Figure 5.5(b). Figure 5.5(c) shows the stationary point which is the saddle. Although the gradient vector vanishes at the saddle point, the dynamical system does not stabilize there. On the figure shown, the system will tend to move toward the upper left or bottom right corner of the map.

Let us find the stationary points of the energy function. Using the gradient vector value computed from (5.14), Equation (5.20a) can be rewritten as

$$-Wv - i + Gu = 0 \quad (5.20b)$$

The additional relationship (5.12b) is mapping the state space into the output space, $u^n \rightarrow v^n$, and along with (5.20b) they can be used for finding v^* , which are the stationary points of the energy function.

Solving Equation (5.20) without consideration of the constraints on the solution as belonging to the v^n hypercube yields

$$Wv = -i \quad (5.21a)$$

and the solution v^* is obtained as

$$v^* = -W^{-1}i \quad (5.21b)$$

The additional assumption has been made in (5.21) of high-gain neurons, which eliminates the last term in (5.20b). It can be also noted in such cases that the Hessian matrix (see the Appendix) of the unconstrained system using high-gain neurons is

$$\nabla^2 E(\mathbf{v}) = \nabla [-\mathbf{W}\mathbf{v} - \mathbf{i}] \quad (5.22a)$$

which yields

$$\nabla^2 E(\mathbf{v}) = -\mathbf{W}. \quad (5.22b)$$

By having previously chosen $w_{ii} = 0$ for the discussed neural network model, the Hessian matrix of the energy function equal to $-\mathbf{W}$ has been made neither positive definite nor negative definite (see the Appendix). This is resulting from our previous assumption that the postulated system has no self-feedback. In fact, symmetric matrix \mathbf{W} with zero diagonal produces scalar-valued energy function of quadratic form $(-\frac{1}{2})\mathbf{v}'\mathbf{W}\mathbf{v} - \mathbf{i}'\mathbf{v}$, which has neither minima nor maxima since \mathbf{W} is neither positive nor negative definite. Thus, the energy function $E(\mathbf{v})$ possesses no unconstrained minima. This, in turn, means that in the case of limitation of the output space of the system to a $[-1, 1]$ hypercube, the constrained minima of $E(\mathbf{v})$ must be located somewhere at the hypercube's boundary, or in other words, on the edges or faces of the cube. If a solution \mathbf{v}^* of the constrained system within the cube $[-1, 1]$ exists, then it must be a saddle point. Clearly, this solution does not represent an attractor for the discussed dynamical system (Park 1989).

Let us try to examine the locations of the stationary points of a high-gain neural network. The energy function (5.14) without its third term, and evaluated on the faces and on the edges simplifies to the following expressions, respectively,

$$E = -\frac{1}{2}(w_{ij}v_i v_j + w_{ji}v_j v_i) - k_i v_i - k_j v_j \quad (5.23a)$$

$$E = -k_i v_i + c \quad (5.23b)$$

where k_i is constant. Indices i and j , for $i, j = 1, 2, \dots, n$, in Equation (5.23) determine the face (i, j) or the edge (i) , of the n -dimensional cube. Since the Hessian matrix for the two-variable function $E(v_i, v_j)$ as in (5.23a) has again a zero diagonal, no minima can exist on any of the faces of the cube. Further, since (5.23b) is a monotonic and linear function of a single variable v_i , the minima may not occur on the edges of the cube either. Thus, the minima, if they exist, must be confined to the vertices of the cube.

We have shown that the assumption of high-gain neurons resulting in neglecting the last term of energy in (5.14) enforces the constrained minima to be exactly in the vertices of the $[-1, 1]$ cube having a total of 2^n vertices. The high-gain network that approaches, in the limit case, the discrete-time system is expected to have its energy minima in the cube corners. The convergence

observed in actual gradient-type network hardware should result in one such cube corner solution.

The gradient-type network with finite neuron gain values also evolves in the continuous-time mode to one of the minima of $E(v)$. However, the minima are now within the cube and are attractors of the system. Such minima are usually desirable if they are as close to the vertices of the hypercube as possible. Understandably, they will always be within the $[-1, 1]$ cube, and for $\lambda \rightarrow \infty$ they will reach the cube corners.

Concluding this discussion of stationary points of energy function for single-layer feedback networks we notice that the solutions of Equations (5.20) and (5.12b) always represent either the constrained minimum or saddle of the energy function. There is, in general, more than one solution to (5.20) and (5.12b). The solution reached by an actual neural network is dependent on network parameters and on the initial condition within the network. Some of the solutions produced by neural networks of this class are useful, so they will be desirable. Other solutions will be less useful and may even be erroneous, although sometimes hard to avoid. This property applies both to discrete-time and continuous-time rules of convergence. We have considered a number of properties of single-layer feedback neural networks that will bring in a number of interesting results and applications. While these results and applications will be studied later, the example below will highlight some of the aspects just discussed.

EXAMPLE 5.2

In this example we design a simple continuous-time network using the concept of computational energy function and also evaluate stationary solutions of the network. The A/D converter circuit discussed below was first proposed in Tank and Hopfield (1986).

Let us assume that the analog input value x is to be converted by a two-neuron network as in Figure 5.1 or 5.4(a) to its binary representation. In this example it will be assumed that the activation function $f(u_i)$ is continuous unipolar and defined between 0 and 1. The A/D conversion error E_c can be considered here as an energy function. We can thus express it by the following formula

$$E_c = \frac{1}{2} \left(x - \sum_{i=0}^1 v_i 2^i \right)^2 \quad (5.24a)$$

where the $v_0 + 2v_1$ is the decimal value of the binary vector $[v_1 \ v_0]^t$ and it corresponds to the analog x value.

The reason for this choice of the energy function is that minimizing the energy will simultaneously minimize the conversion error. Our objective is

to find matrix \mathbf{W} and the input current vector \mathbf{i} for a network minimizing the error E_c . Evaluation of the error (5.24a) indicates, however, that it contains terms v_0^2 and v_1^2 , thus making the w_{ii} terms equal to 2^{2i} instead of equal to zero as postulated by the network definition. Therefore, a supplemental error term E_a , as in (5.24b), will be added. In addition to eliminating the diagonal entries of \mathbf{W} , this term will be minimized close to the corners of the $[0, 1]$ square on v_0, v_1 plane. A suitable choice for E_a is therefore

$$E_a = -\frac{1}{2} \sum_{i=0}^1 2^{2i} v_i(v_i - 1) \quad (5.24b)$$

Indeed, if we note that E_a is nonnegative we can see that it has the lowest value for $v_i = 0$ or $v_i = 1$. Thus, combining E_a and E_c , chosen as in (5.24), results in the following total energy function, which is minimized by the neural network A/D converter:

$$E = \frac{1}{2} \left(x - \sum_{i=0}^1 v_i 2^i \right)^2 - \frac{1}{2} \sum_{i=0}^1 2^{2i} v_i(v_i - 1) \quad (5.25a)$$

This expression for energy should be equated to the general formula for energy as in (5.14), which is equal in this high-gain neuron case:

$$E = -\frac{1}{2} \sum_{i=0}^1 \sum_{j=0, j \neq i}^1 w_{ij} v_i v_j - \sum_{i=0}^1 i_i v_i \quad (5.25b)$$

Note that the energy function (5.14) in the high-gain case retains only two terms, and the term with the integral vanishes because of the high-gain neurons used. Routine rearrangements of the right side of the energy expression (5.25a) yield

$$E = \frac{1}{2} x^2 + \frac{1}{2} \sum_{i=0}^1 \sum_{j=0, j \neq i}^1 2^{i+j} v_i v_j + \sum_{i=0}^1 (2^{2i-1} - 2^i x) v_i \quad (5.26)$$

When the analog input x is fixed, the term $(\frac{1}{2})x^2$ is an additive constant in (5.26), and it is irrelevant for the minimization of E given by (5.26) on the v_0, v_1 plane; therefore, it will be omitted. Comparing the like coefficients of v_0 and v_1 on the right side of energy functions (5.25b) and (5.26) results in the following network parameters:

$$\begin{aligned} w_{01} &= w_{10} = -2 \\ i_0 &= x - \frac{1}{2} \\ i_1 &= 2x - 2 \end{aligned} \quad (5.27)$$

The resulting two-bit A/D converter network is the special case of the general network from Figure 5.4(a) and is designed according to specifications (5.27). The network diagram is shown in Figure 5.6(a). The two

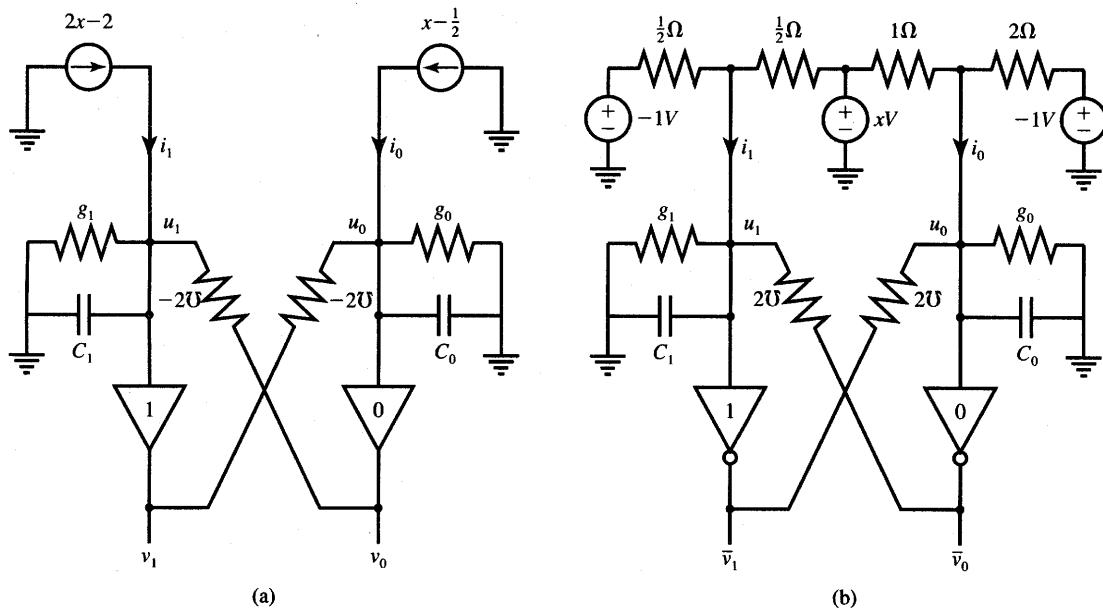


Figure 5.6 Two-bit A/D converter from Example 5.2: (a) using noninverting neurons with activation function $f(\text{net})$ and (b) using inverting neurons with activation function $-f(\text{net})$ and voltage sources producing i_0 and i_1 .

current sources produce the biasing currents i_0 and i_1 and the weights are of -0.5Ω values. Let us note that the KCL equations (5.11) for nodes $i = 0$ and $i = 1$ can be obtained by inspection of Figure 5.6(a) as follows:

$$\begin{aligned} C_0 \frac{du_0}{dt} &= x - \frac{1}{2} + (-2)v_1 - u_0(-2 + g_0) \\ C_1 \frac{du_1}{dt} &= 2x - 2 + (-2)v_0 - u_1(-2 + g_1) \end{aligned} \quad (5.28)$$

As we have already mentioned, the last term of the expression for energy (5.14) vanishes for high-gain neurons since the products $u_i G_i$ become negligibly small. This assumption remains valid for large-gain λ values yielding small activation values u_i , unless the solution is very far from the square's vertices. Under this assumption, the converter diagram from Figure 5.6(a) can be redrawn as shown in Figure 5.6(b). An explicit assumption made for this figure is that $u_0 \approx u_1 \approx 0$, so that currents i_0 and i_1 as in (5.27) can be produced by a voltage source having voltage x combined with the reference voltage source, which delivers the voltage of value -1 V.

An additional change introduced in Figure 5.6(b) compared to Figure 5.6(a) for the sake of practicality has been the replacement of neurons with

activation function $f(\text{net})$ by neurons with its negative activation function $-f(\text{net})$. Neurons with inverting properties can be called *inverting neurons*. Their outputs are now denoted \bar{v}_0 and \bar{v}_1 . As a result of this modification, the previously negative conductances w_{01} and w_{10} of value -2 become positive, as desired. It is left to the reader to verify that Equations (5.28) now can be rewritten as follows

$$\begin{aligned} C_0 \frac{du_0}{dt} &= x - \frac{1}{2} + 2\bar{v}_1 - u_0(2 + g_0) \\ C_1 \frac{du_1}{dt} &= 2x - 2 + 2\bar{v}_0 - u_1(2 + g_1) \end{aligned} \quad (5.29a)$$

Equation (5.29a) can be arranged in matrix form as in (5.12a):

$$\begin{bmatrix} C_0 & 0 \\ 0 & C_1 \end{bmatrix} \begin{bmatrix} \frac{du_0}{dt} \\ \frac{du_1}{dt} \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} \bar{v}_0 \\ \bar{v}_1 \end{bmatrix} - \begin{bmatrix} 2 + g_0 & 0 \\ 0 & 2 + g_1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \end{bmatrix} + \begin{bmatrix} x - \frac{1}{2} \\ 2x - 2 \end{bmatrix} \quad (5.29b)$$

The condition $w_{ij} \geq 0$ ensures the realizability of the network using resistive interconnections of values R_{ij} . The conductance matrix \mathbf{W} now has positive valued elements $w_{ij} = 1/R_{ij}$.

5.4

TRANSIENT RESPONSE OF CONTINUOUS-TIME NETWORKS

Equation (5.11) or (5.12) describes the discussed nonlinear dynamical system in the form of its state equations. Because analytical methods for solving such equations in closed form do not exist, suitable numerical approaches need to be devised for obtaining transients $v(t)$ and the stationary solution thereof $\lim_{t \rightarrow \infty} v(t)$.

A number of numerical integration methods allow for computing the dynamic solution for $v(t)$ starting at initial condition $v(t_0)$. Also, steady-state solutions for the stationary points may be obtained by a number of methods for solving static nonlinear equations. In this section we will address only those among the many numerical approaches that are particularly useful in providing additional insight into network dynamics pattern.

Let us see first how the *vector field method* can be used to illustrate the real-time phenomena in networks with finite gain neurons. The method allows

the transients of the nonlinear dynamical system to be captured. The vector field obtained by using this method involves all possible network trajectories in E^n space.

To obtain the vector field expression of the network of interest, Eq. (5.11) needs to be normalized using the C_i value as a divisor. Thus, the description of the system is given by n equations below:

$$\frac{du_i}{dt} = \frac{1}{C_i} \left(i_i - G_i u_i + \sum_{j=1}^n w_{ij} v_j \right), \quad \text{for } i = 1, 2, \dots, n \quad (5.30)$$

Our vector field analysis objective is to transform the description of the system, originally as in (5.30), to the following form:

$$\frac{dv_i}{dt} = \psi_i[v(t)], \quad \text{for } i = 1, 2, \dots, n \quad (5.31)$$

where ψ_i are functions that should be found. These functions are vector field components. Subsequently, after discretizing differential equations [Equations (5.31)], values $\psi_1(v), \psi_2(v), \dots, \psi_n(v)$ are numerically computed at each point of the space v . As a result, we obtain the vector field diagram, which clearly indicates complete trajectories of the system's dynamics and shows how it evolves in time during transients. While the solution is somewhat more qualitative and the produced vectors contain no explicit time component, the vector field provides complete information about basins of attractions for any given initial point, as well as about the stationary solutions.

To be able to use the vector field method, we note that the specific form of the activation function should be assumed in order to produce Equation (5.31). For the discussion below it has been chosen as

$$v = f(u) = \frac{1}{1 + \exp(-\lambda u)} \quad (5.32a)$$

Simple differentiation of (5.32a) leads to

$$\frac{dv}{du} = \lambda(v - v^2) \quad (5.32b)$$

The increments of v_i as functions of increments of u_i can now be expressed as

$$dv_i = \lambda(v_i - v_i^2) du_i \quad (5.32c)$$

Inserting the result of Equation (5.32c) into (5.30) leads to

$$\frac{dv_i}{dt} = \frac{\lambda(v_i - v_i^2)}{C_i} \left(i_i - G_i f^{-1}(v_i) + \sum_{j=1}^n w_{ij} v_j \right), \quad \text{for } i = 1, 2, \dots, n \quad (5.33)$$

Equation (5.33) is a specific form of the general vector field expression (5.31) for which we have been looking. Vector field components ψ_i are equal to the right side of (5.33). To trace the movement of the system within the output space, the

components of ψ must be computed for the entire cube $v \in [0, 1]$ as follows:

$$\psi_i(v) = \frac{\lambda(v_i - v_i^2)}{C_i} \left(i_i - G_i f^{-1}(v_i) + \sum_{j=1}^n w_{ij} v_j \right), \text{ for } i = 1, 2, \dots, n \quad (5.34)$$

It is interesting to observe that the trajectories depend on the selected λ values, if they are finite, and also on the values of capacitances C_i . In addition to these factors, the steady state, or equilibrium solution may be reached along one of the many trajectories dependent on the initial conditions. The example below will provide more insight into the transient performance of a network with finite λ_i values.

EXAMPLE 5.3

In this example we will evaluate stationary points and discuss the time-domain behavior of the continuous-time example two-bit A/D neural converter designed in Example 5.2. In particular, we will trace the behavior of the converter output versus time and inspect the energy minimization property during the evolution of the system.

Substituting the weights and current values as calculated in (5.27) we can express the truncated energy function given by (5.25b) for very high gains λ_i as follows:

$$E = -\frac{1}{2} \sum_{i=0}^1 \sum_{\substack{j=0 \\ i \neq j}}^1 (-2)v_i v_j - \left(x - \frac{1}{2} \right) v_0 - (2x - 2)v_1 \quad (5.35a)$$

Expanding the double sum and rearranging terms leads to the value for E :

$$E = 2v_0 v_1 + \frac{v_0}{2} + 2v_1 - x(v_0 + 2v_1) \quad (5.35b)$$

Figure 1.10 displays two cases of the truncated energy function of Equation (5.35) for analog input $x = 0$ and $x = 1$. As shown in the figure, the only stationary points are appropriate minima, $v_0 = v_1 = 0$ for $x = 0$, and $v_0 = 1$ and $v_1 = 0$ for $x = 1$. Figure 5.7(a) shows the energy function for $x = 0.5$ and it can be seen that, in contrast to previous cases, the minimum of the energy function exists for $0 < v_0 < 1, v_1 = 0$. It is thus spread over the entire range of output variable v_0 . Indeed, an absence of a definite response by the network is caused by the fact that no digital conversion of the analog input $x = 0.5$ exists that would minimize the conversion error (5.24a).

Another interesting case of energy function behavior occurs for $1 < x < 2$. The energy functions (5.35) are illustrated for $x = 1.25$ and $x = 1.5$ on Figures 5.7(b) and (c), respectively. Inspection of both functions reveals that they both have (1) a desirable minimum at $v_0 = 1$ and $v_1 = 0$, (2) an undesirable minimum at $v_0 = 0$ and $v_1 = 1$, and (3) a saddle point somewhere inside the unity square.

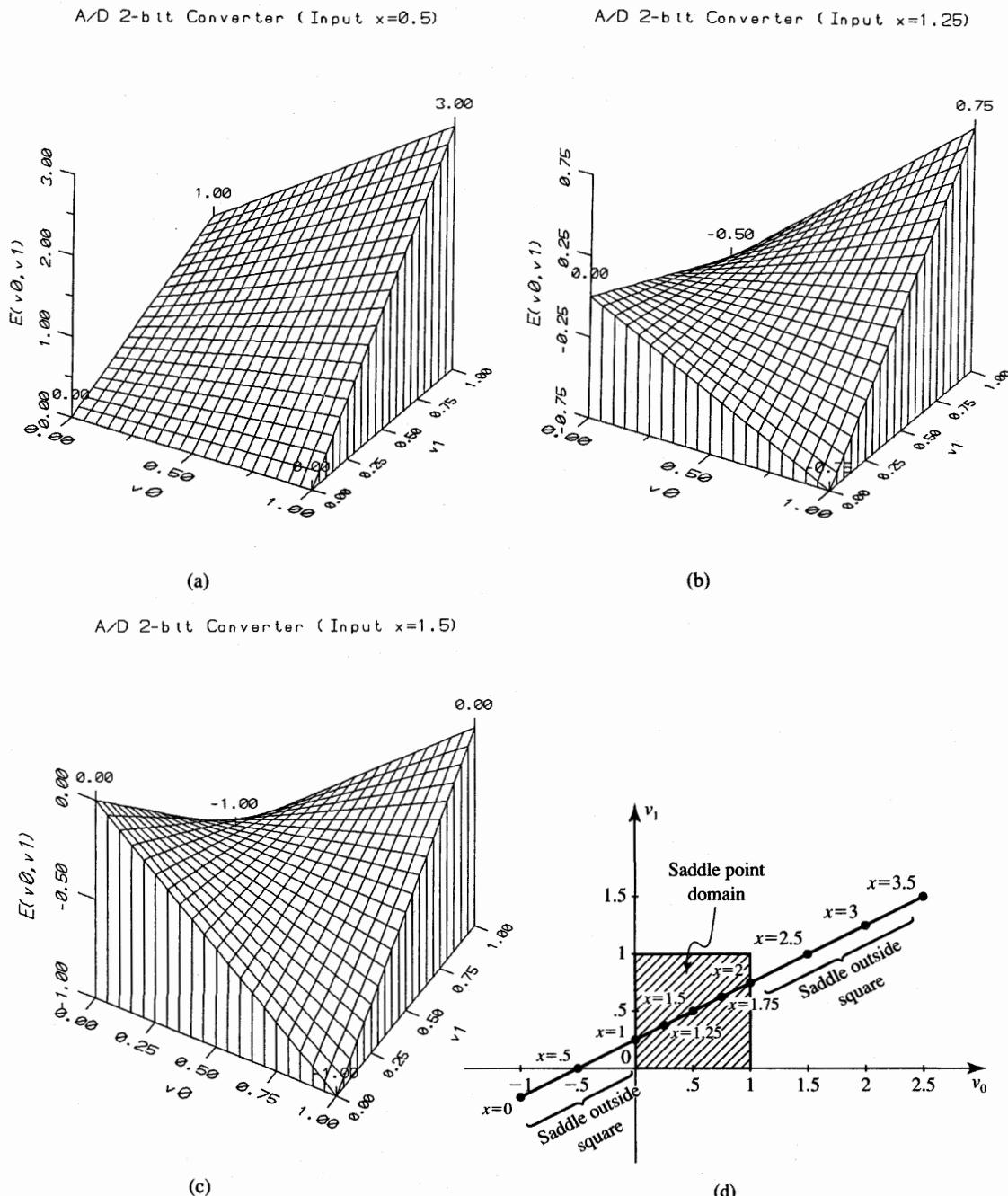


Figure 5.7 Stationary points of the truncated energy function for the two-bit A/D converter of Example 5.3: (a) energy function ($x = 0.5$), (b) energy function ($x = 1.25$), (c) energy function ($x = 1.5$), and (d) saddle point locus ($0 < x < 2.5$).

The discussion below focuses on more detailed analysis of the energy function so that the equilibrium points of the system can be investigated based on the evaluation of the energy function. This should provide us with much insight into the behavior of the system, but even more importantly, will perhaps eliminate the need for solution of differential equations (5.11) and replace it with inspection of the energy function minima.

Let us evaluate the truncated energy function (5.35b). The energy gradient $\nabla E(\mathbf{v})$ can be computed from (5.35b) and equated to zero for the fixed value of x to determine possible high-gain minima at \mathbf{v}^* :

$$\nabla E(\mathbf{v}) = \begin{bmatrix} 2v_1 + \frac{1}{2} - x \\ 2v_0 + 2 - 2x \end{bmatrix} \quad (5.36a)$$

To find the type of stationary points of the network, it is helpful to see that the Hessian matrix \mathbf{H} of the energy function (5.35b) becomes

$$\mathbf{H} = \nabla^2 E(\mathbf{v}) = \begin{bmatrix} 0 & 2 \\ 2 & 0 \end{bmatrix} \quad (5.36b)$$

Result (5.36b) indicates that the Hessian matrix is not positive definite. The energy function has therefore no unconstrained minima. Stationary points \mathbf{v}^* may still be maxima and/or saddles. We can also see that Hessian matrix determinants are

$$\det H_{11} = 0$$

$$\det H_{22} = \begin{bmatrix} 0 & 2 \\ 2 & 0 \end{bmatrix} = -4$$

Thus, the Hessian matrix is not negative definite either. Thus, whatever solutions \mathbf{v}^* are identified within the cube $(0, 1)$ such that

$$\nabla E(\mathbf{v}^*) = \mathbf{0} \quad (5.36c)$$

will be the saddle points of the $E(\mathbf{v})$ function. This coincides with our conclusions related to Equation (5.22) and (5.23) that the minima of the truncated energy function (5.35b) are at cube vertices, and the saddle points may be located anywhere inside it.

The solution for a saddle point of the energy function (5.35b) results directly from (5.36a) and (5.36c) as

$$v_1 = \frac{1}{2}v_0 + \frac{1}{4} \quad (5.37)$$

The saddle point locus is shown in Figure 5.7(d). The saddle point exists within the unity square $0 < v_i < 1$, for $i = 0, 1$, and only for $1 < x < 2$. The cases $0 < x < 1$ and $2 < x < 3.5$ are characterized by a monotonic energy function and the absence of a saddle point within the square of interest. However, there exists a saddle point outside the square and thus outside the domain of definition of vector \mathbf{v} .

Let us see how the study of the energy function and the properties of the converter's vector field can be merged to provide insight into the network behavior. Examples of selected vector field diagrams produced from Equation (5.34) for the discussed converter are shown in Figure 5.8. The figure illustrates four different cases of convergence to the equilibrium point for selected gain values of $\lambda = 10$ and 100 . The remaining convergence parameters of interest are the input conductances g_i of the neurons and the capacitance ratio C_1/C_0 . The convergence has been evaluated for inputs $x = 0$ [Figures 5.8(a) and (b)] and $x = 1.8$ [Figures 5.8(c) and (d)]. The energy function contours, or equipotential lines, have also been marked as background lines. The energy contour maps displayed are for the case of very high-gain neurons and a truncated energy expression as in (5.35b) (Zurada, Kang, and Aronhime 1990).

Unless the terms $u_0 G_0$ and $u_1 G_1$ are excessive, the system evolves in time toward lower values of the truncated energy function as shown. Otherwise, the system minimizes the complete energy function as in (5.14), involving therefore the third term. In any of the cases, the evolution of the system in time is not exactly in the energy function negative gradient direction. It can be seen that for a high value of gain λ and small input conductances g_i of neurons (note that these are the conditions which make a network free from parasitic phenomena) the equilibrium solution approaches one of the corners shown in Figures 5.8(a) and (c). As shown in Figure 5.8(b), for large input conductances $g_0 = g_1 = 10 \Omega^{-1}$, the convergence does not end at any of the corners. The truncated energy function contours do not exactly describe now the evolution of the network in the case of large $g_i u_i$ terms. The stationary point M in Figure 5.8(b) is not in the truncated energy function minimum; however, it is a minimum of the complete energy function involving its last term. It is also the endpoint of all relevant trajectories indicating the dynamics of the network. Point M is the minimum of the complete energy function (5.14), which for this case becomes

$$E_{\text{tot}} = E + E_3 \quad (5.38a)$$

Energy E is given by (5.35b) and the term E_3 can be obtained from (5.14) as

$$E_3 = \sum_{i=0}^1 G_i \int_{1/2}^{v_i} f(\lambda, z)^{-1} dz \quad (5.38b)$$

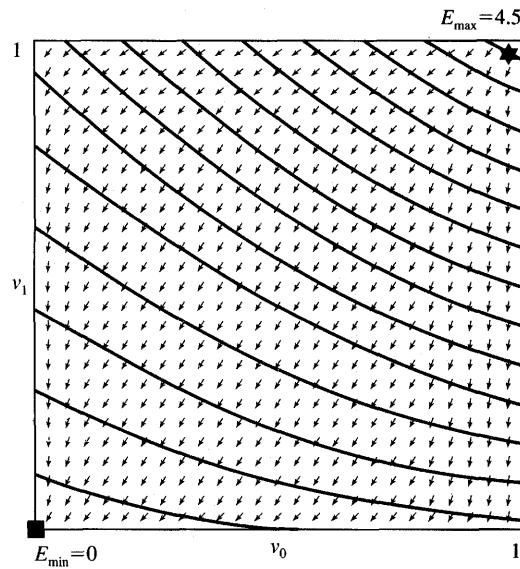
where the inverse of the activation function (5.32a) can be expressed

$$f(\lambda, z)^{-1} = \frac{1}{\lambda} \ln \left(\frac{z}{1-z} \right)$$

This yields the explicit value of the third term of energy as

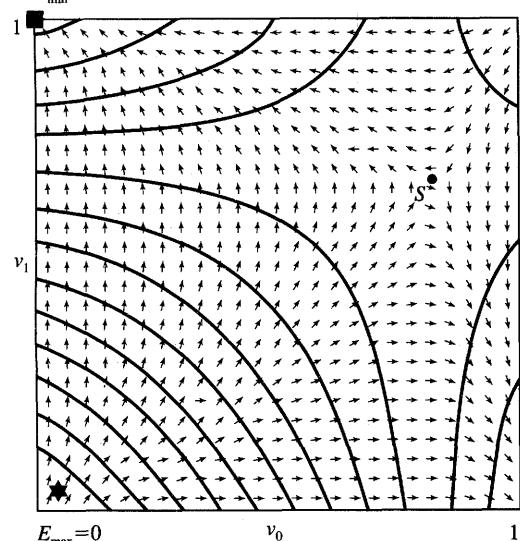
$$E_3 = \sum_{i=0}^1 \frac{G_i}{\lambda_i} [v_i \ln v_i + (1-v_i) \ln (1-v_i) + \ln 2] \quad (5.38c)$$

It should be noted that the total energy function E_{tot} now has minima within the cube. Indeed, when neurons are of finite gains λ , then the minimum of the energy function must exist within the cube, since no walls, edges, or corners



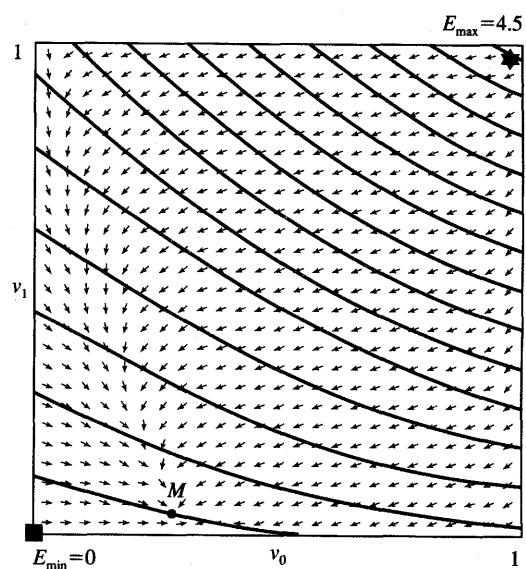
$$x=0, \lambda=100 \\ \frac{C_1}{C_0}=1, g=1$$

(a)



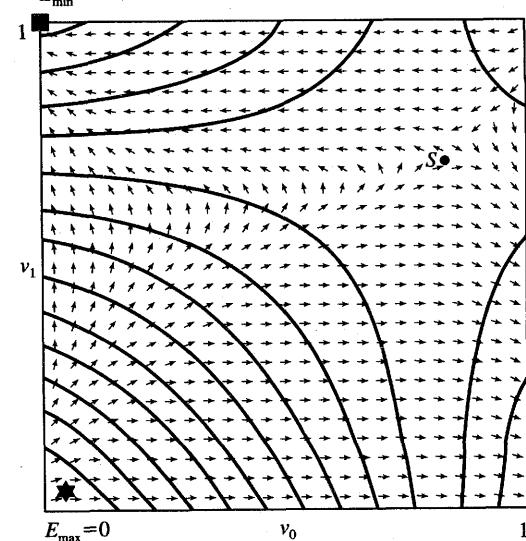
$$x=1.8, \lambda=100 \\ \frac{C_1}{C_0}=1, g=0.1$$

(c)



$$x=0, \lambda=10 \\ \frac{C_1}{C_0}=10, g=10$$

(b)



$$x=1.8, \lambda=10 \\ \frac{C_1}{C_0}=10, g=1$$

(d)

Figure 5.8 Selected vector field method solutions for dynamic behavior of two-bit A/D converter: (a) $x = 0, \lambda = 100, C_1/C_0 = 1, g = 1$, (b) $x = 0, \lambda = 10, C_1/C_0 = 10, g = 10$, (c) $x = 1.8, \lambda = 100, C_1/C_0 = 1, g = 0.1$, and (d) $x = 1.8, \lambda = 10, C_1/C_0 = 10, g = 1$.

are the domain of vector \mathbf{v} . This is in contrast to the case of the truncated energy function for which minima have been confined to vertices and edges of the cube. We can see that the total energy (5.38c) has no truncation error and more realistically characterizes the properties of actual networks for which λ is of finite value. The reader may refer to Problem P5.19 for more illustrations related to the total energy function.

Let us see how we can utilize the vector field trajectories to visualize the phenomena occurring in the network. By comparing the vector fields of Figures 5.8(a) and (b), we can see that the trajectories become horizontally skewed for large C_1/C_0 . This condition promotes movement of the system with slowly varying v_1 and faster varying v_0 . This asymmetry of motion is due to the fact that C_1 holds the majority of charge and the voltage u_1 has to change more slowly than u_0 in this otherwise symmetric network.

In addition, the vector field provides valuable insight for the case of an energy function saddle, as has been shown in Figures 5.8(c) and (d). The analog input which needs to be converted is of value $x = 1.8$. It can be computed from (5.36a) that the saddle point S denoted as \mathbf{v}^* is at $(0.8, 0.65)$. It can also be seen that the trajectories indicate that either of the solutions $v_0 = 0$ and $v_1 = 1$ or $v_0 = 1$ and $v_1 = 0$ can be reached by the network depending on the initial condition selected. Comparison of convergence with zero initial conditions at $\mathbf{v} = [0.5 \ 0.5]^t$ indicates that the right solution for converting the analog input $x = 1.8$ is reached in the case of equal capacitances C_1 and C_0 in Figure 5.8(a). The case shown in Figure 5.8(d) depicts the incorrect solution due to the horizontally biased movement of the system toward the right.

In conclusion, the vector field method provides detailed insight into the transients and stability conditions of the actual network. Although the method can be illustrated by trajectories and provides insight only for cases of networks with up to three neurons, it can be applied for any value of n . In addition, one of the many numerical integration formula can be used for numerical analysis of dynamic performance of continuous-time networks (DiZitti et al. 1989). Selected exercises at the end of the chapter are devoted to this approach.

5.5

RELAXATION MODELING IN SINGLE-LAYER FEEDBACK NETWORKS

The discussion in the preceding sections of this chapter has shown that the gradient-type single-layer network converges, in time, to one of the minima of $E(\mathbf{v})$ located within the hypercube $[-1, 1]$. This assumption is valid for bipolar continuous neurons with outputs defining the components of n -dimensional vector \mathbf{v} . Whether simulated, or reached in actual gradient-type hardware network, the

solution may or may not be satisfactory. The solution may not even be the desired one due to the existence of a much better solution.

Let us devise an efficient numerical approach for finding the solution of differential equations (5.12). As stated before, the solution of differential equations (5.12) can be replaced by solving the purely algebraic equation (5.20b) with the additional nonlinear mapping condition (5.12b). Let us notice that (5.20b) is equivalent to n algebraic equations written as

$$\sum_{j=1}^n w_{ij}v_j + i_i = G_i u_i, \quad \text{for } i = 1, 2, \dots, n \quad (5.39a)$$

Using (5.12b), the above expression can be rewritten in the form involving the nonlinear mapping $v = f(u)$ as follows:

$$v_i = f \left[\frac{1}{G_i} \left(\sum_{j=1}^n w_{ij}v_j + i_i \right) \right], \quad \text{for } i = 1, 2, \dots, n \quad (5.39b)$$

The numerical solution of (5.39b) for v_i can be modeled by the fixed-point iteration method as follows:

$$v_i^{k+1} = f \left[\frac{1}{G_i} \left(\sum_{j=1}^n w_{ij}v_j^k + i_i \right) \right], \quad \text{for } i = 1, 2, \dots, n \quad (5.40a)$$

where superscript k denotes the index of numerical recursion, $k = 0, 1, \dots$. Equation (5.40a) can be briefly rewritten in the vector form as

$$\mathbf{v}^{k+1} = \mathbf{f} \left[\mathbf{G}^{-1} (\mathbf{Wv}^k + \mathbf{i}) \right] \quad (5.40b)$$

Equation (5.40b) represents the computational model for recursive numerical calculations of vector \mathbf{v}^{k+1} . First, an initial output vector \mathbf{v}^0 is submitted to the network to initialize it, then the response \mathbf{v}^1 is computed and is used to replace the value \mathbf{v}^0 on the right side of (5.40b). The algorithm of Equations (5.40) is referred to in literature as the *relaxation*, or method of successive approximation, *algorithm*. As we can see, the algorithm is static, and it represents the solution of a system *without* its dynamic components. Indeed, it is remarkable that the solution (5.40b) does not involve network capacitances. Although the relaxation algorithm is computationally simple, it does not provide any insight into the network dynamics. In addition, the algorithm shown below is numerically stable only under certain conditions.

The recursive computation of \mathbf{v}^{k+1} using fixed-point iterations (5.40) is convergent only under certain conditions, which are discussed in more detail and derived in Zurada and Shen (1990). A sufficient condition for convergence of the relaxation algorithm is

$$\lambda < \min_{i=1,2,\dots,n} \left\{ \frac{4}{n} \left| \frac{G_i}{w_{ij}} \right| \right\}, \quad \text{for } j = 1, 2, \dots, n, \quad j \neq i \quad (5.40c)$$

It can be seen from (5.40c) that to assure unconditional convergence of the

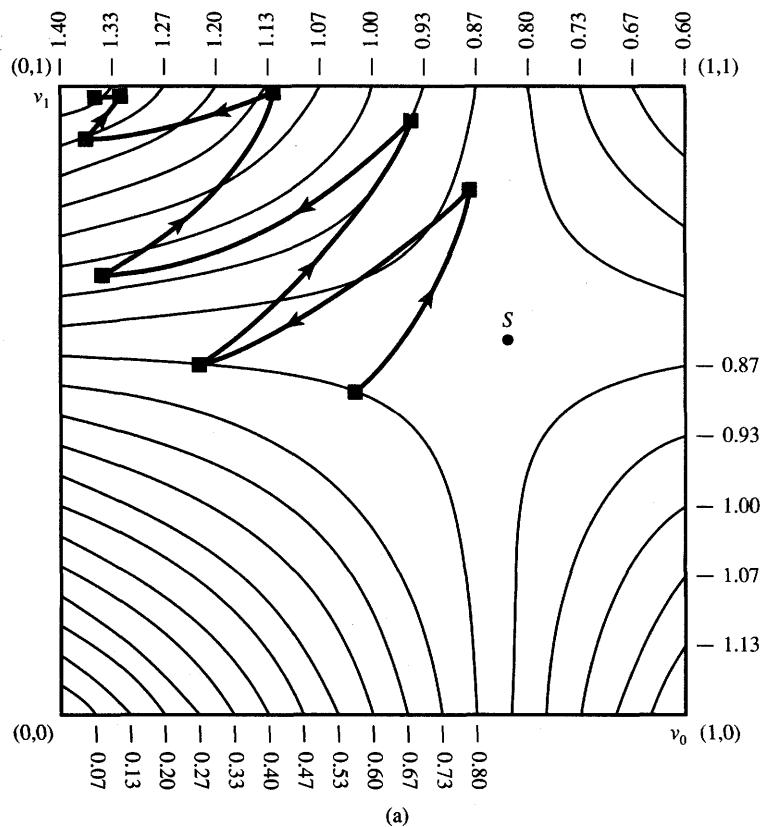


Figure 5.9a Relaxation algorithm using fixed-point iteration (5.40) for two-bit A/D converter, $x = 1.7$: (a) $\lambda = 3.5$, stable numerical solution.

algorithm, λ must be kept below an upper bound value on the right side of the inequality. An illustration of the relaxation algorithm for the two-bit A/D converter is shown in Figure 5.9 for the input $x = 1.7$. Figure 5.9(a) shows stable recursive calculations using (5.40) for $\lambda = 3.5$. The recursive calculations become oscillatory, however, for the λ value of 5, as shown in Figure 5.9(b). Note that both cases from Figure 5.9 originate at zero initial conditions of $v^0 = [0.5 \ 0.5]^T$. This corresponds to the network without an initial energy and starting at a neutral initial point. Specifically, the network capacitances hold no initial charge and provide no initial bias.

In summary, the vector $v(t)$ converges, in time, to one of the minima of $E(v)$ constrained to within the $[0, 1]$ cube, and transients are dependent on the initial condition $v(0) = f[u(0)]$. The system settles there according to its dynamics, and the capacitors and other components determine the actual rate of convergence

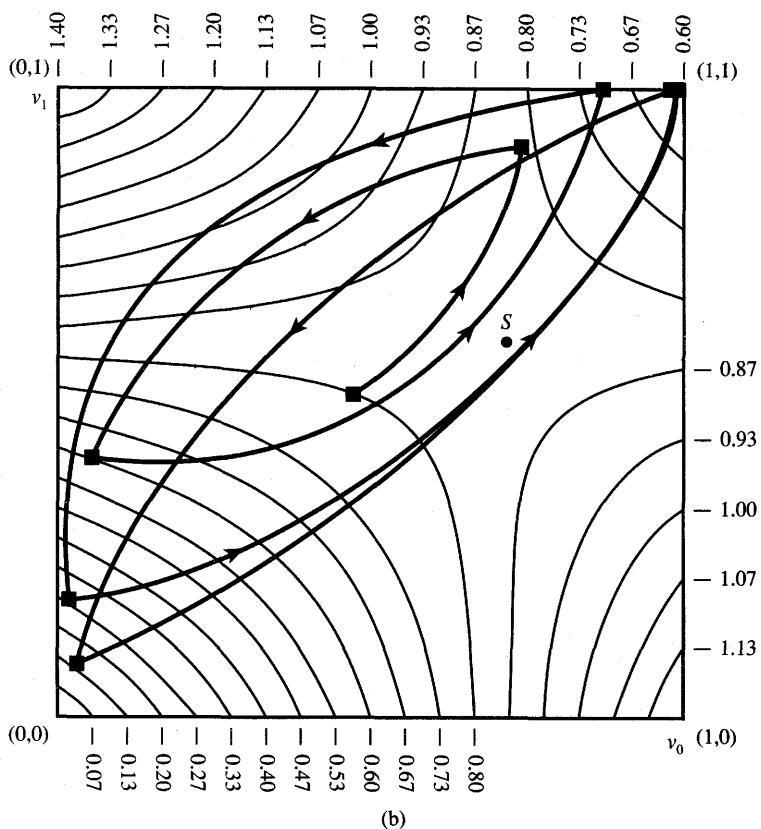


Figure 5.9b Relaxation algorithm using fixed-point iteration (5.40) for two-bit A/D converter, $x = 1.7$ (continued): (b) $\lambda = 5$, unstable numerical solution.

toward the minimum of $E(\mathbf{v})$. The computational relaxation model (5.38), although entirely based on the resistive portion of the network, captures one of the time-domain, or transient, solutions at which the gradient vector vanishes.

It should also be noted that the relaxation algorithm, or any other static algorithm that solves the equation (5.20b) based on zeroing of the gradient vector, should be used for finding stationary points rather than for finding minima only. Indeed, since $\nabla E(\mathbf{v}) = \mathbf{0}$ both at saddle points and at minima, the numerical solutions for cases as shown in Figure 5.9 also produce the saddle point S .

Similar results can be obtained from analysis of single-layer feedback networks using the Newton-Raphson algorithm to solve (5.20b). The Newton-Raphson algorithm is unconditionally stable; however, it is very sensitive to the initial conditions. More insight into application of this algorithm for neural network numerical solutions is available to the reader in Problem P5.18.

5.6

EXAMPLE SOLUTIONS OF OPTIMIZATION PROBLEMS

In this section, two comprehensive examples of continuous-time single-layer feedback networks will be covered. The discussion will emphasize understanding and practical integration of most of the concepts introduced earlier in the chapter. Both examples present the solution of optimization problems modeled with fully coupled single-layer network. The first example covers the design of an electronic network that performs the conversion of a multidimensional analog signal into a four-dimensional binary unipolar vector using the least squared error criterion. The presentation of the first example involves all stages of the design, and is followed by the detailed performance study of the network. The second example approaches a classical and more general optimization problem solution known in technical literature as the *traveling salesman problem*. The presentation of this second example focuses on formulating the most suitable objective function that would need to be substituted for an energy function.

Summing Network With Digital Outputs

The network is based on the concept presented earlier in Example 4.2. The adder/converter discussed in this section and the simple two-bit A/D converter in Example 4.2 belong to the same class of optimization networks. They both attempt to solve specific optimization problems of error minimization between analog input values and their digital output representation using Boolean vectors. Note that the problem essentially reduces to finding a mapping of various input signals into an n -dimensional binary vector such that the mapping objective function, defined as quadratic error, is minimized. To design the network, appropriate responses need to be encoded as its equilibrium states.

Assume that the circuit shown in Figure 5.4 needs to compute the sum of N analog voltages x_k with corresponding weighting coefficients a_k . A digital representation of the output sum using n bits is required, thus $v_i = 0, 1$, for $i = 0, 1, \dots, n - 1$. Let us momentarily assume that the accurate analog sum of the signals is x , thus:

$$x = \sum_{k=0}^{N-1} a_k x_k \quad (5.41)$$

This value has to be approximated by a binary n -component vector. The computational energy that needs to be minimized for this circuit behaving as an n -bit A/D converter can be expressed in terms of the squared error

$$E_c = \frac{1}{2} \left(x - \sum_{i=0}^{n-1} v_i 2^i \right)^2 \quad (5.42a)$$

Expansion of the sum indicates that (5.42a) contains square terms $(v_i 2^i)^2$, thus making the w_{ii} terms equal to 2^{2i} , instead of equal to zero as required. Similarly, as in Example 5.2, a supplemental energy term E_a (5.42b) should be added that, in addition to eliminating diagonal entries of the weight matrix \mathbf{W} , will have minima at v_i equal to 0 or 1.

$$E_a = -\frac{1}{2} \sum_{i=0}^{n-1} 2^{2i} v_i (v_i - 1) \quad (5.42b)$$

Let us sum E_c and E_a and use the truncated energy function (5.14) which is allowed in case of high-gain neurons. We can now require equality between the specific energy function value on the left side of Equation (5.43) and its general form shown on the right side of (5.43):

$$\frac{1}{2} \left(x - \sum_{i=0}^{n-1} v_i 2^i \right)^2 - \frac{1}{2} \sum_{i=0}^{n-1} 2^{2i} v_i (v_i - 1) = -\frac{1}{2} \mathbf{v}' \mathbf{W} \mathbf{v} - \mathbf{i}' \mathbf{v} \quad (5.43)$$

Comparing coefficients on both sides of (5.43) yields the conductance matrix and bias current vector entries of the designed neural network as follows

$$w_{ij} = -2^{i+j} \quad (5.44a)$$

$$i_i = -2^{2i-1} + 2^i x \quad (5.44b)$$

In the case of a four-bit A/D converter, the results of Equations (5.44) can be written in matrix form as follows:

$$\mathbf{W} = - \begin{bmatrix} 0 & 2 & 4 & 8 \\ 2 & 0 & 8 & 16 \\ 4 & 8 & 0 & 32 \\ 8 & 16 & 32 & 0 \end{bmatrix}, \quad \mathbf{i} = - \begin{bmatrix} \frac{1}{2} - x \\ 2 - 2x \\ 8 - 4x \\ 32 - 8x \end{bmatrix} \quad (5.44c)$$

Due to the simplification introduced in (5.41), these are only preliminary answers. To complete the design and to obtain the full diagram of the summing and converting circuit, the value of x as in (5.41) should be plugged into the left side of formula (5.43). This yields the modified energy function in the following form:

$$E = \frac{1}{2} \left(\sum_{k=0}^{N-1} a_k x_k - \sum_{i=0}^{n-1} v_i 2^i \right)^2 - \frac{1}{2} \sum_{i=0}^{n-1} 2^{2i} v_i (v_i - 1) \quad (5.45a)$$

Rearranging the above expression results in the energy function for the summing circuit being equal to

$$\begin{aligned} E = & -\frac{1}{2} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \underset{i \neq j}{-2^{i+j} v_i v_j} - \sum_{i=0}^{n-1} \left(-2^{2i-1} + 2^i \sum_{k=0}^{N-1} a_k x_k \right) v_i \\ & + \frac{1}{2} \left(\sum_{k=0}^{N-1} a_k x_k \right)^2 \end{aligned} \quad (5.45b)$$

Comparing Equation (5.45b) with the right side of Equation (5.43) allows the calculation of the conductance matrix elements w_{ij} and the neuron input currents i_i of network S as

$$w_{ij} = -2^{i+j} \quad (5.46a)$$

$$i_i = -2^{2i-1} + 2^i \sum_{k=0}^{N-1} a_k x_k \quad (5.46b)$$

Figure 5.10 shows the resulting network with appropriate conductance values labeled. The network consists of two parts. The bottom part replicates the fully coupled single-layer network of Figure 5.4a. This is a “neural” part of the network with synaptic weights as in (5.46a). Note that the originally negative signs of the conductance values have been absorbed by the feedback signal derived from inverted neuron outputs. The top part of the network denoted S produces appropriate bias currents by computing the expression (5.41) as indicated by the sum on the right side of (5.46b). Conductances connect voltage sources x_j and a voltage source -1 V with inputs of neurons to produce current according to formula (5.46b).

The approach presented in the preceding paragraphs can be extended to the case of binary addition. For binary coded values of signals x and y , the corresponding energy function similar to that obtained in (5.45a) becomes

$$E = \frac{1}{2} \left(\sum_{k=0}^{n-1} 2^k x_k + \sum_{k=0}^{n-1} 2^k y_k - \sum_{i=0}^n v_i 2^i \right)^2 - \frac{1}{2} \sum_{i=0}^n 2^{2i} v_i (v_i - 1) \quad (5.47)$$

Rearrangement of (5.47) leads to an expression for the values w_{ij} identical to (5.46a). The new conductance values of network S performing the binary addition are now the only difference between the originally considered converting circuit and the joint summing and converting circuit. The input currents to the neurons are specified by the following expression:

$$i_i = -2^{2i-1} + 2^i \left(\sum_{k=0}^{n-1} 2^k x_k + \sum_{k=0}^{n-1} 2^k y_k \right), \quad \text{for } i = 0, 1, \dots, n \quad (5.48)$$

The binary adder network S generating the bias current vector \mathbf{i} consists now of the total of $(2N + 1)(n + 1)$ conductances. The “neural” part of the adder remains unchanged and identical as in previous cases of an A/D converter and an analog signal adder.

To illustrate the theoretical considerations involved in this design, the performance of an electronic network model has been simulated. The convergence of transients for the hardware model has been evaluated for nine different initial conditions. The neurons have been designed using NMOS transistors (Zurada and Kang 1989). For the simple two-dimensional case, contour maps of truncated energy functions, as in (5.35b), have been computed and convergence versus time evaluated for $x = 0, 1, 2$, and 3 . The resulting trajectories obtained from an electronic circuit simulator program SPICE2G1.6 are shown in Figure 5.11.

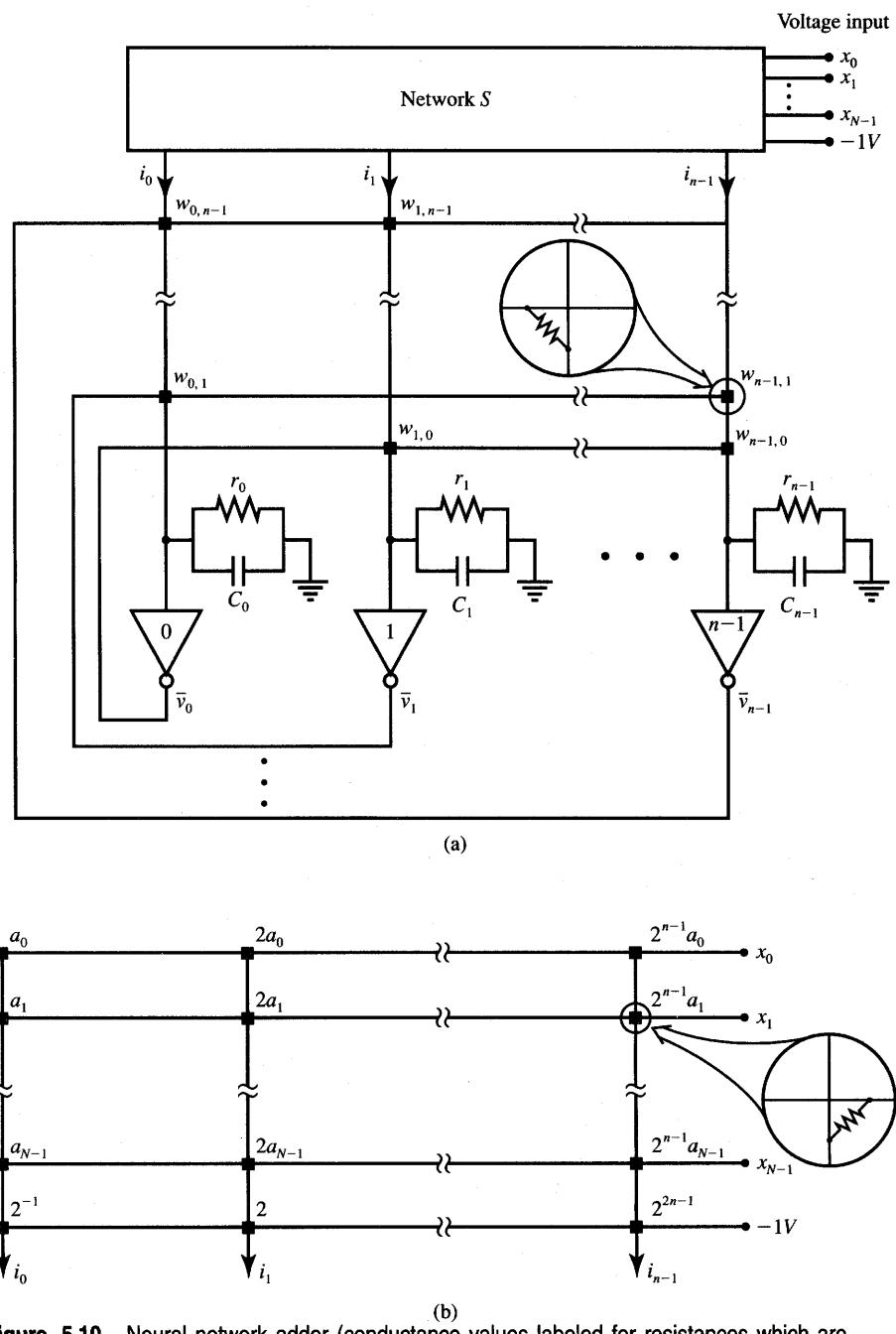


Figure 5.10 Neural network adder (conductance values labeled for resistances which are marked as squares): (a) overall block diagram and (b) network S .

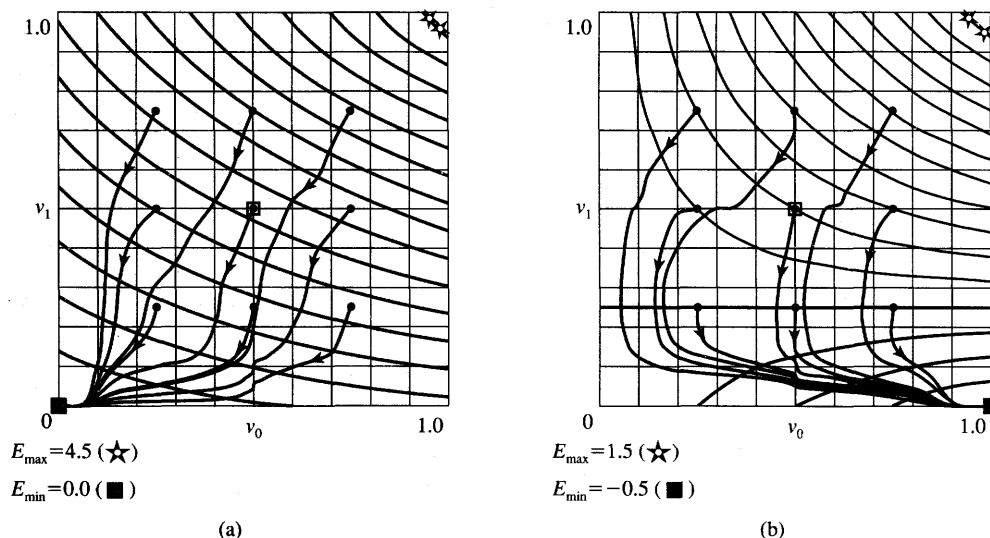


Figure 5.11a,b Equipotential lines and simulated transients for two-bit adder for different inputs (• denotes initial point, □ denotes zero initial conditions point): (a) $x = 0$, (b) $x = 1$.

The distance between each of the two neighboring energy contour lines shown is equal to $1/16$ of the energy difference within the cube. As expected, none of the energy functions has local minima within the cube. Transitions in time take place toward one of the constrained minima and they are generally not in the negative gradient direction.

In the cases of $x = 1.3$ and 1.8 , the corresponding energy functions have a saddle point and two minima at 01 and 10 , respectively. The convergence of transients is to the correct minimum in both cases for $C_0 = C_1$ if zero initial conditions are chosen. For other choices of initial conditions, the transients may decay while yielding an incorrect final response at the output of this neural-type A/D converter. The erroneous response case is depicted in Figures 5.11(e) and (f), which show three trajectories ending up at the incorrect minimum.

For small slopes of the energy surface, the convergence is expected to be slow. This has actually been observed in Figures 5.11(b) and (c) for curves leaving the initial points most distant from the correct solutions, which are 01 and 10 , respectively. The trajectories indicate rather slowly varying transients near $v_1 = 0.25$ [Figure 5.11(b)] and $v_1 = 0.75$ [Figure 5.11(c)], respectively. Slight discontinuities on all curves of Figure 5.11 are also noticeable. They are due to the shape of the voltage transfer characteristics near $u = 0$ for the actual electronic neuron models.

The results of transient simulations at the circuit level using the SPICE2G1.6 program for a four-bit summing network with zero initial conditions have been correct for 145 of 155 simulated cases. Input x applied to a four-bit summing

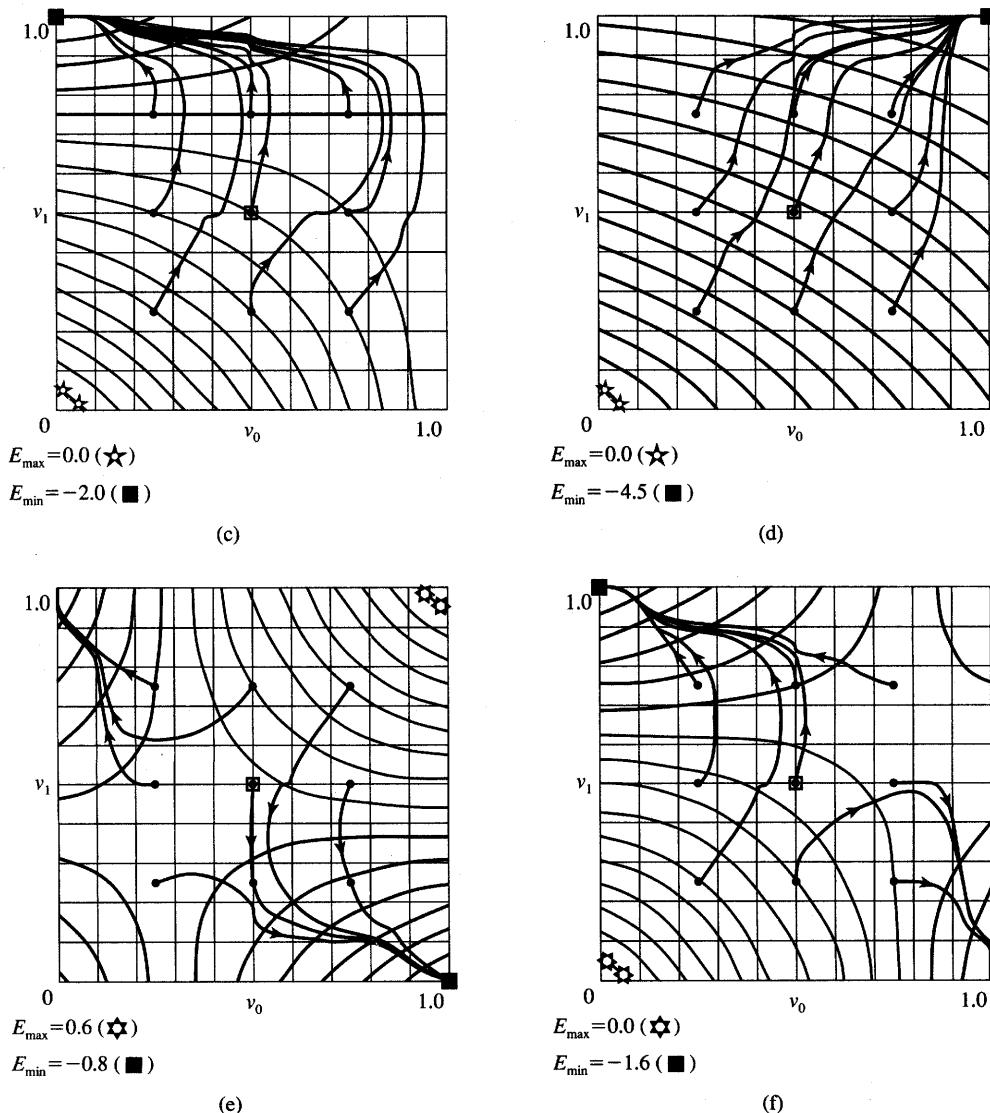


Figure 5.11c-f Equipotential lines and simulated transients for two-bit adder for different inputs (• denotes initial point, □ denotes zero initial conditions point) (continued): (c) $x = 2$, (d) $x = 3$, (e) $x = 1.3$, and (f) $x = 1.8$.

network is varied in the range between 0 and 15.5 with a step of 0.1. In all 10 cases of erroneous convergence, however, the settled binary output is adjacent to the correct answer, thus making the resulting response error bounded. Results of the transient circuit analysis and subsequent thresholding of v are shown in Table 5.1.

TABLE 5.1

Simulation results for input x from the interval $[0, 15.5]$, four-bit adder.

Input Integer Part	Input Decimal Part									
	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0	0	0	0	0	0	0.5	1	1	1	1
1	1	1	1	1	1	1	2	2	2	2
2	2	2	2	2	2	2.5	3	3	3	3
3	3	3	3	3	3	3	3	3	3	4
4	4	4	4	4	4	4.5	5	5	5	5
5	5	5	5	5	5	6	6	6	6	6
6	6	6	6	6	6	6.5	7	7	7	7
7	7	7	7	7	7	8	8	8	8	8
8	8	8	8	8	8	8.5	9	9	9	9
9	9	9	9	9	9	9	9	10	10	10
10	10	10	10	10	10	10.5	11	11	11	11
11	11	11	12	12	12	12	12	12	12	12
12	12	12	12	12	12	12.5	13	13	13	13
13	13	13	14	14	14	14	14	14	14	14
14	14	14	14	14	14	14.5	15	15	15	15
15	15	15	15	15	15					

Interesting results have been observed for simulation of the case when the integer answer for the sum, or A/D conversion problem solution, does not exist because it is exactly equidistant from the adjacent integers. The v_0 bit has remained undetermined and has settled very close to 0.5 for 8 of 15 such cases. Evaluation of the energy value (5.14) shows flat minima spread between adjacent integers. The case is reminiscent to the one depicted in Figure 5.7(a) for the truncated energy function of the two-neuron network.

Sample actual transients illustrating such cases are shown in Figure 5.12, which shows for case $x = 0.5$ that the truncated energy function is monotonic, it has no saddle point within the cube, and the least significant output bit of the network converges to near 0.5. On the other side if $x = 1.5$, the convergence is somewhat erroneous as shown in Figure 5.12(b), because the saddle point at $v_0 = v_1 = 0.5$ divides symmetrically the two monotonic subsurfaces of the error surface E . This case was depicted earlier in Figure 5.7(c) showing the truncated energy surface for $x = 1.5$.

Inspection of corresponding energy surfaces has also shown that for the erroneous results listed in Table 5.1 for the four-dimensional case, the network outputs have converged to erroneous shallow minima. It has also been observed that the convergence to correct minima is usually more reliable when initiated

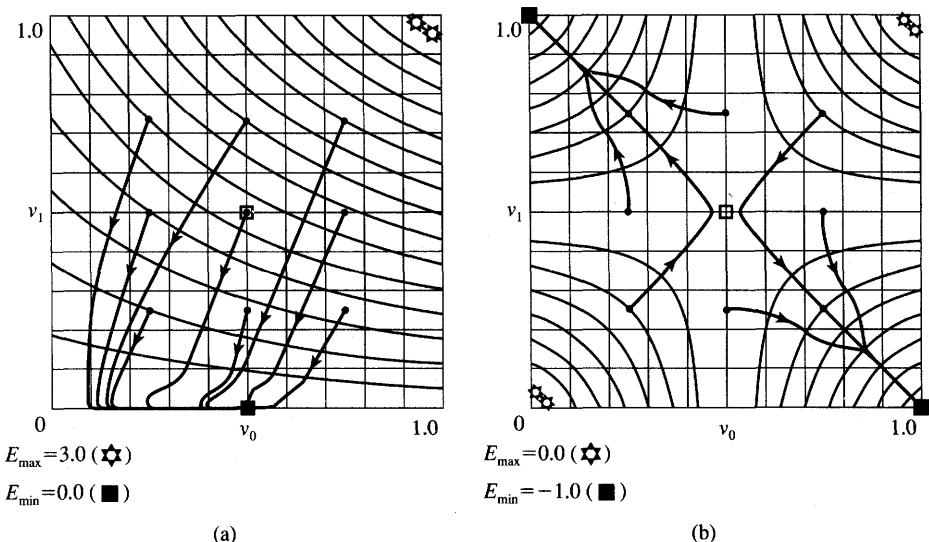


Figure 5.12 Equipotential lines and simulated transients for two-bit adder for different inputs (• denotes initial points, □ denotes zero initial conditions point): (a) $x = 0.5$ and (b) $x = 1.5$.

from a higher initial energy level. It has also been found that the network unmistakably rejected all obviously incorrect suggestions. Additions with negative results have all been rounded to the lowest sum 00, ..., 0. Additions with overflow sums have been rounded by the network to the highest sum 11, ..., 1.

Minimization of the Traveling Salesman Tour Length

The traveling salesman problem of minimization of the tour length through a number of cities with only one visit in each city is one of the classic optimization problems. The objective of the problem is to find a closed tour through n cities such that the tour length is minimized. The problem is NP-complete (nondeterministic polynomial time).

Typically, an optimization problem of size n may have many possible solutions. However, only one of the solutions minimizes the cost, or error, function. Optimization problems of this kind are called *combinational optimization problems*. They are often divided into classes according to the time needed to solve them.

If an algorithm exists that is able to solve the problem in a time that increases polynomially with the size n of the problem, the problem is said to be polynomial (P). NP-complete problems are a class within the P class. Such

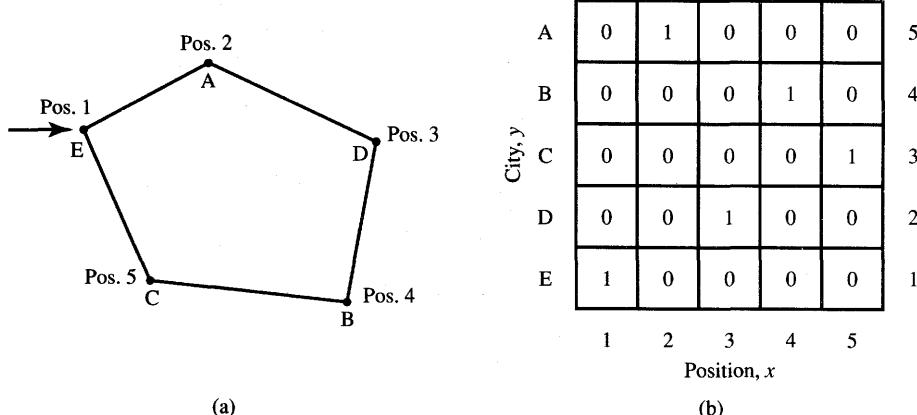


Figure 5.13 Traveling salesman problem example solution: (a) shortest path and (b) city/position matrix.

problems can be tested in polynomial time whether or not a guessed solution of the problem is correct. To solve an NP problem in practice, it usually takes time of order $\exp(n)$. Thus, the traveling salesman problem solution using the conventional combinational optimization approach grows exponentially with the number of cities n . We consider a solution for the problem using continuous-time gradient-type neural networks (Hopfield and Tank 1985).

An example of the cities and the proposed tour is shown in Figure 5.13(a) for $n = 5$. The distances between cities are assumed to be constant for a given set of visited cities. The reader can verify that there are $n!$ tours that can be produced in an n city problem. Among them, $2n$ closed paths are originating at different n cities and going in both directions; therefore, they are of identical lengths. Thus, there are $(n - 1)!/2$ distinct paths that have to be evaluated as far as total tour length minimization is concerned.

A network consisting of n^2 unipolar continuous neurons can be designed to solve this problem. Let us arrange the neurons in an $n \times n$ array. The suggested representation for the $n \times n$ neuron array can be arranged in the form of a matrix, where the j 'th row in the $n \times n$ matrix corresponds to a city y_j , and the i 'th column in the matrix corresponds to a city position x_i . We thus have city rows and position columns. One example solution of the problem is shown in Figure 5.13(b) for $n = 5$. It has the form of a city/position matrix with 0, 1 entries representing the best tour shown in Figure 5.13(a). Since each city can only be visited once and no simultaneous visits are allowed, the solution matrix can have only a single 1 in each column and a single 1 in each row. The neuron turned on, or with output 1, in the square array of neurons indicates a particular position of a particular city.

Note that the example of Figure 5.13 introduced specific notation. The row subscript corresponds to a city name, and the column subscript indicates a position in the tour. Using the correspondence between the neuron's array and the solution matrix, we may say that a valid solution of the problem would have only a single neuron in each column and only a single neuron in each row on. Thus, a total of n of n^2 neurons is distinctly turned on. Accordingly, we will assume high-gain neurons for the network modeling this problem. Note that the truncated energy function as in (5.14), with the third term omitted for simplicity, can be rewritten as follows:

$$E = -\frac{1}{2} \sum_{X_i} \sum_{Y_j} w_{X_i, Y_j} v_{X_i} v_{Y_j} - \sum_{X_i} i_{X_i} v_{X_i} \quad (5.49)$$

Let us observe that the domain of vectors \mathbf{v} where the function E is minimized is the $2^{n \times n}$ corners of the n^2 -dimensional hypercube $[0, 1]$.

Let us attempt to find the suitable expression for the energy function solving this problem (Hopfield and Tank 1985). Our objective is to find weights and currents that are coefficients of the quadratic form of the energy function (5.49) such that the objective function is minimized. The goal of the optimization is to select the shortest tours. Also, the matrix of responses has to be penalized for more than a single "on" neuron per row and per column. In addition, the matrix has to be penalized for trivial solutions of all entries being zero. Let us quantitatively express each of the optimization conditions just spelled out qualitatively by using four following terms of the energy function, each fulfilling a separate role:

$$E_1 = A \sum_X \sum_i \sum_j v_{X_i} v_{X_j}, \quad \text{for } i \neq j \quad (5.50a)$$

$$E_2 = B \sum_i \sum_X \sum_Y v_{X_i} v_{Y_i}, \quad \text{for } X \neq Y \quad (5.50b)$$

$$E_3 = C \left(\sum_X \sum_i v_{X_i} - n \right)^2 \quad (5.50c)$$

$$E_4 = D \sum_X \sum_Y \sum_i d_{XY} v_{X_i} (v_{Y,i+1} + v_{Y,i-1}), \quad X \neq Y \quad (5.50d)$$

Let us discuss the term E_1 , as in (5.50a), of the energy function. A matrix with 0 and 1 entries has no more than one 1 in each row if and only if all possible column-by-column dot products within the matrix are zero. The double internal sum of (5.50a) expresses the $N - 1$ dot products of a column having position X times the remaining columns. Performing the leftmost summation operation (in 5.50a) yields the required sum of all column dot products, which ideally should all be zero. The zero value of E_1 would guarantee that each city will be visited only once. Note that the penalty will progress and become larger if more than a single 1 is contained in one or more rows.

The energy term E_2 proposed as in (5.50b) can be substantiated in a similar way, and it contains a progressive penalty for two or more ones in each column of the matrix. This penalty takes a form of positive value for this term of the energy function and is for simultaneous visits in more than a single city. The

justification for this term is similar to that above for E_1 , however, it applies to controlling the number of ones within columns of the city/position matrix.

Note that penalties E_1 and E_2 are also of zero value for trivial solutions containing no ones, or less ones than required within the city/position matrix. Therefore the term E_3 , as in (5.50c), is required to ensure that the matrix does not simply contain all zeroes. In addition, this term penalizes the objective function for having more or less than n ones in the matrix. The energy function $E_1 + E_2 + E_3$ has minima of value 0 for all matrices that have exactly one 1 per row per column, and all other responses of neurons produce higher energy values. Minimization of the energy function $E_1 + E_2 + E_3$ ensures, or better, favors, that one of the valid tours can be generated.

The discussion of the energy function has focused on formulating constraints and has not yet involved the true goal of tour optimization. The term E_4 , as in (4.50d), needs to be added and minimized for the shortest tours. Simply summing the distance of the tour is a natural choice for E_4 , since minimizing the tour length is the goal. However, denoting the intercity distances as d_{XY} , we want to make sure that only the distances between adjacent cities are counted. To include end effects like the adjacency of city $n - 1$ and 1, the subscripts for summations have to be understood as summed modulo n . This term is numerically equal to the length of the path of the tour.

The resulting weight matrix and bias currents can now be obtained by equating the sum of energies E_1 through E_4 specified by (5.50) with the total energy value as in (5.49). The weights computed in this problem are (Hopfield and Tank 1985):

$$\begin{aligned} W_{Xi,Yj} = & -2A\delta_{XY}(1 - \delta_{ij}) - 2B\delta_{ij}(1 - \delta_{XY}) \\ & - 2C - 2Dd_{XY}(\delta_{j,i+1} + \delta_{j,i-1}) \end{aligned} \quad (5.51a)$$

where δ_{ij} is the Kronecker delta function defined as $\delta_{ij} = 1$, for $i = j$, and $\delta_{ij} = 0$, for $i \neq j$. Positive constants A , B , C , and D are selected heuristically to build an appropriate sum of terms E_1 through E_4 . The constants are responsible for weighting the relative validity of each of the four penalty terms in the overall energy function E to be minimized. The external bias currents are

$$i_{Xi} = 2Cn \quad (5.51b)$$

Observe that the four separate terms of the weight matrix (5.51a) have been generated by the four parts of the energy function as in Equations (5.50a) through (5.50d), respectively. The term E_1 leads to inhibitory (negative) connection of value $-A$ within each row. The term E_2 causes identical inhibitory connections within each column. The term E_3 results in global inhibition provided by each weight, and the term E_4 contains the city distances data weight contribution. Finally, the external input currents are set at an excitation bias level as in (5.51b).

The problem formulated as shown above has been solved numerically for the continuous activation function defined in (2.4a) with $\lambda = 50$, $A = B = D = 250$, and $C = 100$, for $10 \leq n \leq 30$ (Hopfield and Tank 1985). The normalized city

maps were randomly generated within the unity square to initialize each computational experiment using a simulated neural network. The equations solved numerically were

$$\begin{aligned} \frac{du_{X_i}}{dt} = & -\frac{u_{X_i}}{\tau} - 2A \sum_{j \neq i} v_{X_j} - 2B \sum_{Y \neq X} v_{Y_i} \\ & - 2C \left(\sum_X \sum_j v_{X_j} - n \right) - 2D \sum_Y d_{XY} (v_{Y,i+1} + v_{Y,i-1}) \end{aligned} \quad (5.52)$$

where the time constant τ was selected as 1. For $n = 10$, 50% of the trials produced the two best paths among 181440 distinct paths. For $n = 30$, there exists about 4.4×10^{30} possible paths. The simulations of the solutions for this size of problem have routinely produced paths shorter than 7. Statistical evaluation of all paths shows that there are only 10^8 paths shorter than 7. Although no optimal solution has been reported when using this approach, the bad path rejection ratio by the network has been impressive. In numerical experiments by Hopfield and Tank (1985), the bad path rejection ratio was between 10^{-23} and 10^{-22} .

There are practical limitations of the discussed approach. The method becomes more difficult to apply for larger problems, typically larger than 10 cities. However, the example of the traveling salesman problem illustrates that a class of optimization tasks can be approached using the dynamical system model presented in this chapter. Many other possible applications of the model have been reported in the literature. Typical problems solved have been of the resource allocation type, which are subject to different constraints. Job shop scheduling optimization has been reported by Foo and Takefuji (1988). In another application, the concentrator-to-sites assignment problem was mapped onto the single-layer network model by associating each neuron with the hypothesis that a given site should be assigned a particular concentrator (Tagliarini and Page 1988).

Fully coupled single-layer networks can be applied to a variety of other technical problems. Networks of this class are capable of handling optimal routing of calls through a three-stage interconnection network (Melsa, Kenney, and Rohrs 1990). The solution uses a discrete neuron network that seeks a minimum of an energy function for cases in which an open path exists through the interconnection network. When no such path is available, the energy function minimization performed by the network terminates at a null state indicating that the interconnection network is blocked.

Another successful application of energy minimizing networks is for routing communication traffic within a multinode network consisting of many nodes and links. The solution of such a problem requires finding the best of multilink paths for node-to-node traffic to minimize loss. The loss is represented by expected delay or some other traffic inconvenience. The minimization procedure has been implemented using a modification of the traveling salesman problem solution (Rauch and Winarske 1988). Reasonable convergence for a 16-node network has been reported for up to four links from origin to destination.

Other applications of this class of neural networks involve microelectronic circuit module placement on the chip that minimizes the total interconnecting wire length (Sriram and Kang 1990). The two-dimensional problem has been decomposed into two coupled one-dimensional placement problems. The performance of the designed neural network on problems involving placement with up to 64 modules has been very encouraging, with the network being able to find the globally optimal or near-optimal solutions in many cases.

Single-layer networks have also been successfully applied to general linear and nonlinear programming tasks. Nonlinear programming is a basic tool in systems where a set of design parameters is optimized subject to inequality constraints. Both experimental and theoretical results have been presented for completely stable solutions of specific linear and nonlinear programming problems (Kennedy and Chua 1988; Maa and Shanblatt 1989). An economic power dispatch problem has also been solved by using the linear programming network with linear equality and inequality constraints (Maa, Chin, and Shanblatt 1990). A linear programming problem formulation and solution is also demonstrated in Section 8.1.

5.7

CONCLUDING REMARKS

In this chapter we have introduced the basic properties of single-layer feedback neural networks. Neurocomputing algorithms producing convergence toward solutions for both discrete-time and continuous-time networks have been reviewed. The discussion emphasized the dynamical principles of single-layer continuous-time feedback networks. Also, several network design examples have been presented to illustrate the basic concepts of system evolution toward its attractors. Throughout the chapter, the concept of energy function has been stressed. This had two goals in mind: to demonstrate the inherent stability of the networks and the suitability of the networks for producing solutions of certain optimization tasks.

In addition to many questions that have been answered about the performance of this neural network model, some problems still remain open. Let us raise some of the remaining open questions. The stationary solution reached in the time domain does not represent, in general, a global optimum solution to the problem of energy function minimization. This is due to the often highly complex shape of the multidimensional energy function $E(v)$. This limitation is somewhat mitigated by the fact that global solutions of real large-scale minimization problems of $E(v)$ are often mathematically very hard to track anyway, and for large-scale problems cannot be found with certainty.

One of the difficult tasks faced by the designer at the beginning of the design process is the translation of the optimization problem into the energy function minimization. Here, the general energy function must be matched by the criterion

function which is specific for the problem being solved. Given the restrictions on the connectivity matrix imposed by the model, these functions may not match easily. Therefore, problem-specific energy functions are usually hard to find. We have been able to discuss several choices of $E(v)$, but the general solution to the problem does not exist. The only energy form known is the general one stated as in (5.5) or (5.14). Following simplifying modification can often be applied here: If high-gain neurons are used, the third term of the energy expression can be skipped. Also, when no external bias input i is needed, the second energy term vanishes and network weights only need to be computed.

It often takes much of the network designer's ingenuity to devise a meaningful energy function in the neurons output space v and to find its coefficients w_{ij} and i_i . In fact, our ignorance concerning possibly existing energy functions that solve a number of optimization problems creates one of the bottlenecks that makes the use of the gradient-type networks not as simple. For certain classes of systems, such as associative memories, however, $E(v)$ and w_{ij} are easy to find in an explicit form. This will be discussed in the next chapter.

As we have seen, the convergence toward an attractor being a solution for single-layer feedback neural networks corresponds to the transient behavior of a nonlinear dynamical system. Although the concept of a time constant is not directly applicable to nonlinear networks, the transient analysis shows that the convergence of a neural network typically requires several time constants as defined for the network's linear portion. This property is often reported based on numerous experiments both involving actual hardware and numerical simulations.

Figure 5.14 illustrates the procedure for solving optimization problems using single-layer feedback networks. Basic steps that need to be followed by the designer are summarized on the diagram. The problem modeling procedure terminates by implementing transients within the network and accepting the solution produced. In easy optimization tasks, each of the generated solutions is usually correct. This has been the case for A/D conversion networks for low n values. For harder optimization problems such as the traveling salesman problem for large n ($n \geq 30$), the produced solutions are often good but not strictly optimal. This is due to the presence of many local minima in n -dimensional space which are responsible for trapping the evolution of the transients. It should be realized that this class of networks for large n values yields solutions which are not necessarily optimal but acceptable in a statistical sense, i.e., over a population of experiments rather than for a single transient event.

An additional limitation existing within the networks discussed concerns their capacity. There is a limit on the total number of correct solutions that can be stored and expected from the network as its output. Although this number cannot be set precisely, heuristic evaluations of gradient-type networks show that the number of encoded solution vectors that can be retrieved satisfactorily is only a small fraction of the total number of neurons n . The network capacity issues and others related to network performance will be discussed in more detail in the following chapter.

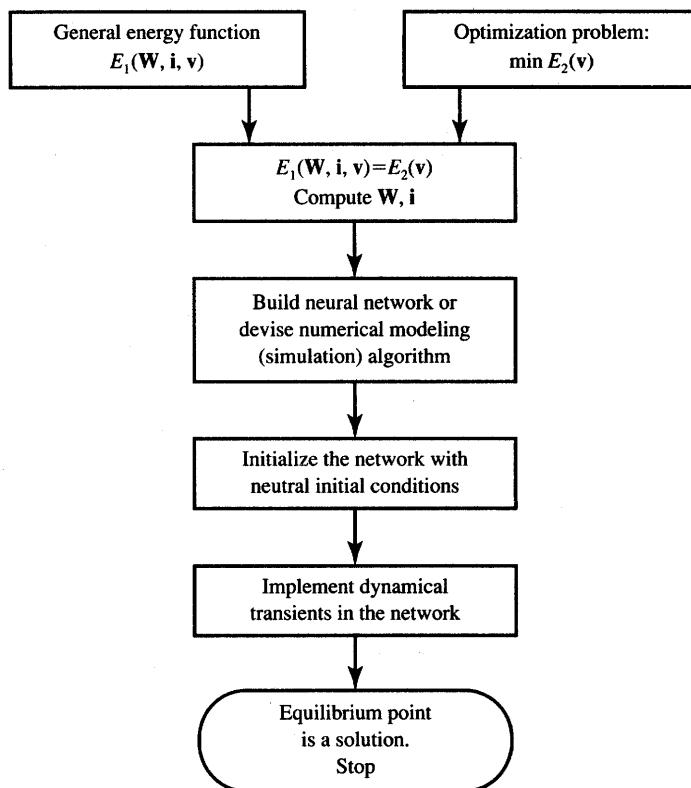


Figure 5.14 Flowchart of producing solutions of optimization problems using fully coupled feed-back networks.

PROBLEMS

Please note that problems highlighted with an asterisk (*) are typically computationally intensive and the use of programs is advisable to solve them.

P5.1 The weight matrix W for a network with bipolar discrete binary neurons is given as

$$W = \begin{bmatrix} 0 & 1 & -1 & -1 & -3 \\ 1 & 0 & 1 & 1 & -1 \\ -1 & 1 & 0 & 3 & 1 \\ -1 & 1 & 3 & 0 & 1 \\ -3 & -1 & 1 & 1 & 0 \end{bmatrix} \Omega^{-1}$$

Knowing that the thresholds and external inputs of neurons are zero, compute the values of energy for $v = [-1 \ 1 \ 1 \ 1 \ 1]^t$ and $v = [-1 \ -1 \ 1 \ -1 \ -1]^t$.

P5.2 Figure P5.2 shows a discrete-time recurrent network with high-gain bipolar neurons.

- (a) Find the weight matrix of the network by inspecting the connections.
- (b) Analyze asynchronous updates from the following initial states:

$$\mathbf{v}^0 = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \mathbf{v}^0 = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ -1 \end{bmatrix}, \mathbf{v}^0 = \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \\ -1 \end{bmatrix}, \mathbf{v}^0 = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}, \mathbf{v}^0 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ 1 \\ -1 \end{bmatrix}$$

Assume that the updates take place in natural order starting with the first neuron, i.e., 1, 2,

- (c) Identify any stable equilibrium state(s) of the network based on part (b).
- (d) Compute the energy value at the stable equilibrium state(s) evaluated in part (c). (Use the truncated energy function.)

*P5.3** Assuming the weight matrix and other input conditions from Problem P5.1, compute the energy values for all 32 bipolar binary vectors (there are five energy levels here). Identify the potential attractors that may have been encoded in the system described by the specified matrix \mathbf{W} by comparing the energy values at each of the $[-1, +1]$ cube vertices. Implement five sample asynchronous discrete-time transitions from high-to low-energy vertices.

P5.4 The weight matrix \mathbf{W} for a single-layer feedback network with three neurons is given as

$$\mathbf{W} = \begin{bmatrix} 0 & -1 & -1 \\ -1 & 0 & 1 \\ -1 & 1 & 0 \end{bmatrix} \Omega^{-1}$$

Calculate the gradient vector, $\nabla E(\mathbf{v})$, for the energy function in three-dimensional output space and its Hessian matrix, $\nabla^2 E(\mathbf{v})$. Prove that the Hessian matrix is not positive definite (see the Appendix).

P5.5 For the two-neuron continuous-time single-layer feedback network shown in Figure P5.5 with high-gain neurons, find the following:

- (a) state equations
- (b) the weight (conductance) matrix, \mathbf{W}
- (c) the truncated energy function, $E(\mathbf{v})$
- (d) the gradient vector of the truncated energy function, $\nabla E(\mathbf{v})$.

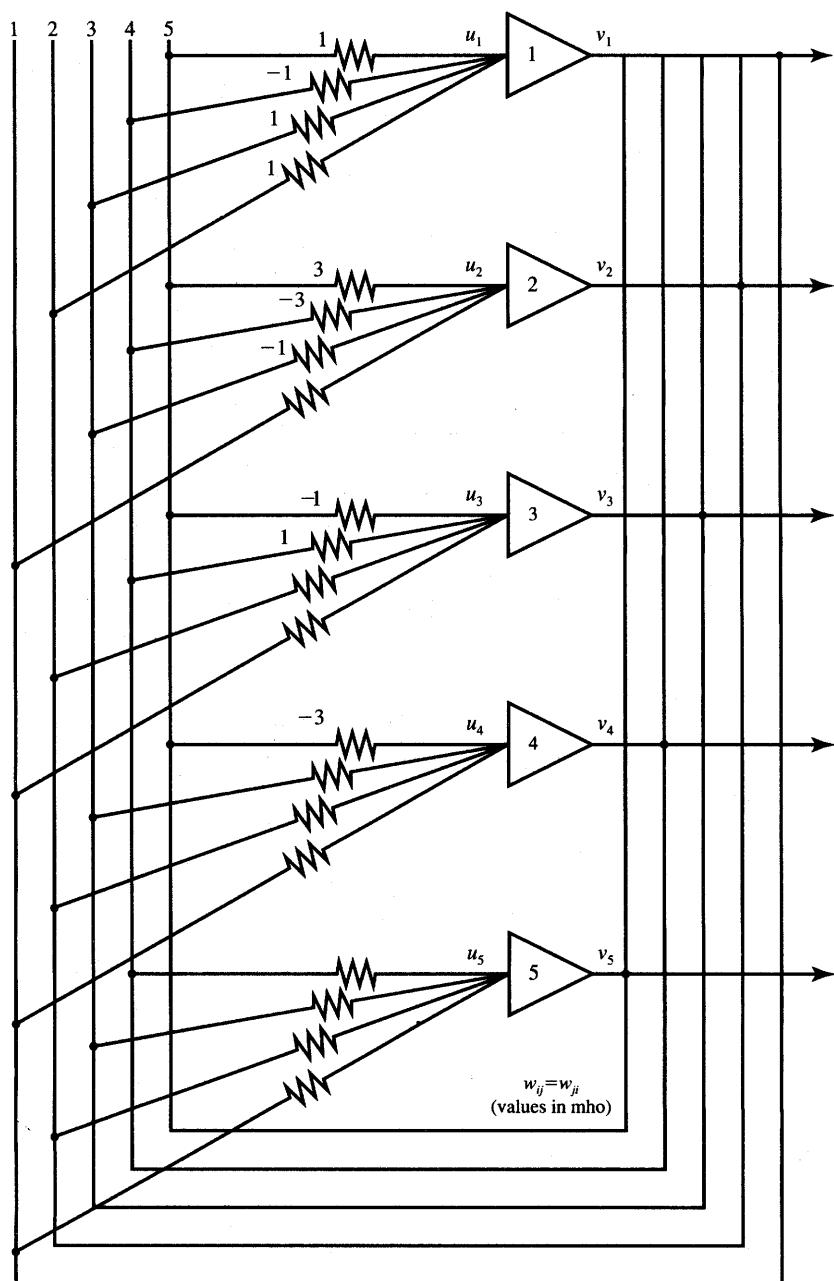


Figure P5.2 Discrete-time recurrent network for analysis.

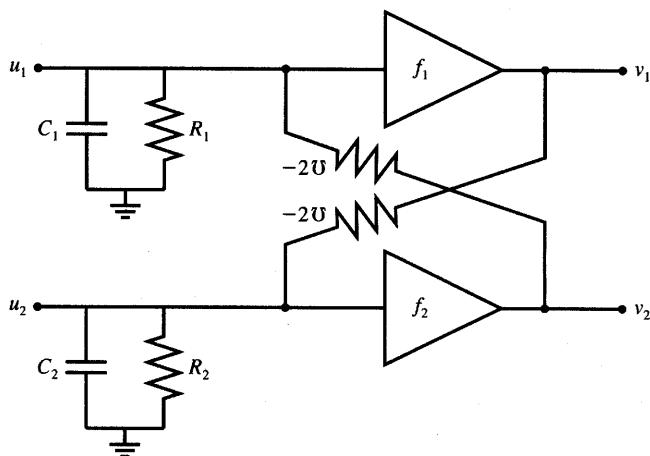


Figure P5.5 Electronic neural network for Problem P5.5.

- P5.6** For the continuous-time single-layer feedback network shown in Figure P5.6 with high-gain neurons, find the following:
- state equations
 - the weight (conductance) matrix, \mathbf{W}
 - the current vector, \mathbf{i}
 - the truncated energy function, $E(\mathbf{v})$
 - the gradient vector of the truncated energy function, $\nabla E(\mathbf{v})$.
- P5.7** Find the weight matrix \mathbf{W} for the four high-gain neuron network shown in Figure P5.7. The network uses inverting neurons and four positive-valued conductances of value $2\Omega^{-1}$ each. Then find the energy values for outputs v_1 , v_2 , and v_3 given as below:
- $$\mathbf{v}_1 = [-1 \ -1 \ 1 \ 1]', \mathbf{v}_2 = [-1 \ -1 \ -1 \ 1]', \mathbf{v}_3 = [-1 \ -1 \ -1 \ -1]'$$
- P5.8** Sketch the appropriate conductances connecting outputs of inverting and noninverting neurons in a network having the overall diagram as shown in Figure P5.7 knowing that

$$\mathbf{W} = \begin{bmatrix} 0 & -1 & -1 \\ -1 & 0 & 1 \\ -1 & 1 & 0 \end{bmatrix} \Omega^{-1}, \quad \mathbf{i} = \mathbf{0}$$

Evaluate the truncated energy function produced by the network for $v_i = \pm 1$, for $i = 1, 2, 3$, and identify possible attractors of the discrete-time network with high-gain neurons.

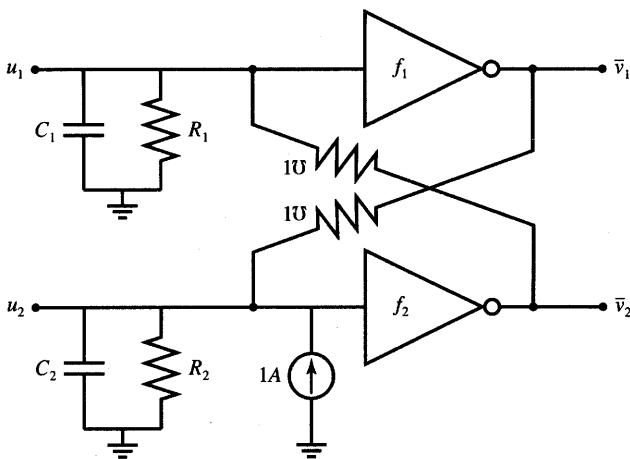


Figure P5.6 Electronic neural network for Problem P5.6.

P5.9 The truncated energy function, $E(\mathbf{v})$, of a certain two-neuron network is specified as

$$E(\mathbf{v}) = -\frac{1}{2} (v_1^2 + 2v_1v_2 + 4v_2^2 + v_1)$$

Assuming high-gain neurons:

- (a) Find the weight matrix \mathbf{W} and the bias current vector \mathbf{i} .
- (b) Determine whether single-layer feedback neural network postulates (symmetry and lack of self-feedback) are fulfilled for \mathbf{W} and \mathbf{i} computed in part (a).

P5.10 Assuming the energy functions and other conditions as in Problem P5.9, find

- (a) the gradient vector of the energy function, $\nabla E(\mathbf{v})$
- (b) the Hessian matrix of the energy function, $\nabla^2 E(\mathbf{v})$
- (c) any unconstrained minima or maxima \mathbf{v}^* the energy function may have.

P5.11 The truncated energy function of a certain three-neuron single-layer network is known as

$$E(\mathbf{v}) = v_1^2 - v_2^2 + v_3^2 - 2v_1v_3 - v_2v_3 + 4v_1 + 12$$

Find the following:

- (a) the gradient vector, $\nabla E(\mathbf{v})$
- (b) the Hessian matrix, $\nabla^2 E(\mathbf{v})$

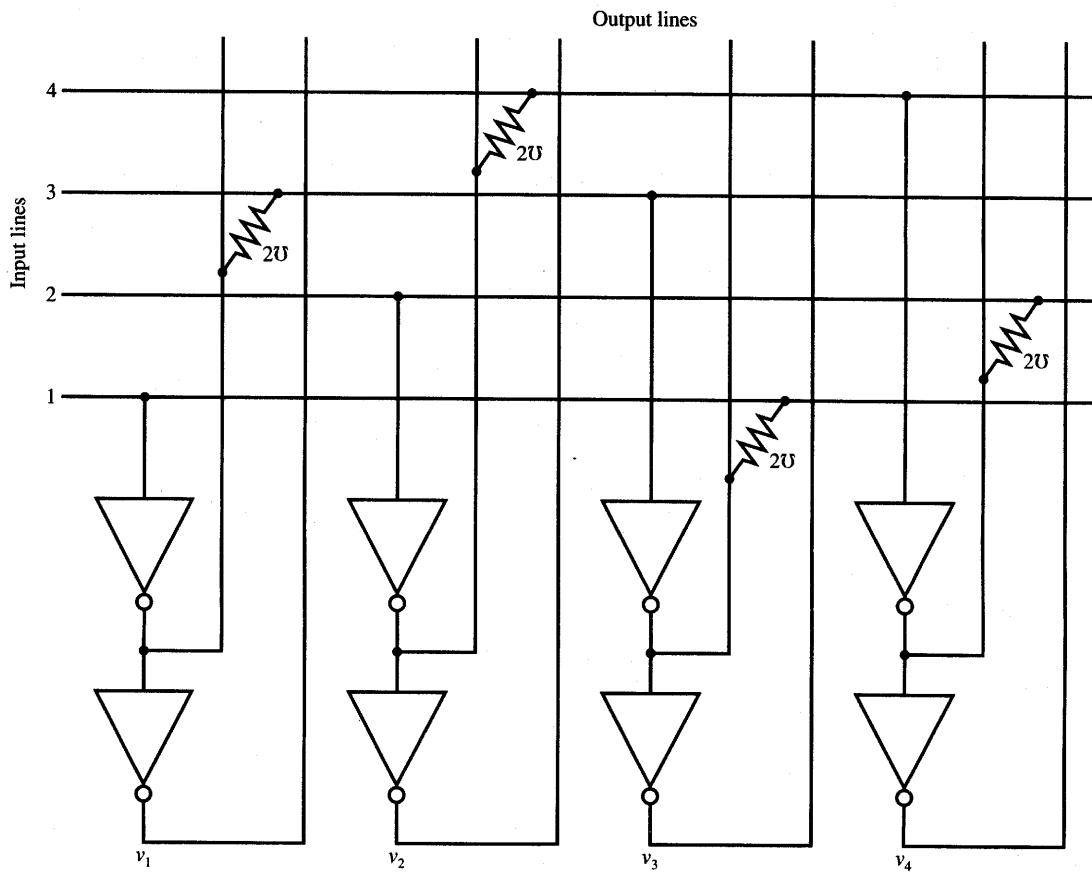


Figure P5.7 Network for Problem P5.7.

(c) any minima, maxima, or saddle points the energy function may have. What are they?

P5.12 The high-gain neuron continuous-time network is shown in Figure P5.12. By evaluating its energy function, find analytically any minima, maxima, or saddle points the function has within the $[-1, 1]$ three-dimensional cube. Use the truncated energy expression containing no third term. Then compute numerical energy values at each of the cube's eight vertices.

P5.13 The energy function $E(v)$ for the two-bit A/D converter discussed in Example 5.3 is specified by Equation (5.35b). It has been derived under the assumption of infinite gain neurons. Calculate the accurate energy function at $v = [0.01 \ 0.01]^T$ using the accurate expression for energy (5.14) which, in addition to (5.35b), involves the additional integral term

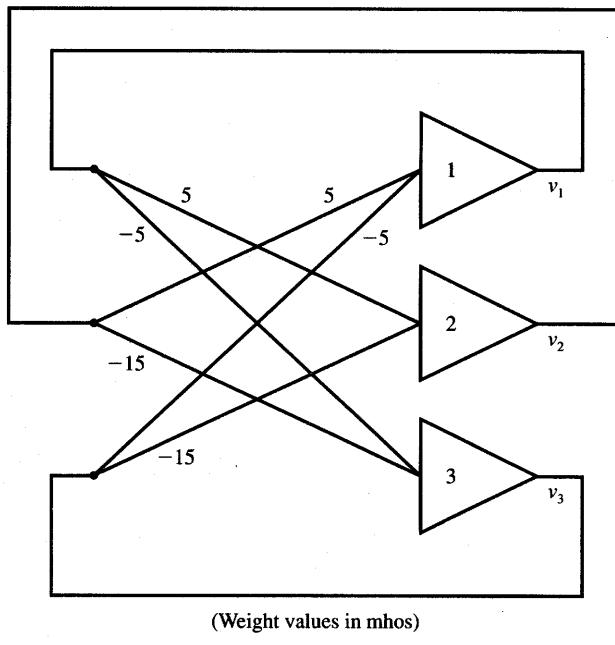


Figure P5.12 High-gain neuron network for Problem P5.12.

(5.38c). This term is equal to

$$\sum_{i=0}^1 G_i \int_{0.5}^{v_i} f_i^{-1}(z) dz$$

Assume the converter as in Figure 5.6(b) with $g_0 = g_1 = 1 \Omega^{-1}$, $x = 1.7$, and

$$v_i = f_i(u_i) = \frac{1}{1 + \exp(-10u_i)}$$

P5.14 Calculate the truncated energy function $E(v_0, v_1, v_2)$ for a three-bit A/D converter that uses high-gain neurons. Then calculate the point v^* for which $\nabla E(v^*) = \mathbf{0}$. Based on the evaluation of the Hessian matrix, $\nabla^2 E(v^*)$, determine what is the type of point v^* computed.

P5.15 A single-layer feedback neural network is described by the nonlinear differential equations, which must be solved numerically

$$\frac{du_i}{dt} = \frac{1}{C_i} \left(\sum_{j=1}^n w_{ij} v_j - u_i G_i + i_i \right), \quad \text{for } i = 1, 2, \dots, n$$

and

$$v_i = f(u_i)$$

- (a) Derive the forward Euler numerical integration formula to find u_i , for $i = 1, 2, \dots, n$.
- (b) Derive the backward Euler numerical integration formula to find u_i , for $i = 1, 2, \dots, n$.

P5.16* Implement the backward Euler numerical integration formula derived in Problem P5.15 to solve for $v(t)$. Obtain the solution for a simple two-bit A/D converter as in Figure 5.6. Assume $C_0 = C_1 = 1F$, $g_0 = g_1 = 1\Omega^{-1}$, $x = 0$, and

$$v_i = f(u_i) = \frac{1}{1 + \exp(-10u_i)}, \quad \text{for } i = 0, 1$$

Test the program for the following initial conditions: $v_0 = [0.25 \ 0.25]^t$, $v_0 = [0.5 \ 0.5]^t$, and $v_0 = [0.75 \ 0.75]^t$.

P5.17 Rearrange the nonlinear differential equations (5.28) describing the simple two-bit A/D converter to the form (5.34) so that the vector field components ψ_0 , and ψ_1

$$\begin{aligned}\frac{dv_0}{dt} &= \psi_0(v_0, v_1) \\ \frac{dv_1}{dt} &= \psi_1(v_0, v_1)\end{aligned}$$

can be explicitly computed for known values of x_i , g_i , C_i , and λ_i ($i = 0, 1$). Find $\psi_0(v_0, v_1)$ and $\psi_1(v_0, v_1)$ for the activation function $v_i = [1 + \exp(-\lambda u_i)]^{-1}$.

P5.18 The stationary point solution of Equations (5.28) describing the simple two-bit A/D converter can be obtained using the Newton-Raphson method by solving the nonlinear algebraic equation $\mathbf{F}(\mathbf{u}) = \mathbf{0}$, where

$$\mathbf{F}(\mathbf{u}) = \begin{bmatrix} -2v_1 + x - 0.5 - (g_0 - 2)u_0 \\ -2v_0 + 2x - 2 - (g_1 - 2)u_1 \end{bmatrix}$$

The iterative solution is obtained in this method as shown:

$$\begin{bmatrix} u_0^{k+1} \\ u_1^{k+1} \end{bmatrix} = \begin{bmatrix} u_0^k \\ u_1^k \end{bmatrix} - [\mathbf{J}(\mathbf{u})]^{-1} \begin{bmatrix} F_0(\mathbf{u}^k) \\ F_1(\mathbf{u}^k) \end{bmatrix}$$

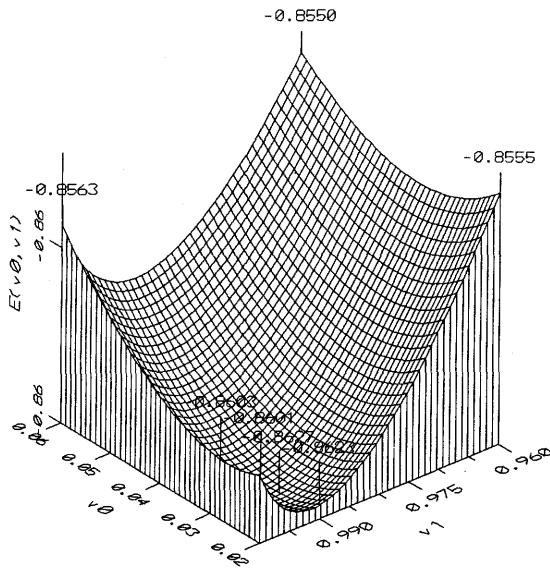
where the Jacobian matrix is defined

$$\mathbf{J} \triangleq \begin{bmatrix} \frac{\partial F_0}{\partial u_0} & \frac{\partial F_0}{\partial u_1} \\ \frac{\partial F_1}{\partial u_0} & \frac{\partial F_1}{\partial u_1} \end{bmatrix}$$

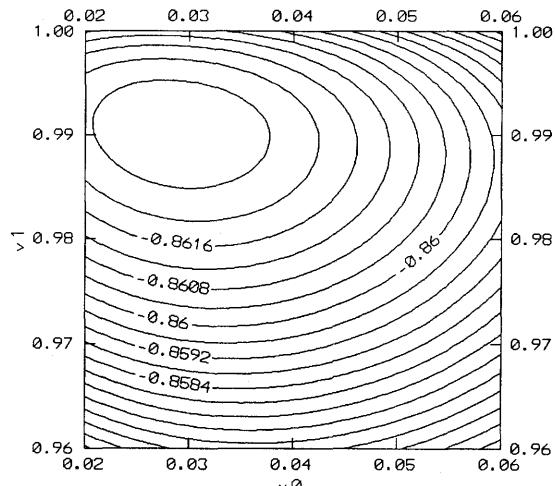
Find the closed form for the iterative solution of the equation $\mathbf{F}(\mathbf{u}) = \mathbf{0}$, including the Jacobian matrix entries (no matrix inversion required). The

A/D 2-bit Converter

Input $x=1.6$, Lambda = 2
Neuron Input Conductance = 2.5



(a)



(b)

Figure P5.19 Total energy function for the two-bit A/D converter as in Problem P5.19 for $x = 1.6$, $\lambda = 2$, and $g_0 = g_1 = 2.5 \Omega^{-1}$ (a) energy surface (b) energy contour map.

activation function to be assumed is

$$v_i = [1 + \exp(-\lambda u_i)]^{-1}, \quad \text{for } i = 0, 1$$

P5.19 Figure P5.19 illustrates the energy surface and the energy contour map near the upper left corner of the unity square. The graphs shown are made for the two-bit A/D converter and the following conditions: $x = 1.6$, $\lambda = 2$, and $g_0 = g_1 = 2.5 \Omega^{-1}$. The displayed energy function expresses the total energy of the network, thus it involves the integral term (5.38c) additive with the truncated energy function (5.35b). Compute the numerical value of the total energy function at the energy minimum located approximately at

$$v_0^* \approx 0.03, \quad v_1^* \approx 0.99$$

P5.20 Figure P5.20 illustrates two cases of vector fields for the two-bit A/D converter. Each of the continuous lines drawn within the fields are for the condition $\partial v_0 / \partial t = 0$ and $\partial v_1 / \partial t = 0$. Obviously, stationary points are

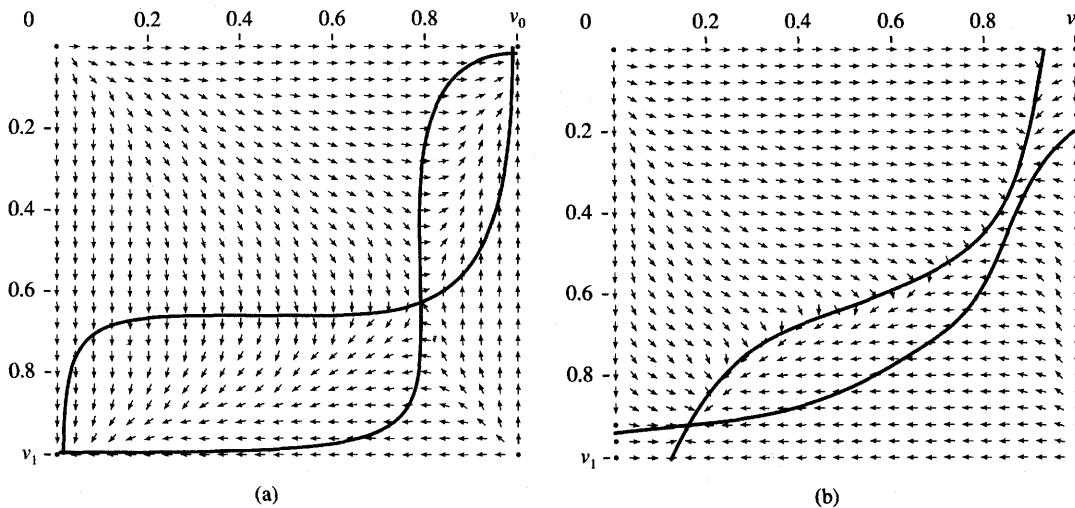


Figure P5.20 Vector fields for Problem P5.20.

located at the intersections of the lines, since both $\partial v_0 / \partial t = \partial v_1 / \partial t = 0$ there. By analyzing the vector field determine the following conditions:

- What are the types of the stationary points displayed in each of the figures?
- What is the approximate analog input value for the vector field of Figure P5.20(a)?

REFERENCES

- Alspector, J., R. B. Allen, V. Hu, S. Satyanarayana. 1988. "Stochastic Learning Networks and Their Electronic Implementation," in *Neural Information Processing Systems*, ed. D. Z. Anderson, New York: American Institute of Physics, pp. 9-21.
- DiZitti, E., et al. 1989. "Analysis of Neural Algorithms for Parallel Architectures," in *Proc. 1989 IEEE Int. Symp. Circuits and Systems*, Portland, Ore., May 9-12, 1989. New York: IEEE, pp. 2187-2190.
- Hopfield, J. J. 1984. "Neurons with Graded Response Have Collective Computational Properties Like Those of Two State Neurons," *Proc. National Academy of Sciences* 81: 3088-3092.
- Hopfield, J. J., and D. W. Tank. 1985. "Neural" Computation of Decisions in Optimization Problems," *Biolog. Cybern.* 52: 141-154.

- Hopfield, J. J., and D. W. Tank. 1986. "Computing with Neural Circuits: A Model," *Science* 233: 625-633.
- Howard, R. E., L. D. Jackel, and H. P. Graf. 1988. "Electronic Neural Networks," *AT&T Tech. J.* (May): 58-64.
- Kamp, Y., and M. Hasler. 1990. *Recursive Neural Networks for Associative Memory*, Chichester, U.K.: John Wiley & Sons.
- Kennedy, M. P., and L. O. Chua. 1988. "Neural Networks for Nonlinear Programming," *IEEE Trans. Circuits and Systems* CAS-35(5): 554-562.
- Maa, C. Y., and M. A. Shanblatt. 1989. "Improved Linear Programming Neural Networks," in *Proc. 31st Midwest Symp. on Circuits and Systems*, Urbana, Ill., August 1989, New York, IEEE, pp. 748-751.
- Maa, C. Y., C. Chin, and M. A. Shanblatt. 1990. "A Constrained Optimization Neural Net Techniques for Economic Power Dispatch," in *Proc. 1990 IEEE Int. Symp. on Circuits and Systems*, New Orleans, La., May 1-3, 1990. New York: IEEE, pp. 2945-2948.
- Melsa, P. J., J. B. Kenney, and C. E. Rohrs. 1990. "A Neural Network Solution for Routing in Three Stage Interconnection Network," in *Proc. 1990 IEEE Int. Symp. on Circuits and Systems*, New Orleans, La., May 1-3, 1990. New York: IEEE, pp. 482-485.
- Park, S. 1989. "Signal Space Interpretation of Hopfield Neural Network for Optimization," in *Proc. 1989 IEEE Int. Symp. Circuits and Systems*, Portland, Ore., May 9-12, 1989. New York: IEEE, pp. 2181-2184.
- Rauch, H. E., and T. Winarske. 1988. "Neural Networks for Routing Communications Traffic," *IEEE Control Systems Magazine* (April): 26-31.
- Roska, T. 1988. "Some Qualitative Aspects of Neural Computing Circuits," in *Proc. 1988 IEEE Int. Symp. on Circuits and Systems*, Helsinki. New York: IEEE, pp. 751-754.
- Sriram, M., and S. M. Kang. 1990. "A Modified Hopfield Network for Two-dimensional Module Placement," in *Proc. 1990 IEEE Int. Symp. on Circuits and Systems*, New Orleans, La., May 1-3, 1990. New York: IEEE, pp. 1663-1666.
- Tagliarini, G. A., and E. W. Page. 1988. "A Neural Network Solution to the Concentrator Assignment Problem," in *Neural Information Processing Systems*, ed. D. Z. Anderson, New York: American Institute of Physics, pp. 775-782.
- Tank, D. W., and J. J. Hopfield. 1986. "Simple 'Neural' Optimization Networks: An A/D Converter, Signal Decision Circuit and a Linear Programming Circuit," *IEEE Trans. Circuits and Systems* CAS-33(5): 533-541.
- Zurada, J. M., and M. J. Kang. 1988. "Summing Networks Using Neural Optimization Concept," *Electron. Lett.* 24(10): 616-617.

- Zurada, J. M., and M. J. Kang. 1988. "Computational Circuits Using Neural Optimization Concept," *Int. J. Electron.* 67(3): 311-320.
- Zurada, J. M., M. J. Kang, and P. B. Aronhime. 1990. "Vector Field Analysis of Single Layer Feedback Neural Networks," in *Proc. Midwest Symp. on Circuits and Systems*, Calgary, Canada, August 12-14, 1990 IEEE, New York: 22-24.
- Zurada, J. M., and W. Shen. 1990. "Sufficient Condition for Convergence of Relaxation Algorithm in Neural Optimization Circuits," *IEEE Trans. Neural Networks* 1(4): 300-303.
- Zurada, J. M. 1992. "Gradient-Type Neural Systems for Computation and Decision-Making," to appear in *Progress in Neural Networks*, ed. O. M. Omidvar. Vol. II. Norwood, New Jersey, Ablex Publishing Company.

6

ASSOCIATIVE MEMORIES

*The clock upbraids me
with a waste of time.*

SHAKESPEARE

- 6.1 Basic Concepts
- 6.2 Linear Associator
- 6.3 Basic Concepts of Recurrent Autoassociative Memory
- 6.4 Performance Analysis of Recurrent Autoassociative Memory
- 6.5 Bidirectional Associative Memory
- 6.6 Associative Memory of Spatio-temporal Patterns
- 6.7 Concluding Remarks

✓

In the preceding chapter we were concerned with dynamical systems that can be used in information processing systems. As we have shown, their dynamic behavior exhibits stable states that act as attractors during the system's evolution in time. Our discussion of dynamical systems thus far has been primarily oriented toward solving optimization problems. In this chapter, we will interpret the system's evolution as a movement of an input pattern toward another stored pattern, called a *prototype* or *stored memory*. Specifically, we will look at building associations for pattern retrieval and restoration. We will also study the dynamics of the discrete-time convergence process. Neural networks of this class are called *associative memories* and are presented in this chapter.

An efficient associative memory can store a large set of patterns as memories. During recall, the memory is excited with a *key pattern* (also called the *search argument*) containing a portion of information about a particular member of a stored pattern set. This particular stored prototype can be recalled through association of the key pattern and the information memorized. A number of architectures and approaches have been devised in the literature to solve effectively the problem of both memory recording and retrieval of its content. While

most of our discussion in this chapter will involve dynamical systems using fully coupled feedback networks from Chapter 5, feedforward memory architectures that employ no feedback will also be discussed. Since feedforward memories were the first developed, their study seems to be both informative and provides insight into the fundamental concepts of neural memories.

Associative memories belong to a class of neural networks that learns according to a certain recording algorithm. They usually acquire information *a priori*, and their connectivity (weight) matrices most often need to be formed in advance. Writing into memory produces changes in the neural interconnections. Reading of the stored information from memory, introduced in Chapter 2 as recall, can be considered as the transformation of the input signals by the network. No usable addressing scheme exists in an associative memory since all memory information is spatially distributed and superimposed throughout the network.

Let us review the expectations regarding associative memory capabilities. The memory should have as large a capacity as possible or a large p value, which denotes the number of stored prototypes. At the same time the memory should be able to store data in a robust manner, so that local damage to its structure does not cause total breakdown and inability to recall. In addition, the ideal memory should truly associate or regenerate stored pattern vectors and do so by means of specific similarity criteria. Another very desirable feature of memory would be its ability to add and eliminate associations as storage requirements change.

Associative memory usually enables a parallel search within a stored data file. The purpose of the search is to output either one or all stored items that match the given search argument, and to retrieve it either entirely or partially. It is also believed that biological memory operates according to associative memory principles. No memory locations have addresses; storage is distributed over a large, densely interconnected, ensemble of neurons. What exactly is meant by that network of interconnections is seldomly defined for biological systems. The operating principles of artificial neural memory models are sometimes also very involved, and their presentation in this chapter is by no means exhaustive. The intention is to provide an understanding of basic associative memory concepts and of the potential benefits, applications, and limitations.

6.1

BASIC CONCEPTS

Figure 6.1 shows a general block diagram of an associative memory performing an associative mapping of an input vector x into an output vector v . The system shown maps vectors x to vectors v , in the pattern space \mathbb{R}^n and output space \mathbb{R}^m , respectively, by performing the transformation

$$v = M[x] \quad (6.1)$$

The operator M denotes a general nonlinear matrix-type operator, and it has

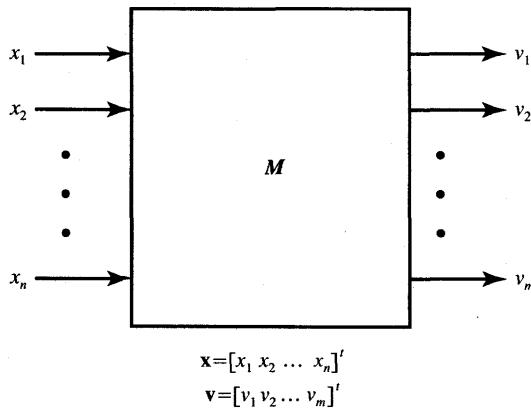


Figure 6.1 Block diagram of an associative memory.

different meaning for each of the memory models. Its form, in fact, defines a specific model that will need to be carefully outlined for each type of memory. The structure of M reflects a specific neural memory paradigm. For dynamic memories, M also involves time variable. Thus, v is available at memory output at a later time than the input has been applied.

For a given memory model, the form of the operator M is usually expressed in terms of given prototype vectors that must be stored. The algorithm allowing the computation of M is called the *recording* or *storage algorithm*. The operator also involves the nonlinear mapping performed by the ensemble of neurons. Usually, the ensemble of neurons is arranged in one or two layers, sometime intertwined with each other.

The mapping as in Equation (6.1) performed on a key vector \mathbf{x} is called a *retrieval*. Retrieval may or may not provide a desired solution prototype, or an undesired prototype, but it may not even provide a stored prototype at all. In such an extreme case, erroneously recalled output does not belong to the set of prototypes. In the following sections we will attempt to define mechanisms and conditions for efficient retrieval of prototype vectors.

Prototype vectors that are stored in memory are denoted with a superscript in parenthesis throughout this chapter. As we will see below, the storage algorithm can be formulated using one or two sets of prototype vectors. The storage algorithm depends on whether an autoassociative or a heteroassociative type of memory is designed.

Let us assume that the memory has certain prototype vectors stored in such a way that once a key input has been applied, an output produced by the memory and associated with the key is the memory response. Assuming that there are p stored pairs of associations defined as

$$\mathbf{x}^{(i)} \rightarrow \mathbf{v}^{(i)}, \quad \text{for } i = 1, 2, \dots, p \quad (6.2a)$$

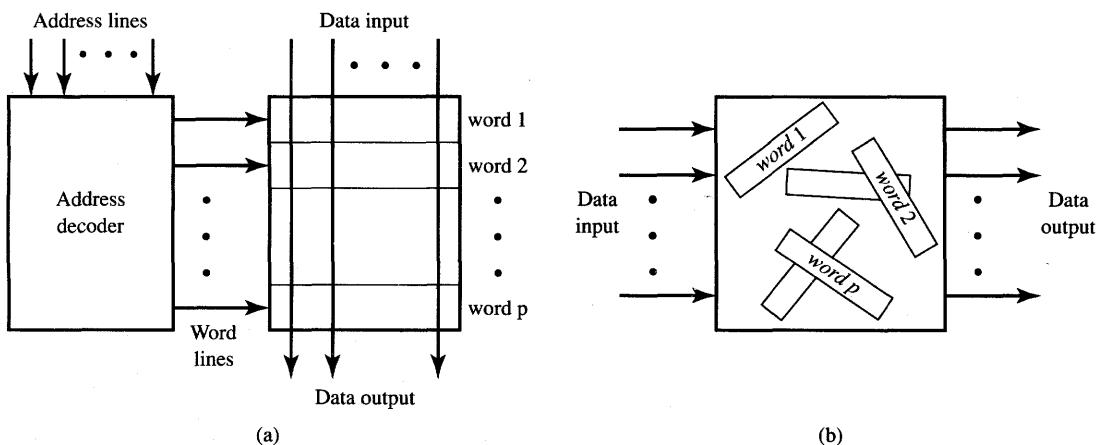


Figure 6.2 Addressing modes for memories: (a) address-addressable memory and (b) content-addressable memory.

and $v^{(i)} \neq x^{(i)}$, for $i = 1, 2, \dots, p$, the network can be termed as *heteroassociative memory*. The association between pairs of two ordered sets of vectors $\{x^{(1)}, x^{(2)}, \dots, x^{(p)}\}$ and $\{v^{(1)}, v^{(2)}, \dots, v^{(p)}\}$ is thus heteroassociative. An example of heteroassociative mapping would be a retrieval of the missing member of the pair $(x^{(i)}, v^{(i)})$ in response to the input $x^{(i)}$ or $v^{(i)}$. If the mapping as in (6.2a) reduces to the form

$$x^{(i)} \rightarrow v^{(i)}|_{v^{(i)}=x^{(i)}}, \quad \text{for } i = 1, 2, \dots, p \quad (6.2b)$$

then the memory is called *autoassociative*. Autoassociative memory associates vectors from within only one set, which is $\{x^{(1)}, x^{(2)}, \dots, x^{(p)}\}$. Obviously, the mapping of a vector $x^{(i)}$ into itself as suggested in (6.2b) cannot be of any significance. A more realistic application of an autoassociative mapping would be the recovery of an undistorted prototype vector in response to the distorted prototype key vector. Vector $x^{(i)}$ can be regarded in such case as stored data and the distorted key serves as a search key or argument.

Associative memory, which uses neural network concepts, bears very little resemblance to digital computer memory. Let us compare their two different addressing modes which are commonly used for memory data retrieval. In digital computers, data are accessed when their correct addresses in the memory are given. As can be seen from Figure 6.2(a), which shows a typical memory organization, data have input and output lines, and a word line accesses and activates the entire word row of binary cells containing word data bits. This activation takes place whenever the binary address is decoded by the address decoder. The addressed word can be either "read" or replaced during the "write" operation. This is called *address-addressable* memory.

In contrast with this mode of addressing, associative memories are *content-addressable*. The words in this memory are accessed based on the content of the key vector. When the network is excited with a portion of the stored data $\mathbf{x}^{(i)}$, $i = 1, 2, \dots, p$, the efficient response of the autoassociative network is the complete $\mathbf{x}^{(i)}$ vector. In the case of heteroassociative memory, the content of vector $\mathbf{x}^{(i)}$ should provide the stored response $\mathbf{v}^{(i)}$. However, there is no storage for prototype $\mathbf{x}^{(i)}$ or $\mathbf{v}^{(i)}$, for $i = 1, 2, \dots, p$, at any location within the network. The entire mapping (6.2) is distributed in the associative network. This is symbolically depicted in Figure 6.2(b). The mapping is implemented through dense connections, sometimes involving feedback, or a nonlinear thresholding operation, or both.

Associative memory networks come in a variety of models. The most important classes of associative memories are static and dynamic memories. The taxonomy is based entirely on their recall principles. Static networks recall an output response after an input has been applied in one feedforward pass, and, theoretically, without delay. They were termed *instantaneous* in Chapter 2. Dynamic memory networks produce recall as a result of output/input feedback interaction, which requires time. Respective block diagrams for both memory classes are shown in Figure 6.3. The static networks implement a feedforward operation of mapping without a feedback, or recursive update, operation. As such they are sometimes also called *non-recurrent*. Static memory with the block diagram shown in Figure 6.3(a) performs the mapping as in Equation (6.1), which can be reduced to the form

$$\mathbf{v}^k = M_1[\mathbf{x}^k] \quad (6.3a)$$

where k denotes the index of recursion and M_1 is an operator symbol. Equation (6.3a) represents a system of nonlinear algebraic equations. Examples of static networks will be discussed in the next section.

Dynamic memory networks exhibit dynamic evolution in the sense that they converge to an equilibrium state according to the recursive formula

$$\mathbf{v}^{k+1} = M_2 [\mathbf{x}^k, \mathbf{v}^k] \quad (6.3b)$$

provided the operator M_2 has been suitably chosen. The operator operates at the present instant k on the present input \mathbf{x}^k and output \mathbf{v}^k to produce the output in the next instant $k + 1$. Equation (6.3b) represents, therefore, a system of nonlinear difference equations. The block diagram of a recurrent network is shown in Figure 6.3(b). The delay element in the feedback loop inserts a unity delay Δ , which is needed for cyclic operation. Autoassociative memory based on the Hopfield model is an example of a recurrent network for which the input \mathbf{x}^0 is used to initialize \mathbf{v}^0 , i.e., $\mathbf{x}^0 = \mathbf{v}^0$, and the input is then removed. The vector retrieved at the instant k can be computed with this initial condition as shown:

$$\mathbf{v}^{k+1} = M_2[\mathbf{v}^k] \quad (6.3c)$$

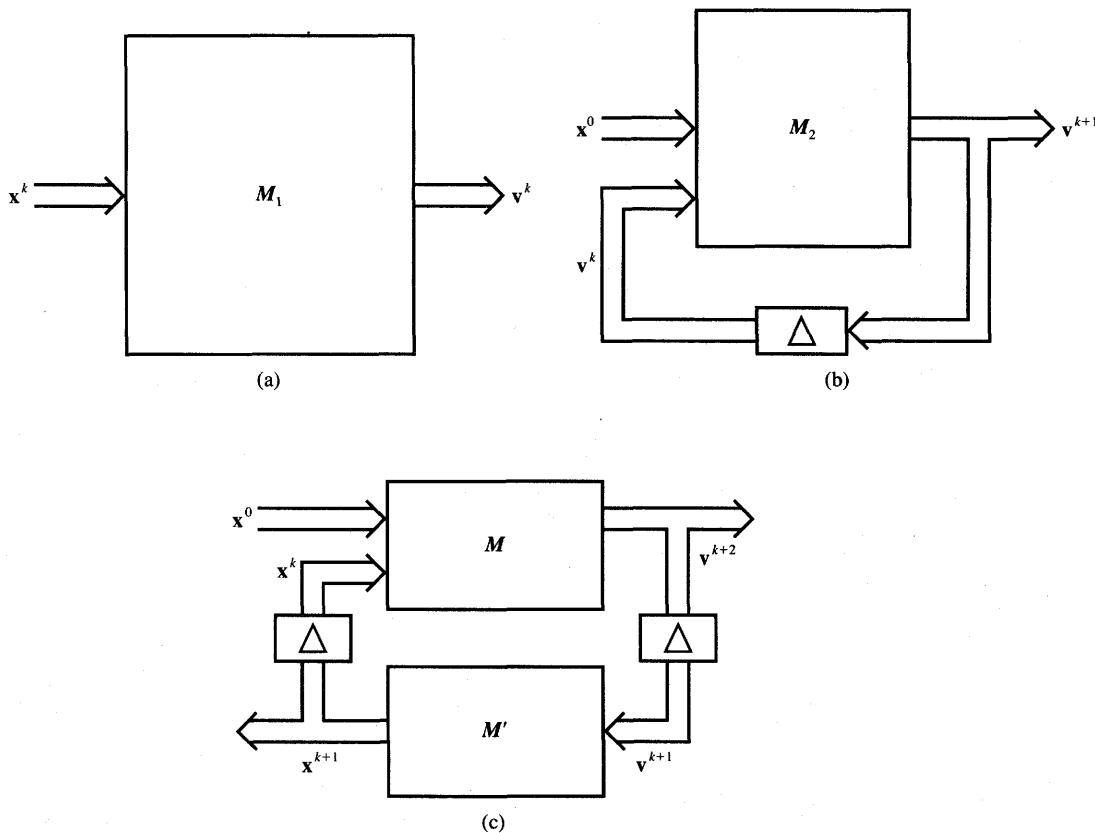


Figure 6.3 Block diagram representation of associative memories: (a) feedforward network, (b) recurrent autoassociative network, and (c) recurrent heteroassociative network.

Figure 6.3(c) shows the block diagram of a recurrent heteroassociative memory that operates with a cycle of 2Δ . The memory associates pairs of vectors $(\mathbf{x}^{(i)}, \mathbf{v}^{(i)})$, $i = 1, 2, \dots, p$, as given in (6.2a).

Figure 6.4 shows Hopfield autoassociative memory without the initializing input \mathbf{x}_0 . The figure also provides additional details on how the recurrent memory network implements Equation (6.3c). Operator M_2 consists of multiplication by a weight matrix followed by the ensemble of nonlinear mapping operations $v_i = f(\text{net}_i)$ performed by the layer of neurons. The details of processing were discussed in earlier chapters.

There is a substantial resemblance of some elements of autoassociative recurrent networks with feedforward networks discussed in Section 4.5 covering the back propagation network architecture. Using the mapping concepts proposed in (4.30c) and (4.31) we can rewrite expression (6.3c) in the following

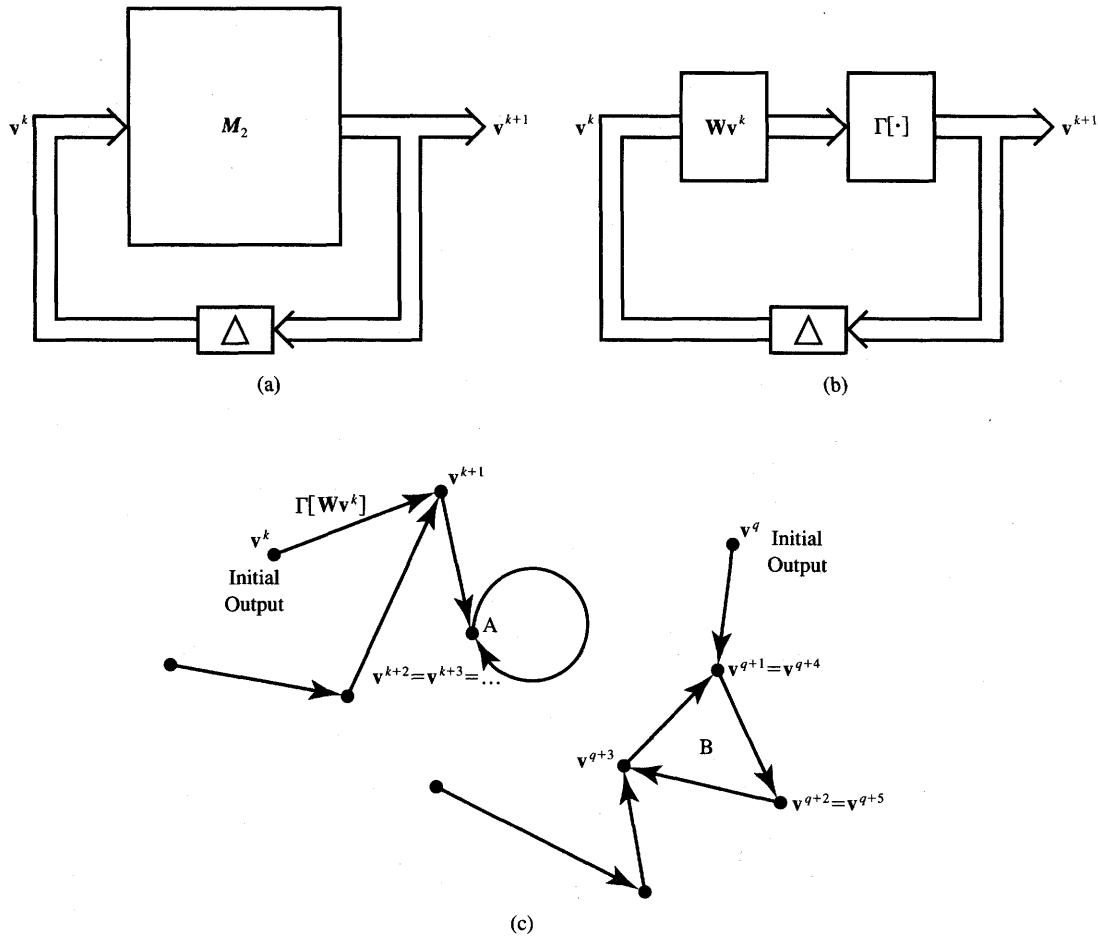


Figure 6.4 Autoassociative recurrent memory: (a) block diagram, (b) expanded block diagram, and (c) example state transition map.

customary form:

$$\mathbf{v}^{k+1} = \Gamma[\mathbf{W}\mathbf{v}^k] \quad (6.4)$$

where \mathbf{W} is the weight matrix of a single layer as defined throughout Chapter 4 or 5. The operator $\Gamma[\cdot]$ is a nonlinear matrix operator with diagonal elements that are hard-limiting (binary) activation functions $f(\cdot)$:

$$\Gamma[\cdot] = \begin{bmatrix} \text{sgn}(\cdot) & 0 & \cdots & 0 \\ 0 & \text{sgn}(\cdot) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \text{sgn}(\cdot) \end{bmatrix} \quad (6.5)$$

The expanded block diagram of the memory is shown in Figure 6.4(b). Although mappings performed by both feedforward and feedback networks are similar, recurrent memory networks respond with bipolar binary values, and operate in a cyclic, recurrent fashion. Their time-domain behavior and properties will therefore no longer be similar.

Regarding the vector $v(k+1)$ as the state of the network at the $(k+1)$ 'th instant, we can consider recurrent Equation (6.4) as defining a mapping of the vector v into itself. The memory state space consists of 2^n n -tuple vectors with components ± 1 . The example state transition map for a memory network is shown in Figure 6.4(c). Each node of the graph is equivalent to a state and has one and only one edge leaving it. If the transitions terminate with a state mapping into itself, as is the case of node A, then the equilibrium A is the fixed point. If the transitions end in a cycle of states as in nodes B, then we have a limit cycle solution with a certain period. The period is defined as the length of the cycle. The figure shows the limit cycle B of length three.

Let us begin with a review of associative memory networks beginning with static networks.

6.2

LINEAR ASSOCIATOR

Traditional associative memories are of the feedforward, instantaneous type. As defined in (6.2a), the task required for the associative memory is to learn the association within p vector pairs $\{x^{(i)}, v^{(i)}\}$, for $i = 1, 2, \dots, p$. For the *linear associative memory*, an input pattern x is presented and mapped to the output by simply performing the matrix multiplication operation

$$v = Wx \quad (6.6a)$$

where x, v, W are matrices of size $n \times 1, m \times 1$, and $m \times n$, respectively. Thus, the general nonlinear mapping relationship (6.3a) has been simplified to the linear form (6.6a), hence the memory name. The linear associative network diagram can be drawn as in Figure 6.5. Only the customary weight matrix W is used to perform the mapping. Noticeably, the network does not involve neuron elements, since no nonlinear or delay operations are involved in the linear association. If, however, the use of neurons is required for the reason of uniform perspective of all neural networks, then the mapping (6.3a) can be rewritten as

$$v = M_1[Wx] \quad (6.6b)$$

where $M_1[\cdot]$ is a dummy linear matrix operator in the form of the $m \times m$ unity matrix. This observation can be used to append an output layer of dummy neurons with identity activation functions $v_i = f(\text{net}_i) = \text{net}_i$. The corresponding network extension is shown within dashed lines in Figure 6.5.

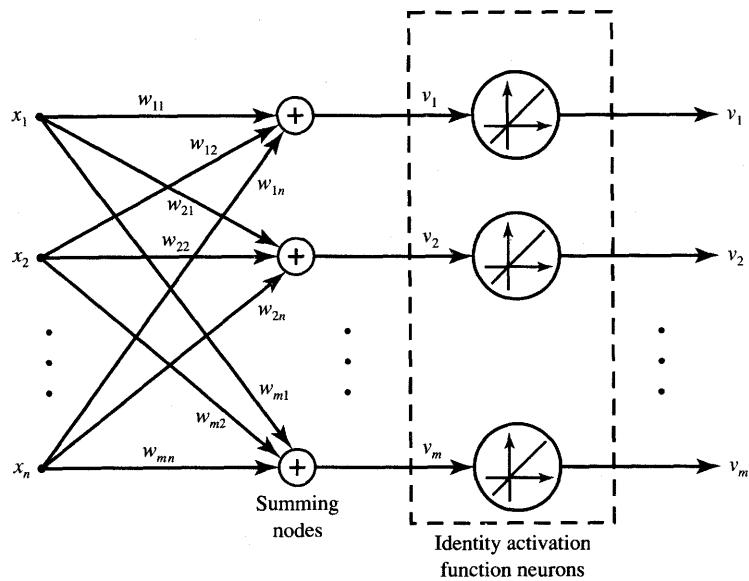


Figure 6.5 Linear associator.

Let us assume that p associations need to be stored in the linear associator. Given are pairs of vectors $\{\mathbf{s}^{(i)}, \mathbf{f}^{(i)}\}$, for $i = 1, 2, \dots, p$, denoting the stored memories, called *stimuli*, and forced responses, respectively. Since this memory is strictly unidirectional, these terms are self-explanatory. We thus have for n -tuple stimuli and m -tuple response vectors of the i 'th pair:

$$\mathbf{s}^{(i)} = [s_1^{(i)} \ s_2^{(i)} \ \dots \ s_n^{(i)}]^t, \quad \text{and} \quad (6.6c)$$

$$\mathbf{f}^{(i)} = [f_1^{(i)} \ f_2^{(i)} \ \dots \ f_m^{(i)}]^t \quad (6.6d)$$

In practice, $\mathbf{s}^{(i)}$ can be patterns and $\mathbf{f}^{(i)}$ can be information about their class membership, or their images, or any other pairwise assigned association with input patterns. The objective of the linear associator is to implement the mapping (6.6a) as follows

$$\mathbf{f}^{(i)} + \boldsymbol{\eta}^i = \mathbf{W}\mathbf{s}^{(i)}$$

or, using the mapping symbol

$$\mathbf{s}^{(i)} \rightarrow \mathbf{f}^{(i)} + \boldsymbol{\eta}^i, \quad \text{for } i = 1, 2, \dots, p \quad (6.7)$$

such that the length of the *noise term vector* denoted as $\boldsymbol{\eta}^i$ is minimized. In general, the solution for this problem aimed at finding the memory weight matrix \mathbf{W} is not very straightforward. First of all, matrix \mathbf{W} should be found such that the Euclidean norm $\sum_i \|\boldsymbol{\eta}^i\|$, is minimized for a large number of observations

of mapping (6.7). This problem is dealt with in the mathematical regression analysis and will not be covered here. The general solution of the problem of optimal mapping (6.7) can be found in references on early associative memories (Kohonen 1977; Kohonen et al. 1981). We will find below, however, partial solutions to the general mapping problem expressed by Equation (6.7) that will also prove to be useful later in this chapter.

Below we focus on finding the matrix \mathbf{W} that allows for efficient storage of data within the memory. Let us apply the Hebbian learning rule in an attempt to train the linear associator network. The weight update rule for the i 'th output node and j 'th input node can be expressed as

$$w'_{ij} = w_{ij} + f_i s_j, \quad \text{for } i = 1, 2, \dots, m \quad \text{and } j = 1, 2, \dots, n \quad (6.8a)$$

where f_i and s_j are the i 'th and j 'th components of association vectors \mathbf{f} and \mathbf{s} and w_{ij} denotes the weight value before the update. The reader should note that the vectors to be associated, \mathbf{f} and \mathbf{s} , must be members of the pair. To generalize formula (6.8a) so it is valid for a single weight matrix entry update to the case of the entire weight matrix update, we can use the outer product formula. We then obtain

$$\mathbf{W}' = \mathbf{W} + \mathbf{f}\mathbf{s}^t \quad (6.8b)$$

where \mathbf{W} denotes the weight matrix before the update. Initializing the weights in their unbiased position $\mathbf{W}_0 = \mathbf{0}$, we obtain for the outer product learning rule:

$$\mathbf{W}' = \mathbf{f}^{(i)} \mathbf{s}^{(i)t} \quad (6.9a)$$

Expression (6.9a) describes the first learning step and involves learning of the i 'th association among p distinct paired associations. Since there are p pairs to be learned, the superposition of weights can be performed as follows

$$\mathbf{W}' = \sum_{i=1}^p \mathbf{f}^{(i)} \mathbf{s}^{(i)t} \quad (6.9b)$$

The memory weight matrix \mathbf{W}' above has the form of a *cross-correlation matrix*. An alternative notation for \mathbf{W}' is provided by the following formula:

$$\mathbf{W}' = \mathbf{F}\mathbf{S}^t \quad (6.9c)$$

where \mathbf{F} and \mathbf{S} are matrices containing vectors of forced responses and stimuli and are defined as follows:

$$\begin{aligned} \mathbf{F} &\triangleq [\mathbf{f}^{(1)} \quad \mathbf{f}^{(2)} \quad \dots \quad \mathbf{f}^{(p)}] \\ \mathbf{S} &\triangleq [\mathbf{s}^{(1)} \quad \mathbf{s}^{(2)} \quad \dots \quad \mathbf{s}^{(p)}] \end{aligned}$$

where the column vectors $\mathbf{f}^{(i)}$ and $\mathbf{s}^{(i)}$ were defined in (6.6c) and (6.6d). The resulting cross-correlation matrix \mathbf{W}' is of size $m \times n$. Integers n and m denote sizes of stimuli and forced responses vectors, respectively, as introduced in (6.6c) and (6.6d). We should now check whether or not the weight matrix \mathbf{W} provides

noise-free mapping as required by expression (6.7). Let us attempt to perform an associative recall of the vector $\mathbf{f}^{(i)}$ when $\mathbf{s}^{(i)}$ is applied as a stimulus. If one of the stored vectors, say $\mathbf{s}^{(j)}$, is now used as key vector at the input, we obtain from the retrieval formula [(6.6a) and (6.6b)]:

$$\mathbf{v} = \left(\sum_{i=1}^p \mathbf{f}^{(i)} \mathbf{s}^{(i)t} \right) \mathbf{s}^{(j)} \quad (6.10a)$$

Expanding the sum of p terms yields

$$\mathbf{v} = \mathbf{f}^{(1)} \mathbf{s}^{(1)t} \mathbf{s}^{(j)} + \dots + \mathbf{f}^{(j)} \mathbf{s}^{(j)t} \mathbf{s}^{(j)} + \dots + \mathbf{f}^{(p)} \mathbf{s}^{(p)t} \mathbf{s}^{(j)} \quad (6.10b)$$

According to the mapping criterion (6.7), the ideal mapping $\mathbf{s}^{(j)} \rightarrow \mathbf{f}^{(j)}$ such that no noise term is present would require

$$\mathbf{v} = \mathbf{f}^{(j)} \quad (6.10c)$$

By inspecting (6.10b) and (6.10c) it can be seen that the ideal mapping can be achieved in the case for which

$$\begin{aligned} \mathbf{s}^{(i)t} \mathbf{s}^{(j)} &= 0, \quad \text{for } i \neq j \\ \mathbf{s}^{(j)t} \mathbf{s}^{(j)} &= 1 \end{aligned} \quad (6.11)$$

Thus, the orthonormal set of p input stimuli vectors $\{\mathbf{s}^{(1)}, \mathbf{s}^{(2)}, \dots, \mathbf{s}^{(p)}\}$ ensures perfect mapping (6.10c). Orthonormality is the condition on the inputs if they are to be ideally associated. However, the condition is rather strict and may not always hold for the set of stimuli vectors.

Let us evaluate the retrieval of associations evoked by stimuli that are not originally encoded. Consider the consequences of a distortion of pattern $\mathbf{s}^{(j)}$ submitted at the memory input as $\mathbf{s}^{(j)'}$ so that

$$\mathbf{s}^{(j)'} = \mathbf{s}^{(j)} + \Delta^{(j)} \quad (6.12)$$

where the distortion term $\Delta^{(j)}$ can be assumed to be statistically independent of $\mathbf{s}^{(j)}$, and thus it can be considered as orthogonal to it. Substituting (6.12) into formula (6.10a), we obtain for orthonormal vectors originally encoded in the memory

$$\mathbf{v} = \mathbf{f}^{(j)} \mathbf{s}^{(j)t} \mathbf{s}^{(j)} + \mathbf{f}^{(j)} \mathbf{s}^{(j)t} \Delta^{(j)} + \sum_{i \neq j}^p (\mathbf{f}^{(i)} \mathbf{s}^{(i)t}) \Delta^{(j)} \quad (6.13a)$$

Due to the orthonormality condition this further reduces to

$$\mathbf{v} = \mathbf{f}^{(j)} + \sum_{i \neq j}^p (\mathbf{f}^{(i)} \mathbf{s}^{(i)t}) \Delta^{(j)} \quad (6.13b)$$

It can be seen that the memory response contains the desired association $\mathbf{f}^{(j)}$ and an additive component, which is due to the distortion term $\Delta^{(j)}$. The second term in the expression above has the meaning of *cross-talk noise* and is caused by the distortion of the input pattern and is present due to the vector $\Delta^{(j)}$. The term

contains, in parentheses, almost all elements of the memory cross-correlation matrix weighted by a distortion term $\Delta^{(j)}$. Therefore, even in the case of stored orthonormal patterns, the cross-talk noise term from all other patterns remains additive at the memory output to the originally stored association. We thus see that the linear associator provides no means for suppression of the cross-talk noise term is of limited use for accurate retrieval of the originally stored association.

Finally, let us notice an interesting property of the linear associator for the case of its autoassociative operation with p distinct n -dimensional prototype patterns $s^{(i)}$. In such a case the network can be called an *autocorrelator*. Plugging $f^{(i)} = s^{(i)}$ in (6.9b) results in the *autocorrelation matrix* \mathbf{W}' :

$$\mathbf{W}' = \sum_{i=1}^p s^{(i)} s^{(i)t} \quad (6.14a)$$

This result can also be expressed using the S matrix from (6.9c) as follows

$$\mathbf{W}' = \mathbf{SS}' \quad (6.14b)$$

The autocorrelation matrix of an autoassociator is of size $n \times n$. Note that this matrix can also be obtained directly from the Hebbian learning rule. Let us examine the attempted regeneration of a stored pattern in response to a distorted pattern $s^{(j)}$ submitted at the input of the linear autocorrelator. Assume again that input is expressed by (6.12). The output can be expressed using (6.10b), and it simplifies for orthonormal patterns $s^{(j)}$, for $j = 1, 2, \dots, p$, to the form

$$\mathbf{v} = s^{(j)} + \sum_{i \neq j}^p s^{(i)} s^{(i)t} \Delta^{(j)} \quad (6.15a)$$

This becomes equal

$$\mathbf{v} = s^{(j)} + (p - 1)\Delta^{(j)} \quad (6.15b)$$

As we can see, the cross-talk noise term again has not been eliminated even for stored orthogonal patterns. The retrieved output is the stored pattern plus the distortion term amplified $p - 1$ times. Therefore, linear associative memories perform rather poorly when retrieving associations due to distorted stimuli vectors.

Linear associator and autoassociator networks can also be used when linearly independent vectors $s^{(1)}, s^{(2)}, \dots, s^{(p)}$, are to be stored. The assumption of linear independence is weaker than the assumption of orthogonality and it allows for consideration of a larger class of vectors to be stored. As discussed by Kohonen (1977) and Kohonen et al. (1981), the weight matrix \mathbf{W} can be expressed for such a case as follows:

$$\mathbf{W} = \mathbf{F}(\mathbf{S}'\mathbf{S})^{-1}\mathbf{S}' \quad (6.16)$$

The weight matrix found from Equation (6.16) minimizes the squared output error between $f^{(j)}$ and $v^{(j)}$ in the case of linearly independent vectors $s^{(j)}$ (see Appendix). Because vectors to be used as stored memories are generally neither orthonormal nor linearly independent, the linear associator and autoassociator may not be efficient memories for many practical tasks.

Linear auto- and heteroassociative memory networks are in a certain limited sense similar to dynamic associative memories. In particular, both memories have similar recording algorithms. However, as shown, linear associators do not provide solutions for the suppression of the noise term in expressions (6.13b) and (6.15). We will show that other memory models utilize a thresholding operation to suppress the noise component present in an output signal. The recurrent autoassociative memory presented in the following section are examples of such memory models.

6.3

BASIC CONCEPTS OF RECURRENT AUTOASSOCIATIVE MEMORY

Discussion of the linear associative memory has pointed out the need for suppressing the output noise at the memory output. This can be done by thresholding the output and by recycling of the output to input in order to produce an improved association. The repetitive process of recycling of the output followed by a feedforward pass through the network can be performed by a recurrent neural network. Such a network is similar to the linear associative memory, however, it has feedback, nonlinear mapping in the form of thresholding, and is dynamical.

Recurrent autoassociative memory has already been introduced as an example of dynamic associative network in Figure 6.4. The reader should by now be familiar with the general theory of such a network. The memory is essentially a single-layer feedback network covered in Section 5.2 as a discrete-time network originally proposed by Hopfield (1982, 1984). In this section, a more detailed view of the model and its performance as an associative memory will be presented. Our focus is mainly on the dynamical performance of recurrent autoassociative memories. In contrast to continuously updating single-layer feedback networks discussed throughout Chapter 5, associative memories are updated in discrete time.

An expanded view of the Hopfield model network from Figure 6.4 is shown in Figure 6.6. Figure 6.6(a) depicts *Hopfield's autoassociative memory*. Under the asynchronous update mode, only one neuron is allowed to compute, or change state, at a time, and then all outputs are delayed by a time Δ produced by the unity delay element in the feedback loop. This symbolic delay allows for the time-stepping of the retrieval algorithm embedded in the update rule of (5.3) or (5.4). Figure 6.6(b) shows a simplified diagram of the network in the form that is often found in the technical literature. Note that the time step and the neurons' thresholding function have been suppressed on the figure. The computing neurons represented in the figure as circular nodes need to perform summation and bipolar thresholding and also need to introduce a unity delay. Note that the recurrent autoassociative memories studied in this chapter provide node responses

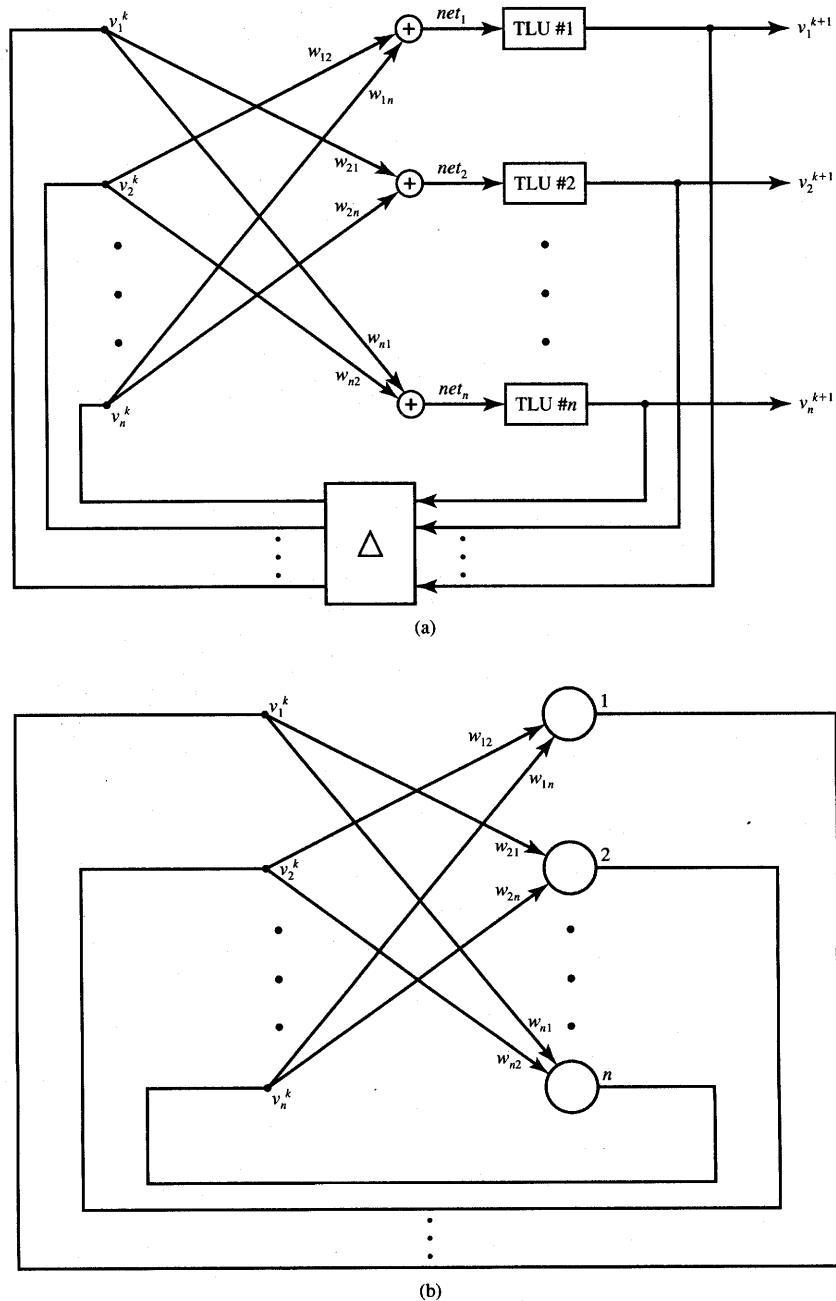


Figure 6.6 Hopfield model autoassociative memory (recurrent autoassociative memory):
 (a) expanded view and (b) simplified diagram.

of discrete values ± 1 . The domain of the n -tuple output vectors in \mathbb{R}^n are thus vertices of the n -dimensional cube $[-1, 1]$.

Retrieval Algorithm

Based on the discussion in Section 5.2 the output update rule for Hopfield autoassociative memory can be expressed in the form

$$v_i^{k+1} = \text{sgn} \left(\sum_{j=1}^n w_{ij} v_j^k \right) \quad (6.17)$$

where k is the index of recursion and i is the number of the neuron currently undergoing an update. The update rule (6.17) has been obtained from (5.4a) under the simplifying assumption that both the external bias i_i and threshold values T_i are zero for $i = 1, 2, \dots, n$. These assumptions will remain valid for the remainder of this chapter. In addition, the asynchronous update sequence considered here is random. Thus, assuming that recursion starts at v^0 , and a random sequence of updating neurons m, p, q, \dots is chosen, the output vectors obtained are as follows

$$\begin{aligned} \text{First update: } \quad & \mathbf{v}^1 = [v_1^0 \ v_2^0 \ \dots \ v_m^1 \ \dots \ v_p^0 \ \dots \ v_q^0 \ \dots \ v_n^0]^t \\ \text{Second update: } \quad & \mathbf{v}^2 = [v_1^0 \ v_2^0 \ \dots \ v_m^1 \ \dots \ v_p^2 \ \dots \ v_q^0 \ \dots \ v_n^0]^t \\ \text{Third update: } \quad & \mathbf{v}^3 = [v_1^0 \ v_2^0 \ \dots \ v_m^1 \ \dots \ v_p^2 \ \dots \ v_q^3 \ \dots \ v_n^0]^t \\ & \vdots \end{aligned} \quad (6.18)$$

Considerable insight into the Hopfield autoassociative memory performance can be gained by evaluating its respective energy function. The energy function (5.5) for the discussed memory network simplifies to

$$E(\mathbf{v}) = -\frac{1}{2} \mathbf{v}' \mathbf{W} \mathbf{v} \quad (6.19a)$$

We consider the memory network to evolve in a discrete-time mode, for $k = 1, 2, \dots$, and its outputs are one of the 2^n bipolar binary n -tuple vectors, each representing a vertex of the n -dimensional $[-1, +1]$ cube. We also discussed in Section 5.2 the fact that the asynchronous recurrent update never increases energy (6.19a) computed for $\mathbf{v} = \mathbf{v}^k$, and that the network settles in one of the local energy minima located at cube vertices.

We can now easily observe that the complement of a stored memory is also a stored memory. For the bipolar binary notation the complement vector of \mathbf{v} is equal to $-\mathbf{v}$. It is easy to see from (6.19a) that

$$E(-\mathbf{v}) = -\frac{1}{2} \mathbf{v}' \mathbf{W} \mathbf{v} \quad (6.19b)$$

and thus both energies $E(\mathbf{v})$ and $E(-\mathbf{v})$ are identical. Therefore, a minimum of $E(\mathbf{v})$ is of the same value as a minimum of $E(-\mathbf{v})$. This provides us with an important conclusion that the memory transitions may terminate as easily at \mathbf{v} as at $-\mathbf{v}$. The crucial factor determining the convergence is the “similarity” between the initializing output vector, and \mathbf{v} and $-\mathbf{v}$.

Storage Algorithm

Let us formulate the information storage algorithm for the recurrent autoassociative memory. Assume that the bipolar binary prototype vectors that need to be stored are $\mathbf{s}^{(m)}$, for $m = 1, 2, \dots, p$. The *storage algorithm* for calculating the weight matrix is

$$\mathbf{W} = \sum_{m=1}^p \mathbf{s}^{(m)} \mathbf{s}^{(m)T} - p \mathbf{I} \quad (6.20a)$$

or

$$w_{ij} = \left(1 - \delta_{ij}\right) \sum_{m=1}^p s_i^{(m)} s_j^{(m)} \quad (6.20b)$$

where, as before, δ_{ij} denotes the usual Kronecker function $\delta_{ij} = 1$ if $i = j$, and $\delta_{ij} = 0$ if $i \neq j$. The weight matrix \mathbf{W} is very similar to the autocorrelation matrix obtained using Hebb's learning rule for the linear associator introduced in (6.14). The difference is that now $w_{ii} = 0$. Note that the system does not remember the individual vectors $\mathbf{s}^{(m)}$ but only the weights w_{ij} , which basically represent correlation terms among the vector entries.

Also, the original Hebb's learning rule does not involve the presence of negative synaptic weight values, which can appear as a result of learning as in (6.20). This is a direct consequence of the condition that *only bipolar binary vectors $\mathbf{s}^{(m)}$ are allowed for building the autocorrelation matrix* in (6.20). Interestingly, additional autoassociations can be added at any time to the existing memory by superimposing new, incremental weight matrices. Autoassociations can also be removed by respective weight matrix subtraction. The storage rule (6.20) is also invariant with respect to the sequence of storing patterns.

The information storage algorithm for unipolar binary vectors $\mathbf{s}^{(m)}$, for $m = 1, 2, \dots, p$, needs to be modified so that a -1 component of the vectors simply replaces the 0 element in the original unipolar vector. This can be formally done by replacing the entries of the original unipolar vector $\mathbf{s}^{(m)}$ with the entries $2s_i^{(m)} - 1$, $i = 1, 2, \dots, n$. The memory storage algorithm (6.20b) for the unipolar binary vectors thus involves scaling and shifting and takes the form

$$w_{ij} = \left(1 - \delta_{ij}\right) \sum_{m=1}^p (2s_i^{(m)} - 1)(2s_j^{(m)} - 1) \quad (6.21)$$

Notice that the information storage rule is invariant under the binary complement