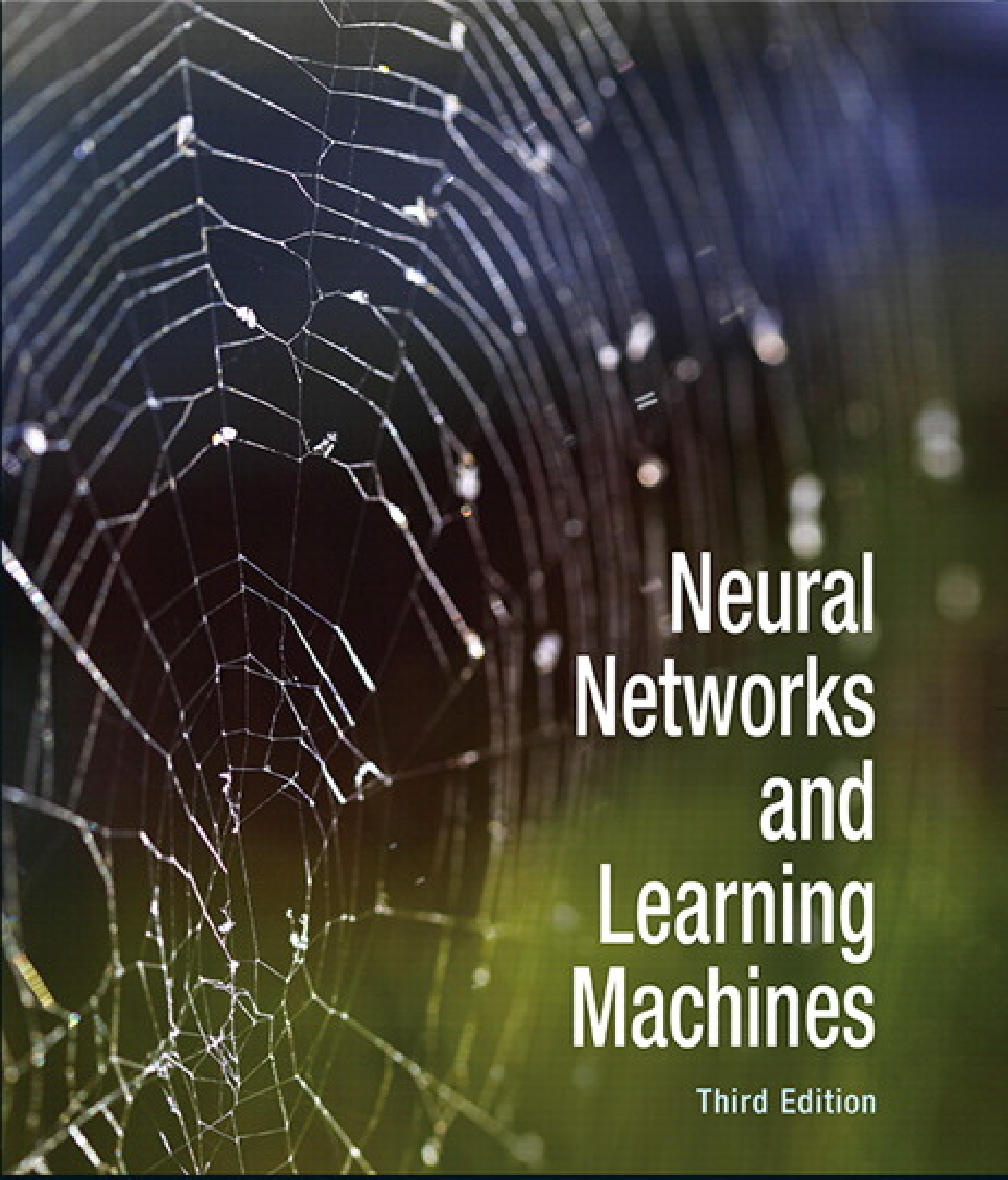


Neural Networks and Learning Machines

Third Edition

Simon Haykin

Neural Networks and Learning Machines

Third Edition

Simon Haykin
McMaster University
Hamilton, Ontario, Canada

New York Boston San Francisco
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

Library of Congress Cataloging-in-Publication Data
Haykin, Simon
Neural networks and learning machines / Simon Haykin.—3rd ed.
p. cm.
Rev. ed of: Neural networks. 2nd ed., 1999.
Includes bibliographical references and index.
ISBN-13: 978-0-13-147139-9
ISBN-10: 0-13-147139-2
1. Neural networks (Computer science) 2. Adaptive filters. I. Haykin, Simon Neural networks. II. Title.
QA76.87.H39 2008
006.3′--dc22
2008034079

Vice President and Editorial Director, ECS: Marcia J. Horton
Associate Editor: Alice Dworkin
Supervisor/Editorial Assistant: Dolores Mars
Editorial Assistant: William Opaluch
Director of Team-Based Project Management: Vince O'Brien
Senior Managing Editor: Scott Disanno
A/V Production Editor: Greg Dulles
Art Director: Jayne Conte
Cover Designer: Bruce Kenselaar
Manufacturing Manager: Alan Fischer
Manufacturing Buyer: Lisa McDowell
Marketing Manager: Tim Galligan

---

10 9 8 7 6 5 4 3 2 1
ISBN-13: 978-0-13-147139-9
ISBN-10: 0-13-147139-2

To my wife, Nancy, for her patience and tolerance,

and

to the countless researchers in neural networks for their original contributions, the many reviewers for their critical inputs, and many of my graduate students for their keen interest.

This page intentionally left blank

Contents

MATLAB codes + solutions to Computer Experiments

Preface

In writing this third edition of a classic book, I have been guided by the same underlying philosophy of the first edition of the book:

> *Write an up-to-date treatment of neural networks in a comprehensive, thorough, and readable manner.*

The new edition has been retitled *Neural Networks and Learning Machines*, in order to reflect two realities:

1. The perceptron, the multilayer perceptron, self-organizing maps, and neurodynamics, to name a few topics, have always been considered integral parts of neural networks, rooted in ideas inspired by the human brain.
2. Kernel methods, exemplified by support-vector machines and kernel principal-components analysis, are rooted in statistical learning theory.

Although, indeed, they share many fundamental concepts and applications, there are some subtle differences between the operations of neural networks and learning machines. The underlying subject matter is therefore much richer when they are studied together, under one umbrella, particularly so when

- ideas drawn from neural networks and machine learning are hybridized to perform improved learning tasks beyond the capability of either one operating on its own, and
- ideas inspired by the human brain lead to new perspectives wherever they are of particular importance.

Moreover, the scope of the book has been broadened to provide detailed treatments of dynamic programming and sequential state estimation, both of which have affected the study of reinforcement learning and supervised learning, respectively, in significant ways.

Organization of the Book

The book begins with an introductory chapter that is motivational, paving the way for the rest of the book which is organized into six parts as follows:

1. Chapters 1 through 4, constituting the first part of the book, follow the classical approach on supervised learning. Specifically,

- Chapter 1 describes Rosenblatt's perceptron, highlighting the perceptron convergence theorem, and the relationship between the perceptron and the Bayesian classifier operating in a Gaussian environment.
- Chapter 2 describes the method of least squares as a basis for model building. The relationship between this method and Bayesian inference for the special case of a Gaussian environment is established. This chapter also includes a discussion of the minimum description length (MDL) principle for model selection.
- Chapter 3 is devoted to the least-mean-square (LMS) algorithm and its convergence analysis. The theoretical framework of the analysis exploits two principles: Kushner's direct method and the Langevin equation (well known in nonequilibrium thermodynamics).

These three chapters, though different in conceptual terms, share a common feature: They are all based on a single computational unit. Most importantly, they provide a great deal of insight into the learning process in their own individual ways—a feature that is exploited in subsequent chapters.

Chapter 4, on the multilayer perceptron, is a generalization of Rosenblatt's perceptron. This rather long chapter covers the following topics:

- the back-propagation algorithm, its virtues and limitations, and its role as an optimum method for computing partial derivations;
- optimal annealing and adaptive control of the learning rate;
- cross-validation;
- convolutional networks, inspired by the pioneering work of Hubel and Wiesel on visual systems;
- supervised learning viewed as an optimization problem, with attention focused on conjugate-gradient methods, quasi-Newton methods, and the Marquardt–Levenberg algorithm;
- nonlinear filtering;
- last, but by no means least, a contrasting discussion of small-scale versus large-scale learning problems.

2. The next part of the book, consisting of Chapters 5 and 6, discusses kernel methods based on radial-basis function (RBF) networks.

In a way, Chapter 5 may be viewed as an insightful introduction to kernel methods. Specifically, it does the following:

- presents Cover's theorem as theoretical justification for the architectural structure of RBF networks;
- describes a relatively simple two-stage hybrid procedure for supervised learning, with stage 1 based on the idea of clustering (namely, the K-means algorithm) for computing the hidden layer, and stage 2 using the LMS or the method of least squares for computing the linear output layer of the network;
- presents kernel regression and examines its relation to RBF networks.

Chapter 6 is devoted to support vector machines (SVMs), which are commonly recognized as a method of choice for supervised learning. Basically, the SVM is a binary classifier, in the context of which the chapter covers the following topics:

- the condition for defining the maximum margin of separation between a pair of linearly separable binary classes;
- quadratic optimization for finding the optimal hyperplane when the two classes are linearly separable and when they are not;
- the SVM viewed as a kernel machine, including discussions of the kernel trick and Mercer's theorem;
- the design philosophy of SVMs;
- the ϵ-insensitive loss function and its role in the optimization of regression problems;
- the Representer Theorem, and the roles of Hilbert space and reproducing kernel Hilbert space (RKHS) in its formulation.

From this description, it is apparent that the underlying theory of support vector machines is built on a strong mathematical background—hence their computational strength as an elegant and powerful tool for supervised learning.

3. The third part of the book involves a single chapter, Chapter 7. This broadly based chapter is devoted to regularization theory, which is at the core of machine learning. The following topics are studied in detail:

- Tikhonov's classic regularization theory, which builds on the RKHS discussed in Chapter 6. This theory embodies some profound mathematical concepts: the Fréchet differential of the Tikhonov functional, the Riesz representation theorem, the Euler–Lagrange equation, Green's function, and multivariate Gaussian functions;
- generalized RBF networks and their modification for computational tractability;
- the regularized least-squares estimator, revisited in light of the Representer Theorem;
- estimation of the regularization parameter, using Wahba's concept of generalized cross-validation;
- semisupervised learning, using labeled as well as unlabeled examples;
- differentiable manifolds and their role in manifold regularization—a role that is basic to designing semisupervised learning machines;
- spectral graph theory for finding a Gaussian kernel in an RBF network used for semisupervised learning;
- a generalized Representer Theorem for dealing with semisupervised kernel machines;
- the Laplacian regularized least-squares (LapRLS) algorithm for computing the linear output layer of the RBF network; here, it should be noted that when the intrinsic regularization parameter (responsible for the unlabeled data) is reduced to zero, the algorithm is correspondingly reduced to the ordinary least-squares algorithm.

This highly theoretical chapter is of profound practical importance. First, it provides the basis for the regularization of supervised-learning machines. Second, it lays down the groundwork for designing regularized semisupervised learning machines.

4. Chapters 8 through 11 constitute the fourth part of the book, dealing with unsupervised learning. Beginning with Chapter 8, four principles of self-organization, intuitively motivated by neurobiological considerations, are presented:

(i) Hebb's postulate of learning for self-amplification;
(ii) Competition among the synapses of a single neuron or a group of neurons for limited resources;
(iii) Cooperation among the winning neuron and its neighbors;
(iv) Structural information (e.g., redundancy) contained in the input data.

The main theme of the chapter is threefold:

- Principles (i), (ii), and (iv) are applied to a single neuron, in the course of which Oja's rule for maximum eigenfiltering is derived; this is a remarkable result obtained through self-organization, which involves bottom-up as well as top-down learning. Next, the idea of maximum eigenfiltering is generalized to principal-components analysis (PCA) on the input data for the purpose of dimensionality reduction; the resulting algorithm is called the generalized Hebbian algorithm (GHA).
- Basically, PCA is a linear method, the computing power of which is therefore limited to second-order statistics. In order to deal with higher-order statistics, the kernel method is applied to PCA in a manner similar to that described in Chapter 6 on support vector machines, but with one basic difference: unlike SVM, kernel PCA is performed in an unsupervised manner.
- Unfortunately, in dealing with natural images, kernel PCA can become unmanageable in computational terms. To overcome this computational limitation, GHA and kernel PCA are hybridized into a new on-line unsupervised learning algorithm called the kernel Hebbian algorithm (KHA), which finds applications in image denoising.

The development of KHA is an outstanding example of what can be accomplished when an idea from machine learning is combined with a complementary idea rooted in neural networks, producing a new algorithm that overcomes their respective practical limitations.

Chapter 9 is devoted to self-organizing maps (SOMs), the development of which follows the principles of self-organization described in Chapter 8. The SOM is a simple algorithm in computational terms, yet highly powerful in its built-in ability to construct organized topographic maps with several useful properties:

- spatially discrete approximation of the input space, responsible for data generation;
- topological ordering, in the sense that the spatial location of a neuron in the topographic map corresponds to a particular feature in the input (data) space;
- input–output density matching;
- input-data feature selection.

The SOM has been applied extensively in practice; the construction of contextual maps and hierarchical vector quantization are presented as two illustrative examples of the SOM's computing power. What is truly amazing is that the SOM exhibits several interesting properties and solves difficult computational tasks, yet it lacks an objective function that could be optimized. To fill this gap and thereby provide the possibility of improved topographic mapping, the self-organizing map is kernelized. This is done by introducing an entropic function as the objective

function to be maximized. Here again, we see the practical benefit of hybridizing ideas rooted in neural networks with complementary kernel-theoretic ones.

Chapter 10 exploits principles rooted in Shannon's information theory as tools for unsupervised learning. This rather long chapter begins by presenting a review of Shannon's information theory, with particular attention given to the concepts of entropy, mutual information, and the Kullback–Leibler divergence (KLD). The review also includes the concept of copulas, which, unfortunately, has been largely overlooked for several decades. Most importantly, the copula provides a measure of the statistical dependence between a pair of correlated random variables. In any event, focusing on mutual information as the objective function, the chapter establishes the following principles:

- The Infomax principle, which maximizes the mutual information between the input and output data of a neural system; Infomax is closely related to redundancy reduction.
- The Imax principle, which maximizes the mutual information between the single outputs of a pair of neural systems that are driven by correlated inputs.
- The Imin principle operates in a manner similar to the Imax principle, except that the mutual information between the pair of output random variables is minimized.
- The independent-components analysis (ICA) principle, which provides a powerful tool for the blind separation of a hidden set of statistically independent source signals. Provided that certain operating conditions are satisfied, the ICA principle affords the basis for deriving procedures for recovering the original source signals from a corresponding set of observables that are linearly mixed versions of the source signals. Two specific ICA algorithms are described:

  (i) the natural-gradient learning algorithm, which, except for scaling and permutation, solves the ICA problem by minimizing the KLD between a parameterized probability density function and the corresponding factorial distribution;

  (ii) the maximum-entropy learning algorithm, which maximizes the entropy of a nonlinearly transformed version of the demixer output; this algorithm, commonly known as the Infomax algorithm for ICA, also exhibits scaling and permutation properties.

Chapter 10 also describes another important ICA algorithm, known as FastICA, which, as the name implies, is computationally fast. This algorithm maximizes a contrast function based on the concept of negentropy, which provides a measure of the non-Gaussianity of a random variable. Continuing with ICA, the chapter goes on to describe a new algorithm known as coherent ICA, the development of which rests on fusion of the Infomax and Imax principles via the use of the copula; coherent ICA is useful for extracting the envelopes of a mixture of amplitude-modulated signals. Finally, Chapter 10 introduces another concept rooted in Shannon's information theory, namely, rate distortion theory, which is used to develop the last concept in the chapter: information bottleneck. Given the joint distribution of an input vector and a (relevant) output vector, the method is formulated as a constrained

optimization problem in such a way that a tradeoff is created between two amounts of information, one pertaining to information contained in the bottleneck vector about the input and the other pertaining to information contained in the bottleneck vector about the output. The chapter then goes on to find an optimal manifold for data representation, using the information bottleneck method.

The final approach to unsupervised learning is described in Chapter 11, using stochastic methods that are rooted in statistical mechanics; the study of statistical mechanics is closely related to information theory. The chapter begins by reviewing the fundamental concepts of Helmholtz free energy and entropy (in a statistical mechanics sense), followed by the description of Markov chains. The stage is then set for describing the Metropolis algorithm for generating a Markov chain, the transition probabilities of which converge to a unique and stable distribution. The discussion of stochastic methods is completed by describing simulated annealing for global optimization, followed by Gibbs sampling, which can be used as a special form of the Metropolis algorithm. With all this background on statistical mechanics at hand, the stage is set for describing the Boltzmann machine, which, in a historical context, was the first multilayer learning machine discussed in the literature. Unfortunately, the learning process in the Boltzmann machine is very slow, particularly when the number of hidden neurons is large—hence the lack of interest in its practical use. Various methods have been proposed in the literature to overcome the limitations of the Boltzmann machine. The most successful innovation to date is the deep belief net, which distinguishes itself in the clever way in which the following two functions are combined into a powerful machine:

- generative modeling, resulting from bottom-up learning on a layer-by-layer basis and without supervision;
- inference, resulting from top-down learning.

Finally, Chapter 10 describes deterministic annealing to overcome the excessive computational requirements of simulated annealing; the only problem with deterministic annealing is that it could get trapped in a local minimum.

5. Up to this point, the focus of attention in the book has been the formulation of algorithms for supervised learning, semisupervised learning, and unsupervised learning. Chapter 12, constituting the next part of the book all by itself, addresses reinforcement learning, in which learning takes place in an on-line manner as the result of an agent (e.g., robot) interacting with its surrounding environment. In reality, however, dynamic programming lies at the core of reinforcement learning. Accordingly, the early part of Chapter 15 is devoted to an introductory treatment of Bellman's dynamic programming, which is then followed by showing that the two widely used methods of reinforcement learning: Temporal difference (TD) learning, and Q-learning can be derived as special cases of dynamic programming. Both TD-learning and Q-learning are relatively simple, on-line reinforcement learning algorithms that do not require knowledge of transition probabilities. However, their practical applications are limited to situations in which the dimensionality of the state space is of moderate size. In large-scale dynamic systems, the curse of dimensionality becomes a serious issue, making not only dynamic programming,

but also its approximate forms, TD-learning and Q-learning, computationally intractable. To overcome this serious limitation, two indirect methods of approximate dynamic programming are described:

- a linear method called the least-squares policy evaluation (LSPV) algorithm, and
- a nonlinear method using a neural network (e.g., multilayer perceptron) as a universal approximator.

6. The last part of the book, consisting of Chapters 13, 14, and 15, is devoted to the study of nonlinear feedback systems, with an emphasis on recurrent neural networks:

(i) Chapter 13 studies neurodynamics, with particular attention given to the stability problem. In this context, the direct method of Lyapunov is described. This method embodies two theorems, one dealing with stability of the system and the other dealing with asymptotic stability. At the heart of the method is a Lyapunov function, for which an energy function is usually found to be adequate. With this background theory at hand, two kinds of associative memory are described:

- the Hopfield model, the operation of which demonstrates that a complex system is capable of generating simple emergent behavior;
- the brain-state-in-a-box model, which provides a basis for clustering.

The chapter also discusses properties of chaotic processes and a regularized procedure for their dynamic reconstruction.

(ii) Chapter 14 is devoted to the Bayesian filter, which provides a unifying basis for sequential state estimation algorithms, at least in a conceptual sense. The findings of the chapter are summarized as follows:

- The classic Kalman filter for a linear Gaussian environment is derived with the use of the minimum mean-square-error criterion; in a problem at the end of the chapter, it is shown that the Kalman filter so derived is a special case of the Bayesian filter;
- square-root filtering is used to overcome the divergence phenomenon that can arise in practical applications of the Kalman filter;
- the extended Kalman filter (EKF) is used to deal with dynamic systems whose nonlinearity is of a mild sort; the Gaussian assumption is maintained;
- the direct approximate form of the Bayesian filter is exemplified by a new filter called the cubature Kalman filter (CKF); here again, the Gaussian assumption is maintained;
- indirect approximate forms of the Bayesian filter are exemplified by particle filters, the implementation of which can accommodate nonlinearity as well as non-Gaussianity.

With the essence of Kalman filtering being that of a predictor–corrector, Chapter 14 goes on to describe the possible role of "Kalman-like filtering" in certain parts of the human brain.

The final chapter of the book, Chapter 15, studies dynamically driven recurrent neural networks. The early part of the chapter discusses different structures (models) for recurrent networks and their computing power, followed by two algorithms for the training of recurrent networks:

- back propagation through time, and
- real-time recurrent learning.

Unfortunately both of these procedures, being gradient based, are likely to suffer from the so-called vanishing-gradients problem. To mitigate the problem, the use of nonlinear sequential state estimators is described at some length for the supervised training of recurrent networks in a rather novel manner. In this context, the advantages and disadvantages of the extended Kalman filter (simple, but derivative dependent) and the cubature Kalman filter (derivative free, but more complicated mathematically) as sequential state estimator for supervised learning are discussed. The emergence of adaptive behavior, unique to recurrent networks, and the potential benefit of using an adaptive critic to further enhance the capability of recurrent networks are also discussed in the chapter.

An important topic featuring prominently in different parts of the book is supervised learning and semisupervised learning applied to large-scale problems. The concluding remarks of the book assert that this topic is in its early stages of development; most importantly, a four-stage procedure is described for its future development.

Distinct Features of the Book

Over and above the broad scope and thorough treatment of the topics summarized under the organization of the book, distinctive features of the text include the following:

1. Chapters 1 through 7 and Chapter 10 include computer experiments involving the double-moon configuration for generating data for the purpose of binary classification. The experiments range from the simple case of linearly separable patterns to difficult cases of nonseparable patterns. The double-moon configuration, as a running example, is used all the way from Chapter 1 to Chapter 7, followed by Chapter 10, thereby providing an experimental means for studying and comparing the learning algorithms described in those eight chapters.
2. Computer experiments are also included in Chapter 8 on PCA, Chapter 9 on SOM and kernel SOM, and Chapter 14 on dynamic reconstruction of the Mackay–Glass attractor using the EKF and CKF algorithms.
3. Several case studies, using real-life data, are presented:
   - Chapter 7 discusses the United States Postal Service (USPS) data for semisupervised learning using the Laplacian RLS algorithm;
   - Chapter 8 examines how PCA is applied to handwritten digital data and describes the coding and denoising of images;
   - Chapter 10 treats the analysis of natural images by using sparse-sensory coding and ICA;
   - Chapter 13 presents dynamic reconstruction applied to the Lorenz attractor by using a regularized RBF network.

Chapter 15 also includes a section on the model reference adaptive control system as a case study.

4. Each chapter ends with notes and references for further study, followed by end-of-chapter problems that are designed to challenge, and therefore expand, the reader's expertise.

   The glossary at the front of the book has been expanded to include explanatory notes on the methodology used on matters dealing with matrix analysis and probability theory.

5. PowerPoint files of all the figures and tables in the book will be available to Instructors and can be found at www.prenhall.com/haykin.

6. Matlab codes for all the computer experiments in the book are available on the Website of the publisher to all those who have purchased copies of the book. These are available to students at www.pearsonhighered.com/haykin.

7. The book is accompanied by a Manual that includes the solutions to all the end-of-chapter problems as well as computer experiments.

   The manual is available from the publisher, Prentice Hall, only to instructors who use the book as the recommended volume for a course, based on the material covered in the book.

Last, but by no means least, every effort has been expended to make the book error free and, most importantly, readable.

Simon Haykin
Ancaster, Ontario

This page intentionally left blank

Acknowledgments

I am deeply indebted to many renowned authorities on neural networks and learning machines around the world, who have provided invaluable comments on selected parts of the book:

Dr. Sun-Ichi Amari, The RIKEN Brain Science Institute, Wako City, Japan

Dr. Susanne Becker, Department of Psychology, Neuroscience & Behaviour, McMaster University, Hamilton, Ontario, Canada

Dr. Dimitri Bertsekas, MIT, Cambridge, Massachusetts

Dr. Leon Bottou, NEC Laboratories America, Princeton, New Jersey

Dr. Simon Godsill, University of Cambridge, Cambridge, England

Dr. Geoffrey Gordon, Carnegie-Mellon University, Pittsburgh, Pennsylvania

Dr. Peter Grünwald, CWI, Amsterdam, the Netherlands

Dr. Geoffrey Hinton, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada

Dr. Timo Honkela, Helsinki University of Technology, Helsinki, Finland

Dr. Tom Hurd, Department of Mathematics and Statistics, McMaster University, Ontario, Canada.

Dr. Eugene Izhikevich, The Neurosciences Institute, San Diego, California

Dr. Juha Karhunen, Helsinki University of Technology, Helsinki, Finland

Dr. Kwang In Kim, Max-Planck-Institut für Biologische Kybernetik, Tübingen, Germany

Dr. James Lo, University of Maryland at Baltimore County, Baltimore, Maryland

Dr. Klaus Müller, University of Potsdam and Fraunhofer Institut FIRST, Berlin, Germany

Dr. Erkki Oja, Helsinki University of Technology, Helsinki, Finland

Dr. Bruno Olshausen, Redwood Center for Theoretical Neuroscience, University of California, Berkeley, California

Dr. Danil Prokhorov, Toyota Technical Center, Ann Arbor, Michigan

Dr. Kenneth Rose, Electrical and Computer Engineering, University of California, Santa Barbara, California

Dr. Bernhard Schölkopf, Max-Planck-Institut für Biologische Kybernetik, Tübingen, Germany

Dr. Vikas Sindhwani, Department of Computer Science, University of Chicago, Chicago, Illinois

Dr. Sergios Theodoridis, Department of Informatics, University of Athens, Athens, Greece

Dr. Naftali Tishby, The Hebrew University, Jerusalem, Israel

Dr. John Tsitsiklis, Massachusetts Institute of Technology, Cambridge, Massachusetts

Dr. Marc Van Hulle, Katholieke Universiteit, Leuven, Belgium

Several photographs and graphs have been reproduced in the book with permissions provided by Oxford University Press and

Dr. Anthony Bell, Redwood Center for Theoretical Neuroscience, University of California, Berkeley, California

Dr. Leon Bottou, NEC Laboratories America, Princeton, New Jersey

Dr. Juha Karhunen, Helsinki University of Technology, Helsinki, Finland

Dr. Bruno Olshausen, Redwood Center for Theoretical Neuroscience, University of California, Berkeley, California

Dr. Vikas Sindhwani, Department of Computer Science, University of Chicago, Chicago, Illinois

Dr. Naftali Tishby, The Hebrew University, Jerusalem, Israel

Dr. Marc Van Hulle, Katholieke Universiteit, Leuven, Belgium

I thank them all most sincerely.

I am grateful to my graduate students:

1. Yanbo Xue, for his tremendous effort devoted to working on nearly all the computer experiments produced in the book, and also for reading the second page proofs of the book.
2. Karl Wiklund, for proofreading the entire book and making valuable comments for improving it.
3. Haran Arasaratnam, for working on the computer experiment dealing with the Mackay–Glass attractor.
4. Andreas Wendel (Graz University of technology, Austria) while he was on leave at McMaster University, 2008.

I am grateful to Scott Disanno and Alice Dworkin of Prentice Hall for their support and hard work in the production of the book. Authorization of the use of color in the book by Marcia Horton is truly appreciated; the use of color has made a tremendous difference to the appearance of the book from cover to cover.

I am grateful to Jackie Henry of Aptara Corp. and her staff, including Donald E. Smith, Jr., the proofreader, for the production of the book. I also wish to thank Brian Baker and the copyeditor, Abigail Lin, at Write With, Inc., for their effort in copy-editing the manuscript of the book.

The tremendous effort by my Technical Coordinator, Lola Brooks, in typing several versions of the chapters in the book over the course of 12 months, almost nonstop, is gratefully acknowledged.

Last, but by no means least, I thank my wife, Nancy, for having allowed me the time and space, which I have needed over the last 12 months, almost nonstop, to complete the book in a timely fashion.

Simon Haykin

Abbreviations and Symbols

ABBREVIATIONS

AR	autoregressive
BBTT	back propagation through time
BM	Boltzmann machine
BP	back propagation
b/s	bits per second
BSB	brain-state-in-a-box
BSS	Blind source (signal) separation
cmm	correlation matrix memory
CV	cross-validation
DFA	deterministic finite-state automata
EKF	extended Kalman filter
EM	expectation-maximization
FIR	finite-duration impulse response
FM	frequency-modulated (signal)
GCV	generalized cross-validation
GHA	generalized Hebbian algorithm
GSLC	generalized sidelobe canceler
Hz	hertz
ICA	independent-components analysis
Infomax	maximum mutual information
Imax	variant of Infomax
Imin	another variant of Infomax
KSOM	kernel self-organizing map
KHA	kernel Hebbian algorithm
LMS	least-mean-square
LR	likelihood ratio

LS	Least-squares
LS-TD	Least-squares, temporal-difference
LTP	long-term potentiation
LTD	long-term depression
LR	likelihood ratio
LRT	Likelihood ratio test
MAP	Maximum a posteriori
MCA	minor-components analysis
MCMC	Markov Chan Monte Carlo
MDL	minimum description length
MIMO	multiple input–multiple output
ML	maximum likelihood
MLP	multilayer perceptron
MRC	model reference control
NARMA	nonlinear autoregressive moving average
NARX	nonlinear autoregressive with exogenous inputs
NDP	neuro-dynamic programming
NW	Nadaraya–Watson (estimator)
NWKR	Nadaraya–Watson kernal regression
OBD	optimal brain damage
OBS	optimal brain surgeon
OCR	optical character recognition
PAC	probably approximately correct
PCA	principal-components analysis
PF	Particle Filter
pdf	probability density function
pmf	probability mass function
QP	quadratic programming
RBF	radial basis function
RLS	recursive least-squares
RLS	regularized least-squares
RMLP	recurrent multilayer perceptron
RTRL	real-time recurrent learning
SIMO	single input–multiple output
SIR	sequential importance resampling
SIS	sequential important sampling
SISO	single input–single output
SNR	signal-to-noise ratio
SOM	self-organizing map
SRN	simple recurrent network (also referred to as Elman's recurrent network)

SVD	singular value decomposition
SVM	support vector machine
TD	temporal difference
TDNN	time-delay neural network
TLFN	time-lagged feedforward network
VC	Vapnik–Chervononkis (dimension)
VLSI	very-large-scale integration
XOR	exclusive OR

IMPORTANT SYMBOLS

a	action
$\mathbf{a}^T\mathbf{b}$	inner product of vectors **a** and **b**
$\mathbf{a}\mathbf{b}^T$	outer product of vectors **a** and **b**
$\binom{l}{m}$	binomial coefficient
$A \cup B$	unions of A and B
B	inverse of temperature
b_k	bias applied to neuron k
$\cos(\mathbf{a},\mathbf{b})$	cosine of the angle between vectors **a** and **b**
$c_{u,v}(u, v)$	probability density function of copula
D	depth of memory
$D_{f\|\|g}$	Kullback–Leibler divergence between probability density functions f and g
$\tilde{\mathbf{D}}$	adjoint of operator **D**
E	energy function
E_i	energy of state i in statistical mechanics
$\mathbb{E}$	statistical expectation operator
$\langle E \rangle$	average energy
exp	exponential
$\mathscr{E}_{\text{av}}$	average squared error, or sum of squared errors
$\mathscr{E}(n)$	instantaneous value of the sum of squared errors
$\mathscr{E}_{\text{total}}$	total sum of error squares
F	free energy
$\mathscr{F}^*$	subset (network) with minimum empirical risk
$\mathbf{H}$	Hessian (matrix)
$\mathbf{H}^{-1}$	inverse of Hessian **H**
i	square root of -1, also denoted by j
$\mathbf{I}$	identity matrix
$\mathbf{I}$	Fisher's information matrix
J	mean-square error
$\mathbf{J}$	Jacobian (matrix)

$\mathbf{P}^{1/2}$	square root of matrix $\mathbf{P}$
$\mathbf{P}^{T/2}$	transpose of square root of matrix $\mathbf{P}$
$\mathbf{P}n,n-1$	error covariance matrix in Kalman filter theory
k_{B}	Boltzmann constant
log	logarithm
$L(\mathbf{w})$	log-likelihood function of weight vector $\mathbf{w}$
$\mathscr{L}(\mathbf{w})$	log-likelihood function of weight vector $\mathbf{w}$ based on a single example
$\mathbf{M}_{\mathrm{c}}$	controllability matrix
$\mathbf{M}_o$	observability matrix
n	discrete time
p_i	probability of state i in statistical mechanics
p_{ij}	transition probability from state i to state j
$\mathbf{P}$	stochastic matrix
$P(e\|\mathscr{C})$	conditional probability of error e given that the input is drawn from class $\mathscr{C}$
P_{α}^{+}	probability that the visible neurons of a Boltzmann machine are in state α , given that the network is in its clamped condition (i.e., positive phase)
P_{α}^{-}	probability that the visible neurons of a Boltzmann machine are in state α, given that the network is in its free-running condition (i.e., negative phase)
$\hat{r}_x(j, k;n)]$	estimate of autocorrelation function of $\mathrm{x}_j(n)$ and $x_k(n)$
$\hat{r}_{dx}(k;n)$	estimate of cross-correlation function of $d(n)$ and $x_k(n)$
$\mathbf{R}$	correlation matrix of an input vector
t	continuous time
T	temperature
$\mathscr{T}$	training set (sample)
tr	operator denoting the trace of a matrix
var	variance operator
$V(\mathbf{x})$	Lyapunov function of state vector $\mathbf{x}$
v_j	induced local field or activation potential of neuron j
$\mathbf{w}_o$	optimum value of synaptic weight vector
w_{kj}	weight of synapse j belonging to neuron k
$\mathbf{w}^*$	optimum weight vector
$\bar{\mathbf{x}}$	equilibrium value of state vector $\mathbf{x}$
$\langle x_j \rangle$	average of state x_j in a "thermal" sense
$\hat{x}$	estimate of x, signified by the use of a caret (hat)
$\|x\|$	absolute value (magnitude) of x
x^*	complex conjugate of x, signified by asterisk as superscript
$\|\|\mathbf{x}\|\|$	Euclidean norm (length) of vector $\mathbf{x}$
$\mathbf{x}^T$	transpose of vector $\mathbf{x}$, signified by the superscript T
z^{-1}	unit-time delay operator
Z	partition function
$\delta_j(n)$	local gradient of neuron j at time n
Δw	small change applied to weight w
∇	gradient operator

∇^2	Laplacian operator
$\nabla_w J$	gradient of J with respect to w
$\boldsymbol{\nabla} \cdot \mathbf{F}$	divergence of vector $\mathbf{F}$
η	learning-rate parameter
κ	cumulant
μ	policy
θ_k	threshold applied to neuron k (i.e., negative of bias b_k)
λ	regularization parameter
λ_k	kth eigenvalue of a square matrix
$\varphi_k(\cdot)$	nonlinear activation function of neuron k
$\in$	symbol for "belongs to"
$\cup$	symbol for "union of"
$\cap$	symbol for "intersection of"
$*$	symbol for convolution
$+$	superscript symbol for pseudoinverse of a matrix
$+$	superscript symbol for updated estimate

Open and closed intervals

- The open interval (a, b) of a variable x signifies that $a < x < b$.
- The closed interval $[a, b]$ of a variable x signifies that $a \leq \mathrm{x} \leq b$.
- The closed-open interval $[a, b)$ of a variable x signifies that $a \leq x < b$; likewise for the open-closed interval $(a, b]$, $a < x \leq b$.

Minima and Maxima

- The symbol $\arg\min_{\mathbf{w}} f(\mathbf{w})$ signifies the minimum of the function $f(\mathbf{w})$ with respect to the argument vector $\mathbf{w}$.
- The symbol $\arg\max_{\mathbf{w}} f(\mathbf{w})$ signifies the maximum of the function $f(\mathbf{w})$ with respect to the argument vector $\mathbf{w}$.

This page intentionally left blank

GLOSSARY

NOTATIONS I: MATRIX ANALYSIS

Scalars: Italic lowercase symbols are used for scalars.

Vectors: Bold lowercase symbols are used for vectors.

A vector is defined as a *column* of scalars. Thus, the *inner product* of a pair of m-dimensional vectors, $\mathbf{x}$ and $\mathbf{y}$, is written as

$$\mathbf{x}^T\mathbf{y} = [x_1, x_2, ..., x_m]\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \sum_{i=1}^{m} x_i y_i$$

where the superscript T denotes *matrix transposition*. With the inner product being a scalar, we therefore have

$$\mathbf{y}^T\mathbf{x} = \mathbf{x}^T\mathbf{y}$$

Matrices: Bold uppercase symbols are used for matrices.

Matrix multiplication is carried out on a *row multiplied by column basis*. To illustrate, consider an m-by-k matrix $\mathbf{X}$ and a k-by-l matrix $\mathbf{Y}$. The product of these two matrices yields the m-by-l matrix

$$\mathbf{Z} = \mathbf{XY}$$

More specifically, the ij-th component of matrix $\mathbf{Z}$ is obtained by multiplying the ith row of matrix $\mathbf{X}$ by the jth column of matrix $\mathbf{Y}$, both of which are made up of k scalars.

The outer product of a pair of m-dimensional vectors, $\mathbf{x}$ and $\mathbf{y}$, is written as $\mathbf{x}\mathbf{y}^T$, which is an m-by-m matrix.

NOTATIONS II: PROBABILITY THEORY

Random variables: Italic uppercase symbols are used for random variables. The sample value (i.e., one-shot realization) of a random variable is denoted by the corresponding

italic lowercase symbol. For example, we write X for a random variable and x for its sample value.

Random vectors: Bold uppercase symbols are used for random vectors. Similarly, the sample value of a random vector is denoted by the corresponding bold lowercase symbol. For example, we write $\mathbf{X}$ for a random vector and $\mathbf{x}$ for its sample value.

The *probability density function* (pdf) of a random variable $\mathbf{X}$ is thus denoted by $p_{\mathbf{X}}(\mathbf{x})$, which is a function of the sample value $\mathbf{x}$; the subscript $\mathbf{X}$ is included as a reminder that the pdf pertains to random vector $\mathbf{X}$.

Introduction

1 WHAT IS A NEURAL NETWORK?

Work on artificial neural networks, commonly referred to as "neural networks," has been motivated right from its inception by the recognition that the human brain computes in an entirely different way from the conventional digital computer. The brain is a highly *complex, nonlinear, and parallel computer* (information-processing system). It has the capability to organize its structural constituents, known as *neurons*, so as to perform certain computations (e.g., pattern recognition, perception, and motor control) many times faster than the fastest digital computer in existence today. Consider, for example, human *vision*, which is an information-processing task. It is the function of the visual system to provide a *representation* of the environment around us and, more important, to supply the information we need to *interact* with the environment. To be specific, the brain routinely accomplishes perceptual recognition tasks (e.g., recognizing a familiar face embedded in an unfamiliar scene) in approximately 100–200 ms, whereas tasks of much lesser complexity take a great deal longer on a powerful computer.

For another example, consider the *sonar* of a bat. Sonar is an active echolocation system. In addition to providing information about how far away a target (e.g., a flying insect) is, bat sonar conveys information about the relative velocity of the target, the size of the target, the size of various features of the target, and the azimuth and elevation of the target. The complex neural computations needed to extract all this information from the target echo occur within a brain the size of a plum. Indeed, an echolocating bat can pursue and capture its target with a facility and success rate that would be the envy of a radar or sonar engineer.

How, then, does a human brain or the brain of a bat do it? At birth, a brain already has considerable structure and the ability to build up its own rules of behavior through what we usually refer to as "experience." Indeed, experience is built up over time, with much of the development (i.e., hardwiring) of the human brain taking place during the first two years from birth, but the development continues well beyond that stage.

A "developing" nervous system is synonymous with a plastic brain: *Plasticity* permits the developing nervous system to *adapt* to its surrounding environment. Just as plasticity appears to be essential to the functioning of neurons as information-processing units in the human brain, so it is with neural networks made up of artificial neurons. In

its most general form, a *neural network* is a machine that is designed to *model* the way in which the brain performs a particular task or function of interest; the network is usually implemented by using electronic components or is simulated in software on a digital computer. In this book, we focus on an important class of neural networks that perform useful computations through a process of *learning*. To achieve good performance, neural networks employ a massive interconnection of simple computing cells referred to as "neurons" or "processing units." We may thus offer the following definition of a neural network viewed as an adaptive machine[1]:

> *A neural network is a massively parallel distributed processor made up of simple processing units that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:*
>
> 1. *Knowledge is acquired by the network from its environment through a learning process.*
> 2. *Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.*

The procedure used to perform the learning process is called a *learning algorithm*, the function of which is to modify the synaptic weights of the network in an orderly fashion to attain a desired design objective.

The modification of synaptic weights provides the traditional method for the design of neural networks. Such an approach is the closest to linear adaptive filter theory, which is already well established and successfully applied in many diverse fields (Widrow and Stearns, 1985; Haykin, 2002). However, it is also possible for a neural network to modify its own topology, which is motivated by the fact that neurons in the human brain can die and new synaptic connections can grow.

Benefits of Neural Networks

It is apparent that a neural network derives its computing power through, first, its massively parallel distributed structure and, second, its ability to learn and therefore generalize. *Generalization* refers to the neural network's production of reasonable outputs for inputs not encountered during training (learning). These two information-processing capabilities make it possible for neural networks to find good approximate solutions to complex (large-scale) problems that are *intractable*. In practice, however, neural networks cannot provide the solution by working individually. Rather, they need to be integrated into a consistent system engineering approach. Specifically, a complex problem of interest is *decomposed* into a number of relatively simple tasks, and neural networks are assigned a subset of the tasks that *match* their inherent capabilities. It is important to recognize, however, that we have a long way to go (if ever) before we can build a computer architecture that mimics the human brain.

Neural networks offer the following useful properties and capabilities:

1. *Nonlinearity.* An artificial neuron can be linear or nonlinear. A neural network, made up of an interconnection of nonlinear neurons, is itself nonlinear. Moreover, the nonlinearity is of a special kind in the sense that it is *distributed* throughout the network. Nonlinearity is a highly important property, particularly if the underlying physical

mechanism responsible for generation of the input signal (e.g., speech signal) is inherently nonlinear.

2. *Input–Output Mapping.* A popular paradigm of learning, called *learning with a teacher*, or *supervised learning*, involves modification of the synaptic weights of a neural network by applying a set of labeled *training examples*, or *task examples*. Each example consists of a unique *input signal* and a corresponding *desired (target) response*. The network is presented with an example picked at random from the set, and the synaptic weights (free parameters) of the network are modified to minimize the difference between the desired response and the actual response of the network produced by the input signal in accordance with an appropriate statistical criterion. The training of the network is repeated for many examples in the set, until the network reaches a steady state where there are no further significant changes in the synaptic weights. The previously applied training examples may be reapplied during the training session, but in a different order. Thus the network learns from the examples by constructing an *input–output mapping* for the problem at hand. Such an approach brings to mind the study of *nonparametric statistical inference*, which is a branch of statistics dealing with model-free estimation, or, from a biological viewpoint, *tabula rasa* learning (Geman et al., 1992); the term "nonparametric" is used here to signify the fact that no prior assumptions are made on a statistical model for the input data. Consider, for example, a *pattern classification* task, where the requirement is to assign an input signal representing a physical object or event to one of several prespecified categories (classes). In a nonparametric approach to this problem, the requirement is to "estimate" arbitrary decision boundaries in the input signal space for the pattern-classification task using a set of examples, and to do so *without* invoking a probabilistic distribution model. A similar point of view is implicit in the supervised learning paradigm, which suggests a close analogy between the input–output mapping performed by a neural network and nonparametric statistical inference.

3. *Adaptivity.* Neural networks have a built-in capability to *adapt* their synaptic weights to changes in the surrounding environment. In particular, a neural network trained to operate in a specific environment can be easily *retrained* to deal with minor changes in the operating environmental conditions. Moreover, when it is operating in a *nonstationary* environment (i.e., one where statistics change with time), a neural network may be designed to change its synaptic weights in real time. The natural architecture of a neural network for pattern classification, signal processing, and control applications, coupled with the adaptive capability of the network, makes it a useful tool in adaptive pattern classification, adaptive signal processing, and adaptive control. As a general rule, it may be said that the more adaptive we make a system, all the time ensuring that the system remains stable, the more robust its performance will likely be when the system is required to operate in a nonstationary environment. It should be emphasized, however, that adaptivity does not always lead to robustness; indeed, it may do the very opposite. For example, an adaptive system with short-time constants may change rapidly and therefore tend to respond to spurious disturbances, causing a drastic degradation in system performance. To realize the full benefits of adaptivity, the principal time constants of the system should be long enough for the system to ignore spurious disturbances, and yet short enough to respond to meaningful changes in the

environment; the problem described here is referred to as the *stability–plasticity dilemma* (Grossberg, 1988).

4. *Evidential Response.* In the context of pattern classification, a neural network can be designed to provide information not only about which particular pattern to *select*, but also about the *confidence* in the decision made. This latter information may be used to reject ambiguous patterns, should they arise, and thereby improve the classification performance of the network.

5. *Contextual Information.* Knowledge is represented by the very structure and activation state of a neural network. Every neuron in the network is potentially affected by the global activity of all other neurons in the network. Consequently, contextual information is dealt with naturally by a neural network.

6. *Fault Tolerance.* A neural network, implemented in hardware form, has the potential to be inherently *fault tolerant*, or capable of robust computation, in the sense that its performance degrades gracefully under adverse operating conditions. For example, if a neuron or its connecting links are damaged, recall of a stored pattern is impaired in quality. However, due to the distributed nature of information stored in the network, the damage has to be extensive before the overall response of the network is degraded seriously. Thus, in principle, a neural network exhibits a graceful degradation in performance rather than catastrophic failure. There is some empirical evidence for robust computation, but usually it is uncontrolled. In order to be assured that the neural network is, in fact, fault tolerant, it may be necessary to take corrective measures in designing the algorithm used to train the network (Kerlirzin and Vallet, 1993).

7. *VLSI Implementability.* The massively parallel nature of a neural network makes it potentially fast for the computation of certain tasks. This same feature makes a neural network well suited for implementation using *very-large-scale-integrated* (VLSI) technology. One particular beneficial virtue of VLSI is that it provides a means of capturing truly complex behavior in a highly hierarchical fashion (Mead, 1989).

8. *Uniformity of Analysis and Design.* Basically, neural networks enjoy universality as information processors. We say this in the sense that the same notation is used in all domains involving the application of neural networks. This feature manifests itself in different ways:

- Neurons, in one form or another, represent an ingredient *common* to all neural networks.
- This commonality makes it possible to *share* theories and learning algorithms in different applications of neural networks.
- Modular networks can be built through a *seamless integration of modules*.

9. *Neurobiological Analogy.* The design of a neural network is motivated by analogy with the brain, which is living proof that fault-tolerant parallel processing is not only physically possible, but also fast and powerful. Neurobiologists look to (artificial) neural networks as a research tool for the interpretation of neurobiological phenomena. On the other hand, engineers look to neurobiology for new ideas to solve problems more complex than those based on conventional hardwired design

techniques. These two viewpoints are illustrated by the following two respective examples:

- In Anastasio (1993), linear system models of the *vestibulo-ocular reflex* (VOR) are compared to neural network models based on *recurrent networks*, which are described in Section 6 and discussed in detail in Chapter 15. The vestibulo-ocular reflex is part of the oculomotor system. The function of VOR is to maintain visual (i.e., retinal) image stability by making eye rotations that are opposite to head rotations. The VOR is mediated by premotor neurons in the vestibular nuclei that receive and process head rotation signals from vestibular sensory neurons and send the results to the eye muscle motor neurons. The VOR is well suited for modeling because its input (head rotation) and its output (eye rotation) can be precisely specified. It is also a relatively simple reflex, and the neurophysiological properties of its constituent neurons have been well described. Among the three neural types, the premotor neurons (reflex interneurons) in the vestibular nuclei are the most complex and therefore most interesting. The VOR has previously been modeled using lumped, linear system descriptors and control theory. These models were useful in explaining some of the overall properties of the VOR, but gave little insight into the properties of its constituent neurons. This situation has been greatly improved through neural network modeling. Recurrent network models of VOR (programmed using an algorithm called real-time recurrent learning, described in Chapter 15) can reproduce and help explain many of the static, dynamic, nonlinear, and distributed aspects of signal processing by the neurons that mediate the VOR, especially the vestibular nuclei neurons.
- The *retina*, more than any other part of the brain, is where we begin to put together the relationships between the outside world represented by a visual sense, its *physical image* projected onto an array of receptors, and the first *neural images*. The retina is a thin sheet of neural tissue that lines the posterior hemisphere of the eyeball. The retina's task is to convert an optical image into a neural image for transmission down the optic nerve to a multitude of centers for further analysis. This is a complex task, as evidenced by the synaptic organization of the retina. In all vertebrate retinas, the transformation from optical to neural image involves three stages (Sterling, 1990):

  (i) photo transduction by a layer of receptor neurons;
  (ii) transmission of the resulting signals (produced in response to light) by chemical synapses to a layer of bipolar cells;
  (iii) transmission of these signals, also by chemical synapses, to output neurons that are called ganglion cells.

  At both synaptic stages (i.e., from receptor to bipolar cells, and from bipolar to ganglion cells), there are specialized laterally connected neurons called *horizontal cells* and *amacrine cells*, respectively. The task of these neurons is to modify the transmission across the synaptic layers. There are also centrifugal elements called *inter-plexiform cells*; their task is to convey signals from the inner synaptic layer back to the outer one. Some researchers have built electronic chips that mimic the structure of the retina. These electronic chips are called *neuromorphic* integrated circuits, a term coined by Mead (1989). A neuromorphic imaging sensor

consists of an array of photoreceptors combined with analog circuitry at each picture element (pixel). It emulates the retina in that it can adapt locally to changes in brightness, detect edges, and detect motion. The neurobiological analogy, exemplified by neuromorphic integrated circuits, is useful in another important way: It provides a hope and belief, and to a certain extent an existence of proof, that physical understanding of neurobiological structures could have a productive influence on the art of electronics and VLSI technology for the implementation of neural networks.

With inspiration from neurobiology in mind, it seems appropriate that we take a brief look at the human brain and its structural levels of organization.[2]

2 THE HUMAN BRAIN

The human nervous system may be viewed as a three-stage system, as depicted in the block diagram of Fig. 1 (Arbib, 1987). Central to the system is the *brain*, represented by the *neural (nerve) net*, which continually receives information, perceives it, and makes appropriate decisions. Two sets of arrows are shown in the figure. Those pointing from left to right indicate the *forward* transmission of information-bearing signals through the system. The arrows pointing from right to left (shown in red) signify the presence of *feedback* in the system. The *receptors* convert stimuli from the human body or the external environment into electrical impulses that convey information to the neural net (brain). The *effectors* convert electrical impulses generated by the neural net into discernible responses as system outputs.

The struggle to understand the brain has been made easier because of the pioneering work of Ramón y Cajál (1911), who introduced the idea of *neurons* as structural constituents of the brain. Typically, neurons are five to six orders of magnitude slower than silicon logic gates; events in a silicon chip happen in the nanosecond range, whereas neural events happen in the millisecond range. However, the brain makes up for the relatively slow rate of operation of a neuron by having a truly staggering number of neurons (nerve cells) with massive interconnections between them. It is estimated that there are approximately 10 billion neurons in the human cortex, and 60 trillion synapses or connections (Shepherd and Koch, 1990). The net result is that the brain is an enormously efficient structure. Specifically, the *energetic efficiency* of the brain is approximately 10^{-16} joules (J) per operation per second, whereas the corresponding value for the best computers is orders of magnitude larger.

Synapses, or *nerve endings*, are elementary structural and functional units that mediate the interactions between neurons. The most common kind of synapse is a *chemical synapse*, which operates as follows: A presynaptic process liberates a *transmitter* substance that diffuses across the synaptic junction between neurons and then acts on a postsynaptic process. Thus a synapse converts a presynaptic electrical signal into a chemical

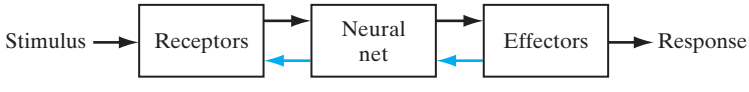


FIGURE 1 Block diagram representation of nervous system.

signal and then back into a postsynaptic electrical signal (Shepherd and Koch, 1990). In electrical terminology, such an element is said to be a *nonreciprocal two-port device.* In traditional descriptions of neural organization, it is assumed that a synapse is a simple connection that can impose *excitation* or *inhibition*, but not both on the receptive neuron.

Earlier we mentioned that plasticity permits the developing nervous system to adapt to its surrounding environment (Eggermont, 1990; Churchland and Sejnowski, 1992). In an adult brain, plasticity may be accounted for by two mechanisms: the creation of new synaptic connections between neurons, and the modification of existing synapses. *Axons*, the transmission lines, and *dendrites*, the receptive zones, constitute two types of cell filaments that are distinguished on morphological grounds; an axon has a smoother surface, fewer branches, and greater length, whereas a dendrite (so called because of its resemblance to a tree) has an irregular surface and more branches (Freeman, 1975). Neurons come in a wide variety of shapes and sizes in different parts of the brain. Figure 2 illustrates the shape of a *pyramidal cell*, which is one of the most common types of cortical neurons. Like many other types of neurons, it receives most of its inputs through dendritic spines; see the segment of dendrite in the insert in Fig. 2 for detail. The pyramidal cell can receive 10,000 or more synaptic contacts, and it can project onto thousands of target cells.

The majority of neurons encode their outputs as a series of brief voltage pulses. These pulses, commonly known as *action potentials*, or *spikes*,[3] originate at or close to the cell body of neurons and then propagate across the individual neurons at constant velocity and amplitude. The reasons for the use of action potentials for communication among neurons are based on the physics of axons. The axon of a neuron is very long and thin and is characterized by high electrical resistance and very large capacitance. Both of these elements are distributed across the axon. The axon may therefore be modeled as resistance-capacitance (RC) transmission line, hence the common use of "cable equation" as the terminology for describing signal propagation along an axon. Analysis of this propagation mechanism reveals that when a voltage is applied at one end of the axon, it decays exponentially with distance, dropping to an insignificant level by the time it reaches the other end. The action potentials provide a way to circumvent this transmission problem (Anderson, 1995).

In the brain, there are both small-scale and large-scale anatomical organizations, and different functions take place at lower and higher levels. Figure 3 shows a hierarchy of interwoven levels of organization that has emerged from the extensive work done on the analysis of local regions in the brain (Shepherd and Koch, 1990; Churchland and Sejnowski, 1992). The *synapses* represent the most fundamental level, depending on molecules and ions for their action. At the next levels, we have neural microcircuits, dendritic trees, and then neurons. A *neural microcircuit* refers to an assembly of synapses organized into patterns of connectivity to produce a functional operation of interest. A neural microcircuit may be likened to a silicon chip made up of an assembly of transistors. The smallest size of microcircuits is measured in micrometers (μm), and their fastest speed of operation is measured in milliseconds. The neural microcircuits are grouped to form *dendritic subunits* within the *dendritic trees* of individual neurons. The whole *neuron*, about 100 μm in size, contains several dendritic subunits. At the next level of complexity, we have *local circuits* (about 1 mm in size) made up of neurons with similar or different properties; these neural

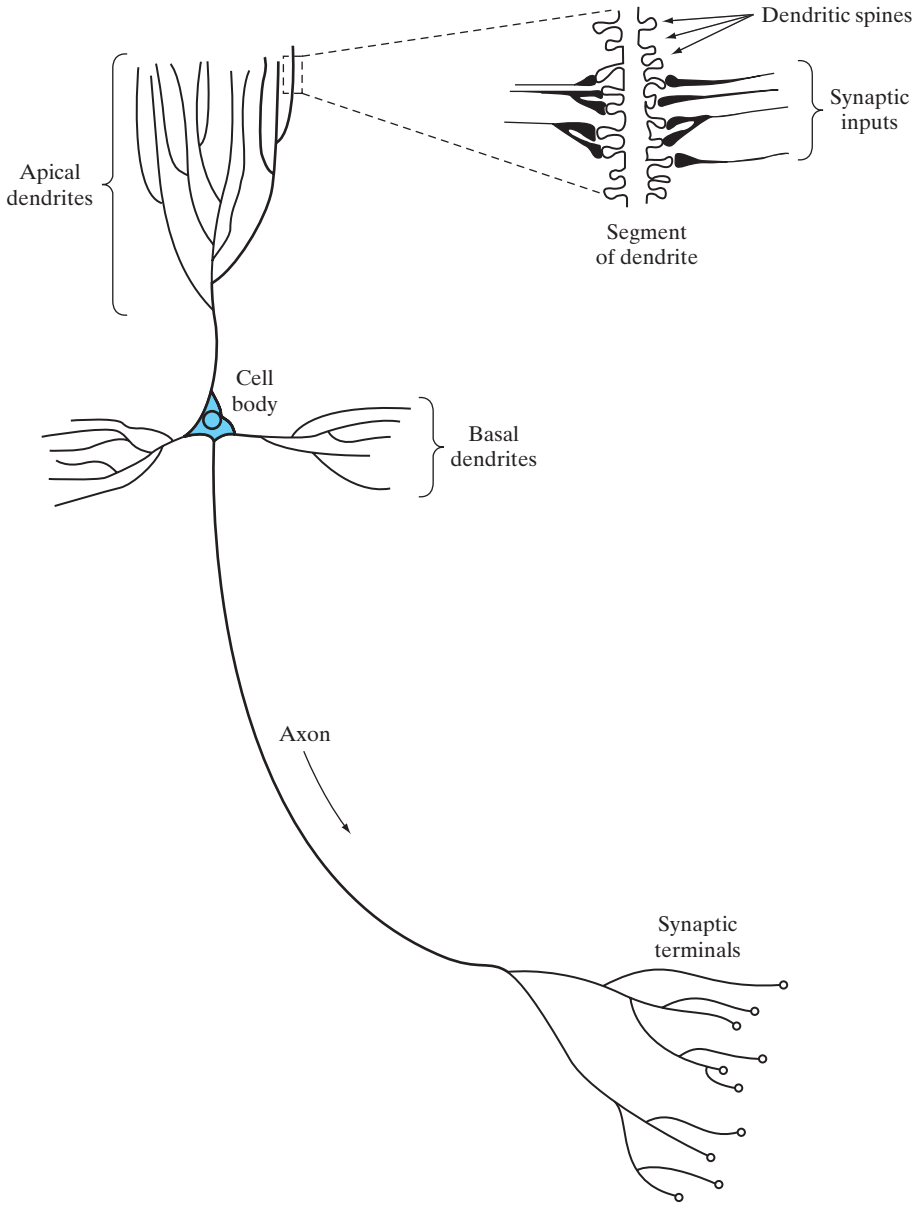


FIGURE 2 The pyramidal cell.

assemblies perform operations characteristic of a localized region in the brain. They are followed by *interregional circuits* made up of pathways, columns, and topographic maps, which involve multiple regions located in different parts of the brain.

Topographic maps are organized to respond to incoming sensory information. These maps are often arranged in sheets, as in the *superior colliculus*, where the visual,

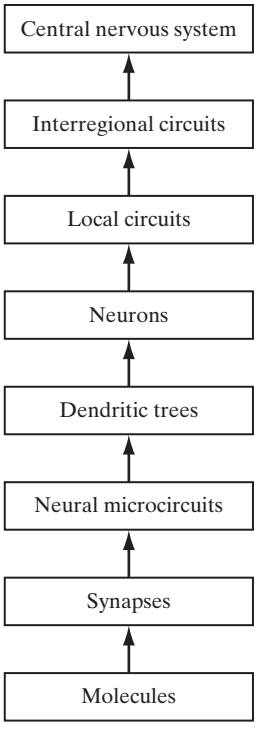


FIGURE 3 Structural organization of levels in the brain.

auditory, and somatosensory maps are stacked in adjacent layers in such a way that stimuli from corresponding points in space lie above or below each other. Figure 4 presents a cytoarchitectural map of the cerebral cortex as worked out by Brodmann (Brodal, 1981). This figure shows clearly that different sensory inputs (motor, somatosensory, visual, auditory, etc.) are mapped onto corresponding areas of the cerebral cortex in an orderly fashion. At the final level of complexity, the topographic maps and other interregional circuits mediate specific types of behavior in the *central nervous system*.

It is important to recognize that the structural levels of organization described herein are a unique characteristic of the brain. They are nowhere to be found in a digital computer, and we are nowhere close to re-creating them with artificial neural networks. Nevertheless, we are inching our way toward a hierarchy of computational levels similar to that described in Fig. 3. The artificial neurons we use to build our neural networks are truly primitive in comparison with those found in the brain. The neural networks we are presently able to design are just as primitive compared with the local circuits and the interregional circuits in the brain. What is really satisfying, however, is the remarkable progress that we have made on so many fronts. With neurobiological analogy as the source of inspiration, and the wealth of theoretical and computational tools that we are bringing together, it is certain that our understanding of artificial neural networks and their applications will continue to grow in depth as well as breadth, year after year.

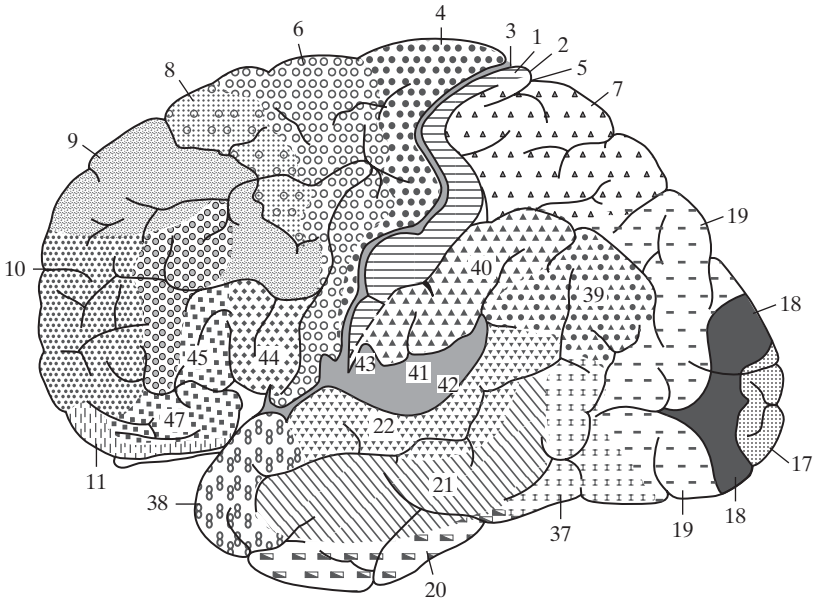


FIGURE 4 Cytoarchitectural map of the cerebral cortex. The different areas are identified by the thickness of their layers and types of cells within them. Some of the key sensory areas are as follows: Motor cortex: motor strip, area 4; premotor area, area 6; frontal eye fields, area 8. Somatosensory cortex: areas 3, 1, and 2. Visual cortex: areas 17, 18, and 19. Auditory cortex: areas 41 and 42. *(From A. Brodal, 1981; with permission of Oxford University Press.)*

3 MODELS OF A NEURON

A *neuron* is an information-processing unit that is fundamental to the operation of a neural network. The block diagram of Fig. 5 shows the *model* of a neuron, which forms the basis for designing a large family of neural networks studied in later chapters. Here, we identify three basic elements of the neural model:

1. A set of *synapses*, or *connecting links*, each of which is characterized by a *weight* or *strength* of its own. Specifically, a signal x_j at the input of synapse j connected to neuron k is multiplied by the synaptic weight w_{kj}. It is important to make a note of the manner in which the subscripts of the synaptic weight w_{kj} are written. The first subscript in w_{kj} refers to the neuron in question, and the second subscript refers to the input end of the synapse to which the weight refers. Unlike the weight of a synapse in the brain, the synaptic weight of an artificial neuron may lie in a range that includes negative as well as positive values.
2. An *adder* for summing the input signals, weighted by the respective synaptic strengths of the neuron; the operations described here constitute a *linear combiner*.
3. An *activation function* for limiting the amplitude of the output of a neuron. The activation function is also referred to as a *squashing function*, in that it squashes (limits) the permissible amplitude range of the output signal to some finite value.

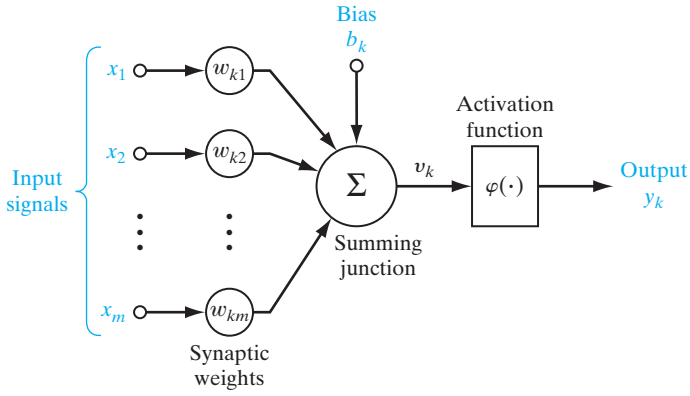


FIGURE 5 Nonlinear model of a neuron, labeled k.

Typically, the normalized amplitude range of the output of a neuron is written as the closed unit interval [0,1], or, alternatively, [−1,1].

The neural model of Fig. 5 also includes an externally applied *bias*, denoted by b_k. The bias b_k has the effect of increasing or lowering the net input of the activation function, depending on whether it is positive or negative, respectively.

In mathematical terms, we may describe the neuron k depicted in Fig. 5 by writing the pair of equations:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{1}$$

and

$$y_k = \varphi(u_k + b_k) \tag{2}$$

where $x_1, x_2, \ldots, x_m$ are the input signals; $w_{k1}, w_{k2}, \ldots, w_{km}$ are the respective synaptic weights of neuron k; u_k (not shown in Fig. 5) is the *linear combiner output* due to the input signals; b_k is the bias; $\varphi(\cdot)$ is the *activation function*; and y_k is the output signal of the neuron. The use of bias b_k has the effect of applying an *affine transformation* to the output u_k of the linear combiner in the model of Fig. 5, as shown by

$$v_k = u_k + b_k \tag{3}$$

In particular, depending on whether the bias b_k is positive or negative, the relationship between the *induced local field*, or *activation potential*, v_k of neuron k and the linear combiner output u_k is modified in the manner illustrated in Fig. 6; hereafter, these two terms are used interchangeably. Note that as a result of this affine transformation, the graph of v_k versus u_k no longer passes through the origin.

The bias b_k is an external parameter of neuron k. We may account for its presence as in Eq. (2). Equivalently, we may formulate the combination of Eqs. (1) to (3) as follows:

$$v_k = \sum_{j=0}^{m} w_{kj} x_j \tag{4}$$

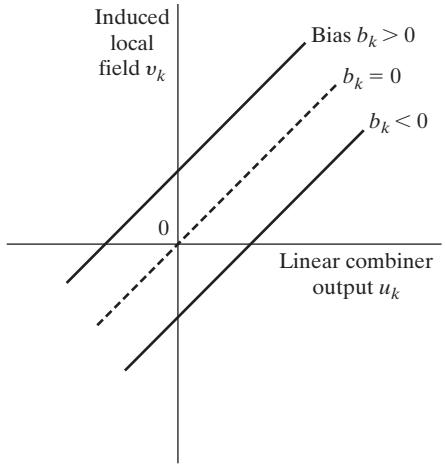


FIGURE 6 Affine transformation produced by the presence of a bias; note that $v_k = b_k$ at $u_k = 0$.

and

$$y_k = \varphi(v_k) \tag{5}$$

In Eq. (4), we have added a new synapse. Its input is

$$x_0 = +1 \tag{6}$$

and its weight is

$$w_{k0} = b_k \tag{7}$$

We may therefore reformulate the model of neuron k as shown in Fig. 7. In this figure, the effect of the bias is accounted for by doing two things: (1) adding a new input signal fixed at +1, and (2) adding a new synaptic weight equal to the bias b_k. Although the models of Figs. 5 and 7 are different in appearance, they are mathematically equivalent.

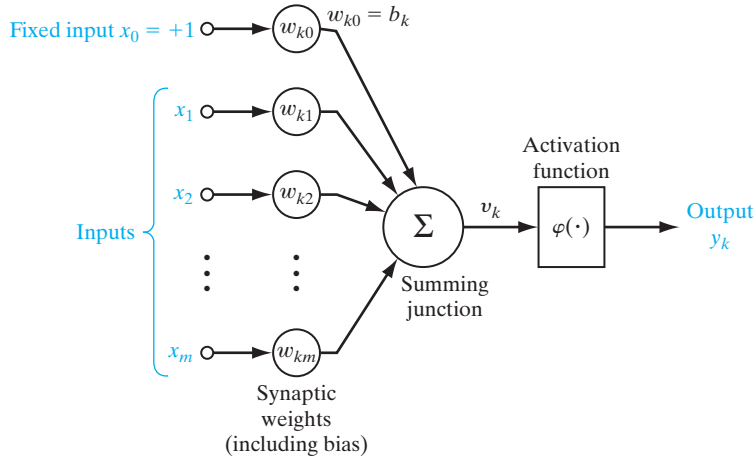


FIGURE 7 Another nonlinear model of a neuron; w_{k0} accounts for the bias b_k.

Types of Activation Function

The activation function, denoted by $\varphi(v)$, defines the output of a neuron in terms of the induced local field v. In what follows, we identify two basic types of activation functions:

1. *Threshold Function.* For this type of activation function, described in Fig. 8a, we have

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases} \tag{8}$$

In engineering, this form of a threshold function is commonly referred to as a *Heaviside function.* Correspondingly, the output of neuron k employing such a threshold function is expressed as

$$y_k = \begin{cases} 1 & \text{if } v_k \geq 0 \\ 0 & \text{if } v_k < 0 \end{cases} \tag{9}$$

where v_k is the induced local field of the neuron; that is,

$$v_k = \sum_{j=1}^{m} w_{kj} x_j + b_k \tag{10}$$

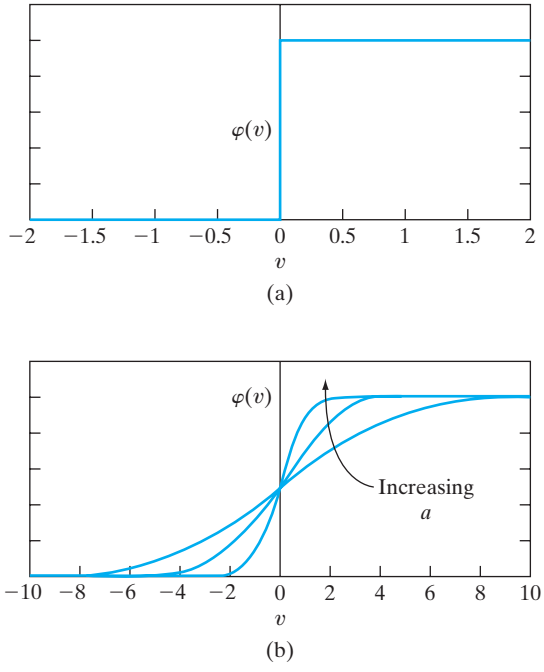


FIGURE 8 (a) Threshold function. (b) Sigmoid function for varying slope parameter a.

In neural computation, such a neuron is referred to as the *McCulloch–Pitts model*, in recognition of the pioneering work done by McCulloch and Pitts (1943). In this model, the output of a neuron takes on the value of 1 if the induced local field of that neuron is nonnegative, and 0 otherwise. This statement describes the *all-or-none property* of the McCulloch–Pitts model.

2. *Sigmoid Function.*[4] The sigmoid function, whose graph is "S"-shaped, is by far the most common form of activation function used in the construction of neural networks. It is defined as a strictly increasing function that exhibits a graceful balance between linear and nonlinear behavior. An example of the sigmoid function is the *logistic function*,[5] defined by

$$\varphi(v) = \frac{1}{1 + \exp(-av)} \tag{11}$$

where a is the *slope parameter* of the sigmoid function. By varying the parameter a, we obtain sigmoid functions of different slopes, as illustrated in Fig. 8b. In fact, the slope at the origin equals $a/4$. In the limit, as the slope parameter approaches infinity, the sigmoid function becomes simply a threshold function. Whereas a threshold function assumes the value of 0 or 1, a sigmoid function assumes a continuous range of values from 0 to 1. Note also that the sigmoid function is differentiable, whereas the threshold function is not. (Differentiability is an important feature of neural network theory, as described in Chapter 4).

The activation functions defined in Eqs. (8) and (11) range from 0 to +1. It is sometimes desirable to have the activation function range from −1 to +1, in which case, the activation function is an odd function of the induced local field. Specifically, the threshold function of Eq. (8) is now defined as

$$\varphi(v) = \begin{cases} 1 & \text{if } v > 0 \\ 0 & \text{if } v = 0 \\ -1 & \text{if } v < 0 \end{cases} \tag{12}$$

which is commonly referred to as the *signum function.* For the corresponding form of a sigmoid function, we may use the *hyperbolic tangent function*, defined by

$$\varphi(v) = \tanh(v) \tag{13}$$

Allowing an activation function of the sigmoid type to assume negative values as prescribed by Eq. (13) may yield practical benefits over the logistic function of Eq. (11).

Stochastic Model of a Neuron

The neural model described in Fig. 7 is deterministic in that its input–output behavior is precisely defined for all inputs. For some applications of neural networks, it is desirable to base the analysis on a stochastic neural model. In an analytically tractable approach, the activation function of the McCulloch–Pitts model is given a probabilistic interpretation. Specifically, a neuron is permitted to reside in only one of two states: +1

or −1, say. The decision for a neuron to *fire* (i.e., switch its state from "off" to "on") is probabilistic. Let x denote the state of the neuron and $P(v)$ denote the *probability* of firing, where v is the induced local field of the neuron. We may then write

$$x = \begin{cases} +1 & \text{with probability } P(v) \\ -1 & \text{with probability } 1 - P(v) \end{cases} \tag{14}$$

A standard choice for $P(v)$ is the sigmoid-shaped function

$$P(v) = \frac{1}{1 + \exp(-v/T)} \tag{15}$$

where T is a *pseudotemperature* used to control the noise level and therefore the uncertainty in firing (Little, 1974). It is important to realize, however, that T is *not* the physical temperature of a neural network, be it a biological or an artificial neural network. Rather, as already stated, we should think of T merely as a parameter that controls the thermal fluctuations representing the effects of synaptic noise. Note that when $T \to 0$, the stochastic neuron described by Eqs. (14) and (15) reduces to a noiseless (i.e., deterministic) form, namely, the McCulloch–Pitts model.

4 NEURAL NETWORKS VIEWED AS DIRECTED GRAPHS

The *block diagram* of Fig. 5 or that of Fig. 7 provides a functional description of the various elements that constitute the model of an artificial neuron. We may simplify the appearance of the model by using the idea of signal-flow graphs without sacrificing any of the functional details of the model. Signal-flow graphs, with a well-defined set of rules, were originally developed by Mason (1953, 1956) for linear networks. The presence of nonlinearity in the model of a neuron limits the scope of their application to neural networks. Nevertheless, signal-flow graphs do provide a neat method for the portrayal of the flow of signals in a neural network, which we pursue in this section.

A *signal-flow graph* is a network of directed *links* (*branches*) that are interconnected at certain points called *nodes*. A typical node j has an associated *node signal* x_j. A typical directed link originates at node j and terminates on node k; it has an associated *transfer function*, or *transmittance*, that specifies the manner in which the signal y_k at node k depends on the signal x_j at node j. The flow of signals in the various parts of the graph is dictated by three basic rules:

Rule 1. A signal flows along a link only in the direction defined by the arrow on the link.

Two different types of links may be distinguished:

- *Synaptic links*, whose behavior is governed by a *linear* input–output relation. Specifically, the node signal x_j is multiplied by the synaptic weight w_{kj} to produce the node signal y_k, as illustrated in Fig. 9a.
- *Activation links*, whose behavior is governed in general by a *nonlinear* input–output relation. This form of relationship is illustrated in Fig. 9b, where $\varphi(\cdot)$ is the nonlinear activation function.

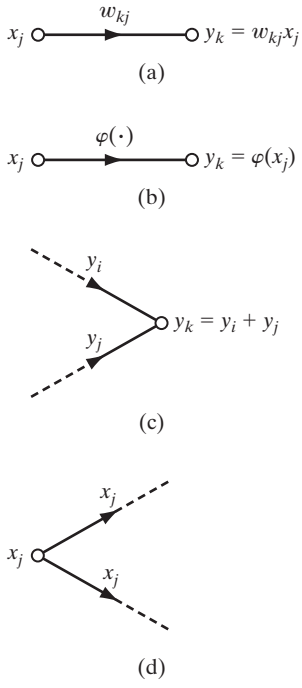


FIGURE 9 Illustrating basic rules for the construction of signal-flow graphs.

Rule 2. A node signal equals the algebraic sum of all signals entering the pertinent node via the incoming links.

This second rule is illustrated in Fig. 9c for the case of *synaptic convergence*, or *fan-in*.

Rule 3. The signal at a node is transmitted to each outgoing link originating from that node, with the transmission being entirely independent of the transfer functions of the outgoing links.

This third rule is illustrated in Fig. 9d for the case of *synaptic divergence*, or *fan-out*.

For example, using these rules, we may construct the signal-flow graph of Fig. 10 as the model of a neuron, corresponding to the block diagram of Fig. 7. The representation shown in Fig. 10 is clearly simpler in appearance than that of Fig. 7, yet it contains all the functional details depicted in the latter diagram. Note that in both figures, the input $x_0 = +1$ and the associated synaptic weight $w_{k0} = b_k$, where b_k is the bias applied to neuron k.

Indeed, based on the signal-flow graph of Fig. 10 as the model of a neuron, we may now offer the following mathematical definition of a neural network:

A neural network is a directed graph consisting of nodes with interconnecting synaptic and activation links and is characterized by four properties:

1. *Each neuron is represented by a set of linear synaptic links, an externally applied bias, and a possibly nonlinear activation link. The bias is represented by a synaptic link connected to an input fixed at +1.*
2. *The synaptic links of a neuron weight their respective input signals.*

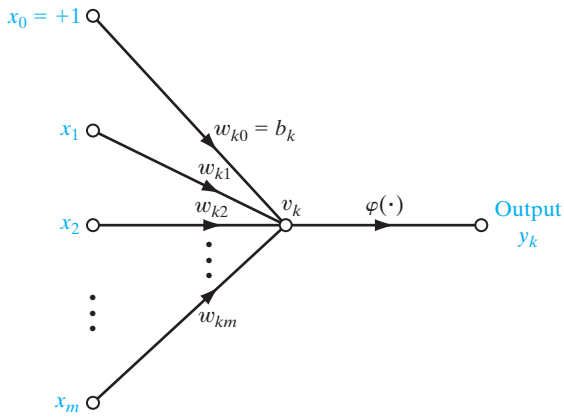


FIGURE 10 Signal-flow graph of a neuron.

3. *The weighted sum of the input signals defines the induced local field of the neuron in question.*
4. *The activation link squashes the induced local field of the neuron to produce an output.*

A directed graph, defined in this manner is *complete* in the sense that it describes not only the signal flow from neuron to neuron, but also the signal flow inside each neuron. When, however, the focus of attention is restricted to signal flow from neuron to neuron, we may use a reduced form of this graph by omitting the details of signal flow inside the individual neurons. Such a directed graph is said to be *partially complete.* It is characterized as follows:

1. *Source nodes* supply input signals to the graph.
2. Each neuron is represented by a single node called a *computation node.*
3. The *communication links* interconnecting the source and computation nodes of the graph carry no weight; they merely provide directions of signal flow in the graph.

A partially complete directed graph defined in this way is referred to as an *architectural graph*, describing the layout of the neural network. It is illustrated in Fig. 11 for the simple case of a single neuron with m source nodes and a single node fixed at +1 for the bias. Note that the computation node representing the neuron is shown shaded, and the source node is shown as a small square. This convention is followed throughout the book. More elaborate examples of architectural layouts are presented later in Section 6.

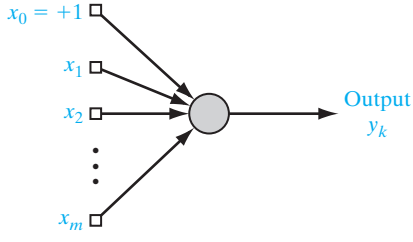


FIGURE 11 Architectural graph of a neuron.

To sum up, we have three graphical representations of a neural network:

- block diagram, providing a functional description of the network;
- architectural graph, describing the network layout;
- signal-flow graph, providing a complete description of signal flow in the network.

5 FEEDBACK

Feedback is said to exist in a dynamic system whenever the output of an element in the system influences in part the input applied to that particular element, thereby giving rise to one or more closed paths for the transmission of signals around the system. Indeed, feedback occurs in almost every part of the nervous system of every animal (Freeman, 1975). Moreover, it plays a major role in the study of a special class of neural networks known as *recurrent networks*. Figure 12 shows the signal-flow graph of a *single-loop feedback system*, where the input signal $x_j(n)$, internal signal $x_j'(n)$, and output signal $y_k(n)$ are functions of the discrete-time variable n. The system is assumed to be *linear*, consisting of a forward path and a feedback path that are characterized by the "operators" $\mathbf{A}$ and $\mathbf{B}$, respectively. In particular, the output of the forward channel determines in part its own output through the feedback channel. From Fig. 12, we readily note the input–output relationships

$$y_k(n) = \mathbf{A}[x_j'(n)] \tag{16}$$

and

$$x_j'(n) = x_j(n) + \mathbf{B}[y_k(n)] \tag{17}$$

where the square brackets are included to emphasize that $\mathbf{A}$ and $\mathbf{B}$ act as *operators*. Eliminating $x_j'(n)$ between Eqs. (16) and (17), we get

$$y_k(n) = \frac{\mathbf{A}}{1 - \mathbf{AB}}[x_j(n)] \tag{18}$$

We refer to $\mathbf{A}/(1 - \mathbf{AB})$ as the *closed-loop operator* of the system, and to $\mathbf{AB}$ as the *open-loop operator*. In general, the open-loop operator is noncommutative in that $\mathbf{BA} \neq \mathbf{AB}$.

Consider, for example, the single-loop feedback system shown in Fig. 13a, for which $\mathbf{A}$ is a fixed weight w and $\mathbf{B}$ is a *unit-delay operator* z^{-1}, whose output is delayed with respect to the input by one time unit. We may then express the closed-loop operator of the system as

$$\begin{aligned} \frac{\mathbf{A}}{1 - \mathbf{AB}} &= \frac{w}{1 - wz^{-1}} \\ &= w(1 - wz^{-1})^{-1} \end{aligned}$$

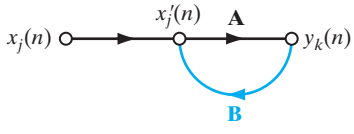


FIGURE 12 Signal-flow graph of a single-loop feedback system.

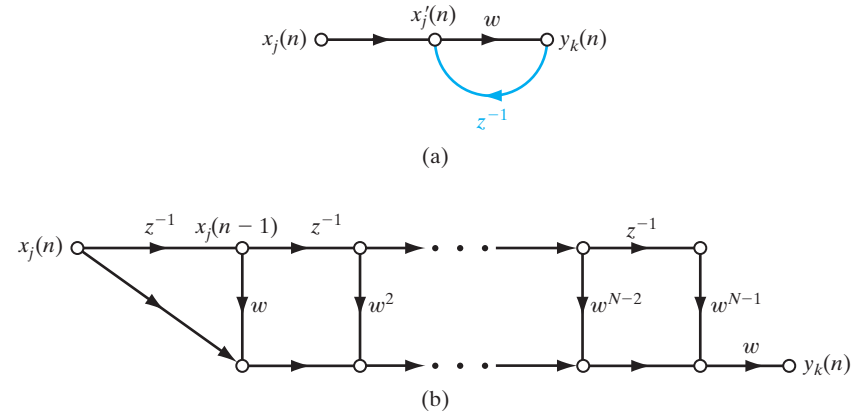


FIGURE 13 (a) Signal-flow graph of a first-order, infinite-duration impulse response (IIR) filter. (b) Feedforward approximation of part (a) of the figure, obtained by truncating Eq. (20).

Using the binomial expansion for $(1 - wz^{-1})^{-1}$, we may rewrite the closed-loop operator of the system as

$$\frac{\mathbf{A}}{1 - \mathbf{AB}} = w \sum_{l=0}^{\infty} w^l z^{-l} \tag{19}$$

Hence, substituting Eq. (19) into (18), we get

$$y_k(n) = w \sum_{l=0}^{\infty} w^l z^{-l}[x_j(n)] \tag{20}$$

where again we have included square brackets to emphasize the fact that z^{-1} is an operator. In particular, from the definition of z^{-1}, we have

$$z^{-l}[x_j(n)] = x_j(n - l) \tag{21}$$

where $x_j(n - l)$ is a sample of the input signal delayed by l time units. Accordingly, we may express the output signal $y_k(n)$ as an infinite weighted summation of present and past samples of the input signal $x_j(n)$, as shown by

$$y_k(n) = \sum_{l=0}^{\infty} w^{l+1} x_j(n - l) \tag{22}$$

We now see clearly that the dynamic behavior of a feedback system represented by the signal-flow graph of Fig. 13 is controlled by the weight w. In particular, we may distinguish two specific cases:

1. $|w| < 1$, for which the output signal $y_k(n)$ is exponentially *convergent*; that is, the system is *stable*. This case is illustrated in Fig. 14a for a positive w.
2. $|w| \geq 1$, for which the output signal $y_k(n)$ is *divergent*; that is, the system is *unstable*. If $|w| = 1$ the divergence is linear, as in Fig. 14b, and if $|w| > 1$ the divergence is exponential, as in Fig. 14c.

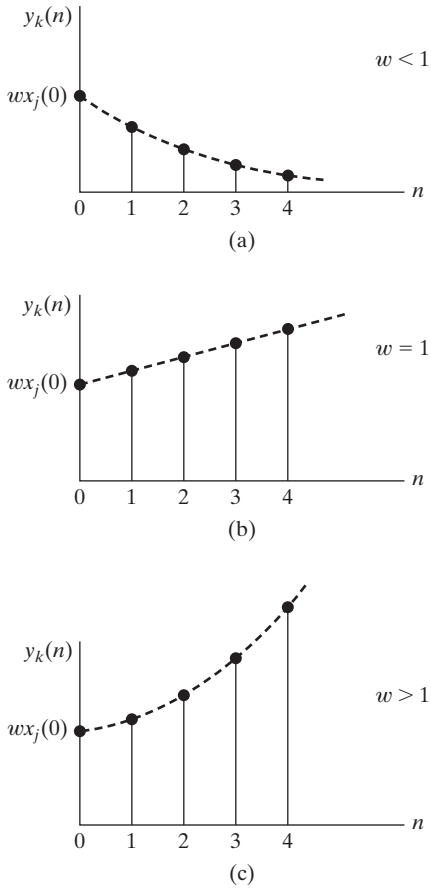


FIGURE 14 Time response of Fig. 13 for three different values of feedforward weight w.
(a) Stable.
(b) Linear divergence.
(c) Exponential divergence.

The issue of stability features prominently in the study of closed-loop feedback systems.

The case of $|w| < 1$ corresponds to a system with *infinite memory* in the sense that the output of the system depends on samples of the input extending into the infinite past. Moreover, the memory is *fading* in that the influence of a past sample is reduced exponentially with time n. Suppose that, for some power N, $|w|$ is small enough relative to unity such that w^N is negligible for all practical purposes. In such a situation, we may approximate the output y_k by the finite sum

$$\begin{aligned} y_k(n) &\approx \sum_{l=0}^{N-1} w^{l+1} x_j(n-l) \\ &= wx_j(n) + w^2 x_j(n-1) + w^3 x_j(n-2) + \cdots + w^N x_j(n-N+1) \end{aligned}$$

In a corresponding way, we may use the feedforward signal-flow graph of Fig. 13b as the approximation for the feedback signal-flow graph of Fig. 13a. In making this approximation, we speak of the "unfolding" of a feedback system. Note, however, that the unfolding operation is of practical value only when the feedback system is stable.

The analysis of the dynamic behavior of neural networks involving the application of feedback is unfortunately complicated by the fact that the processing units used for the construction of the network are usually *nonlinear*. Further consideration of this important issue is deferred to the latter part of the book.

6 NETWORK ARCHITECTURES

The manner in which the neurons of a neural network are structured is intimately linked with the learning algorithm used to train the network. We may therefore speak of learning algorithms (rules) used in the design of neural networks as being *structured*. The classification of learning algorithms is considered in Section 8. In this section, we focus attention on network architectures (structures).

In general, we may identify three fundamentally different classes of network architectures:

(i) Single-Layer Feedforward Networks

In a *layered* neural network, the neurons are organized in the form of layers. In the simplest form of a layered network, we have an *input layer* of source nodes that projects directly onto an *output layer* of neurons (computation nodes), but not vice versa. In other words, this network is strictly of a *feedforward* type. It is illustrated in Fig. 15 for the case of four nodes in both the input and output layers. Such a network is called a *single-layer network*, with the designation "single-layer" referring to the output layer of computation nodes (neurons). We do not count the input layer of source nodes because no computation is performed there.

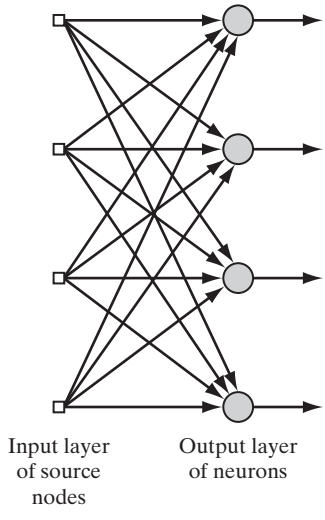


FIGURE 15 Feedforward network with a single layer of neurons.

(ii) Multilayer Feedforward Networks

The second class of a feedforward neural network distinguishes itself by the presence of one or more *hidden layers*, whose computation nodes are correspondingly called *hidden neurons* or *hidden units*; the term "hidden" refers to the fact that this part of the neural network is not seen directly from either the input or output of the network. The function of hidden neurons is to intervene between the external input and the network output in some useful manner. By adding one or more hidden layers, the network is enabled to extract higher-order statistics from its input. In a rather loose sense, the network acquires a *global* perspective despite its local connectivity, due to the extra set of synaptic connections and the extra dimension of neural interactions (Churchland and Sejnowski, 1992).

The source nodes in the input layer of the network supply respective elements of the activation pattern (input vector), which constitute the input signals applied to the neurons (computation nodes) in the second layer (i.e., the first hidden layer). The output signals of the second layer are used as inputs to the third layer, and so on for the rest of the network. Typically, the neurons in each layer of the network have as their inputs the output signals of the preceding layer only. The set of output signals of the neurons in the output (final) layer of the network constitutes the overall response of the network to the activation pattern supplied by the source nodes in the input (first) layer. The architectural graph in Fig. 16 illustrates the layout of a multilayer feedforward neural network for the case of a single hidden layer. For the sake of brevity, the network in Fig. 16 is referred to as a 10–4–2 network because it has 10 source nodes, 4 hidden neurons, and 2 output neurons. As another example, a feedforward network with m source nodes, h_1 neurons in the first hidden layer, h_2 neurons in the second hidden layer, and q neurons in the output layer is referred to as an m–h_1–h_2–q network.

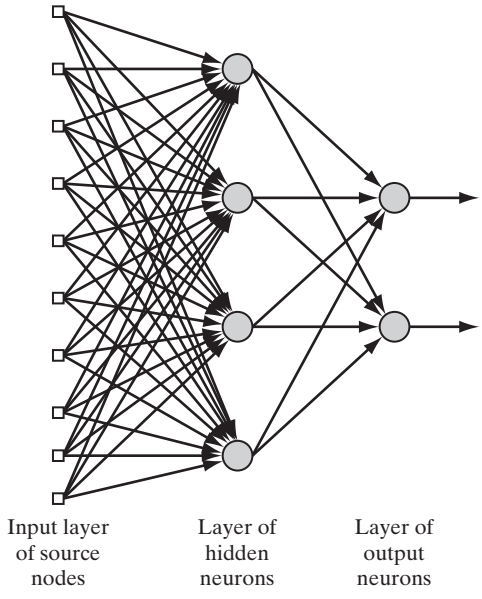


FIGURE 16 Fully connected feedforward network with one hidden layer and one output layer.

The neural network in Fig. 16 is said to be *fully connected* in the sense that every node in each layer of the network is connected to every other node in the adjacent forward layer. If, however, some of the communication links (synaptic connections) are missing from the network, we say that the network is *partially connected.*

(iii) Recurrent Networks

A *recurrent neural network* distinguishes itself from a feedforward neural network in that it has at least one *feedback* loop. For example, a recurrent network may consist of a single layer of neurons with each neuron feeding its output signal back to the inputs of all the other neurons, as illustrated in the architectural graph in Fig. 17. In the structure depicted in this figure, there are *no* self-feedback loops in the network; self-feedback refers to a situation where the output of a neuron is fed back into its own input. The recurrent network illustrated in Fig. 17 also has *no* hidden neurons.

In Fig. 18 we illustrate another class of recurrent networks with hidden neurons. The feedback connections shown in Fig. 18 originate from the hidden neurons as well as from the output neurons.

The presence of feedback loops, be it in the recurrent structure of Fig. 17 or in that of Fig. 18, has a profound impact on the learning capability of the network and on its performance. Moreover, the feedback loops involve the use of particular branches composed of unit-time delay elements (denoted by z^{-1}), which result in a nonlinear dynamic behavior, assuming that the neural network contains nonlinear units.

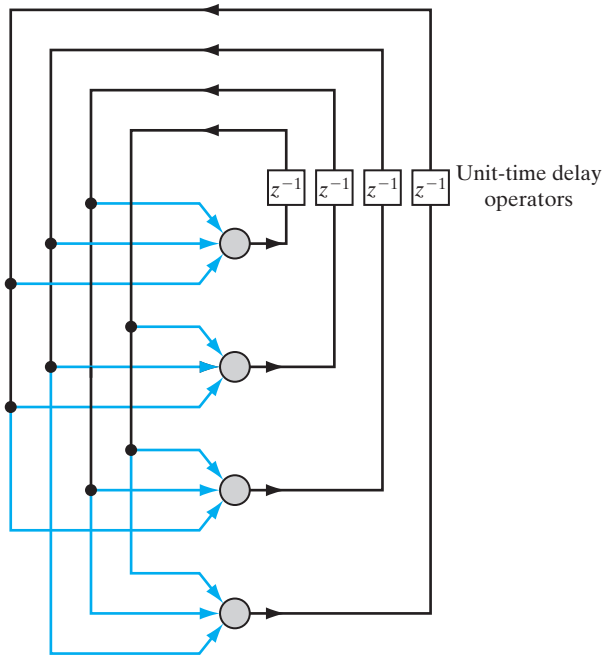


FIGURE 17 Recurrent network with no self-feedback loops and no hidden neurons.

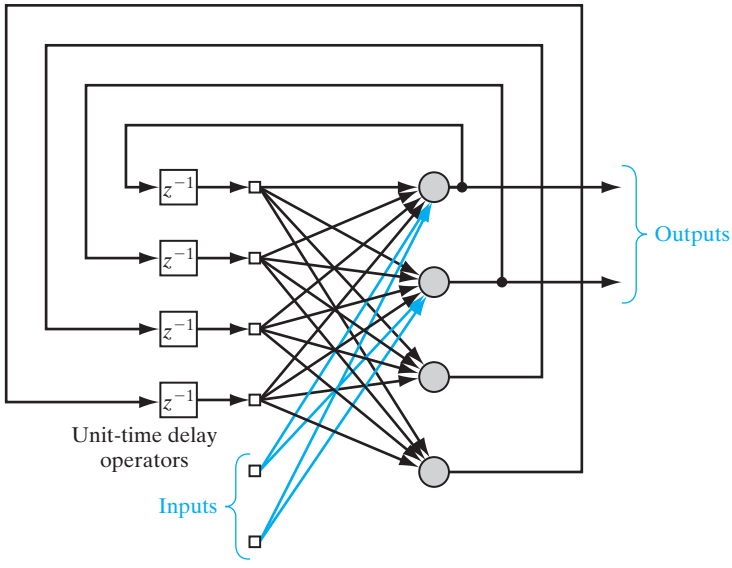


FIGURE 18 Recurrent network with hidden neurons.

7 KNOWLEDGE REPRESENTATION

In Section 1, we used the term "knowledge" in the definition of a neural network without an explicit description of what we mean by it. We now take care of this matter by offering the following generic definition (Fischler and Firschein, 1987):

> *Knowledge refers to stored information or models used by a person or machine to interpret, predict, and appropriately respond to the outside world.*

The primary characteristics of *knowledge representation* are twofold: (1) what information is actually made explicit; and (2) how the information is physically encoded for subsequent use. By the very nature of it, therefore, knowledge representation is goal directed. In real-world applications of "intelligent" machines, it can be said that a good solution depends on a good representation of knowledge (Woods, 1986). So it is with neural networks. Typically, however, we find that the possible forms of representation from the inputs to internal network parameters are highly diverse, which tends to make the development of a satisfactory solution by means of a neural network a real design challenge.

A major task for a neural network is to learn a model of the world (environment) in which it is embedded, and to maintain the model sufficiently consistently with the real world so as to achieve the specified goals of the application of interest. Knowledge of the world consists of two kinds of information:

1. The known world state, represented by facts about what is and what has been known; this form of knowledge is referred to as *prior information.*
2. Observations (measurements) of the world, obtained by means of sensors designed to probe the environment, in which the neural network is supposed to operate.

Ordinarily, these observations are inherently noisy, being subject to errors due to sensor noise and system imperfections. In any event, the observations so obtained provide the pool of information, from which the *examples* used to train the neural network are drawn.

The examples can be *labeled* or *unlabeled.* In labeled examples, each example representing an *input signal* is paired with a corresponding *desired response* (i.e., target output). On the other hand, unlabeled examples consist of different realizations of the input signal all by itself. In any event, a set of examples, labeled or otherwise, represents knowledge about the environment of interest that a neural network can learn through training. Note, however, that labeled examples may be expensive to collect, as they require the availability of a "teacher" to provide a desired response for each labeled example. In contrast, unlabeled examples are usually abundant as there is no need for supervision.

A set of input–output pairs, with each pair consisting of an input signal and the corresponding desired response, is referred to as a *set of training data*, or simply *training sample.* To illustrate how such a data set can be used, consider, for example, the *handwritten-digit recognition problem.* In this problem, the input signal consists of an image with black or white pixels, with each image representing one of 10 digits that are well separated from the background. The desired response is defined by the "identity" of the particular digit whose image is presented to the network as the input signal. Typically, the training sample consists of a large variety of handwritten digits that are representative of a real-world situation. Given such a set of examples, the design of a neural network may proceed as follows:

- An appropriate architecture is selected for the neural network, with an input layer consisting of source nodes equal in number to the pixels of an input image, and an output layer consisting of 10 neurons (one for each digit). A subset of examples is then used to train the network by means of a suitable algorithm. This phase of the network design is called *learning.*
- The recognition performance of the trained network is *tested* with data not seen before. Specifically, an input image is presented to the network, but this time the network is not told the identity of the digit which that particular image represents. The performance of the network is then assessed by comparing the digit recognition reported by the network with the actual identity of the digit in question. This second phase of the network operation is called *testing*, and successful performance on the test patterns is called *generalization*, a term borrowed from psychology.

Herein lies a fundamental difference between the design of a neural network and that of its classical information-processing counterpart: the pattern classifier. In the latter case, we usually proceed by first formulating a mathematical model of environmental observations, validating the model with real data, and then building the design on the basis of the model. In contrast, the design of a neural network is based directly on real-life data, with the *data set being permitted to speak for itself.* Thus, the neural network not only provides the implicit model of the environment in which it is embedded, but also performs the information-processing function of interest.

The examples used to train a neural network may consist of both *positive* and *negative* examples. For instance, in a passive sonar detection problem, positive examples pertain to input training data that contain the target of interest (e.g., a submarine). Now,

in a passive sonar environment, the possible presence of marine life in the test data is known to cause occasional false alarms. To alleviate this problem, negative examples (e.g., echos from marine life) are included purposely in the training data to teach the network not to confuse marine life with the target.

In a neural network of specified architecture, knowledge representation of the surrounding environment is defined by the values taken on by the free parameters (i.e., synaptic weights and biases) of the network. The form of this knowledge representation constitutes the very design of the neural network, and therefore holds the key to its performance.

Roles of Knowledge Representation

The subject of how knowledge is actually represented inside an artificial network is, however, very complicated. Nevertheless, there are four rules for knowledge representation that are of a general commonsense nature, as described next.

Rule 1. Similar inputs (i.e., patterns drawn) from similar classes should usually produce similar representations inside the network, and should therefore be classified as belonging to the same class.

There is a plethora of measures for determining the similarity between inputs. A commonly used *measure of similarity* is based on the concept of Euclidian distance. To be specific, let $\mathbf{x}_i$ denote an m-by-1 vector

$$\mathbf{x}_i = [x_{i1}, x_{i2}, ..., x_{im}]^T$$

all of whose elements are real; the superscript T denotes matrix *transposition*. The vector $\mathbf{x}_i$ defines a point in an m-dimensional space called *Euclidean space* and denoted by $\mathbb{R}^m$. As illustrated in Fig. 19, the *Euclidean distance* between a pair of m-by-1 vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ is defined by

$$\begin{aligned} d(\mathbf{x}_i, \mathbf{x}_j) &= \|\mathbf{x}_i - \mathbf{x}_j\| \\ &= \left[\sum_{k=1}^{m} (x_{ik} - x_{jk})^2\right]^{1/2} \end{aligned} \tag{23}$$

where x_{ik} and x_{jk} are the kth elements of the input vectors $\mathbf{x}_i$ and $\mathbf{x}_j$, respectively. Correspondingly, the similarity between the inputs represented by the vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ is defined as the Euclidean distance $d(\mathbf{x}_i, \mathbf{x}_j)$. The closer the individual elements of the input vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ are to each other, the smaller the Euclidean distance $d(\mathbf{x}_i, \mathbf{x}_j)$ is and therefore the greater the similarity between the vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ will be. Rule 1 states that if the vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ are similar, they should be assigned to the same class.

Another measure of similarity is based on the idea of a *dot product*, or *inner product*, which is also borrowed from matrix algebra. Given a pair of vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ of the same dimension, their inner product is $\mathbf{x}_i^T\mathbf{x}_j$, defined as the *projection* of the vector $\mathbf{x}_i$ onto the vector $\mathbf{x}_j$, as illustrated in Fig. 19. We thus write

$$\begin{aligned} (\mathbf{x}_i, \mathbf{x}_j) &= \mathbf{x}_i^T\mathbf{x}_j \\ &= \sum_{k=1}^{m} x_{ik}x_{jk} \end{aligned} \tag{24}$$

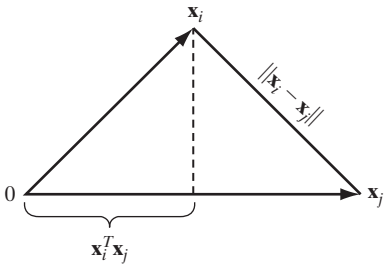


FIGURE 19 Illustrating the relationship between inner product and Euclidean distance as measures of similiarity between patterns.

The inner product $(\mathbf{x}_i, \mathbf{x}_j)$ divided by the product $\|\mathbf{x}_i\|\,\|\mathbf{x}_j\|$ is the *cosine of the angle* subtended between the vectors $\mathbf{x}_i$ and $\mathbf{x}_j$.

The two measures of similarity defined here are indeed intimately related to each other, as illustrated in Fig. 19. This figure shows clearly that the smaller the Euclidean distance $\|\mathbf{x}_i - \mathbf{x}_j\|$, and therefore the more similar the vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ are, the larger the inner product $\mathbf{x}_i^T\mathbf{x}_j$ will be.

To put this relationship on a formal basis, we first normalize the vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ to have unit length, that is,

$$\|\mathbf{x}_i\| = \|\mathbf{x}_j\| = 1$$

We may then use Eq. (23) to write

$$\begin{aligned} d^2(\mathbf{x}_i, \mathbf{x}_j) &= (\mathbf{x}_i - \mathbf{x}_j)^T(\mathbf{x}_i - \mathbf{x}_j) \\ &= 2 - 2\mathbf{x}_i^T\mathbf{x}_j \end{aligned} \tag{25}$$

Equation (25) shows that minimization of the Euclidean distance $d(\mathbf{x}_i, \mathbf{x}_i)$ corresponds to maximization of the inner product $(\mathbf{x}_i, \mathbf{x}_j)$ and, therefore, the similarity between the vectors $\mathbf{x}_i$, and $\mathbf{x}_j$.

The Euclidean distance and inner product described here are defined in deterministic terms. What if the vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ are *stochastic*, drawn from two different populations, or ensembles, of data? To be specific, suppose that the difference between these two populations lies solely in their mean vectors. Let $\boldsymbol{\mu}_i$ and $\boldsymbol{\mu}_j$ denote the mean values of the vectors $\mathbf{x}_i$ and $\mathbf{x}_j$, respectively. That is,

$$\boldsymbol{\mu}_i = \mathbb{E}[\mathbf{x}_i] \tag{26}$$

where $\mathbb{E}$ is the *statistical expectation operator* over the *ensemble* of data vectors $\mathbf{x}_i$. The mean vector $\boldsymbol{\mu}_j$ is similarly defined. For a measure of the distance between these two populations, we may use the *Mahalanobis distance*, denoted by d_{ij}. The squared value of this distance from $\mathbf{x}_i$ to $\mathbf{x}_j$ is defined by

$$d_{ij}^2 = (\mathbf{x}_i - \boldsymbol{\mu}_i)^T\mathbf{C}^{-1}(\mathbf{x}_j - \boldsymbol{\mu}_j) \tag{27}$$

where $\mathbf{C}^{-1}$ is the *inverse* of the covariance matrix $\mathbf{C}$. It is assumed that the *covariance matrix* is the same for both populations, as shown by

$$\begin{aligned} \mathbf{C} &= \mathbb{E}[(\mathbf{x}_i - \boldsymbol{\mu}_i)(\mathbf{x}_i - \boldsymbol{\mu}_i)^T] \\ &= \mathbb{E}[(\mathbf{x}_j - \boldsymbol{\mu}_j)(\mathbf{x}_j - \boldsymbol{\mu}_j)^T] \end{aligned} \tag{28}$$

Then, for a prescribed $\mathbf{C}$, the smaller the distance d_{ij} is, the more similar the vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ will be.

For the special case when $\mathbf{x}_j = \mathbf{x}_i$, $\boldsymbol{\mu}_i = \boldsymbol{\mu}_j = \boldsymbol{\mu}$, and $\mathbf{C} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix, the Mahalanobis distance reduces to the Euclidean distance between the sample vector $\mathbf{x}_i$ and the mean vector $\boldsymbol{\mu}$.

Regardless of whether the data vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ are deterministic or stochastic, Rule 1 addresses the issue of how these two vectors are *correlated* to each other. *Correlation* plays a key role not only in the human brain, but also in signal processing of various kinds (Chen et al., 2007).

> **Rule 2.** Items to be categorized as separate classes should be given widely different representations in the network.

According to Rule 1, patterns drawn from a particular class have an algebraic measure (e.g., Euclidean distance) that is small. On the other hand, patterns drawn from different classes have a large algebraic measure. We may therefore say that Rule 2 is the dual of Rule 1.

> **Rule 3.** If a particular feature is important, then there should be a large number of neurons involved in the representation of that item in the network.

Consider, for example, a radar application involving the detection of a target (e.g., aircraft) in the presence of clutter (i.e., radar reflections from undesirable targets such as buildings, trees, and weather formations). The detection performance of such a radar system is measured in terms of two probabilities:

- *probability of detection*, defined as the probability that the system decides that a target is present when it is;
- *probability of false alarm*, defined as the probability that the system decides that a target is present when it is not.

According to the *Neyman–Pearson criterion*, the probability of detection is maximized, subject to the constraint that the probability of false alarm does not exceed a prescribed value (Van Trees, 1968). In such an application, the actual presence of a target in the received signal represents an important feature of the input. Rule 3, in effect, states that there should be a large number of neurons involved in making the decision that a target is present when it actually is. By the same token, there should be a very large number of neurons involved in making the decision that the input consists of clutter only when it actually does. In both situations, the large number of neurons assures a high degree of accuracy in decision making and tolerance with respect to faulty neurons.

> **Rule 4.** Prior information and invariances should be built into the design of a neural network whenever they are available, so as to simplify the network design by its not having to learn them.

Rule 4 is particularly important because proper adherence to it results in a neural network with a *specialized structure*. This is highly desirable for several reasons:

1. Biological visual and auditory networks are known to be very specialized.

2. A neural network with specialized structure usually has a smaller number of free parameters available for adjustment than a fully connected network. Consequently, the specialized network requires a smaller data set for training, learns faster, and often generalizes better.
3. The rate of information transmission through a specialized network (i.e., the network throughput) is accelerated.
4. The cost of building a specialized network is reduced because of its smaller size, relative to that of its fully connected counterpart.

Note, however, that the incorporation of prior knowledge into the design of a neural network *restricts* application of the network to the particular problem being addressed by the knowledge of interest.

How to Build Prior Information into Neural Network Design

An important issue that has to be addressed, of course, is how to develop a specialized structure by building prior information into its design. Unfortunately, there are currently no well-defined rules for doing this; rather, we have some *ad hoc* procedures that are known to yield useful results. In particular, we may use a combination of two techniques:

1. *restricting the network architecture*, which is achieved through the use of local connections known as *receptive fields*[6];
2. *constraining the choice of synaptic weights*, which is implemented through the use of *weight-sharing*.[7]

These two techniques, particularly the latter one, have a profitable side benefit: The number of free parameters in the network could be reduced significantly.

To be specific, consider the partially connected feedforward network of Fig. 20. This network has a restricted architecture by construction. The top six source nodes constitute

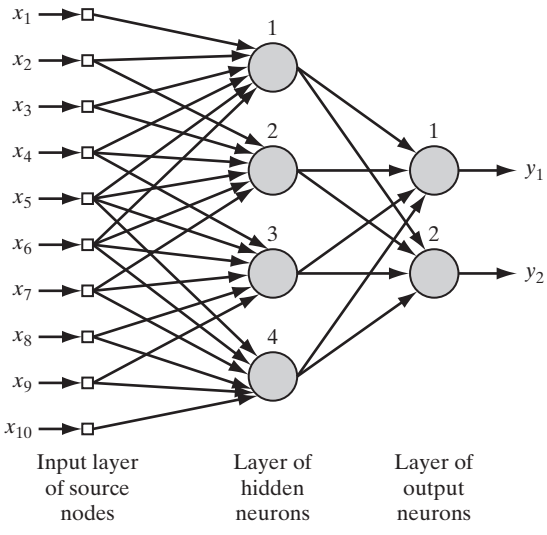


FIGURE 20 Illustrating the combined use of a receptive field and weight sharing. All four hidden neurons share the same set of weights exactly for their six synaptic connections.

the receptive field for hidden neuron 1, and so on for the other hidden neurons in the network. The *receptive* field of a neuron is defined as that region of the input field over which the incoming stimuli can influence the output signal produced by the neuron. The mapping of the receptive field is a powerful and shorthand description of the neuron's behavior, and therefore its output.

To satisfy the weight-sharing constraint, we merely have to use the same set of synaptic weights for each one of the neurons in the hidden layer of the network. Then, for the example shown in Fig. 20 with six local connections per hidden neuron and a total of four hidden neurons, we may express the induced local field of hidden neuron j as

$$v_j = \sum_{i=1}^{6} w_i x_{i+j-1}, \qquad j = 1, 2, 3, 4 \tag{29}$$

where $\{w_i\}_{i=1}^{6}$ constitutes the same set of weights shared by all four hidden neurons, and x_k is the signal picked up from source node $k = i + j - 1$. Equation (29) is in the form of a *convolution sum.* It is for this reason that a feedforward network using local connections and weight sharing in the manner described herein is referred to as a *convolutional network* (LeCun and Bengio, 2003).

The issue of building prior information into the design of a neural network pertains to one part of Rule 4; the remaining part of the rule involves the issue of invariances, which is discussed next.

How to Build Invariances into Neural Network Design

Consider the following physical phenomena:

- When an object of interest rotates, the image of the object as perceived by an observer usually changes in a corresponding way.
- In a coherent radar that provides amplitude as well as phase information about its surrounding environment, the echo from a moving target is shifted in frequency, due to the Doppler effect that arises from the radial motion of the target in relation to the radar.
- The utterance from a person may be spoken in a soft or loud voice, and in a slow or quick manner.

In order to build an object-recognition system, a radar target-recognition system, and a speech-recognition system for dealing with these phenomena, respectively, the system must be capable of coping with a range of *transformations* of the observed signal. Accordingly, a primary requirement of pattern recognition is to design a classifier that is *invariant* to such transformations. In other words, a class estimate represented by an output of the classifier must not be affected by transformations of the observed signal applied to the classifier input.

There are at least three techniques for rendering classifier-type neural networks invariant to transformations (Barnard and Casasent, 1991):

1. *Invariance by Structure.* Invariance may be imposed on a neutral network by structuring its design appropriately. Specifically, synaptic connections between the

neurons of the network are created so that transformed versions of the same input are forced to produce the same output. Consider, for example, the classification of an input image by a neural network that is required to be independent of in-plane rotations of the image about its center. We may impose rotational invariance on the network structure as follows: Let w_{ji} be the synaptic weight of neuron j connected to pixel i in the input image. If the condition $w_{ji} = w_{jk}$ is enforced for all pixels i and k that lie at equal distances from the center of the image, then the neural network is invariant to in-plane rotations. However, in order to maintain rotational invariance, the synaptic weight w_{ji} has to be duplicated for every pixel of the input image at the same radial distance from the origin. This points to a shortcoming of invariance by structure: The number of synaptic connections in the neural network becomes prohibitively large even for images of moderate size.

2. *Invariance by Training.* A neural network has a natural ability for pattern classification. This ability may be exploited directly to obtain transformation invariance as follows: The network is trained by presenting it with a number of different examples of the same object, with the examples being chosen to correspond to different transformations (i.e., different aspect views) of the object. Provided that the number of examples is sufficiently large, and if the the network is trained to learn to discriminate between the different aspect views of the object, we may then expect the network to generalize correctly to transformations other than those shown to it. However, from an engineering perspective, invariance by training has two disadvantages. First, when a neural network has been trained to recognize an object in an invariant fashion with respect to known transformations, it is not obvious that this training will also enable the network to recognize other objects of different classes invariantly. Second, the computational demand imposed on the network may be too severe to cope with, especially if the dimensionality of the feature space is high.

3. *Invariant Feature Space.* The third technique of creating an invariant classifier-type neural network is illustrated in Fig. 21. It rests on the premise that it may be possible to extract *features* that characterize the essential information content of an input data set and that are invariant to transformations of the input. If such features are used, then the network as a classifier is relieved of the burden of having to delineate the range of transformations of an object with complicated decision boundaries. Indeed, the only differences that may arise between different instances of the same object are due to unavoidable factors such as noise and occlusion. The use of an invariant-feature space offers three distinct advantages. First, the number of features applied to the network may be reduced to realistic levels. Second, the requirements imposed on network design are relaxed. Third, invariance for all objects with respect to known transformations is assured.

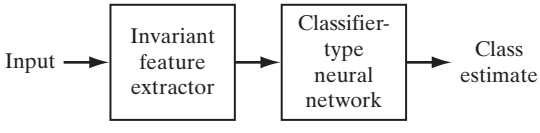


FIGURE 21 Block diagram of an invariant-feature-space type of system.

EXAMPLE 1: Autoregressive Models

To illustrate the idea of invariant-feature space, consider the example of a coherent radar system used for air surveillance, where the targets of interest include aircraft, weather systems, flocks of migrating birds, and ground objects. The radar echoes from these targets possess different spectral characteristics. Moreover, experimental studies have shown that such radar signals can be modeled fairly closely as an *autoregressive* (AR) *process* of moderate order (Haykin and Deng, 1991). An AR model is a special form of regressive model defined for complex-valued data by

$$x(n) = \sum_{i=1}^{M} a_i^* x(n - i) + e(n) \tag{30}$$

where $\{a_i\}_{i=1}^{M}$ are the *AR coefficients*, M is the *model order*, $x(n)$ is the *input*, and $e(n)$ is the *error* described as white noise. Basically, the AR model of Eq. (30) is represented by a *tapped-delay-line filter* as illustrated in Fig. 22a for $M = 2$. Equivalently, it may be represented by a *lattice filter* as shown in Fig. 22b, the coefficients of which are called *reflection coefficients*. There is a one-to-one correspondence between the AR coefficients of the model in Fig. 22a and the reflection coefficients of the model in Fig. 22b. The two models depicted here assume that the input $x(n)$ is complex valued, as in the case of a coherent radar, in which case the AR coefficients and the reflection coefficients are all complex valued. The asterisk in Eq. (30) and Fig. 22 signifies *complex conjugation*. For now, it suffices to say that the coherent radar data may be described by a set of *autoregressive coefficients*, or by a corresponding set of *reflection coefficients*. The latter set of coefficients has

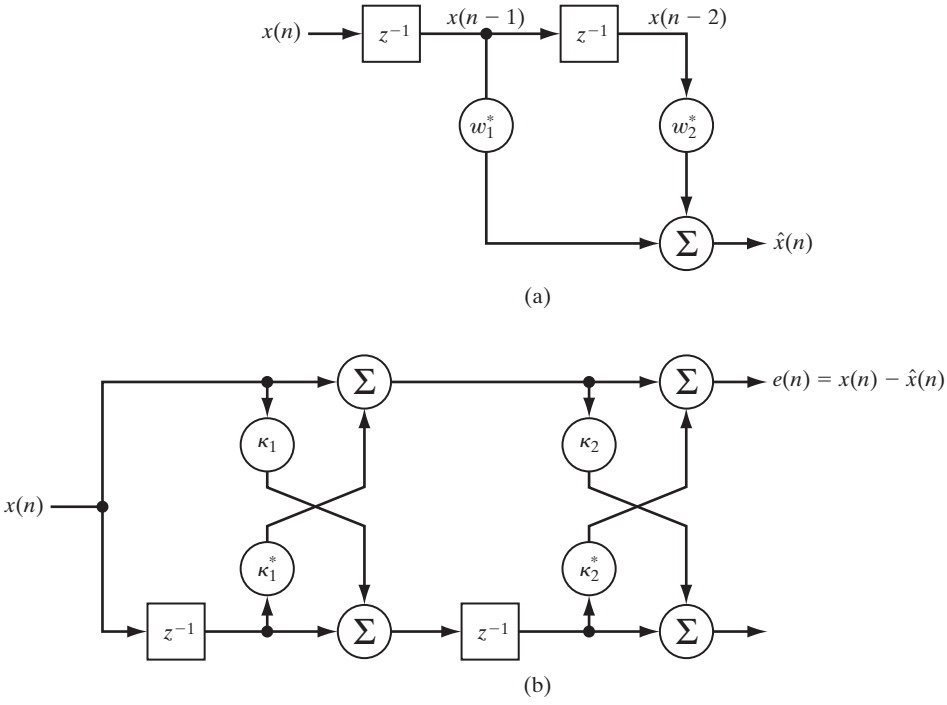


FIGURE 22 Autoregressive model of order 2: (a) tapped-delay-line model; (b) lattice-filter model. (The asterisk denotes complex conjugation.)

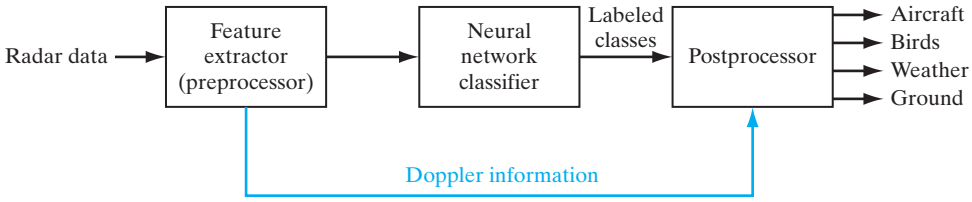


FIGURE 23 Doppler-shift-invariant classifier of radar signals.

a computational advantage in that efficient algorithms exist for their computation directly from the input data. The feature extraction problem, however, is complicated by the fact that moving objects produce varying Doppler frequencies that depend on their radial velocities measured with respect to the radar, and that tend to obscure the spectral content of the reflection coefficients as feature discriminants. To overcome this difficulty, we must build *Doppler invariance* into the computation of the reflection coefficients. The phase angle of the first reflection coefficient turns out to be equal to the Doppler frequency of the radar signal. Accordingly, Doppler frequency *normalization* is applied to all coefficients so as to remove the mean Doppler shift. This is done by defining a new set of reflection coefficients $\{\kappa'_m\}$ related to the set of ordinary reflection coefficients $\{\kappa_m\}$ computed from the input data as:

$$\kappa'_m = \kappa_m e^{-jm\theta} \qquad \text{for } m = 1, 2, ..., M \tag{31}$$

where θ is the phase angle of the first reflection coefficient. The operation described in Eq. (31) is referred to as *heterodyning*. A set of *Doppler-invariant radar features* is thus represented by the normalized reflection coefficients $\kappa'_1, \kappa'_2, ..., \kappa'_M$, with κ'_1 being the only real-valued coefficient in the set. As mentioned previously, the major categories of radar targets of interest in air surveillance are weather, birds, aircraft, and ground. The first three targets are moving, whereas the last one is not. The *heterodyned* spectral parameters of radar echoes from ground have echoes similar in characteristic to those from aircraft. A ground echo can be discriminated from an aircraft echo because of its small Doppler shift. Accordingly, the radar classifier includes a postprocessor as shown in Fig. 23, which operates on the classified results (encoded labels) for the purpose of identifying the ground class (Haykin and Deng, 1991). Thus, the *preprocessor* in Fig. 23 takes care of Doppler-shift-invariant feature extraction at the classifier input, whereas the *postprocessor* uses the stored Doppler signature to distinguish between aircraft and ground returns. ■

EXAMPLE 2: Echolocating Bat

A much more fascinating example of knowledge representation in a neural network is found in the biological sonar system of echolocating bats. Most bats use *frequency-modulated* (FM, or "chirp") signals for the purpose of acoustic imaging; in an FM signal, the instantaneous frequency of the signal varies with time. Specifically, the bat uses its mouth to broadcast short-duration FM sonar signals and uses its auditory system as the sonar receiver. Echoes from targets of interest are represented in the auditory system by the activity of neurons that are selective to different combinations of acoustic parameters. There are three principal neural dimensions of the bat's auditory representation (Simmons et al., 1992):

- *Echo frequency*, which is encoded by "place" originating in the frequency map of the cochlea; it is preserved throughout the entire auditory pathway as an orderly arrangement across certain neurons tuned to different frequencies.

- *Echo amplitude*, which is encoded by other neurons with different dynamic ranges; it is manifested both as amplitude tuning and as the number of discharges per stimulus.
- *Echo delay*, which is encoded through neural computations (based on cross-correlation) that produce delay-selective responses; it is manifested as target-range tuning.

The two principal characteristics of a target echo for image-forming purposes are *spectrum* for target shape and *delay* for target range. The bat perceives "shape" in terms of the arrival time of echoes from different reflecting surfaces (glints) within the target. For this to occur, *frequency* information in the echo spectrum is converted into estimates of the *time* structure of the target. Experiments conducted by Simmons and coworkers on the big brown bat, *Eptesicus fuscus,* critically identify this conversion process as consisting of parallel time-domain and frequency-to-time-domain transforms whose converging outputs create the common delay of range axis of a perceived image of the target. It appears that the unity of the bat's perception is due to certain properties of the transforms themselves, despite the separate ways in which the auditory time representation of the echo delay and frequency representation of the echo spectrum are initially performed. Moreover, feature invariances are built into the sonar image-forming process so as to make it essentially independent of the target's motion and the bat's own motion. ■

Some Final Remarks

The issue of knowledge representation in a neural network is directly related to that of network architecture. Unfortunately, there is no well-developed theory for optimizing the architecture of a neural network required to interact with an environment of interest, or for evaluating the way in which changes in the network architecture affect the representation of knowledge inside the network. Indeed, satisfactory answers to these issues are usually found through an exhaustive experimental study for a specific application of interest, with the designer of the neural network becoming an essential part of the structural learning loop.

8 LEARNING PROCESSES

Just as there are different ways in which we ourselves learn from our own surrounding environments, so it is with neural networks. In a broad sense, we may categorize the learning processes through which neural networks function as follows: learning with a teacher and learning without a teacher. By the same token, the latter form of learning may be subcategorized into unsupervised learning and reinforcement learning. These different forms of learning as performed on neural networks parallel those of human learning.

Learning with a Teacher

Learning with a teacher is also referred to as *supervised learning.* Figure 24 shows a block diagram that illustrates this form of learning. In conceptual terms, we may think of the teacher as having knowledge of the environment, with that knowledge being represented by a set of *input–output examples.* The environment is, however, *unknown* to the neural network. Suppose now that the teacher and the neural network are both exposed to a training vector (i.e., example) drawn from the same environment. By virtue of built-in

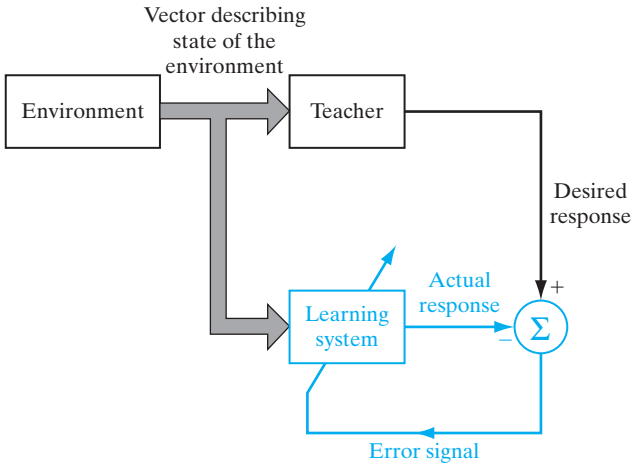


FIGURE 24 Block diagram of learning with a teacher; the part of the figure printed in red constitutes a feedback loop.

knowledge, the teacher is able to provide the neural network with a desired response for that training vector. Indeed, the desired response represents the "optimum" action to be performed by the neural network. The network parameters are adjusted under the combined influence of the training vector and the error signal. The *error signal* is defined as the difference between the desired response and the actual response of the network. This adjustment is carried out iteratively in a step-by-step fashion with the aim of eventually making the neural network *emulate* the teacher; the emulation is presumed to be optimum in some statistical sense. In this way, knowledge of the environment available to the teacher is transferred to the neural network through training and stored in the form of "fixed" synaptic weights, representing *long-term memory*. When this condition is reached, we may then dispense with the teacher and let the neural network deal with the environment completely by itself.

The form of supervised learning we have just described is the basis of *error-correction learning*. From Fig. 24, we see that the supervised-learning process constitutes a closed-loop feedback system, but the unknown environment is outside the loop. As a performance measure for the system, we may think in terms of the *mean-square error*, or the *sum of squared errors* over the training sample, defined as a function of the free parameters (i.e., synaptic weights) of the system. This function may be visualized as a multidimensional *error-performance surface*, or simply *error surface*, with the free parameters as coordinates. The true error surface is *averaged* over all possible input–output examples. Any given operation of the system under the teacher's supervision is represented as a point on the error surface. For the system to improve performance over time and therefore learn from the teacher, the operating point has to move down successively toward a minimum point of the error surface; the minimum point may be a local minimum or a global minimum. A supervised learning system is able to do this with the useful information it has about the *gradient* of the error surface corresponding to the current behavior of the system. The gradient

of the error surface at any point is a vector that points in the direction of *steepest descent.* In fact, in the case of supervised learning from examples, the system may use an *instantaneous estimate* of the gradient vector, with the example indices presumed to be those of time. The use of such an estimate results in a motion of the operating point on the error surface that is typically in the form of a "random walk." Nevertheless, given an algorithm designed to minimize the cost function, an adequate set of input–output examples, and enough time in which to do the training, a supervised learning system is usually able to approximate an unknown input–output mapping reasonably well.

Learning without a Teacher

In supervised learning, the learning process takes place under the tutelage of a teacher. However, in the paradigm known as *learning without a teacher*, as the name implies, there is *no teacher* to oversee the learning process. That is to say, there are no labeled examples of the function to be learned by the network. Under this second paradigm, two subcategories are identified:

1. Reinforcement Learning

In *reinforcement learning*, the learning of an input–output mapping is performed through continued interaction with the environment in order to minimize a scalar index of performance. Figure 25 shows the block diagram of one form of a reinforcement-learning system built around a *critic* that converts a *primary reinforcement signal* received from the environment into a higher quality reinforcement signal called the *heuristic reinforcement signal*, both of which are scalar inputs (Barto et al., 1983). The system is designed to learn under *delayed reinforcement*, which means that the system observes a temporal sequence of stimuli also received from the environment, which eventually result in the generation of the heuristic reinforcement signal.

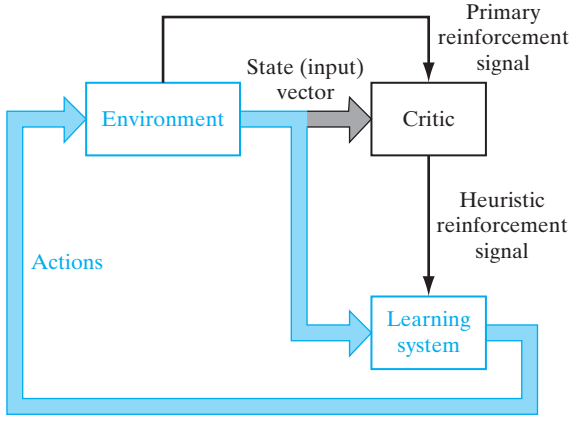


FIGURE 25 Block diagram of reinforcement learning; the learning system and the environment are both inside the feedback loop.

The goal of reinforcement learning is to minimize a *cost-to-go function*, defined as the expectation of the cumulative cost of *actions* taken over a sequence of steps instead of simply the immediate cost. It may turn out that certain actions taken earlier in that sequence of time steps are in fact the best determinants of overall system behavior. The function of the *learning system* is to *discover* these actions and feed them back to the environment.

Delayed-reinforcement learning is difficult to perform for two basic reasons:

- There is no teacher to provide a desired response at each step of the learning process.
- The delay incurred in the generation of the primary reinforcement signal implies that the learning machine must solve a *temporal credit assignment problem.* By this we mean that the learning machine must be able to assign credit and blame individually to each action in the sequence of time steps that led to the final outcome, while the primary reinforcement may only evaluate the outcome.

Notwithstanding these difficulties, delayed-reinforcement learning is appealing. It provides the basis for the learning system to interact with its environment, thereby developing the ability to learn to perform a prescribed task solely on the basis of the outcomes of its experience that result from the interaction.

2. Unsupervised Learning

In *unsupervised*, or *self-organized*, *learning*, there is no external teacher or critic to oversee the learning process, as indicated in Fig. 26. Rather, provision is made for a *task-independent measure* of the quality of representation that the network is required to learn, and the free parameters of the network are optimized with respect to that measure. For a specific task-independent measure, once the network has become tuned to the statistical regularities of the input data, the network develops the ability to form internal representations for encoding features of the input and thereby to create new classes automatically (Becker, 1991).

To perform unsupervised learning, we may use a competitive-learning rule. For example, we may use a neural network that consists of two layers—an input layer and a competitive layer. The input layer receives the available data. The competitive layer consists of neurons that compete with each other (in accordance with a learning rule) for the "opportunity" to respond to features contained in the input data. In its simplest form, the network operates in accordance with a "winner-takes-all" strategy. In such a strategy, the neuron with the greatest total input "wins" the competition and turns on; all the other neurons in the network then switch off.

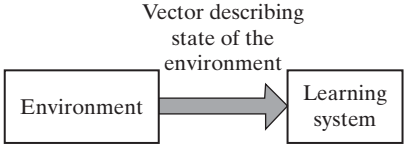


FIGURE 26 Block diagram of unsupervised learning.

9 LEARNING TASKS

In the previous section, we discussed different learning paradigms. In this section, we describe some basic learning tasks. The choice of a particular learning rule, is of course, influenced by the learning task, the diverse nature of which is testimony to the universality of neural networks.

Pattern Association

An *associative memory* is a brainlike distributed memory that learns by *association.* Association has been known to be a prominent feature of human memory since the time of Aristotle, and all models of cognition use association in one form or another as the basic operation (Anderson, 1995).

Association takes one of two forms: *autoassociation* and *heteroassociation.* In autoassociation, a neural network is required to *store* a set of patterns (vectors) by repeatedly presenting them to the network. The network is subsequently presented with a partial description or distorted (noisy) version of an original pattern stored in it, and the task is to *retrieve* (*recall*) that particular pattern. Heteroassociation differs from autoassociation in that an arbitrary set of input patterns is *paired* with another arbitrary set of output patterns. Autoassociation involves the use of unsupervised learning, whereas the type of learning involved in heteroassociation is supervised.

Let $\mathbf{x}_k$ denote a *key pattern* (vector) applied to an associative memory and $\mathbf{y}_k$ denote a *memorized pattern* (vector). The pattern association performed by the network is described by

$$\mathbf{x}_k \rightarrow \mathbf{y}_k, \qquad k = 1, 2, ..., q \tag{32}$$

where q is the number of patterns stored in the network. The key pattern $\mathbf{x}_k$ acts as a stimulus that not only determines the storage location of memorized pattern $\mathbf{y}_k$, but also holds the key for its retrieval.

In an autoassociative memory, $\mathbf{y}_k = \mathbf{x}_k$, so the input and output (data) spaces of the network have the same dimensionality. In a heteroassociative memory, $\mathbf{y}_k \neq \mathbf{x}_k$; hence, the dimensionality of the output space in this second case may or may not equal the dimensionality of the input space.

There are two phases involved in the operation of an associative memory:

- *storage phase*, which refers to the training of the network in accordance with Eq. (32);
- *recall phase*, which involves the retrieval of a memorized pattern in response to the presentation of a noisy or distorted version of a key pattern to the network.

Let the stimulus (input) $\mathbf{x}$ represent a noisy or distorted version of a key pattern $\mathbf{x}_j$. This stimulus produces a response (output) $\mathbf{y}$, as indicated in Fig. 27. For perfect recall, we should find that $\mathbf{y} = \mathbf{y}_j$, where $\mathbf{y}_j$ is the memorized pattern associated with the key pattern $\mathbf{x}_j$. When $\mathbf{y} \neq \mathbf{y}_j$ for $\mathbf{x} = \mathbf{x}_j$, the associative memory is said to have made an *error in recall.*

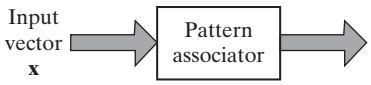


FIGURE 27 Input–output relation of pattern associator.

The number of patterns q stored in an associative memory provides a direct measure of the *storage capacity* of the network. In designing an associative memory, the challenge is to make the storage capacity q (expressed as a percentage of the total number N of neurons used to construct the network) as large as possible, yet insist that a large fraction of the memorized patterns is recalled correctly.

Pattern Recognition

Humans are good at pattern recognition. We receive data from the world around us via our senses and are able to recognize the source of the data. We are often able to do so almost immediately and with practically no effort. For example, we can recognize the familiar face of a person even though that person has aged since our last encounter, identify a familiar person by his or her voice on the telephone despite a bad connection, and distinguish a boiled egg that is good from a bad one by smelling it. Humans perform pattern recognition through a learning process; so it is with neural networks.

Pattern recognition is formally defined as *the process whereby a received pattern/signal is assigned to one of a prescribed number of classes.* A neural network performs pattern recognition by first undergoing a training session during which the network is repeatedly presented with a set of input patterns along with the category to which each particular pattern belongs. Later, the network is presented with a new pattern that has not been seen before, but which belongs to the same population of patterns used to train the network. The network is able to identify the class of that particular pattern because of the information it has extracted from the training data. Pattern recognition performed by a neural network is statistical in nature, with the patterns being represented by points in a multidimensional *decision space.* The decision space is divided into regions, each one of which is associated with a class. The decision boundaries are determined by the training process. The construction of these boundaries is made statistical by the inherent variability that exists within and between classes.

In generic terms, pattern-recognition machines using neural networks may take one of two forms:

- The machine is split into two parts, an unsupervised network for *feature extraction* and a supervised network for *classification*, as shown in the hybridized system of Fig. 28a. Such a method follows the traditional approach to statistical pattern recognition (Fukunaga, 1990; Duda et al., 2001; Theodoridis and Koutroumbas, 2003). In conceptual terms, a pattern is represented by a set of m observables, which may be viewed as a point $\mathbf{x}$ in an m-dimensional *observation (data) space.*

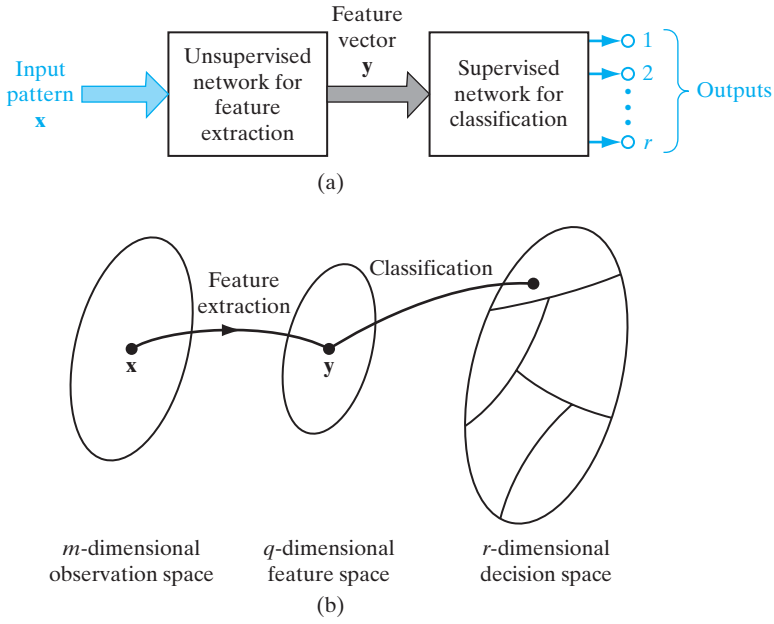


FIGURE 28 Illustration of the classical approach to pattern classification.

Feature extraction is described by a transformation that maps the point $\mathbf{x}$ into an intermediate point $\mathbf{y}$ in a q-dimensional *feature space* with $q < m$, as indicated in Fig. 28b. This transformation may be viewed as one of dimensionality reduction (i.e., data compression), the use of which is justified on the grounds that it simplifies the task of classification. The classification is itself described as a transformation that maps the intermediate point $\mathbf{y}$ into one of the classes in an r-dimensional decision space, where r is the number of classes to be distinguished.

- The machine is designed as a feedforward network using a supervised learning algorithm. In this second approach, the task of feature extraction is performed by the computational units in the hidden layer(s) of the network.

Function Approximation

The third learning task of interest is that of function approximation. Consider a nonlinear input–output mapping described by the functional relationship

$$\mathbf{d} = \mathbf{f}(\mathbf{x}) \tag{33}$$

where the vector $\mathbf{x}$ is the input and the vector $\mathbf{d}$ is the output. The vector-valued function $\mathbf{f}(\cdot)$ is assumed to be unknown. To make up for the lack of knowledge about the function $\mathbf{f}(\cdot)$, we are given the set of labeled examples:

$$\mathcal{T} = \{(\mathbf{x}_i, \mathbf{d}_i)\}_{i=1}^{N} \tag{34}$$

The requirement is to design a neural network that approximates the unknown function $\mathbf{f}(\cdot)$ such that the function $\mathbf{F}(\cdot)$ describing the input–output mapping actually realized by the network, is close enough to $\mathbf{f}(\cdot)$ in a Euclidean sense over all inputs, as shown by

$$\|\mathbf{F}(\mathbf{x}) - \mathbf{f}(\mathbf{x})\| < \varepsilon \quad \text{for all } \mathbf{x} \tag{35}$$

where ε is a small positive number. Provided that the size N of the training sample $\mathcal{T}$ is large enough and the network is equipped with an adequate number of free parameters, then the approximation error ε can be made small enough for the task.

The approximation problem described here is a perfect candidate for supervised learning, with $\mathbf{x}_i$ playing the role of input vector and $\mathbf{d}_i$ serving the role of desired response. We may turn this issue around and view supervised learning as an approximation problem.

The ability of a neural network to approximate an unknown input–output mapping may be exploited in two important ways:

(i) *System identification.* Let Eq. (33) describe the input–output relation of an unknown memoryless *multiple input–multiple output (MIMO) system*; by a "memoryless" system, we mean a system that is time invariant. We may then use the set of labeled examples in Eq. (34) to train a neural network as a model of the system. Let the vector $\mathbf{y}_i$ denote the actual output of the neural network produced in response to an input vector $\mathbf{x}_i$. The difference between $\mathbf{d}_i$ (associated with $\mathbf{x}_i$) and the network output $\mathbf{y}_i$ provides the error signal vector $\mathbf{e}_i$, as depicted in Fig. 29. This error signal is, in turn, used to adjust the free parameters of the network to minimize the squared difference between the outputs of the unknown system and the neural network in a statistical sense, and is computed over the entire training sample $\mathcal{T}$.

(ii) *Inverse modeling.* Suppose next we are given a known memoryless MIMO system whose input–output relation is described by Eq. (33). The requirement in this case is to construct an *inverse model* that produces the vector $\mathbf{x}$ in response to the vector $\mathbf{d}$. The inverse system may thus be described by

$$\mathbf{x} = \mathbf{f}^{-1}(\mathbf{d}) \tag{36}$$

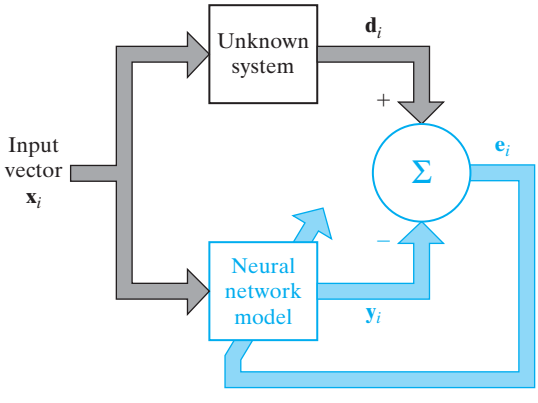


FIGURE 29 Block diagram of system identification: The neural network, doing the identification, is part of the feedback loop.

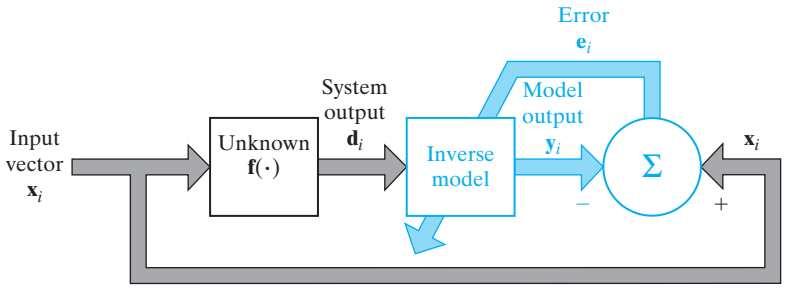


FIGURE 30 Block diagram of inverse system modeling. The neural network, acting as the inverse model, is part of the feedback loop.

where the vector-valued function $\mathbf{f}^{-1}(\cdot)$ denotes the inverse of $\mathbf{f}(\cdot)$. Note, however, that $\mathbf{f}^{-1}(\cdot)$ is not the reciprocal of $\mathbf{f}(\cdot)$; rather, the use of superscript -1 is merely a flag to indicate an inverse. In many situations encountered in practice, the vector-valued function $\mathbf{f}(\cdot)$ is much too complex and inhibits a straightforward formulation of the inverse function $\mathbf{f}^{-1}(\cdot)$. Given the set of labeled examples in Eq. (34), we may construct a neural network approximation of $\mathbf{f}^{-1}(\cdot)$ by using the scheme shown in Fig. 30. In the situation described here, the roles of $\mathbf{x}_i$ and $\mathbf{d}_i$ are interchanged: The vector $\mathbf{d}_i$ is used as the input, and $\mathbf{x}_i$ is treated as the desired response. Let the error signal vector $\mathbf{e}_i$ denote the difference between $\mathbf{x}_i$ and the actual output $\mathbf{y}_i$ of the neural network produced in response to $\mathbf{d}_i$. As with the system identification problem, this error signal vector is used to adjust the free parameters of the neural network to minimize the squared difference between the outputs of the unknown inverse system and the neural network in a statistical sense, and is computed over the complete training set $\mathcal{T}$. Typically, inverse modeling is a more difficult learning task than system identification, as there may not be a unique solution for it.

Control

The control of a *plant* is another learning task that is well suited for neural networks; by a "plant" we mean a process or critical part of a system that is to be maintained in a controlled condition. The relevance of learning to control should not be surprising because, after all, the human brain is a computer (i.e., information processor), the outputs of which as a whole system are *actions*. In the context of control, the brain is living proof that it is possible to build a generalized controller that takes full advantage of parallel distributed hardware, can control many thousands of actuators (muscle fibers) in parallel, can handle nonlinearity and noise, and can optimize over a long-range planning horizon (Werbos, 1992).

Consider the *feedback control system* shown in Fig. 31. The system involves the use of unity feedback around a plant to be controlled; that is, the plant output is fed back directly to the input. Thus, the plant output $\mathbf{y}$ is subtracted from a *reference signal* $\mathbf{d}$ supplied from an external source. The error signal $\mathbf{e}$ so produced is applied to a neural *controller* for the purpose of adjusting its free parameters. The primary objective of the controller is to supply appropriate inputs to the plant to make its output $\mathbf{y}$ track the

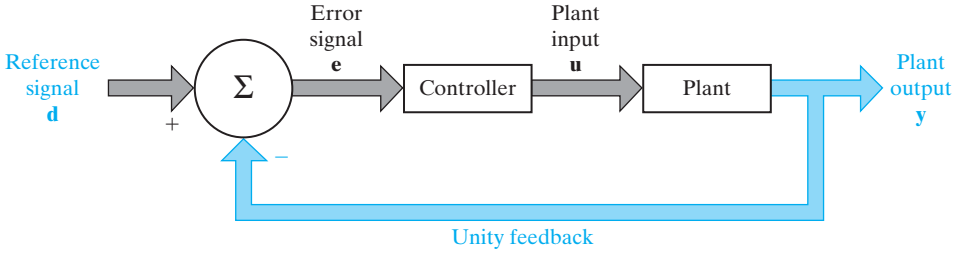


FIGURE 31 Block diagram of feedback control system.

reference signal **d**. In other words, the controller has to invert the plant's input–output behavior.

We note that in Fig. 31, the error signal **e** has to propagate through the neural controller before reaching the plant. Consequently, to perform adjustments on the free parameters of the plant in accordance with an error-correction learning algorithm, we need to know the *Jacobian,* made up of a matrix of partial derivatives as shown by

$$\mathbf{J} = \left\{ \frac{\partial y_k}{\partial u_j} \right\}_{j,k} \tag{37}$$

where y_k is an element of the plant output **y** and u_j is an element of the plant input **u**. Unfortunately, the partial derivatives $\partial y_k / \partial u_j$ for the various k and j depend on the operating point of the plant and are therefore not known. We may use one of two approaches to account for them:

(i) *Indirect learning.* Using actual input–output measurements on the plant, we first construct a neural model to produce a copy of it. This model is, in turn, used to provide an estimate of the Jacobian **J**. The partial derivatives constituting this Jacobian are subsequently used in the error-correction learning algorithm for computing the adjustments to the free parameters of the neural controller (Nguyen and Widrow, 1989; Suykens et al., 1996; Widrow and Walach, 1996).

(ii) *Direct learning.* The signs of the partial derivatives $\partial y_k / \partial u_j$ are generally known and usually remain constant over the dynamic range of the plant. This suggests that we may approximate these partial derivatives by their individual signs. Their absolute values are given a distributed representation in the free parameters of the neural controller (Saerens and Soquet, 1991; Schiffman and Geffers, 1993). The neural controller is thereby enabled to learn the adjustments to its free parameters directly from the plant.

Beamforming

Beamforming is used to distinguish between the spatial properties of a target signal and background noise. The device used to do the beamforming is called a *beamformer.*

The task of beamforming is compatible, for example, with feature mapping in the cortical layers of auditory systems of echolocating bats (Suga, 1990a; Simmons et al.,

1992). The echolocating bat illuminates the surrounding environment by broadcasting short-duration frequency-modulated (FM) sonar signals and then uses its auditory system (including a pair of ears) to focus attention on its prey (e.g., flying insect). The ears provide the bat with a beamforming capability that is exploited by the auditory system to produce *attentional selectivity*.

Beamforming is commonly used in radar and sonar systems where the primary task is to detect and track a target of interest in the combined presence of receiver noise and interfering signals (e.g., jammers). This task is complicated by two factors:

- the target signal originates from an unknown direction, and
- there is no *prior* information available on the interfering signals.

One way of coping with situations of this kind is to use a *generalized sidelobe canceller* (GSLC), the block diagram of which is shown in Fig. 32. The system consists of the following components (Griffiths and Jim, 1982; Haykin, 2002):

- An *array of antenna elements*, which provides a means of sampling the observation-space signal at discrete points in space.
- A *linear combiner* defined by a set of fixed weights $\{w_i\}_{i=1}^{m}$, the output of which performs the role of a desired response. This linear combiner acts like a "spatial filter," characterized by a radiation pattern (i.e., a polar plot of the amplitude of the antenna output versus the incidence angle of an incoming signal). The mainlobe of this radiation pattern is pointed along a prescribed direction, for which the GSLC is *constrained* to produce a distortionless response. The output of the linear combiner, denoted by $d(n)$, provides a desired response for the beamformer.
- A *signal-blocking matrix* $\mathbf{C}_a$, the function of which is to cancel interference that leaks through the sidelobes of the radiation pattern of the spatial filter representing the linear combiner.

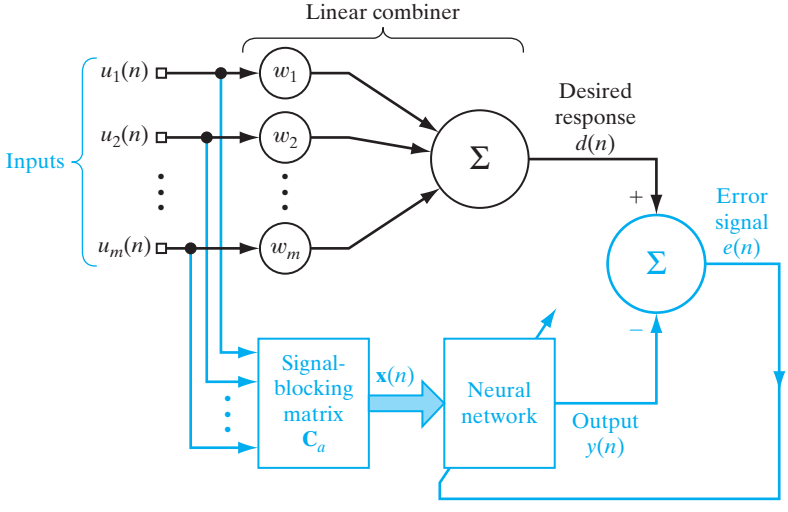


FIGURE 32 Block diagram of generalized sidelobe canceller.

- A *neural network* with adjustable parameters, which is designed to accommodate statistical variations in the interfering signals.

The adjustments to the free parameters of the neural network are performed by an error-correcting learning algorithm that operates on the error signal $e(n)$, defined as the difference between the linear combiner output $d(n)$ and the actual output $y(n)$ of the neural network. Thus the GSLC operates under the supervision of the linear combiner that assumes the role of a "teacher." As with ordinary supervised learning, notice that the linear combiner is outside the feedback loop acting on the neural network. A beamformer that uses a neural network for learning is called a *neuro-beamformer.* This class of learning machines comes under the general heading of *attentional neurocomputers* (Hecht-Nielsen, 1990).

10 CONCLUDING REMARKS

In the material covered in this introductory chapter, we have focused attention on neural networks, the study of which is motivated by the human brain. The one important property of neural networks that stands out is that of *learning*, which is categorized as follows:

(i) *supervised learning*, which requires the availability of a target or desired response for the realization of a specific input–output mapping by minimizing a cost function of interest;

(ii) *unsupervised learning*, the implementation of which relies on the provision of a task-independent measure of the quality of representation that the network is required to learn in a self-organized manner;

(iii) *reinforcement learning*, in which input–output mapping is performed through the continued interaction of a learning system with its environment so as to minimize a scalar index of performance.

Supervised learning relies on the availability of a training sample of *labeled examples*, with each example consisting of an input signal (stimulus) and the corresponding desired (target) response. In practice, we find that the collection of labeled examples is a time-consuming and expensive task, especially when we are dealing with large-scale learning problems; typically, we therefore find that labeled examples are in short supply. On the other hand, unsupervised learning relies solely on unlabeled examples, consisting simply of a set of input signals or stimuli, for which there is usually a plentiful supply. In light of these realities, there is a great deal of interest in another category of learning: *semisupervised learning*, which employs a training sample that consists of labeled as well as unlabeled examples. The challenge in semisupervised learning, discussed in a subsequent chapter, is to design a learning system that scales reasonably well for its implementation to be practically feasible when dealing with large-scale pattern-classification problems.

Reinforcement learning lies between supervised learning and unsupervised learning. It operates through continuing interactions between a learning system (agent) and the environment. The learning system performs an action and learns from

the response of the environment to that action. In effect, the role of the teacher in supervised learning is replaced by a critic, for example, that is integrated into the learning machinery.

NOTES AND REFERENCES

1. This definition of a neural network is adapted from Aleksander and Morton (1990).
2. For a readable account of computational aspects of the brain, see Churchland and Sejnowski (1992). For more detailed descriptions, see Kandel et al. (1991), Shepherd (1990), Kuffler et al. (1984), and Freeman (1975).
3. For detailed treatment of spikes and spiking neurons, see Rieke et al. (1997). For a biophysical perspective of computation and information-processing capability of single neurons, see Koch (1999).
4. For a thorough account of sigmoid functions and related issues, see Mennon et al. (1996).
5. The logistic function, or more precisely, the *logistic distribution function*, derives its name from a transcendental "law of logistic growth" that has a huge literature. Measured in appropriate units, all growth processes are supposed to be represented by the logistic distribution function

$$F(t) = \frac{1}{1 + e^{\alpha t - \beta}}$$

   where t represents time, and α and β are constants.
6. According to Kuffler et al. (1984), the term "receptive field" was coined originally by Sherrington (1906) and reintroduced by Hartline (1940). In the context of a visual system, the receptive field of a neuron refers to the restricted area on the retinal surface, which influences the discharges of that neuron due to light.
7. The weight-sharing technique was originally described in Rumelhart et al. (1986b).

CHAPTER 1

Rosenblatt's Perceptron

ORGANIZATION OF THE CHAPTER

The perceptron occupies a special place in the historical development of neural networks: It was the first algorithmically described neural network. Its invention by Rosenblatt, a psychologist, inspired engineers, physicists, and mathematicians alike to devote their research effort to different aspects of neural networks in the 1960s and the 1970s. Moreover, it is truly remarkable to find that the perceptron (in its basic form as described in this chapter) is as valid today as it was in 1958 when Rosenblatt's paper on the perceptron was first published.

The chapter is organized as follows:

1. Section 1.1 expands on the formative years of neural networks, going back to the pioneering work of McCulloch and Pitts in 1943.
2. Section 1.2 describes Rosenblatt's perceptron in its most basic form. It is followed by Section 1.3 on the perceptron convergence theorem. This theorem proves convergence of the perceptron as a linearly separable pattern classifier in a finite number time-steps.
3. Section 1.4 establishes the relationship between the perceptron and the Bayes classifier for a Gaussian environment.
4. The experiment presented in Section 1.5 demonstrates the pattern-classification capability of the perceptron.
5. Section 1.6 generalizes the discussion by introducing the perceptron cost function, paving the way for deriving the batch version of the perceptron convergence algorithm.

Section 1.7 provides a summary and discussion that conclude the chapter.

1.1 INTRODUCTION

In the formative years of neural networks (1943–1958), several researchers stand out for their pioneering contributions:

- McCulloch and Pitts (1943) for introducing the idea of neural networks as computing machines.

- Hebb (1949) for postulating the first rule for self-organized learning.
- Rosenblatt (1958) for proposing the perceptron as the first model for learning with a teacher (i.e., supervised learning).

The idea of Hebbian learning will be discussed at some length in Chapter 8. In this chapter, we discuss Rosenblatt's *perceptron*.

The perceptron is the simplest form of a neural network used for the classification of patterns said to be *linearly separable* (i.e., patterns that lie on opposite sides of a hyperplane). Basically, it consists of a single neuron with adjustable synaptic weights and bias. The algorithm used to adjust the free parameters of this neural network first appeared in a learning procedure developed by Rosenblatt (1958, 1962) for his perceptron brain model.[1] Indeed, Rosenblatt proved that if the patterns (vectors) used to train the perceptron are drawn from two linearly separable classes, then the perceptron algorithm converges and positions the decision surface in the form of a hyperplane between the two classes. The proof of convergence of the algorithm is known as the *perceptron convergence theorem*.

The perceptron built around a *single neuron* is limited to performing pattern classification with only two classes (hypotheses). By expanding the output (computation) layer of the perceptron to include more than one neuron, we may correspondingly perform classification with more than two classes. However, the classes have to be linearly separable for the perceptron to work properly. The important point is that insofar as the basic theory of the perceptron as a pattern classifier is concerned, we need consider only the case of a single neuron. The extension of the theory to the case of more than one neuron is trivial.

1.2 PERCEPTRON

Rosenblatt's perceptron is built around a nonlinear neuron, namely, the *McCulloch–Pitts model* of a neuron. From the introductory chapter we recall that such a neural modeling consists of a linear combiner followed by a hard limiter (performing the signum function), as depicted in Fig. 1.1. The summing node of the neural model computes a linear combination of the inputs applied to its synapses, as well as incorporates an externally applied bias. The resulting sum, that is, the induced local field, is applied to a hard

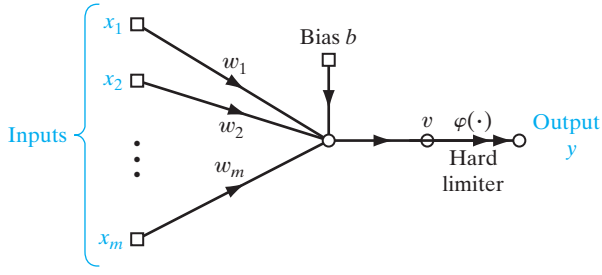


FIGURE 1.1 Signal-flow graph of the perceptron.

limiter. Accordingly, the neuron produces an output equal to +1 if the hard limiter input is positive, and −1 if it is negative.

In the signal-flow graph model of Fig. 1.1, the synaptic weights of the perceptron are denoted by $w_1, w_2, \ldots, w_m$. Correspondingly, the inputs applied to the perceptron are denoted by $x_1, x_2, \ldots, x_m$. The externally applied bias is denoted by b. From the model, we find that the hard limiter input, or induced local field, of the neuron is

$$v = \sum_{i=1}^{m} w_i x_i + b \tag{1.1}$$

The goal of the perceptron is to correctly classify the set of externally applied stimuli $x_1, x_2, \ldots, x_m$ into one of two classes, $\mathscr{C}_1$ or $\mathscr{C}_2$. The decision rule for the classification is to assign the point represented by the inputs $x_1, x_2, \ldots, x_m$ to class $\mathscr{C}_1$ if the perceptron output y is +1 and to class $\mathscr{C}_2$ if it is −1.

To develop insight into the behavior of a pattern classifier, it is customary to plot a map of the decision regions in the m-dimensional signal space spanned by the m input variables $x_1, x_2, \ldots, x_m$. In the simplest form of the perceptron, there are two decision regions separated by a *hyperplane*, which is defined by

$$\sum_{i=1}^{m} w_i x_i + b = 0 \tag{1.2}$$

This is illustrated in Fig. 1.2 for the case of two input variables x_1 and x_2, for which the decision boundary takes the form of a straight line. A point (x_1, x_2) that lies above the boundary line is assigned to class $\mathscr{C}_1$, and a point (x_1, x_2) that lies below the boundary line is assigned to class $\mathscr{C}_2$. Note also that the effect of the bias b is merely to shift the decision boundary away from the origin.

The synaptic weights $w_1, w_2, \ldots, w_m$ of the perceptron can be adapted on an iteration-by-iteration basis. For the adaptation, we may use an error-correction rule known as the perceptron convergence algorithm, discussed next.

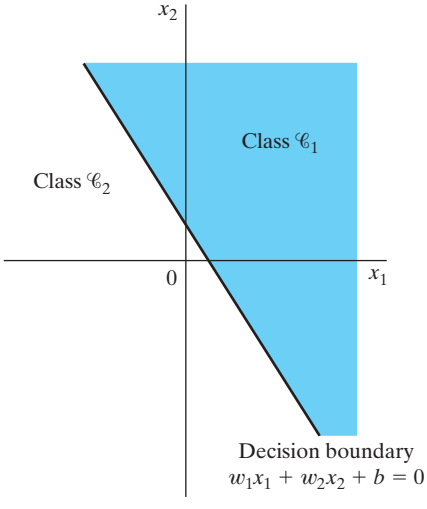


FIGURE 1.2 Illustration of the hyperplane (in this example, a straight line) as decision boundary for a two-dimensional, two-class pattern-classification problem.

1.3 THE PERCEPTRON CONVERGENCE THEOREM

To derive the error-correction learning algorithm for the perceptron, we find it more convenient to work with the modified signal-flow graph model in Fig. 1.3. In this second model, which is equivalent to that of Fig. 1.1, the bias $b(n)$ is treated as a synaptic weight driven by a fixed input equal to +1. We may thus define the $(m + 1)$-by-1 input vector

$$\mathbf{x}(n) = [+1, x_1(n), x_2(n), ..., x_m(n)]^T$$

where n denotes the time-step in applying the algorithm. Correspondingly, we define the $(m + 1)$-by-1 weight vector as

$$\mathbf{w}(n) = [b, w_1(n), w_2(n), ..., w_m(n)]^T$$

Accordingly, the linear combiner output is written in the compact form

$$\begin{aligned} v(n) &= \sum_{i=0}^{m} w_i(n)x_i(n) \\ &= \mathbf{w}^T(n)\mathbf{x}(n) \end{aligned} \tag{1.3}$$

where, in the first line, $w_0(n)$, corresponding to $i = 0$, represents the bias b. For fixed n, the equation $\mathbf{w}^T\mathbf{x} = 0$, plotted in an m-dimensional space (and for some prescribed bias) with coordinates $x_1, x_2, \ldots, x_m$, defines a hyperplane as the decision surface between two different classes of inputs.

For the perceptron to function properly, the two classes $\mathscr{C}_1$ and $\mathscr{C}_2$ must be *linearly separable*. This, in turn, means that the patterns to be classified must be sufficiently separated from each other to ensure that the decision surface consists of a hyperplane. This requirement is illustrated in Fig. 1.4 for the case of a two-dimensional perceptron. In Fig. 1.4a, the two classes $\mathscr{C}_1$ and $\mathscr{C}_2$ are sufficiently separated from each other for us to draw a hyperplane (in this case, a striaght line) as the decision boundary. If, however, the two classes $\mathscr{C}_1$ and $\mathscr{C}_2$ are allowed to move too close to each other, as in Fig. 1.4b, they become nonlinearly separable, a situation that is beyond the computing capability of the perceptron.

Suppose then that the input variables of the perceptron originate from two linearly separable classes. Let $\mathscr{H}_1$ be the subspace of training vectors $\mathbf{x}_1(1), \mathbf{x}_1(2), \ldots$ that belong to class $\mathscr{C}_1$, and let $\mathscr{H}_2$ be the subspace of training vectors $\mathbf{x}_2(1), \mathbf{x}_2(2), \ldots$ that belong to class $\mathscr{C}_2$. The union of $\mathscr{H}_1$ and $\mathscr{H}_2$ is the complete space denoted by $\mathscr{H}$. Given the sets

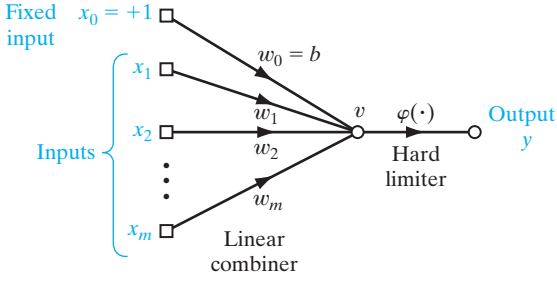


FIGURE 1.3 Equivalent signal-flow graph of the perceptron; dependence on time has been omitted for clarity.

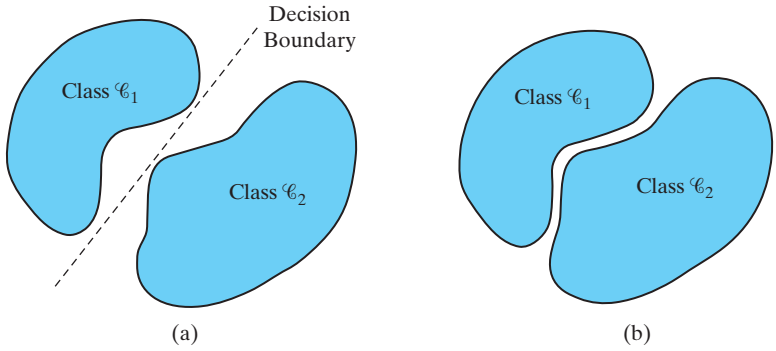


FIGURE 1.4 (a) A pair of linearly separable patterns. (b) A pair of non-linearly separable patterns.

of vectors $\mathscr{H}_1$ and $\mathscr{H}_2$ to train the classifier, the training process involves the adjustment of the weight vector $\mathbf{w}$ in such a way that the two classes $\mathscr{C}_1$ and $\mathscr{C}_2$ are linearly separable. That is, there exists a weight vector $\mathbf{w}$ such that we may state

$$\begin{aligned} \mathbf{w}^T\mathbf{x} &> 0 \text{ for every input vector } \mathbf{x} \text{ belonging to class } \mathscr{C}_1 \\ \mathbf{w}^T\mathbf{x} &\leq 0 \text{ for every input vector } \mathbf{x} \text{ belonging to class } \mathscr{C}_2 \end{aligned} \tag{1.4}$$

In the second line of Eq. (1.4), we have arbitrarily chosen to say that the input vector $\mathbf{x}$ belongs to class $\mathscr{C}_2$ if $\mathbf{w}^T\mathbf{x} = 0$. Given the subsets of training vectors $\mathscr{H}_1$ and $\mathscr{H}_2$, the training problem for the perceptron is then to find a weight vector $\mathbf{w}$ such that the two inequalities of Eq. (1.4) are satisfied.

The algorithm for adapting the weight vector of the elementary perceptron may now be formulated as follows:

1. If the nth member of the training set, $\mathbf{x}(n)$, is correctly classified by the weight vector $\mathbf{w}(n)$ computed at the nth iteration of the algorithm, no correction is made to the weight vector of the perceptron in accordance with the rule:

$$\begin{aligned} \mathbf{w}(n+1) &= \mathbf{w}(n) \quad \text{if } \mathbf{w}^T(n)\mathbf{x}(n) > 0 \text{ and } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_1 \\ \mathbf{w}(n+1) &= \mathbf{w}(n) \quad \text{if } \mathbf{w}^T(n)\mathbf{x}(n) \leq 0 \text{ and } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_2 \end{aligned} \tag{1.5}$$

2. Otherwise, the weight vector of the perceptron is updated in accordance with the rule

$$\begin{aligned} \mathbf{w}(n+1) &= \mathbf{w}(n) - \eta(n)\mathbf{x}(n) \quad \text{if } \mathbf{w}^T(n)\mathbf{x}(n) > 0 \text{ and } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_2 \\ \mathbf{w}(n+1) &= \mathbf{w}(n) + \eta(n)\mathbf{x}(n) \quad \text{if } \mathbf{w}^T(n)\mathbf{x}(n) \leq 0 \text{ and } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_1 \end{aligned} \tag{1.6}$$

where the *learning-rate parameter* $\eta(n)$ controls the adjustment applied to the weight vector at iteration n.

If $\eta(n) = \eta > 0$, where η is a constant independent of the iteration number n, then we have a *fixed-increment adaptation rule* for the perceptron.

In the sequel, we first prove the convergence of a fixed-increment adaptation rule for which $\eta = 1$. Clearly, the value of η is unimportant, so long as it is positive. A value

of $\eta \neq 1$ merely scales the pattern vectors without affecting their separability. The case of a variable $\eta(n)$ is considered later.

Proof of the perceptron convergence algorithm[2] is presented for the initial condition $\mathbf{w}(0) = \mathbf{0}$. Suppose that $\mathbf{w}^T(n)\mathbf{x}(n) < 0$ for $n = 1, 2, ...$, and the input vector $\mathbf{x}(n)$ belongs to the subset $\mathcal{H}_1$. That is, the perceptron incorrectly classifies the vectors $\mathbf{x}(1)$, $\mathbf{x}(2)$, ..., since the first condition of Eq. (1.4) is violated. Then, with the constant $\eta(n) = 1$, we may use the second line of Eq. (1.6) to write

$$\mathbf{w}(n + 1) = \mathbf{w}(n) + \mathbf{x}(n) \qquad \text{for } \mathbf{x}(n) \text{ belonging to class } \mathcal{C}_1 \tag{1.7}$$

Given the initial condition $\mathbf{w}(0) = \mathbf{0}$, we may iteratively solve this equation for $\mathbf{w}(n + 1)$, obtaining the result

$$\mathbf{w}(n + 1) = \mathbf{x}(1) + \mathbf{x}(2) + \cdots + \mathbf{x}(n) \tag{1.8}$$

Since the classes $\mathcal{C}_1$ and $\mathcal{C}_2$ are assumed to be linearly separable, there exists a solution $\mathbf{w}_o$ for which $\mathbf{w}^T\mathbf{x}(n) > 0$ for the vectors $\mathbf{x}(1), ..., \mathbf{x}(n)$ belonging to the subset $\mathcal{H}_1$. For a fixed solution $\mathbf{w}_o$, we may then define a positive number α as

$$\alpha = \min_{\mathbf{x}(n) \in \mathcal{H}_1} \mathbf{w}_o^T \mathbf{x}(n) \tag{1.9}$$

Hence, multiplying both sides of Eq. (1.8) by the row vector $\mathbf{w}_o^T$, we get

$$\mathbf{w}_o^T\mathbf{w}(n + 1) = \mathbf{w}_o^T\mathbf{x}(1) + \mathbf{w}_o^T\mathbf{x}(2) + \cdots + \mathbf{w}_o^T\mathbf{x}(n)$$

Accordingly, in light of the definition given in Eq. (1.9), we have

$$\mathbf{w}_o^T\mathbf{w}(n + 1) \geq n\alpha \tag{1.10}$$

Next we make use of an inequality known as the Cauchy–Schwarz inequality. Given two vectors $\mathbf{w}_0$ and $\mathbf{w}(n + 1)$, the *Cauchy–Schwarz inequality* states that

$$\|\mathbf{w}_o\|^2 \|\mathbf{w}(n + 1)\|^2 \geq [\mathbf{w}_o^T\mathbf{w}(n + 1)]^2 \tag{1.11}$$

where $\|\cdot\|$ denotes the Euclidean norm of the enclosed argument vector, and the inner product $\mathbf{w}_o^T\mathbf{w}(n + 1)$ is a scalar quantity. We now note from Eq. (1.10) that $[\mathbf{w}_o^T\mathbf{w}(n + 1)]^2$ is equal to or greater than $n^2\alpha^2$. From Eq. (1.11) we note that $\|\mathbf{w}_o\|^2\|\mathbf{w}(n + 1)\|^2$ is equal to or greater than $[\mathbf{w}_o^T\mathbf{w}(n + 1)]^2$. It follows therefore that

$$\|\mathbf{w}_o\|^2\|\mathbf{w}(n + 1)\|^2 \geq n^2\alpha^2$$

or, equivalently,

$$\|\mathbf{w}(n + 1)\|^2 \geq \frac{n^2\alpha^2}{\|\mathbf{w}_o\|^2} \tag{1.12}$$

We next follow another development route. In particular, we rewrite Eq. (1.7) in the form

$$\mathbf{w}(k + 1) = \mathbf{w}(k) + \mathbf{x}(k) \qquad \text{for } k = 1, ..., n \quad \text{and} \quad \mathbf{x}(k) \in \mathcal{H}_1 \tag{1.13}$$

By taking the squared Euclidean norm of both sides of Eq. (1.13), we obtain

$$\|\mathbf{w}(k + 1)\|^2 = \|\mathbf{w}(k)\|^2 + \|\mathbf{x}(k)\|^2 + 2\mathbf{w}^T(k)\mathbf{x}(k) \tag{1.14}$$

But, $\mathbf{w}^T(k)\mathbf{x}(k) \leq 0$. We therefore deduce from Eq. (1.14) that

$$\|\mathbf{w}(k+1)\|^2 \leq \|\mathbf{w}(k)\|^2 + \|\mathbf{x}(k)\|^2$$

or, equivalently,

$$\|\mathbf{w}(k+1)\|^2 - \|\mathbf{w}(k)\|^2 \leq \|\mathbf{x}(k)\|^2, \qquad k = 1, \ldots, n \tag{1.15}$$

Adding these inequalities for $k = 1, \ldots, n$, and invoking the assumed initial condition $\mathbf{w}(0) = \mathbf{0}$, we get the inequality

$$\begin{aligned} \|\mathbf{w}(n+1)\|^2 &\leq \sum_{k=1}^{n} \|\mathbf{x}(k)\|^2 \\ &\leq n\beta \end{aligned} \tag{1.16}$$

where β is a positive number defined by

$$\beta = \max_{\mathbf{x}(k) \in \mathcal{H}_1} \|\mathbf{x}(k)\|^2 \tag{1.17}$$

Equation (1.16) states that the squared Euclidean norm of the weight vector $\mathbf{w}(n+1)$ grows at most linearly with the number of iterations n.

The second result of Eq. (1.16) is clearly in conflict with the earlier result of Eq. (1.12) for sufficiently large values of n. Indeed, we can state that n cannot be larger than some value $n_{\max}$ for which Eqs. (1.12) and (1.16) are both satisfied with the equality sign. That is, $n_{\max}$ is the solution of the equation

$$\frac{n_{\max}^2 \alpha^2}{\|\mathbf{w}_o\|^2} = n_{\max}\beta$$

Solving for $n_{\max}$, given a solution vector $\mathbf{w}_o$, we find that

$$n_{\max} = \frac{\beta \|\mathbf{w}_o\|^2}{\alpha^2} \tag{1.18}$$

We have thus proved that for $\eta(n) = 1$ for all n and $\mathbf{w}(0) = \mathbf{0}$, and given that a solution vector $\mathbf{w}_o$ exists, the rule for adapting the synaptic weights of the perceptron must terminate after at most $n_{\max}$ interations. Surprisingly, this statement, proved for hypothesis $\mathcal{H}_1$, also holds for huypothesis $\mathcal{H}_2$. Note however,

We may now state the *fixed-increment covergence theorem* for the perceptron as follows (Rosenblatt, 1962):

> *Let the subsets of training vectors $\mathcal{H}_1$ and $\mathcal{H}_2$ be linearly separable. Let the inputs presented to the perceptron originate from these two subsets. The perceptron converges after some n_o iterations, in the sense that*
>
> $$\mathbf{w}(n_o) = \mathbf{w}(n_o + 1) = \mathbf{w}(n_o + 2) = \cdots$$
>
> *is a solution vector for $n_0 \leq n_{\max}$.*

Consider next the *absolute error-correction procedure* for the adaptation of a single-layer perceptron, for which $\eta(n)$ is variable. In particular, let $\eta(n)$ be the smallest integer for which the condition

$$\eta(n)\mathbf{x}^T(n)\mathbf{x}(n) > |\mathbf{w}^T(n)\mathbf{x}(n)|$$

holds. With this procedure we find that if the inner product $\mathbf{w}^T(n)\mathbf{x}(n)$ at iteration n has an incorrect sign, then $\mathbf{w}^T(n+1)\mathbf{x}(n)$ at iteration $n+1$ would have the correct sign. This suggests that if $\mathbf{w}^T(n)\mathbf{x}(n)$ has an incorrect sign, at iteration n, we may modify the training sequence at iteration $n+1$ by setting $\mathbf{x}(n+1) = \mathbf{x}(n)$. In other words, each pattern is presented repeatedly to the perceptron until that pattern is classified correctly.

Note also that the use of an initial value $\mathbf{w}(0)$ different from the null condition merely results in a decrease or increase in the number of iterations required to converge, depending on how $\mathbf{w}(0)$ relates to the solution $\mathbf{w}_o$. Regardless of the value assigned to $\mathbf{w}(0)$, the perceptron is assured of convergence.

In Table 1.1, we present a summary of the *perceptron convergence algorithm* (Lippmann, 1987). The symbol $\text{sgn}(\cdot)$, used in step 3 of the table for computing the actual response of the perceptron, stands for the *signum function*:

$$\text{sgn}(v) = \begin{cases} +1 & \text{if } v > 0 \\ -1 & \text{if } v < 0 \end{cases} \tag{1.19}$$

We may thus express the *quantized response* $y(n)$ of the perceptron in the compact form

$$y(n) = \text{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)] \tag{1.20}$$

Notice that the input vector $\mathbf{x}(n)$ is an $(m+1)$-by-1 vector whose first element is fixed at +1 throughout the computation. Correspondingly, the weight vector $\mathbf{w}(n)$ is an

TABLE 1.1 Summary of the Perceptron Convergence Algorithm

Variables and Parameters:

$\mathbf{x}(n)$ = $(m+1)$-by-1 input vector
$= [+1, x_1(n), x_2(n), \ldots, x_m(n)]^T$
$\mathbf{w}(n)$ = $(m+1)$-by-1 weight vector
$= [b, w_1(n), w_2(n), \ldots, w_m(n)]^T$
b = bias
$y(n)$ = actual response (quantized)
$d(n)$ = desired response
η = learning-rate parameter, a positive constant less than unity

1. *Initialization.* Set $\mathbf{w}(0) = \mathbf{0}$. Then perform the following computations for time-step $n = 1, 2, \ldots$.
2. *Activation.* At time-step n, activate the perceptron by applying continuous-valued input vector $\mathbf{x}(n)$ and desired response $d(n)$.
3. *Computation of Actual Response.* Compute the actual response of the perceptron as
$$y(n) = \text{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)]$$
where $\text{sgn}(\cdot)$ is the signum function.
4. *Adaptation of Weight Vector.* Update the weight vector of the perceptron to obtain
$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta[d(n) - y(n)]\mathbf{x}(n)$$
where
$$d(n) = \begin{cases} +1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_1 \\ -1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_2 \end{cases}$$
5. *Continuation.* Increment time step n by one and go back to step 2.

$(m+1)$-by-1 vector whose first element equals the bias b. One other important point to note in Table 1.1 is that we have introduced a *quantized desired response* $d(n)$, defined by

$$d(n) = \begin{cases} +1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_1 \\ -1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_2 \end{cases} \tag{1.21}$$

Thus, the adaptation of the weight vector $\mathbf{w}(n)$ is summed up nicely in the form of the *error-correction learning rule*

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta[d(n) - y(n)]\mathbf{x}(n) \tag{1.22}$$

where η is the *learning-rate parameter* and the difference $d(n) - y(n)$ plays the role of an *error signal*. The learning-rate parameter is a positive constant limited to the range $0 < \eta \leq 1$. When assigning a value to it inside this range, we must keep in mind two conflicting requirements (Lippmann, 1987):

- *averaging* of past inputs to provide stable weight estimates, which requires a small η;
- *fast adaptation* with respect to real changes in the underlying distributions of the process responsible for the generation of the input vector $\mathbf{x}$, which requires a large η.

1.4 RELATION BETWEEN THE PERCEPTRON AND BAYES CLASSIFIER FOR A GAUSSIAN ENVIRONMENT

The perceptron bears a certain relationship to a classical pattern classifier known as the Bayes classifier. When the environment is Gaussian, the Bayes classifier reduces to a linear classifier. This is the same form taken by the perceptron. However, the linear nature of the perceptron is *not* contingent on the assumption of Gaussianity. In this section, we study this relationship and thereby develop further insight into the operation of the perceptron. We begin the discussion with a brief review of the Bayes classifier.

Bayes Classifier

In the *Bayes classifier*, or *Bayes hypothesis testing procedure*, we minimize the *average risk*, denoted by $\mathscr{R}$. For a two-class problem, represented by classes $\mathscr{C}_1$ and $\mathscr{C}_2$, the average risk is defined by Van Trees (1968) as

$$\begin{aligned} \mathscr{R} = {} & c_{11}p_1 \int_{\mathscr{H}_1} p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_1)d\mathbf{x} + c_{22}p_2 \int_{\mathscr{H}_2} p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_2)d\mathbf{x} \\ & + c_{21}p_1 \int_{\mathscr{H}_2} p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_1)d\mathbf{x} + c_{12}p_2 \int_{\mathscr{H}_1} p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_2)d\mathbf{x} \end{aligned} \tag{1.23}$$

where the various terms are defined as follows:

p_i = *prior probability* that the observation vector $\mathbf{x}$ (representing a realization of the random vector $\mathbf{X}$) corresponds to an object in class $\mathscr{C}_i$, with $i = 1, 2$, and $p_1 + p_2 = 1$

c_{ij} = cost of deciding in favor of class $\mathscr{C}_i$ represented by subspace $\mathscr{H}_i$ when class $\mathscr{C}_j$ is true (i.e., observation vector $\mathbf{x}$ corresponds to an object in class $\mathcal{C}_1$), with $i, j = 1, 2$

$p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_i)$ = conditional probability density function of the random vector $\mathbf{X}$, given that the observation vector $\mathbf{x}$ corresponds to an object in class $\mathcal{C}_1$, with $i = 1, 2$.

The first two terms on the right-hand side of Eq. (1.23) represent *correct* decisions (i.e., correct classifications), whereas the last two terms represent *incorrect* decisions (i.e., misclassifications). Each decision is weighted by the product of two factors: the cost involved in making the decision and the relative frequency (i.e., *prior* probability) with which it occurs.

The intention is to determine a strategy for the *minimum average risk*. Because we require that a decision be made, each observation vector $\mathbf{x}$ must be assigned in the overall observation space $\mathscr{X}$ to either $\mathscr{X}_1$ or $\mathscr{X}_2$. Thus,

$$\mathscr{X} = \mathscr{X}_1 + \mathscr{X}_2 \tag{1.24}$$

Accordingly, we may rewrite Eq. (1.23) in the equivalent form

$$\begin{aligned}\mathscr{R} = {} & c_{11}p_1 \int_{\mathscr{X}_1} p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_1)d\mathbf{x} + c_{22}p_2 \int_{\mathscr{X}-\mathscr{X}_1} p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_2)d\mathbf{x} \\ & + c_{21}p_1 \int_{\mathscr{X}-\mathscr{X}_1} p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_1)\mathbf{x} + c_{12}p_2 \int_{\mathscr{X}_1} p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_2)d\mathbf{x}\end{aligned} \tag{1.25}$$

where $c_{11} < c_{21}$ and $c_{22} < c_{12}$. We now observe the fact that

$$\int_{\mathscr{X}} p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_1)d\mathbf{x} = \int_{\mathscr{X}} p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_2)d\mathbf{x} = 1 \tag{1.26}$$

Hence, Eq. (1.25) reduces to

$$\begin{aligned}\mathscr{R} = {} & c_{21}p_1 + c_{22}p_2 \\ & + \int_{\mathscr{X}_1} [p_2(c_{12} - c_{22})\, p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_2) - p_1(c_{21} - c_{11})\, p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_1)]d\mathbf{x}\end{aligned} \tag{1.27}$$

The first two terms on the right-hand side of Eq. (1.27) represent a fixed cost. Since the requirement is to minimize the average risk $\mathscr{R}$, we may therefore deduce the following strategy from Eq.(1.27) for optimum classification:

1. All values of the observation vector $\mathbf{x}$ for which the integrand (i.e., the expression inside the square brackets) is negative should be assigned to subset $\mathscr{X}_1$ (i.e., class $\mathscr{C}_1$), for the integral would then make a negative contribution to the risk $\mathscr{R}$.
2. All values of the observation vector $\mathbf{x}$ for which the integrand is positive should be excluded from subset $\mathscr{X}_1$ (i.e., assigned to class $\mathscr{C}_2$), for the integral would then make a positive contribution to the risk $\mathscr{R}$.
3. Values of $\mathbf{x}$ for which the integrand is zero have no effect on the average risk $\mathscr{R}$ and may be assingned arbitrarily. We shall assume that these points are assigned to subset $\mathscr{X}_2$ (i.e., class $\mathscr{C}_2$).

On this basis, we may now formulate the Bayes classifier as follows:

If the condition

$$p_1(c_{21} - c_{11})\, p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_1) > p_2(c_{12} - c_{22})\, p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_2)$$

holds, assign the observation vector $\mathbf{x}$ *to subspace* $\mathscr{X}_1$ *(i.e., class* $\mathscr{C}_1$*). Otherwise assign* $\mathbf{x}$ *to* $\mathscr{X}_2$ *(i.e., class* $\mathscr{C}_2$*).*

To simplify matters, define

$$\Lambda(\mathbf{x}) = \frac{p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_1)}{p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_2)} \tag{1.28}$$

and

$$\xi = \frac{p_2(c_{12} - c_{22})}{p_1(c_{21} - c_{11})} \tag{1.29}$$

The quantity $\Lambda(\mathbf{x})$, the ratio of two conditional probability density functions, is called the *likelihood ratio*. The quantity ξ is called the *threshold* of the test. Note that both $\Lambda(\mathbf{x})$ and ξ are always positive. In terms of these two quantities, we may now reformulate the Bayes classifier by stating the following

If, for an observation vector $\mathbf{x}$*, the likelihood ratio* $\Lambda(\mathbf{x})$ *is greater than the threshold* ξ*, assign* $\mathbf{x}$ *to class* $\mathscr{C}_1$*. Otherwise, assign it to class* $\mathscr{C}_2$*.*

Figure 1.5a depicts a block-diagram representation of the Bayes classifier. The important points in this block diagram are twofold:

1. The data processing involved in designing the Bayes classifier is confined entirely to the computation of the likelihood ratio $\Lambda(\mathbf{x})$.

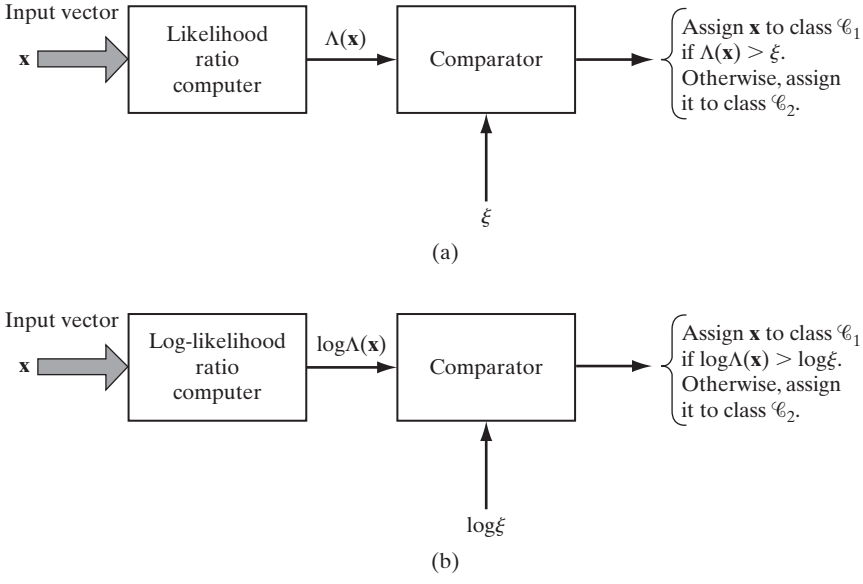


FIGURE 1.5 Two equivalent implementations of the Bayes classifier: (a) Likelihood ratio test, (b) Log-likelihood ratio test.

2. This computation is completely invariant to the values assigned to the prior probabilities and costs involved in the decision-making process. These quantities merely affect the value of the threshold ξ.

From a computational point of view, we find it more convenient to work with the logarithm of the likelihood ratio rather than the likelihood ratio itself. We are permitted to do this for two reasons. First, the logarithm is a monotonic function. Second, the likelihood ratio $\Lambda(\mathbf{x})$ and threshold ξ are both positive. Therefore, the Bayes classifier may be implemented in the equivalent form shown in Fig. 1.5b. For obvious reasons, the test embodied in this latter figure is called the *log-likelihood ratio test.*

Bayes Classifier for a Gaussian Distribution

Consider now the special case of a two-class problem, for which the underlying distribution is Gaussian. The random vector $\mathbf{X}$ has a mean value that depends on whether it belongs to class $\mathscr{C}_1$ or class $\mathscr{C}_2$, but the covariance matrix of $\mathbf{X}$ is the same for both classes. That is to say,

$$\begin{aligned}
\text{Class } \mathscr{C}_1: \quad & \mathbb{E}[\mathbf{X}] = \boldsymbol{\mu}_1 \\
& \mathbb{E}[(\mathbf{X} - \boldsymbol{\mu}_1)(\mathbf{X} - \boldsymbol{\mu}_1)^T] = \mathbf{C} \\
\text{Class } \mathscr{C}_2: \quad & \mathbb{E}[\mathbf{X}] = \boldsymbol{\mu}_2 \\
& \mathbb{E}[(\mathbf{X} - \boldsymbol{\mu}_2)(\mathbf{X} - \boldsymbol{\mu}_2)^T] = \mathbf{C}
\end{aligned}$$

The covariance matrix $\mathbf{C}$ is nondiagonal, which means that the samples drawn from classes $\mathscr{C}_1$ and $\mathscr{C}_2$ are *correlated.* It is assumed that $\mathbf{C}$ is nonsingular, so that its inverse matrix $\mathbf{C}^{-1}$ exists.

With this background, we may express the conditional probability density function of $\mathbf{X}$ as the multivariate Gaussian distribution

$$p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_i) = \frac{1}{(2\pi)^{m/2}(\det(\mathbf{C}))^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_i)^T \mathbf{C}^{-1}(\mathbf{x} - \boldsymbol{\mu}_i)\right), \quad i = 1, 2 \tag{1.30}$$

where m is the dimensionality of the observation vector $\mathbf{x}$.

We further assume the following:

1. The two classes $\mathscr{C}_1$ and $\mathscr{C}_2$ are equiprobable:

$$p_1 = p_2 = \frac{1}{2} \tag{1.31}$$

2. Misclassifications carry the same cost, and no cost is incurred on correct classifications:

$$c_{21} = c_{12} \quad \text{and} \quad c_{11} = c_{22} = 0 \tag{1.32}$$

We now have the information we need to design the Bayes classifier for the two-class problem. Specifically, by substituting Eq. (1.30) into (1.28) and taking the natural logarithm, we get (after simplifications)

$$\begin{aligned}\log\Lambda(\mathbf{x}) &= -\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_1)^T\mathbf{C}^{-1}(\mathbf{x}-\boldsymbol{\mu}_1) + \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_2)^T\mathbf{C}^{-1}(\mathbf{x}-\boldsymbol{\mu}_2) \\ &= (\boldsymbol{\mu}_1-\boldsymbol{\mu}_2)^T\mathbf{C}^{-1}\mathbf{x} + \frac{1}{2}(\boldsymbol{\mu}_2^T\mathbf{C}^{-1}\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1^T\mathbf{C}^{-1}\boldsymbol{\mu}_1)\end{aligned} \tag{1.33}$$

By substituting Eqs. (1.31) and (1.32) into Eq. (1.29) and taking the natural logarithm, we get

$$\log\xi = 0 \tag{1.34}$$

Equations (1.33) and (1.34) state that the Bayes classifier for the problem at hand is a *linear classifier*, as described by the relation

$$y = \mathbf{w}^T\mathbf{x} + b \tag{1.35}$$

where

$$\mathbf{y} = \log\Lambda(\mathbf{x}) \tag{1.36}$$

$$\mathbf{w} = \mathbf{C}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \tag{1.37}$$

$$b = \frac{1}{2}(\boldsymbol{\mu}_2^T\mathbf{C}^{-1}\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1^T\mathbf{C}^{-1}\boldsymbol{\mu}_1) \tag{1.38}$$

More specifically, the classifier consists of a linear combiner with weight vector $\mathbf{w}$ and bias b, as shown in Fig. 1.6.

On the basis of Eq. (1.35), we may now describe the log-likelihood ratio test for our two-class problem as follows:

If the output y of the linear combiner (including the bias b) is positive, assign the observation vector $\mathbf{x}$ to class $\mathscr{C}_1$. Otherwise, assign it to class $\mathscr{C}_2$.

The operation of the Bayes classifier for the Gaussian environment described herein is analogous to that of the perceptron in that they are both linear classifiers; see Eqs. (1.1) and (1.35). There are, however, some subtle and important differences between them, which should be carefully examined (Lippmann, 1987):

- The perceptron operates on the premise that the patterns to be classified are *linearly separable*. The Gaussian distributions of the two patterns assumed in the derivation of the Bayes classifier certainly do *overlap* each other and are therefore *not* separable. The extent of the overlap is determined by the mean vectors

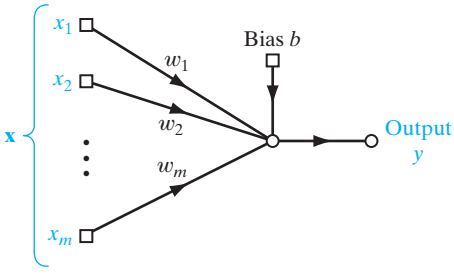


FIGURE 1.6 Signal-flow graph of Gaussian classifier.

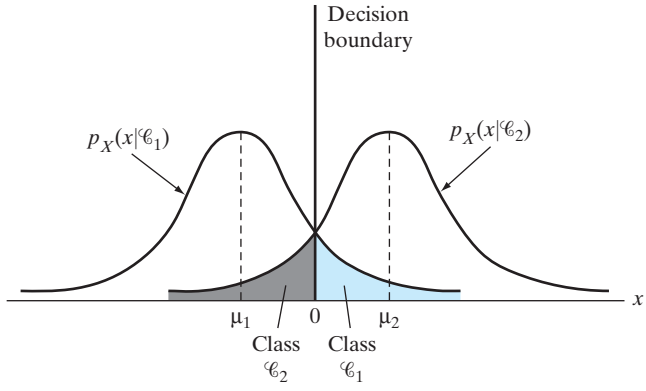


FIGURE 1.7 Two overlapping, one-dimensional Gaussian distributions.

$\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ and the covariance matrix $\mathbf{C}$. The nature of this overlap is illustrated in Fig. 1.7 for the special case of a scalar random variable (i.e., dimensionality $m = 1$). When the inputs are nonseparable and their distributions overlap as illustrated, the perceptron convergence algorithm develops a problem because decision boundaries between the different classes may oscillate continuously.

- The Bayes classifier minimizes the probability of classification error. This minimization is independent of the overlap between the underlying Gaussian distributions of the two classes. For example, in the special case illustrated in Fig. 1.7, the Bayes classifier always positions the decision boundary at the point where the Gaussian distributions for the two classes $\mathscr{C}_1$ and $\mathscr{C}_2$ cross each other.
- The perceptron convergence algorithm is *nonparametric* in the sense that it makes no assumptions concerning the form of the underlying distributions. It operates by concentrating on errors that occur where the distributions overlap. It may therefore work well when the inputs are generated by nonlinear physical mechanisms and when their distributions are heavily skewed and non-Gaussian. In contrast, the Bayes classifier is *parametric*; its derivation is contingent on the assumption that the underlying distributions be Gaussian, which may limit its area of application.
- The perceptron convergence algorithm is both adaptive and simple to implement; its storage requirement is confined to the set of synaptic weights and bias. On the other hand, the design of the Bayes classifier is fixed; it can be made adaptive, but at the expense of increased storage requirements and more complex computations.

1.5 COMPUTER EXPERIMENT: PATTERN CLASSIFICATION

The objective of this computer experiment is twofold:

(i) to lay down the specifications of a *double-moon classification problem* that will serve as the basis of a prototype for the part of the book that deals with pattern-classification experiments;

(ii) to demonstrate the capability of Rosenblatt's perceptron algorithm to correctly classify linearly separable patterns and to show its breakdown when the condition of linear separability is violated.

Specifications of the Classification Problem

Figure 1.8 shows a pair of "moons" facing each other in an *asymmetrically* arranged manner. The moon labeled "Region A" is positioned symmetrically with respect to the y-axis, whereas the moon labeled "Region B" is displaced to the right of the y-axis by an amount equal to the radius r and below the x-axis by the distance d. The two moons have identical parameters:

$$\text{radius of each moon, } r = 10$$
$$\text{width of each moon, } w = 6$$

The vertical distance d separating the two moons is adjustable; it is measured with respect to the x-axis, as indicated in Fig. 1.8:

- Increasingly positive values of d signify increased separation between the two moons;
- increasingly negative values of d signify the two moons' coming closer to each other.

The training sample $\mathcal{T}$ consists of 1,000 pairs of data points, with each pair consisting of one point picked from region A and another point picked from region B, both randomly. The test sample consists of 2,000 pairs of data points, again picked in a random manner.

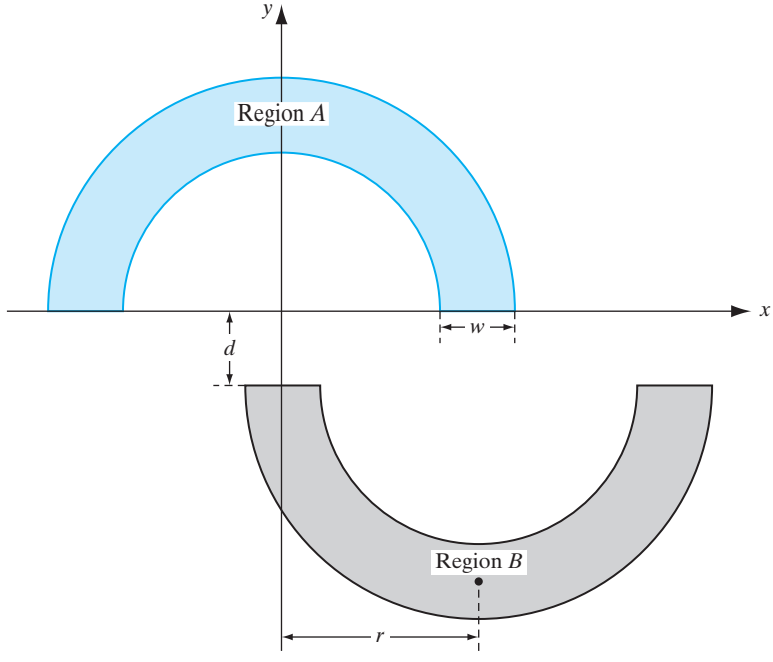


FIGURE 1.8 The double-moon classification problem.

The Experiment

The perceptron parameters picked for the experiment were as follows:

$$\text{size of the input layer} = 2$$
$$\beta = 50; \text{ see Eq. (1.17)}$$

The learning-rate parameter η was varied linearly from 10^{-1} down to 10^{-5}.

The weights were initially all set at zero.

Figure 1.9 presents the results of the experiment for $d = 1$, which corresponds to perfect linear separability. Part (a) of the figure presents the *learning curve*, where the mean-square error (MSE) is plotted versus the number of epochs; the figure shows convergence of the algorithm in three iterations. Part (b) of the figure shows the decision boundary computed through training of the perceptron algorithm, demonstrating perfect separability of all 2,000 test points.

In Fig. 1.10, the separation between the two moons was set at $d = -4$, a condition that violates linear separability. Part (a) of the figure displays the learning curve where the perceptron algorithm is now found to fluctuate continuously, indicating breakdown of the algorithm. This result is confirmed in part (b) of the figure, where the decision boundary (computed through training) intersects both moons, with a classification error rate of $(186/2000) \times 100\% = 9.3\%$.

1.6 THE BATCH PERCEPTRON ALGORITHM

The derivation of the perceptron convergence algorithm summarized in Table 1.1 was presented without reference to a cost function. Moreover, the derivation focused on a single-sample correction. In this section, we will do two things:

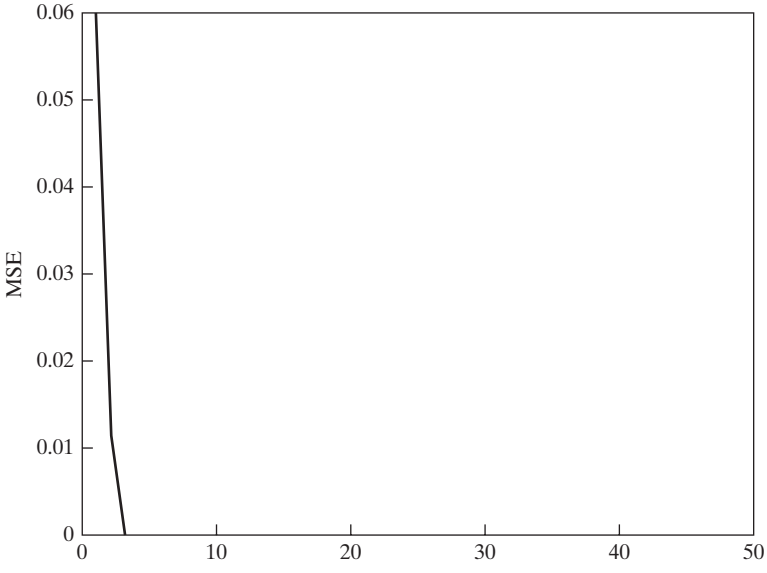
1. introduce the generalized form of a perceptron cost function;
2. use the cost function to formulate a batch version of the perceptron convergence algorithm.

The cost function we have in mind is a function that permits the application of a gradient search. Specifically, we define the *perceptron cost function* as

$$J(\mathbf{w}) = \sum_{\mathbf{x}(n) \in \mathcal{X}} (-\mathbf{w}^T \mathbf{x}(n) d(n)) \tag{1.39}$$

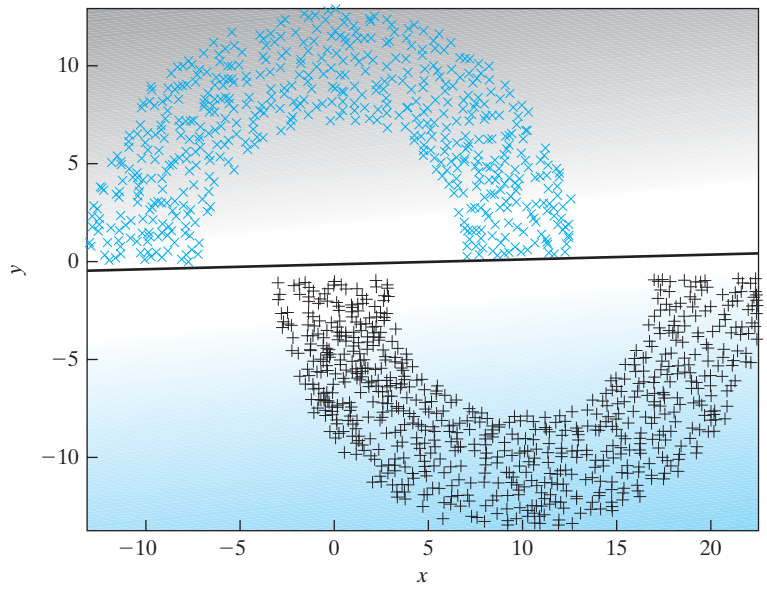
where $\mathcal{X}$ is the set of samples $\mathbf{x}$ *misclassified* by a perceptron using $\mathbf{w}$ as its weight vector (Duda et al., 2001). If all the samples are classified correctly, then the set $\mathcal{X}$ is empty, in which case the cost function $J(\mathbf{w})$ is zero. In any event, the nice feature of the cost function $J(\mathbf{w})$ is that it is *differentiable* with respect to the weight vector $\mathbf{w}$. Thus, differentiating $J(\mathbf{w})$ with respect to $\mathbf{w}$ yields the *gradient vector*

$$\nabla J(\mathbf{w}) = \sum_{\mathbf{x}(n) \in \mathcal{X}} (-\mathbf{x}(n) d(n)) \tag{1.40}$$



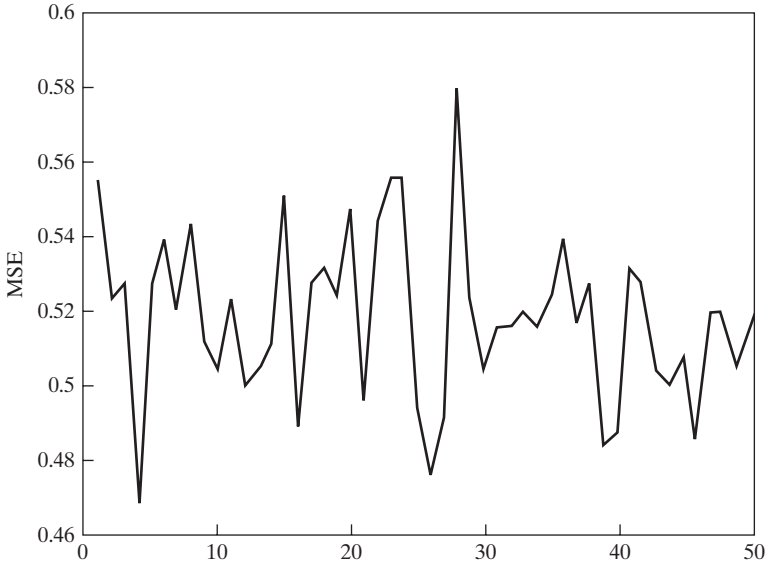
(a) learning curve

Classification using perceptron with distance = 1, radius = 10, and width = 6

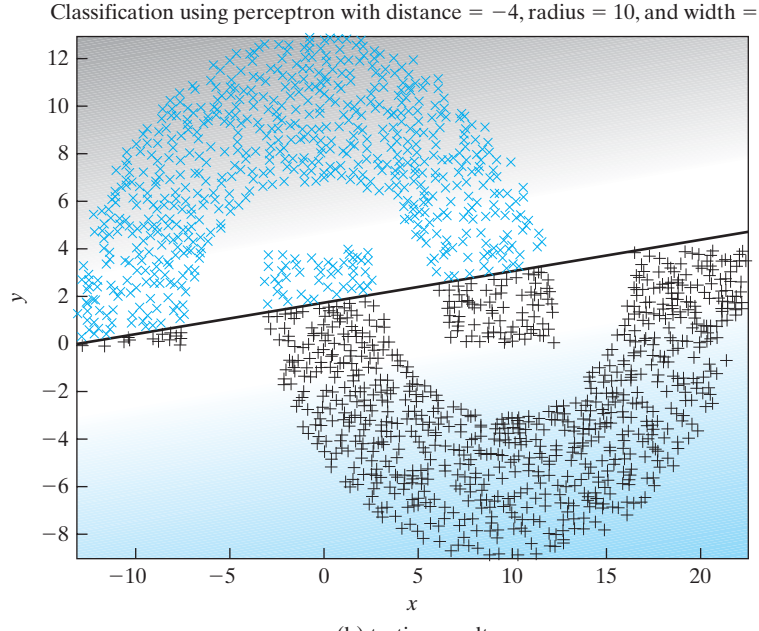


(b) testing result

FIGURE 1.9 Perceptron with the double-moon set at distance $d = 1$.



(a) learning curve



(b) testing result

FIGURE 1.10 Perceptron with the double-moon set at distance $d = -4$.

where the *gradient operator*

$$\nabla = \left[\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, ..., \frac{\partial}{\partial w_m}\right]^T \tag{1.41}$$

In the *method of steepest descent,* the adjustment to the weight vector $\mathbf{w}$ at each time-step of the algorithm is applied in a direction *opposite* to the gradient vector $\nabla J(\mathbf{w})$. Accordingly, the algorithm takes the form

$$\begin{aligned}\mathbf{w}(n+1) &= \mathbf{w}(n) - \eta(n)\nabla J(\mathbf{w}) \\ &= \mathbf{w}(n) + \eta(n)\sum_{\mathbf{x}(n)\in\mathcal{X}} \mathbf{x}(n)d(n)\end{aligned} \tag{1.42}$$

which includes the single-sample correction version of the perceptron convergence algorithm as a special case. Moreover, Eq. (1.42) embodies the *batch perceptron algorithm* for computing the weight vector, given the sample set $\mathbf{x}(1), \mathbf{x}(2), ...$ In particular, the adjustment applied to the weight vector at time-step $n+1$ is defined by the sum of all the samples misclassified by the weight vector $\mathbf{w}(n)$, with the sum being scaled by the learning-rate parameter $\eta(n)$. The algorithm is said to be of the "batch" kind because at each time-step of the algorithm, a batch of misclassified samples is used to compute the adjustment.

1.7 SUMMARY AND DISCUSSION

The perceptron is a single-layer neural network, the operation of which is based on error-correlation learning. The term "single layer" is used here to signify the fact that the computation layer of the network consists of a single neuron for the case of binary classification. The learning process for pattern classification occupies a finite number of iterations and then stops. For the classification to be successful, however, the patterns would have to be linearly separable.

The perceptron uses the McCulloch–Pitts model of a neuron. In this context, it is tempting to raise the question, would the perceptron perform better if it used a sigmoidal nonlinearity in place of the hard limiter? It turns out that the steady-state, decision-making characteristics of the perceptron are basically the same, regardless of whether we use hard limiting or soft limiting as the source of nonlinearity in the neural model (Shynk, 1990; Shynk and Bershad, 1991). We may therefore state formally that so long as we limit ourselves to the model of a neuron that consists of a linear combiner followed by a nonlinear element, then regardless of the form of nonlinearity used, a single-layer perceptron can perform pattern classification only on linearly separable patterns.

The first real critique of Rosenblatt's perceptron was presented by Minsky and Selfridge (1961). Minsky and Selfridge pointed out that the perceptron as defined by Rosenblatt could not even generalize toward the notion of binary parity, let alone make general abstractions. The computational limitations of Rosenblatt's perceptron were subsequently put on a solid mathematical foundation in the famous book *Perceptrons*, by Minsky and Papert (1969, 1988). After the presentation of some brilliant and highly detailed mathematical analyses of the perceptron, Minsky and Papert proved that the perceptron as defined by Rosenblatt is inherently incapable of making some global

generalizations on the basis of locally learned examples. In the last chapter of their book, Minsky and Papert make the conjecture that the limitations they had discovered for Rosenblatt's perceptron would also hold true for its variants—more specifically, multilayer neural networks. Section 13.2 of their book (1969) says the following:

> *The perceptron has shown itself worthy of study despite (and even because of!) its severe limitations. It has many features to attract attention: its linearity; its intriguing learning theorem; its clear paradigmatic simplicity as a kind of parallel computation. There is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension to multilayer systems is sterile.*

This conclusion was largely responsible for casting serious doubt on the computational capabilities of not only the perceptron, but also neural networks in general up to the mid-1980s.

History has shown, however, that the conjecture made by Minsky and Papert seems to be unjustified in that we now have several advanced forms of neural networks and learning machines that are computationally more powerful than Rosenblatt's perceptron. For example, multilayer perceptrons trained with the back-propagation algorithm discussed in Chapter 4, the radial basis-function networks discussed in Chapter 5, and the support vector machines discussed in Chapter 6 overcome the computational limitations of the single-layer perceptron in their own individual ways.

In closing the discussion, we may say that the perceptron is an elegant neural network designed for the classification of linearly separable patterns. Its importance is not only historical but also of practical value in the classification of linearly separable patters.

NOTES AND REFERENCES

1. The network organization of the original version of the perceptron as envisioned by Rosenblatt (1962) has three types of units: sensory units, association units, and response units. The connections from the sensory units to the association units have fixed weights, and the connections from the association units to the response units have variable weights. The association units act as preprocessors designed to extract a pattern from the environmental input. Insofar as the variable weights are concerned, the operation of Rosenblatt's original perceptron is essentially the same as that for the case of a single response unit (i.e., single neuron).
2. Proof of the perceptron convergence algorithm presented in Section 1.3 follows the classic book of Nilsson (1965).

PROBLEMS

1.1 Verify that Eqs. (1.19)–(1.22), summarizing the perceptron convergence algorithm, are consistent with Eqs. (1.5) and (1.6).

1.2 Suppose that in the signal-flow graph of the perceptron shown in Fig. 1.1, the hard limiter is replaced by the sigmoidal nonlinearity

$$\varphi(v) = \tanh\left(\frac{v}{2}\right)$$

where v is the induced local field. The classification decisions made by the perceptron are defined as follows:

> *Observation vector* $\mathbf{x}$ *belongs to class* $\mathscr{C}_1$ *if the output* $y > \xi$, *where* ξ *is a threshold; otherwise,* $\mathbf{x}$ *belongs to class* $\mathscr{C}_2$.

Show that the decision boundary so constructed is a hyperplane.

1.3 **(a)** The perceptron may be used to perform numerous logic functions. Demonstrate the implementation of the binary logic functions AND, OR, and COMPLEMENT.

(b) A basic limitation of the perceptron is that it cannot implement the EXCLUSIVE OR function. Explain the reason for this limitation.

1.4 Consider two one-dimensional, Gaussian-distributed classes $\mathscr{C}_1$ and $\mathscr{C}_2$ that have a common variance equal to 1. Their mean values are

$$\mu_1 = -10$$
$$\mu_2 = +10$$

These two classes are essentially linearly separable. Design a classifier that separates these two classes.

1.5 Equations (1.37) and (1.38) define the weight vector and bias of the Bayes classifier for a Gaussian environment. Determine the composition of this classifier for the case when the covariance matrix $\mathbf{C}$ is defined by

$$\mathbf{C} = \sigma^2\mathbf{I}$$

where σ^2 is a constant and $\mathbf{I}$ is the identity matrix.

Computer Experiment

1.6 Repeat the computer experiment of Section 1.5, this time, however, positioning the two moons of Figure 1.8 to be on the edge of separability, that is, $d = 0$. Determine the classification error rate produced by the algorithm over 2,000 test data points.

CHAPTER 2

Model Building through Regression

ORGANIZATION OF THE CHAPTER

The theme of this chapter is how to use *linear regression*, a special form of function approximation, to model a given set of random variables.

The chapter is organized as follows:

1. Section 2.1 is introductory, followed by Section 2.2 that sets the stage for the rest of the chapter by describing the mathematical framework of linear regression models.
2. Section 2.3 derives the *maximum a posteriori* (*MAP*) *estimate* of the parameter vector of a linear regression model.
3. Section 2.4 tackles the parameter estimation problem using the method of least squares and discusses this method's relationship to the Bayesian approach.
4. In Section 2.5, we revisit the pattern-classification experiment considered in Chapter 1, this time using the method of least squares.
5. Section 2.6 addresses the problem of model-order selection.
6. Section 2.7 discusses consequences of finite sample size in parameter estimation, including the bias–variance dilemma.
7. Section 2.8 introduces the notion of instrumental variables to deal with the "errors-in-variables" problem.

Section 2.9 provides a summary and discussion that conclude the chapter.

2.1 INTRODUCTION

The idea of model building shows up practically in every discipline that deals with *statistical data* analysis. Suppose, for example, we are given a set of random variables and the assigned task is to find the relationships that may exist between them, if any. In *regression*, which is a special form of function approximation, we typically find the following scenario:

- One of the random variables is considered to be of particular interest; that random variable is referred to as a dependent variable, or *response*.
- The remaining random variables are called independent variables, or *regressors*; their role is to explain or predict the statistical behavior of the response.
- The dependence of the response on the regressors includes an additive *error* term, to account for uncertainties in the manner in which this dependence is formulated;

the error term is called the *expectational error*, or *explanational error*, both of which are used interchangeably.

Such a model is called the *regression model.*[1]

There are two classes of regression models: linear and nonlinear. In *linear regression models*, the dependence of the response on the regressors is defined by a linear function, which makes their statistical analysis mathematically tractable. On the other hand, in *nonlinear regression models*, this dependence is defined by a nonlinear function, hence the mathematical difficulty in their analysis. In this chapter, we focus attention on linear regression models. Nonlinear regression models are studied in subsequent chapters.

The mathematical tractability of linear regression models shows up in this chapter in two ways. First, we use *Bayesian theory*[2] to derive the *maximum a posteriori estimate* of the vector that parameterizes a linear regression model. Next, we view the parameter estimation problem using another approach, namely, the *method of least squares*, which is perhaps the oldest parameter-estimation procedure; it was first derived by Gauss in the early part of the 19th century. We then demonstrate the equivalence between these two approaches for the special case of a Gaussian environment.

2.2 LINEAR REGRESSION MODEL: PRELIMINARY CONSIDERATIONS

Consider the situation depicted in Fig. 2.1a, where an *unknown stochastic environment* is the focus of attention. The environment is *probed* by applying a set of inputs, constituting the regressor

$$\mathbf{x} = [x_1, x_2, ..., x_M]^T \tag{2.1}$$

where the superscript T denotes matrix *transposition.* The resulting output of the environment, denoted by d, constitutes the corresponding *response*, which is assumed to be scalar merely for the convenience of presentation. Ordinarily, we do not know the functional dependence of the response d on the regressor $\mathbf{x}$, so we propose a linear regression model, parameterized as:

$$d = \sum_{j=1}^{M} w_j x_j + \varepsilon \tag{2.2}$$

where $w_1, w_2, ..., w_M$ denote a set of *fixed*, but *unknown*, *parameters*, meaning that the environment is *stationary.* The additive term ε, representing the expectational error of the model, accounts for our ignorance about the environment. A signal-flow graph depiction of the input–output behavior of the model described in Eq. (2.2) is presented in Fig. 2.1b.

Using matrix notation, we may rewrite Eq. (2.2) in the compact form

$$d = \mathbf{w}^T\mathbf{x} + \varepsilon \tag{2.3}$$

where the regressor $\mathbf{x}$ is defined in terms of its elements in Eq. (2.1). Correspondingly, the *parameter vector* $\mathbf{w}$ is defined by

$$\mathbf{w} = [w_1, w_2, ..., w_M]^T \tag{2.4}$$

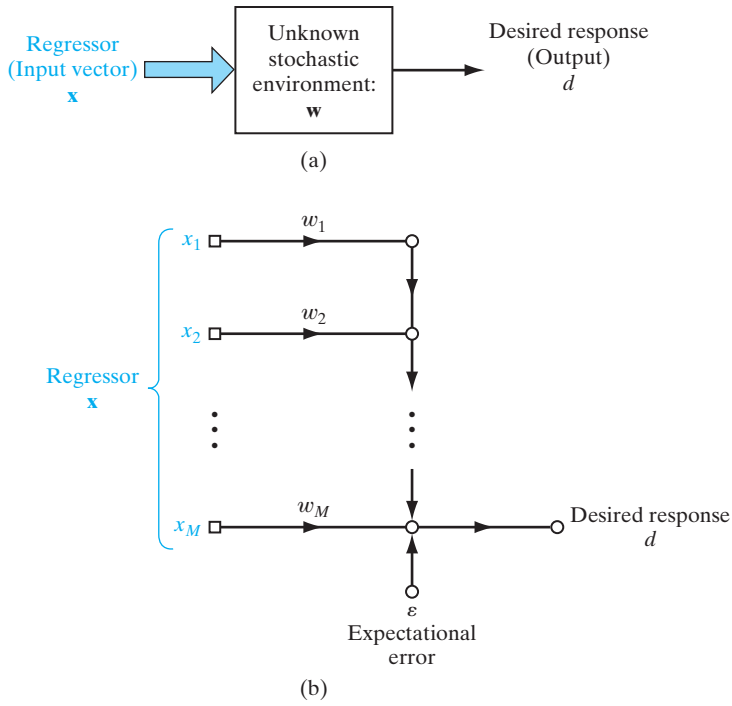


FIGURE 2.1 (a) Unknown stationary stochastic environment. (b) Linear regression model of the environment.

whose dimensionality is the same as that of the regressor $\mathbf{x}$; the common dimension M is called the *model order*. The matrix term $\mathbf{w}^T\mathbf{x}$ is the *inner product* of the vectors $\mathbf{w}$ and $\mathbf{x}$.

With the environment being stochastic, it follows that the regressor $\mathbf{x}$, the response d, and the expectational error ε are *sample* values (i.e., single-shot realizations) of the random vector $\mathbf{X}$, the random variable D, and the random variable E, respectively. With such a stochastic setting as the background, the problem of interest may now be stated as follows:

> *Given the joint statistics of the regressor* $\mathbf{X}$ *and the corresponding response* D*, estimate the unknown parameter vector* $\mathbf{w}$.

When we speak of the joint statistics, we mean the following set of statistical parameters:

- the correlation matrix of the regressor $\mathbf{X}$;
- the variance of the desired response D;
- the cross-correlation vector of the regressor $\mathbf{X}$ and the desired response D.

It is assumed that the means of both $\mathbf{X}$ and D are zero.

In Chapter 1, we discussed one important facet of Bayesian inference in the context of pattern classification. In this chapter, we study another facet of Bayesian inference that addresses the *parameter estimation* problem just stated.

2.3 MAXIMUM *A POSTERIORI* ESTIMATION OF THE PARAMETER VECTOR

The Bayesian paradigm provides a powerful approach for addressing and quantifying the *uncertainty* that surrounds the choice of the parameter vector $\mathbf{w}$ in the linear regression model of Eq. (2.3). Insofar as this model is concerned, the following two remarks are noteworthy:

1. The regressor $\mathbf{X}$ acts as the "excitation," bearing no relation whatsoever to the parameter vector $\mathbf{w}$.
2. Information about the unknown parameter vector $\mathbf{W}$ is contained solely in the desired response D that acts as the "observable" of the environment.

Accordingly, we focus attention on the joint probablity density function of $\mathbf{W}$ and D, conditional on $\mathbf{X}$.

Let this density function be denoted by $p_{\mathbf{W},D|\mathbf{X}}(\mathbf{w}, d|\mathbf{x})$. From probability theory, we know that this density function may be expressed as

$$p_{\mathbf{W},D|\mathbf{X}}(\mathbf{w}, d|\mathbf{x}) = p_{\mathbf{W}|D,\mathbf{X}}(\mathbf{w}|d, \mathbf{x})p_D(d) \tag{2.5}$$

Moreover, we may also express it in the equivalent form

$$p_{\mathbf{W},D|\mathbf{X}}(\mathbf{w}, d|\mathbf{x}) = p_{D|\mathbf{W},\mathbf{X}}(d|\mathbf{w}, \mathbf{x})p_{\mathbf{W}}(\mathbf{w}) \tag{2.6}$$

In light of this pair of equations, we may go on to write

$$p_{\mathbf{W}|D,\mathbf{X}}(\mathbf{w}|d, \mathbf{x}) = \frac{p_{D|\mathbf{W},\mathbf{X}}(d|\mathbf{w}, \mathbf{x})p_{\mathbf{W}}(\mathbf{w})}{p_D(d)} \tag{2.7}$$

provided that $p_D(d) \neq 0$. Equation (2.7) is a special form of *Bayes's theorem*; it embodies four density functions, characterized as follows:

1. *Observation density*: This stands for the conditional probability density function $p_{D|\mathbf{W},\mathbf{X}}(d|\mathbf{w}, \mathbf{x})$, referring to the "observation" of the environmental response d due to the regressor $\mathbf{x}$, given the parameter vector $\mathbf{w}$.
2. *Prior*: This stands for the probability density function $p_{\mathbf{W}}(\mathbf{w})$, referring to information about the parameter vector $\mathbf{w}$, prior to any observations made on the environment. Hereafter, the prior is simply denoted by $\pi(\mathbf{w})$.
3. *Posterior density*: This stands for the conditional probability density function $p_{\mathbf{W}|D,\mathbf{X}}(\mathbf{w}|d, \mathbf{x})$, referring to the parameter vector $\mathbf{w}$ "after" observation of the environment has been completed. Hereafter, the posterior density is denoted by $\pi(\mathbf{w}|d, \mathbf{x})$. The conditioning response–regressor pair $(\mathbf{x}, d)$ is the "observation model," embodying the response d of the environment due to the regressor $\mathbf{x}$.
4. *Evidence*: This stands for the probability density function $p_D(d)$, referring to the "information" contained in the response d for statistical analysis.

The observation density $p_{D|\mathbf{W},\mathbf{X}}(d|\mathbf{w}, \mathbf{x})$ is commonly reformulated mathematically as the *likelihood function*, defined by

$$l(\mathbf{w}|d, \mathbf{x}) = p_{D|\mathbf{W},\mathbf{X}}(d|\mathbf{w}, \mathbf{x}) \tag{2.8}$$

Moreover, insofar as the estimation of the parameter vector $\mathbf{w}$ is concerned, the evidence $p_D(d)$ in the denominator of the right-hand side of Eq. (2.7) plays merely the role of a *normalizing constant.* Accordingly, we may express Eq. (2.7) in words by stating the following:

> *The posterior density of the vector* $\mathbf{w}$ *parameterizing the regression model is proportional to the product of the likelihood function and the prior.*

That is,

$$\pi(\mathbf{w}|d, \mathbf{x}) \propto l(\mathbf{w}|d, \mathbf{x})\pi(\mathbf{w}) \tag{2.9}$$

where the symbol $\propto$ signifies proportionality.

The likelihood function $l(\mathbf{w}|d, \mathbf{x})$, considered on its own, provides the basis for the *maximum-likelihood (ML) estimate* of the parameter vector $\mathbf{w}$, as shown by

$$\mathbf{w}_{ML} = \arg\max_{\mathbf{w}} l(\mathbf{w}|d, \mathbf{x}) \tag{2.10}$$

For a more profound estimate of the parameter vector $\mathbf{w}$, however, we look to the posterior density $\pi(\mathbf{w}|d, \mathbf{x})$. Specifically, we define the *maximum a posteriori (MAP) estimate* of the parameter vector $\mathbf{w}$ by the formula

$$\mathbf{w}_{\text{MAP}} = \arg\max_{\mathbf{w}} \pi(\mathbf{w}|d, \mathbf{x}) \tag{2.11}$$

We say that the MAP estimator is more profound than the ML estimator for two important reasons:

1. The Bayesian paradigm for parameter estimation, rooted in the Bayes' theorem as shown in Eq. (2.7) and exemplified by the MAP estimator of Eq. (2.11), *exploits all the conceivable information about the parameter vector* $\mathbf{w}$. In contrast, the ML estimator of Eq. (2.10) lies on the *fringe* of the Bayesian paradigm, ignoring the prior.
2. The ML estimator relies solely on the observation model $(d, \mathbf{x})$ and may therefore lead to a nonunique solution. To enforce uniqueness and stability on the solution, *the prior* $\pi(\mathbf{w})$ *has to be incorporated into the formulation of the estimator*; this is precisely what is done in the MAP estimator.

Of course, the challenge in applying the MAP estimation procedure is how to come up with an appropriate prior, which makes MAP more computationally demanding than ML.

One last comment is in order. From a computational perspective, we usually find it more convenient to work with the logarithm of the posterior density rather than the posterior density itself. We are permitted to do this, since the logarithm is a monotonically increasing function of its argument. Accordingly, we may express the MAP estimator in the desired form by writing

$$\mathbf{w}_{\text{MAP}} = \arg\max_{\mathbf{w}} \log(\pi(\mathbf{w}|d, \mathbf{x})) \tag{2.12}$$

where "log" denotes the natural logarithm. A similar statement applies to the ML estimator.

Parameter Estimation in a Gaussian Environment

Let $\mathbf{x}_i$ and d_i denote the regressor applied to the environment and the resulting response, respectively, on the ith trial of an experiment performed on the environment. Let the experiment be repeated a total of N times. We thus express the training sample, available for parameter estimation, as

$$\mathcal{T} = \{\mathbf{x}_i, d_i\}_{i=1}^{N} \tag{2.13}$$

To proceed with the task of parameter estimation, we make the following assumptions:

Assumption 1: Statistical Independence and Identical Distribution

The N examples, constituting the training sample, are statistically *independent and identically distributed* (iid).

Assumption 2: Gaussianity

The environment, responsible for generation of the training sample $\mathcal{T}$, is *Gaussian distributed.*

More specifically, the expectational error in the linear regression model of Eq. (2.3) is described by a Gaussian density function of zero mean and common variance σ^2, as shown by

$$p_E(\varepsilon_i) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(-\frac{\varepsilon_i^2}{2\sigma^2}\right), \qquad i = 1, 2, ..., N \tag{2.14}$$

Assumption 3: Stationarity

The environment is stationary, which means that the parameter vector $\mathbf{w}$ is *fixed*, but unknown, throughout the N trials of the experiment.

More specifically, the M elements of the weight vector $\mathbf{w}$ are themselves assumed to be iid, with each element being governed by a Gaussian density function of zero mean and common variance σ_w^2. We may therefore express the prior for the kth element of the parameter vector $\mathbf{w}$ as

$$\pi(w_k) = \frac{1}{\sqrt{2\pi}\,\sigma_w} \exp\left(-\frac{w_k^2}{2\sigma_w^2}\right), \qquad k = 1, 2, ..., M \tag{2.15}$$

Rewriting Eq. (2.3) for the ith trial of the experiment performed on the environment, we have

$$d_i = \mathbf{w}^T\mathbf{x}_i + \varepsilon_i, \qquad i = 1, 2, ..., N \tag{2.16}$$

where d_i, $\mathbf{x}_i$, and ε_i are sample values (i.e., single-shot realizations) of the random variable D, the random vector $\mathbf{X}$, and the random variable E, respectively.

Let $\mathbb{E}$ denote the statistical expectation operator. Since, under Assumption 2, we have

$$\mathbb{E}[E_i] = 0, \qquad \text{for all } i \tag{2.17}$$

and

$$\text{var}[E_i] = \mathbb{E}[E_i^2] = \sigma^2, \qquad \text{for all } i \tag{2.18}$$

it follows from Eq. (2.16) that, for a given regressor $\mathbf{x}_i$,

$$\mathbb{E}[D_i] = \mathbf{w}^T\mathbf{x}_i, \qquad i = 1, 2, ..., N \tag{2.19}$$

$$\begin{aligned}\text{var}[D_i] &= \mathbb{E}[(D_i - \mathbb{E}[D_i])^2] \\ &= \mathbb{E}[E_i^2] \\ &= \sigma^2\end{aligned} \tag{2.20}$$

We thus complete the Gaussian implication of Assumption 2 by expressing the likelihood function for the ith trial, in light of Eq. (2.14), as

$$l(\mathbf{w}|d_i, x_i) = \frac{1}{\sqrt{2\pi}\,\sigma}\exp\left(-\frac{1}{2\sigma^2}(d_i - \mathbf{w}^T\mathbf{x}_i)^2\right), \qquad i = 1, 2, ..., N \tag{2.21}$$

Next, invoking the iid characterization of the N trials of the experiment on the environment under Assumption 1, we express the overall likelihood function for the experiment as

$$\begin{aligned}l(\mathbf{w}|d, \mathbf{x}) &= \prod_{i=1}^{N} l(\mathbf{w}|d_i, \mathbf{x}_i) \\ &= \frac{1}{(\sqrt{2\pi}\sigma)^N}\prod_{i=1}^{N}\exp\left(-\frac{1}{2\sigma^2}(d_i - \mathbf{w}^T\mathbf{x}_i)^2\right) \\ &= \frac{1}{(\sqrt{2\pi}\sigma)^N}\exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{N}(d_i - \mathbf{w}^T\mathbf{x}_i)^2\right)\end{aligned} \tag{2.22}$$

which accounts for the total empirical knowledge about the weight vector $\mathbf{w}$ contained in the training sample $\mathcal{T}$ of Eq. (2.13).

The only other source of information that remains to be accounted for is that contained in the prior $\pi(\mathbf{w})$. Invoking the zero-mean Gaussian characterization of the kth element of $\mathbf{w}$ described in Eq. (2.15), followed by the iid characterization of the M elements of $\mathbf{w}$ under Assumption 3, we write

$$\begin{aligned}\pi(\mathbf{w}) &= \prod_{k=1}^{M}\pi(w_k) \\ &= \frac{1}{(\sqrt{2\pi}\sigma_w)^M}\prod_{k=1}^{M}\exp\left(-\frac{w_k^2}{2\sigma_w^2}\right) \\ &= \frac{1}{(\sqrt{2\pi}\sigma_w)^M}\exp\left(-\frac{1}{2\sigma_w^2}\sum_{i=1}^{M}w_k^2\right) \\ &= \frac{1}{(\sqrt{2\pi}\sigma_w)^M}\exp\left(-\frac{1}{2\sigma_w^2}\|\mathbf{w}\|^2\right)\end{aligned} \tag{2.23}$$

where $\|\mathbf{w}\|$ is the *Euclidean norm* of the unknown parameter vector $\mathbf{w}$, defined by

$$\|\mathbf{w}\| = \left(\sum_{k=1}^{M} w_k^2 \right)^{1/2} \tag{2.24}$$

Hence, substituting Eqs. (2.22) and (2.23) into Eq. (2.9), and then simplifying the result, we get the posterior density

$$\pi(\mathbf{w}|d, \mathbf{x}) \propto \exp\left[-\frac{1}{2\sigma^2} \sum_{i=1}^{N} (d_i - \mathbf{w}^T \mathbf{x}_i)^2 - \frac{1}{2\sigma_w^2} \|\mathbf{w}\|^2 \right] \tag{2.25}$$

We are now fully equipped to apply the MAP formula of Eq. (2.12) to the estimation problem at hand. Specifically, substituting Eq. (2.25) into this formula, we get

$$\hat{\mathbf{w}}_{\text{MAP}}(N) = \max_{\mathbf{w}} \left[-\frac{1}{2} \sum_{i=1}^{N} (d_i - \mathbf{w}^T \mathbf{x}_i)^2 - \frac{\lambda}{2} \|\mathbf{w}\|^2 \right] \tag{2.26}$$

where we have introduced the new parameter

$$\lambda = \frac{\sigma^2}{\sigma_w^2} \tag{2.27}$$

Now we define the *quadratic function*

$$\mathscr{E}(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{N} (d_i - \mathbf{w}^T \mathbf{x}_i)^2 + \frac{\lambda}{2} \|\mathbf{w}\|^2 \tag{2.28}$$

Clearly, maximization of the argument in Eq. (2.26) with respect to $\mathbf{w}$ is equivalent to minimization of the quadratic function $\mathscr{E}(\mathbf{w})$. Accordingly, the optimum estimate $\hat{\mathbf{w}}_{\text{MAP}}$ is obtained by differentiating the function $\mathscr{E}(\mathbf{w})$ with respect to $\mathbf{w}$ and setting the result equal to zero. In so doing, we obtain the desired MAP estimate of the M-by-1 parameter vector as

$$\hat{\mathbf{w}}_{\text{MAP}}(N) = [\mathbf{R}_{xx}(N) + \lambda \mathbf{I}]^{-1} \mathbf{r}_{dx}(N) \tag{2.29}$$

where we have introduced two matrices and a vector:

1. the *time-averaged* M-by-M *correlation matrix* of the regressor $\mathbf{x}$, which is defined by

$$\hat{\mathbf{R}}_{xx}(N) = \sum_{i=1}^{N} \sum_{j=1}^{N} \mathbf{x}_i \mathbf{x}_j^T \tag{2.30}$$

   where $\mathbf{x}_i \mathbf{x}_j^T$ is the outer product of the regressors $\mathbf{x}_i$ and $\mathbf{x}_j$, applied to the environment on the ith and jth experimental trials;
2. the M-by-M *identity matrix* $\mathbf{I}$ whose M diagonal elements are unity and the remaining elements are all zero;
3. the *time-averaged* M-by-1 cross-correlation vector of the regressor $\mathbf{x}$ and the desired response d, which is defined by

$$\hat{\mathbf{r}}_{dx}(N) = \sum_{j=1}^{N} \mathbf{x}_i d_i \tag{2.31}$$

The correlations $\hat{\mathbf{R}}_{xx}(N)$ and $\hat{\mathbf{r}}_{dx}(N)$ are both averaged over all the N examples of the training sample $\mathcal{T}$—hence the use of the term "time averaged."

Suppose we assign a large value to the variance σ_w^2, which has the implicit effect of saying that the prior distribution of each element of the parameter vector $\mathbf{w}$ is essentially *uniform* over a wide range of possible values. Under this condition, the parameter λ is essentially zero and the formula of Eq. (2.29) reduces to the ML estimate

$$\hat{\mathbf{w}}_{\text{ML}}(N) = \hat{\mathbf{R}}_{xx}^{-1}(N)\hat{\mathbf{r}}_{dx}(N) \tag{2.32}$$

which supports the point we made previously: The ML estimator relies solely on the observation model exemplified by the training sample $\mathcal{T}$. In the statistics literature on linear regression, the equation

$$\hat{\mathbf{R}}_{xx}(N)\,\hat{\mathbf{w}}_{\text{ML}}(N) = \hat{\mathbf{r}}_{dx}(N) \tag{2.33}$$

is commonly referred to as the *normal equation*; the ML estimator $\hat{\mathbf{w}}_{\text{ML}}$ is, of course, the solution of this equation. It is also of interest that the ML estimator is an *unbiased* estimator, in the sense that for an infinitely large training sample $\mathcal{T}$, we find that, in the limit, $\hat{\mathbf{w}}_{\text{ML}}$ converges to the parameter vector $\mathbf{w}$ of the unknown stochastic environment, provided that the regressor $\mathbf{x}(n)$ and the response $d(n)$ are drawn from *jointly ergodic processes*, in which case time averages may be substituted for ensemble averages. Under this condition, in Problem 2.4, it is shown that

$$\lim_{N\to\infty} \hat{\mathbf{w}}_{\text{ML}}(N) = \mathbf{w}$$

In contrast, the MAP estimator of Eq. (2.29) is a *biased* estimator, which therefore prompts us to make the following statement:

> *In improving the stability of the maximum likelihood estimator through the use of regularization (i.e., the incorporation of prior knowledge), the resulting maximum* a posteriori *estimator becomes biased.*

In short, we have a tradeoff between stability and bias.

2.4 RELATIONSHIP BETWEEN REGULARIZED LEAST-SQUARES ESTIMATION AND MAP ESTIMATION

We may approach the estimation of the parameter vector $\mathbf{w}$ in another way by focusing on a *cost* function $\mathscr{E}_0(\mathbf{w})$ defined as the *squared expectational errors summed over the N experimental trials on the environment.* Specifically, we write

$$\mathscr{E}_0(\mathbf{w}) = \sum_{i=1}^{N} \varepsilon_i^2(\mathbf{w})$$

where we have included $\mathbf{w}$ in the argument of ε_i to stress the fact that the uncertainty in the regression model is due to the vector $\mathbf{w}$. Rearranging terms in Eq. (2.16), we obtain

$$\varepsilon_i(\mathbf{w}) = d_i - \mathbf{w}^T\mathbf{x}_i, \qquad i = 1, 2, \ldots, N \tag{2.34}$$

Substituting this equation into the expression for $\mathscr{E}_0(\mathbf{w})$ yields

$$\mathscr{E}_0(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(d_i - \mathbf{w}^T\mathbf{x}_i)^2 \tag{2.35}$$

which relies solely on the training sample $\mathcal{T}$. Minimizing this cost function with respect to $\mathbf{w}$ yields a formula for the *ordinary least-squares estimator* that is identical to the maximum-likelihood estimator of Eq. (2.32), and hence there is a distinct possibility of obtaining a solution that lacks uniqueness and stability.

To overcome this serious problem, the customary practice is to expand the cost function of Eq. (2.35) by adding a new term as follows:

$$\begin{aligned}\mathscr{E}(\mathbf{w}) &= \mathscr{E}_0(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2 \\ &= \frac{1}{2}\sum_{i=1}^{N}(d_i - \mathbf{w}^T\mathbf{x}_i)^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2\end{aligned} \tag{2.36}$$

This expression is identical to the function defined in Eq. (2.28). The inclusion of the squared Euclidean norm $\|\mathbf{w}\|^2$ is referred to as *structural regularization.* Correspondingly, the scalar λ is referred to as the *regularization parameter.*

When $\lambda = 0$, the implication is that we have complete confidence in the observation model exemplified by the training sample $\mathcal{T}$. At the other extreme, when $\lambda = \infty$, the implication is that we have no confidence in the observation model. In practice, the regularization parameter λ is chosen somewhere between these two limiting cases.

In any event, for a prescribed value of the regularization parameter λ, the solution of the *regularized method of least squares*, obtained by minimizing the regularized cost function of Eq. (2.36) with respect to the parameter vector $\mathbf{w}$, is identical to the MAP estimate of Eq. (2.29). This particular solution is referred to as the *regularized least-squares* (RLS) *solution*.

2.5 COMPUTER EXPERIMENT: PATTERN CLASSIFICATION

In this section, we repeat the computer experiment performed on the pattern-classification problem studied in Chapter 1, where we used the perceptron algorithm. As before, the double-moon structure, providing the training as well as the test data, is that shown in Fig. 1.8. This time, however, we use the method of least squares to perform the classification.

Figure 2.2 presents the results of training the least squares algorithm for the separation distance between the two moons set at $d = 1$. The figure shows the decision boundary constructed between the two moons. The corresponding results obtained using the perceptron algorithm for the same setting $d = 1$ were presented in Fig. 1.9. Comparing these two figures, we make the following interesting observations:

1. The decision boundaries constructed by the two algorithms are both linear, which is intuitively satisfying. The least-squares algorithm discovers the asymmetric

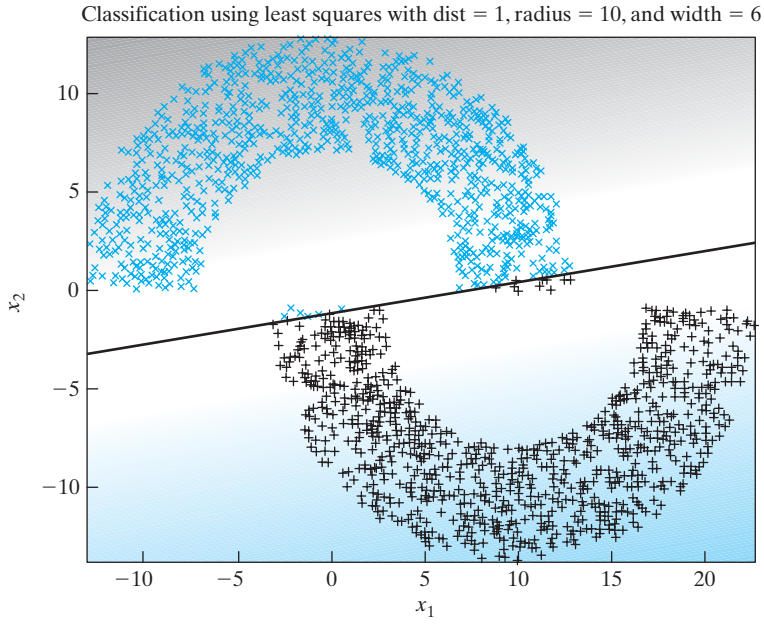


FIGURE 2.2 Least Squares classification of the double-moon of Fig. 1.8 with distance $d = 1$.

manner in which the two moons are positioned relative to each other, as seen by the positive slope of the decision boundary in Fig. 2.2. Interestingly enough, the perceptron algorithm completely ignores this asymmetry by constructing a decision boundary that is parallel to the x-axis.

2. For the separation distance $d = 1$, the two moons are linearly separable. The perceptron algorithm responds perfectly to this setting; on the other hand, in discovering the asymmetric feature of the double-moon figure, the method of least squares ends up misclassifying the test data, incurring a classification error of 0.8%.
3. Unlike the perceptron algorithm, the method of least squares computes the decision boundary in one shot.

Figure 2.3 presents the results of the experiment performed on the double-moon patterns for the separation distance $d = -4$, using the method of least squares. As expected, there is now a noticeable increase in the classification error, namely, 9.5%. Comparing this performance with the 9.3% classification error of the perceptron algorithm for the same setting, which was reported in Fig. 1.10, we see that the classification performance of the method of least squares has degraded slightly.

The important conclusion to be drawn from the pattern-classification computer experiments of Sections 1.5 and 2.5 is as follows:

Although the perceptron and the least-squares algorithms are both linear, they operate differently in performing the task of pattern classification.

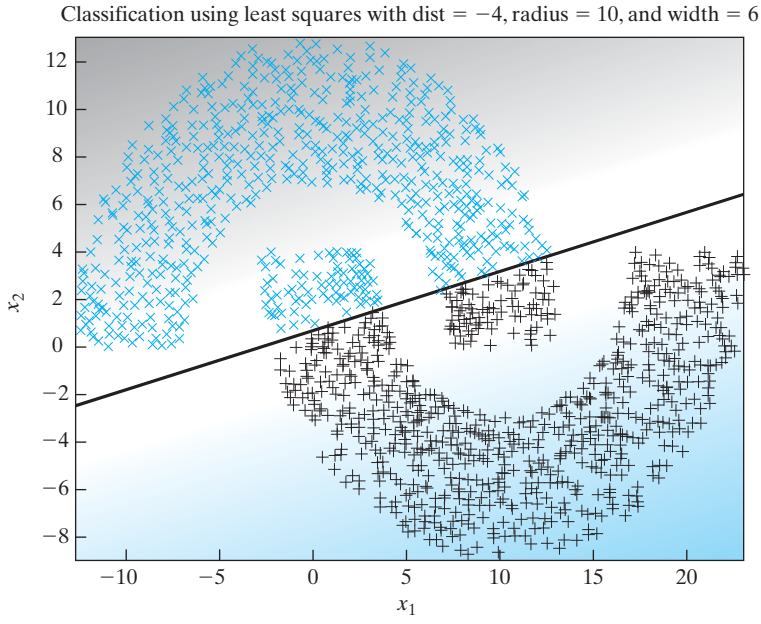


FIGURE 2.3 Least-squares classification of the double-moon of Fig. 1.8 with distance $d = -4$.

2.6 THE MINIMUM-DESCRIPTION-LENGTH PRINCIPLE

The representation of a stochastic process by a linear model may be used for synthesis or analysis. In *synthesis*, we generate a desired time series by assigning a formulated set of values to the parameters of the model and feeding it with *white noise* of zero mean and prescribed variance; the model so obtained is referred to as a *generative model.* In *analysis*, on the other hand, we *estimate* the parameters of the model by processing a given time series of finite length, using the Bayesian approach or the regularized method of least squares. Insofar as the estimation is statistical, we need an appropriate measure of the fit between the model and the observed data. We refer to this second problem as that of *model selection.* For example, we may want to estimate the number of degrees of freedom (i.e., adjustable parameters) of the model, or even the general structure of the model.

A plethora of methods for model selection has been proposed in the statistics literature, with each one of them having a goal of its own. With the goals being different, it is not surprising to find that the different methods yield wildly different results when they are applied to the same data set (Grünwald, 2007).

In this section, we describe a well-proven method, called the *minimum-description-length (MDL) principle* for model selection, which was pioneered by Rissanen (1978).

Inspiration for the development of the MDL principle is traced back to *Kolmogorov complexity theory.* In this remarkable theory, the great mathematician

Kolmogorov defined complexity as follows (Kolmogorov, 1965; Li and Vitányi, 1993; Cover and Thomas, 2006; Grünwald, 2007):

> *The algorithmic (descriptive) complexity of a data sequence is the length of the shortest binary computer program that prints out the sequence and then halts.*

What is truly amazing about this definition of complexity is the fact that it looks to the computer, the most general form of data compressor, rather than the notion of probability distribution for its basis.

Using the fundamental concept of Kolmogorov complexity, we may develop a *theory of idealized inductive inference*, the goal of which is to find "regularity" in a given data sequence. The idea of viewing learning as trying to find "regularity" provided the first insight that was used by Rissanen in formulating the MDL principle. The second insight used by Rissanen is that regularity itself may be identified with the "ability to compress."

Thus, the MDL principle combines these two insights, one on regularity and the other on the ability to compress, to view the process of *learning as data compression*, which, in turn, teaches us the following:

> *Given a set of hypotheses, $\mathcal{H}$, and a data sequence d, we should try to find the particular hypothesis or some combination of hypotheses in $\mathcal{H}$, that compresses the data sequence d the most.*

This statement sums up what the MDL principle is all about very succinctly. The symbol d for a sequence should not be confused with the symbol d used previously for desired response.

There are several versions of the MDL principle that have been described in the literature. We will focus on the oldest, but simplest and most well-known version, known as the *simplistic two-part code MDL principle* for probabilistic modeling. By the term "simplistic," we mean that the codelengths under consideration are not determined in an optimal fashion. The terms "code" and "codelengths" used herein pertain to the process of encoding the data sequence in the shortest or *least redundant* manner.

Suppose that we are given a candidate model or model class $\mathcal{M}$. With all the elements of $\mathcal{M}$ being probabilistic sources, we henceforth refer to a point hypothesis as p rather than $\mathcal{H}$. In particular, we look for the probability density function $p \in \mathcal{M}$ that best explains a given data sequence d. The two-part code MDL principle then tells us to look for the (point) hypothesis $p \in \mathcal{M}$ that minimizes the description length of p, which we denote by $L_1(p)$, and the description length of the data sequence d when it is encoded with the help of p, which we denote as $L_2(d \mid p)$. We thus form the sum

$$L_{12}(p, d) = L_1(p) + L_2(d|p)$$

and pick the particular point hypothesis $p \in \mathcal{M}$ that minimizes $L_{12}(p, d)$.

It is crucial that p itself be encoded as well here. Thus, in finding the hypothesis that compresses the data sequence d the most, we must encode (describe or compress) the data in such a way that a decoder can retrieve the data even without knowing the hypothesis in advance. This can be done by explicitly encoding a hypothesis, as in the foregoing two-part code principle; it can also be done in quite different ways—for example, by averaging over hypotheses (Grünwald, 2007).

Model-Order Selection

Let $\mathcal{M}^{(1)}, \mathcal{M}^{(2)}, ..., \mathcal{M}^{(k)}, ...$, denote a family of linear regression models that are associated with the parameter vector $\mathbf{w}^k \in \mathcal{W}^k$, where the model order $k = 1, 2, ...$; that is, the weight spaces $\mathcal{W}^{(1)}, \mathcal{W}^{(2)}, ..., \mathcal{W}^{(k)}, ...$, are of increasing dimensionality. The issue of interest is to identify the model that best explains an unknown environment that is responsible for generating the training sample $\{\mathbf{x}_i, d_i\}_{i=1}^N$, where $\mathbf{x}_i$ is the stimulus and d_i is the corresponding response. What we have just described is the *model-order selection problem.*

In working through the statistical characterization of the composite length $L_{12}(p, d)$, the two-part code MDL principle tells us to pick the kth model that is the mimimizer

$$\min_k \Bigg\{ \overbrace{-\log p(d_i|\mathbf{w}^{(k)})\pi(\mathbf{w}^{(k)})}^{\text{Error term}} + \overbrace{\frac{k}{2}\log(N) + O(k)}^{\text{Complexity term}} \Bigg\}, \qquad \begin{matrix} k = 1, 2, ... \\ i = 1, 2, ..., N \end{matrix} \tag{2.37}$$

where $\pi(\mathbf{w}^{(k)})$ is the prior distribution of the parameter vector $\mathbf{w}^{(k)}$, and the last term of the expression is of the order of model order k (Rissanen, 1989; Grünwald, 2007). For a large sample size N, this last term gets overwhelmed by the second term of the expression $\frac{k}{2}\log(N)$. The expression in Eq. (2.37) is usually partitioned into two terms:

- the *error term*, denoted by $-\log(p(d_i|\mathbf{w}^{(k)})\, \pi(\mathbf{w}^{(k)})$, which relates to the model and the data;
- the *hypothesis complexity term,* denoted by $\frac{k}{2}\log(N) + O(k)$, which relates to the model alone.

In practice, the $O(k)$ term is often ignored to simplify matters when applying Eq. (2.37), with mixed results. The reason for mixed results is that the $O(k)$ term can be rather large. For linear regression models, however, it can be explicitly and efficiently computed, and the resulting procedures tend to work quite well in practice.

Note also that the expression of Eq. (2.37) without the prior distribution $\pi(\mathbf{w}^{(k)})$ was first formulated in Rissanen (1978).

If it turns out that we have more than one minimizer of the expression in Eq. (2.37), then we pick the model with the smallest hypothesis complexity term. And if this move still leaves us with several candidate models, then we do not have any further choice but to work with one of them (Grünwald, 2007).

Attributes of the MDL Principle

The MDL principle for model selection offers two important attributes (Grünwald, 2007):

1. When we have two models that fit a given data sequence equally well, the MDL principle will pick the one that is the "simplest" in the sense that it allows the use of a shorter description of the data. In other words, the MDL principle implements a precise form of *Occam's razor*, which states a preference for simple theories:

   Accept the simplest explanation that fits the data.

2. The MDL principle is a *consistent* model selection estimator in the sense that it converges to the true model order as the sample size increases.

Perhaps the most significant point to note is that, in nearly all of the applications involving the MDL principle, few, if any, anomalous results or models with undesirable properties have been reported in the literature.

2.7 FINITE SAMPLE-SIZE CONSIDERATIONS

A serious limitation of the maximum-likelihood or ordinary least-squares approach to parameter estimation is the nonuniqueness and instability of the solution, which is attributed to complete reliance on the observation model (i.e., the training sample $\mathcal{T}$); the traits of nonuniqueness and instability in characterizing a solution are also referred to as an *overfitting* problem in the literature. To probe more deeply into this practical issue, consider the generic regressive model

$$d = f(\mathbf{x}, \mathbf{w}) + \varepsilon \tag{2.38}$$

where $f(\mathbf{x}, \mathbf{w})$ is a deterministic function of the regressor $\mathbf{x}$ for some $\mathbf{w}$ parameterizing the model and ε is the expectational error. This model, depicted in Fig. 2.4a, is a *mathematical* description of a stochastic environment; its purpose is to *explain* or *predict* the response d produced by the regressor $\mathbf{x}$.

Figure 2.4b is the corresponding *physical* model of the environment, where $\hat{\mathbf{w}}$ denotes an *estimate* of the unknown parameter vector $\mathbf{w}$. The purpose of this second model is to *encode the empirical knowledge represented by the training sample* $\mathcal{T}$, as shown by

$$\mathcal{T} \rightarrow \hat{\mathbf{w}} \tag{2.39}$$

In effect, the physical model provides an *approximation* to the regression model of Fig. 2.4a. Let the actual response of the physical model, produced in response to the input vector $\mathbf{x}$, be denoted by

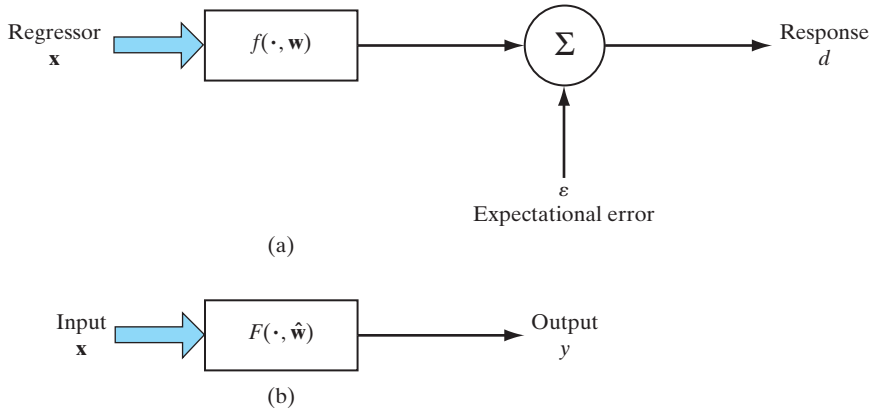


FIGURE 2.4 (a) Mathematical model of a stochastic environment, parameterized by the vector $\mathbf{w}$. (b) Physical model of the environment, where $\hat{\mathbf{w}}$ is an estimate of the unknown parameter vector $\mathbf{w}$.

$$y = F(\mathbf{x}, \hat{\mathbf{w}}) \tag{2.40}$$

where $F(\cdot, \hat{\mathbf{w}})$ is the input–output function realized by the physical model; the y in Eq. (2.40) is a sample value of random variable Y. Given the training sample $\mathcal{T}$ of Eq. (2.39), the estimator $\hat{\mathbf{w}}$ is the minimizer of the cost function

$$\mathscr{E}(\hat{\mathbf{w}}) = \frac{1}{2}\sum_{i=1}^{N}(d_i - F(\mathbf{x}_i, \hat{\mathbf{w}}))^2 \tag{2.41}$$

where the factor $\frac{1}{2}$ has been used to be consistent with earlier notations. Except for the scaling factor $\frac{1}{2}$, the cost function $\mathscr{E}(\hat{\mathbf{w}})$ is the squared difference between the environmental (desired) response d and the actual response y of the physical model, computed over the entire training sample $\mathcal{T}$.

Let the symbol $\mathbb{E}_{\mathcal{T}}$ denote the *average operator* taken over the entire training sample $\mathcal{T}$. The variables or their functions that come under the average operator $\mathbb{E}_{\mathcal{T}}$ are denoted by $\mathbf{x}$ and d; the pair $(\mathbf{x}, d)$ represents an example in the training sample $\mathcal{T}$. In contrast, the statistical expectation operator $\mathbb{E}$ acts on the whole ensemble of $\mathbf{x}$ and d, which includes $\mathcal{T}$ as a subset. The difference between the operators $\mathbb{E}$ and $\mathbb{E}_{\mathcal{T}}$ should be very carefully noted in what follows.

In light of the transformation described in Eq. (2.39), we may interchangeably use $F(\mathbf{x}, \hat{\mathbf{w}})$ and $F(\mathbf{x}, \mathcal{T})$ and therefore rewrite Eq. (2.41) in the equivalent form

$$\mathscr{E}(\hat{\mathbf{w}}) = \frac{1}{2}\mathbb{E}_{\mathcal{T}}[(d - F(\mathbf{x}, \mathcal{T}))^2] \tag{2.42}$$

By adding and then subtracting $f(\mathbf{x}, \mathbf{w})$ to the argument $(d - F(\mathbf{x}, \mathcal{T}))$ and next using Eq. (2.38), we may write

$$\begin{aligned} d - f(\mathbf{x}, \mathcal{T}) &= [d - f(\mathbf{x}, \mathbf{w})] + [f(\mathbf{x}, \mathbf{w}) - F(\mathbf{x}, \mathcal{T})] \\ &= \varepsilon + [f(\mathbf{x}, \mathbf{w}) - F(\mathbf{x}, \mathcal{T})] \end{aligned}$$

By substituting this expression into Eq. (2.42) and then expanding terms, we may recast the cost function $\mathscr{E}(\hat{\mathbf{w}})$ in the equivalent form

$$\mathscr{E}(\hat{\mathbf{w}}) = \frac{1}{2}\mathbb{E}_{\mathcal{T}}[\varepsilon^2] + \frac{1}{2}\mathbb{E}_{\mathcal{T}}[(f(\mathbf{x}, \mathbf{w}) - F(\mathbf{x}, \mathcal{T}))^2] + \mathbb{E}_{\mathcal{T}}[\varepsilon f(\mathbf{x}, \mathbf{w}) - \varepsilon F(\mathbf{x}, \mathcal{T})] \tag{2.43}$$

However, the last expectation term on the right-hand side of Eq. (2.43) is zero, for two reasons:

- The expectational error ε is uncorrelated with the regression function $f(\mathbf{x}, \mathbf{w})$.
- The expectational error ε pertains to the regression model of Fig. 2.4a, whereas the approximating function $F(\mathbf{x},\hat{\mathbf{w}})$ pertains to the physical model of Fig. 2.4b.

Accordingly, Eq. (2.43) reduces to

$$\mathscr{E}(\hat{\mathbf{w}}) = \frac{1}{2}\mathbb{E}_{\mathcal{T}}[\varepsilon^2] + \frac{1}{2}\mathbb{E}_{\mathcal{T}}[(f(\mathbf{x}, \mathbf{w}) - F(\mathbf{x}, \mathcal{T}))^2] \tag{2.44}$$

The term $\mathbb{E}_{\mathcal{T}}[\varepsilon^2]$ on the right-hand side of Eq. (2.44) is the *variance* of the expectational (regressive modeling) error ε, evaluated over the training sample $\mathcal{T}$; here it is assumed that ε has zero mean. This variance represents the *intrinsic error* because it is independent of the estimate $\hat{\mathbf{w}}$. Hence, the estimator $\hat{\mathbf{w}}$ that is the minimizer of the cost function $\mathscr{E}(\hat{\mathbf{w}})$ will also minimize the ensemble average of the squared distance between the regression function $f(\mathbf{x}, \mathbf{w})$ and the approximating function $F(\mathbf{x}, \hat{\mathbf{w}})$. In other words, the *natural measure* of the effectiveness of $F(\mathbf{x}, \hat{\mathbf{w}})$ as a predictor of the desired response d is defined as follows (ignoring the scaling factor ½):

$$L_{\text{av}}(f(\mathbf{x}, \mathbf{w}), F(\mathbf{x}, \hat{\mathbf{w}})) = \mathbb{E}_{\mathcal{T}}[(f(\mathbf{x}, \mathbf{w}) - F(\mathbf{x}, \mathcal{T}))^2] \tag{2.45}$$

This natural measure is fundamentally important because it provides the mathematical basis for the tradeoff between the bias and variance that results from the use of $F(\mathbf{x}, \hat{\mathbf{w}})$ as the approximation to $f(\mathbf{x}, \mathbf{w})$.

Bias–Variance Dilemma

From Eq. (2.38), we find that the function $f(\mathbf{x}, \mathbf{w})$ is equal to the conditional expectation $\mathbb{E}(d|\mathbf{x})$. We may therefore redefine the squared distance between $f(\mathbf{x})$ and $F(\mathbf{x}, \hat{\mathbf{w}})$ as

$$L_{\text{av}}(f(\mathbf{x}, \mathbf{w}), F(\mathbf{x}, \hat{\mathbf{w}})) = \mathbb{E}_{\mathcal{T}}[(\mathbb{E}[d|\mathbf{x}] - F(\mathbf{x}, \mathcal{T}))^2] \tag{2.46}$$

This expression may therefore be viewed as the average value of the estimation error between the regression function $f(\mathbf{x}, \mathbf{w}) = \mathbb{E}[d|\mathbf{x}]$ and the approximating function $F(\mathbf{x}, \hat{\mathbf{w}})$, evaluated over the entire training sample $\mathcal{T}$. Notice that the conditional mean $\mathbb{E}[d|\mathbf{x}]$ has a constant expectation with respect to the training sample $\mathcal{T}$. Next we write

$$\mathbb{E}[d|\mathbf{x}] - F(\mathbf{x}, \mathcal{T}) = (\mathbb{E}[d|\mathbf{x}] - \mathbb{E}_{\mathcal{T}}[F(\mathbf{x}, \mathcal{T})]) + (\mathbb{E}_{\mathcal{T}}[F(\mathbf{x}, \mathcal{T})] - F(\mathbf{x}, \mathcal{T}))$$

where we have simply added and then subtracted the average $\mathbb{E}_{\mathcal{T}}[F(\mathbf{x}, \mathcal{T})]$. By proceeding in a manner similar to that described for deriving Eq. (2.43) from Eq. (2.42), we may reformulate Eq. (2.46) as the sum of two terms (see Problem 2.5):

$$L_{\text{av}}(f(\mathbf{x}), F(\mathbf{x}, \mathcal{T})) = B^2(\hat{\mathbf{w}}) + V(\hat{\mathbf{w}}) \tag{2.47}$$

where $B(\hat{\mathbf{w}})$ and $V(\hat{\mathbf{w}})$ are themselves respectively defined by

$$B(\hat{\mathbf{w}}) = \mathbb{E}_{\mathcal{T}}[F(\mathbf{x}, \mathcal{T})] - \mathbb{E}[d|\mathbf{x}] \tag{2.49}$$

and

$$V(\hat{\mathbf{w}}) = \mathbb{E}_{\mathcal{T}}[(F(\mathbf{x}, \mathcal{T}) - \mathbb{E}_{\mathcal{T}}[F(\mathbf{x}, \mathcal{T}])^2] \tag{2.49}$$

We now make two important observations:

1. The first term, $B(\hat{\mathbf{w}})$, is the *bias* of the average value of the approximating function $F(\mathbf{x}, \mathcal{T})$, measured with respect to the regression function $f(\mathbf{x}, \mathbf{w}) = \mathbb{E}[d|\mathbf{x}]$. Thus, $B(\hat{\mathbf{w}})$ represents the inability of the physical model defined by the function $F(\mathbf{x}, \hat{\mathbf{w}})$ to accurately approximate the regression function $f(\mathbf{x}, \mathbf{w}) = \mathbb{E}[d|\mathbf{x}]$. We may therefore view the bias $B(\hat{\mathbf{w}})$ as an *approximation error.*
2. The second term, $V(\hat{\mathbf{w}})$, is the *variance* of the approximating function $F(\mathbf{x}, \mathcal{T})$, measured over the entire training sample $\mathcal{T}$. Thus, $V(\hat{\mathbf{w}})$ represents the inadequacy

of the empirical knowledge contained in the training sample $\mathcal{T}$ about the regression function $f(\mathbf{x}, \mathbf{w})$. We may therefore view the variance $V(\hat{\mathbf{w}})$ as the manifestation of an *estimation error*.

Figure 2.5 illustrates the relations between the target (desired) and approximating functions; it shows how the estimation errors, namely, the bias and variance, accumulate. To achieve good overall performance, the bias $B(\hat{\mathbf{w}})$ and the variance $V(\hat{\mathbf{w}})$ of the approximating function $F(\mathbf{x}, \hat{\mathbf{w}}) = F(\mathbf{x}, \mathcal{T})$ would both have to be small.

Unfortunately, we find that in a complex physical model that learns by example and does so with a training sample of limited size, the price for achieving a small bias is a large variance. For any physical model, it is only when the size of the training sample becomes infinitely large that we can hope to eliminate both bias and variance at the same time. Accordingly, we have a *bias–variance dilemma*, the consequence of which is prohibitively slow convergence (Geman et al., 1992). The bias–variance dilemma may be circumvented if we are willing to *purposely* introduce bias, which then makes it possible to eliminate the variance or to reduce it significantly. Needless to say, we must be sure that the bias built into the design of the physical model is harmless. In the context of pattern classification, for example, the bias is said to be harmless in the sense that it will contribute significantly to the mean-square error only if we try to infer regressions that are not in the anticipated class.

Explanatory notes on what Fig. 2.5 is depicting:

1. The shaded inner space of the figure is a subset of the outer space:
   The outer space represents the ensemble of regression functions $f(\cdot, \mathbf{w})$.
   The inner space represents the ensemble of approximating functions $F(\cdot, \hat{\mathbf{w}})$.
2. The figure shows three points, two fixed and one random:
   $\mathbb{E}[d|\mathbf{x}]$, fixed-point, is averaged over the outer space
   $\mathbb{E}_{\mathcal{T}}[F(\mathbf{x}, \mathcal{T})]$, second fixed-point, is averaged over the inner space
   $F(\mathbf{x}, \mathcal{T})$ is randomly distributed inside the inner space
3. Statistical parameters, embodied in the figure:
   $B(\mathbf{w})$ = bias, denoting the distance between $\mathbb{E}[d|\mathbf{x}]$ and $\mathbb{E}_{\mathcal{T}}[F(\mathbf{x}, \mathcal{T})]$.
   $V(\mathbf{w})$ = variance, denoting the squared distance between $F(\mathbf{x}, \mathcal{T})$ and $\mathbb{E}_{\mathcal{T}}[F(\mathbf{x}, \mathcal{T})]$, averaged over the training sample $\mathcal{T}$.
   $B^2(\mathbf{w}) + V(\mathbf{w})$ = squared distance between $F(\mathbf{x}, \mathcal{T})$ and $\mathbb{E}[d|\mathbf{x}]$ averaged over the training sample $\mathcal{T}$.

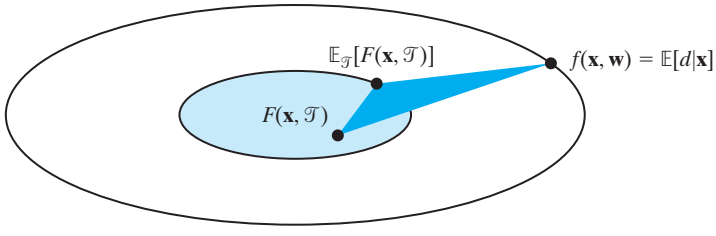


FIGURE 2.5 Decomposition of the natural measure $L_{\text{av}}(f(\mathbf{x}, \mathbf{w}), F(\mathbf{x}, \hat{\mathbf{w}}))$, defined in Eq. (2.46), into bias and variance terms for linear regression models.

In general, the bias must be *designed* for each specific application of interest. A practical way of achieving such an objective is to use a *constrained* network architecture, which usually performs better than a general-purpose architecture.

2.8 THE INSTRUMENTAL-VARIABLES METHOD

In studying the linear regression model, first from the perspective of Bayesian theory in Section 2.3 and then from the perspective of the method of least squares in Section 2.4, we pointed out that both approaches yield the same solution for the parameter vector $\mathbf{w}$ of the unknown stochastic environment depicted in Fig. 2.1, namely, Eq. (2.29) for the regularized linear regression model and Eq. (2.32) for the unregularized version. Both of these formulas were derived for a Gaussian environment, on the premise that the regressor (i.e., the input signal) $\mathbf{x}$ and the desired response d are both noiseless. What if, however, we find that the regressor $\mathbf{x}$ can be observed only in the presence of additive noise, as could happen in practice? That is, the noisy regressor is now defined by

$$\mathbf{z}_i = \mathbf{x}_i + \boldsymbol{\nu}_i \tag{2.50}$$

where $\boldsymbol{\nu}_i$ is the measurement noise associated with the observation of $\mathbf{x}_i$ in the ith realization of the training sample $\mathcal{T}$. If we were to apply the unregularized formula of Eq. (2.32), we would obtain a modified solution for the parameter vector $\mathbf{w}$ of the unknown stochastic environment:

$$\hat{\mathbf{w}}_{\text{ML}} = \hat{\mathbf{R}}_{zz}^{-1}\hat{\mathbf{r}}_{dz} \tag{2.51}$$

where $\hat{\mathbf{R}}_{zz}$ is the time-averaged correlation function of the noisy regressor $\mathbf{z}$ and $\hat{\mathbf{r}}_{dz}$ is the corresponding time-averaged cross-correlation function between the desired response d and $\mathbf{z}$. To simplify matters, we have ignored the dependence of these two correlation functions on the size of the training sample. Assuming that the measurement noise vector $\boldsymbol{\nu}$ is white noise with zero mean and correlation matrix $\sigma_\nu^2\mathbf{I}$, where $\mathbf{I}$ is the identity matrix, we obtain the following correlation functions:

$$\hat{\mathbf{R}}_{zz} = \hat{\mathbf{R}}_{xx} + \sigma_\nu^2\mathbf{I}$$

and

$$\hat{\mathbf{r}}_{dz} = \hat{\mathbf{r}}_{dx}$$

Correspondingly, the maximum-likelihood estimator assumes the new form

$$\hat{\mathbf{w}}_{\text{ML}} = (\hat{\mathbf{R}}_{xx} + \sigma_\nu^2\mathbf{I})^{-1}\hat{\mathbf{r}}_{dx} \tag{2.52}$$

which, in mathematical terms, is identical to the MAP formula of Eq. (2.29) with the regularization parameter λ set equal to the noise variance σ_ν^2. This observation leads us to make the following statement:

> *The presence of additive noise in the regressor* $\mathbf{z}$ *(with the right noise variance) has the beneficial effect of stabilizing the maximum-likelihood estimator, but at the expense of introducing a bias into the solution.*

This is quite an ironic statement: The addition of noise acts as a regularizer (stabilizer)!

Suppose, however, the requirement is to produce a solution for the unknown parameter vector $\mathbf{w}$ that is desirably *asymptotically unbiased.* In such a situation, we may resort to the *method of instrumental variables* (Young, 1984). This method relies on the introduction of a set of instrumental variables, denoted by the vector $\hat{\mathbf{x}}$ that has the same dimensionality as the noisy regressor $\mathbf{z}$ and satisfies the following two properties:

Property 1. The instrumental vector $\hat{\mathbf{x}}$ is highly correlated with the noiseless regressor $\mathbf{x}$, as shown by

$$\mathbb{E}[x_j \hat{x}_k] \neq 0 \qquad \text{for all } j \text{ and } k \tag{2.53}$$

where x_j is the jth element of the noiseless regressor $\mathbf{x}$ and $\hat{x}_k$ is the kth element of the instrumental vector $\hat{\mathbf{x}}$.

Property 2. The instrumental vector $\hat{\mathbf{x}}$ and the measurement noise vector $\boldsymbol{\nu}$ are statistically independent, as shown by

$$\mathbb{E}[\nu_j \hat{x}_k] = 0 \qquad \text{for all } j \text{ and } k \tag{2.54}$$

Equipped with the instrumental vector $\hat{\mathbf{x}}$ that satisfies these two properties, we compute the following correlation functions:

1. The noisy regressor $\mathbf{z}$ is correlated with the instrumental vector $\hat{\mathbf{x}}$, obtaining the cross-correlation matrix

$$\hat{\mathbf{R}}_{z\hat{x}} = \sum_{i=1}^{N} \hat{\mathbf{x}}_i \mathbf{z}_i^T \tag{2.55}$$

   where $\mathbf{z}_i$ is the ith regressor of the noisy training sample $\{\mathbf{z}_i, d_i\}_{i=1}^N$, and $\hat{\mathbf{x}}_i$ is the corresponding instrumental vector.

2. The desired response d is correlated with the instrumental vector $\hat{\mathbf{x}}$, obtaining the cross-correlation vector

$$\hat{\mathbf{r}}_{d\hat{x}} = \sum_{i=1}^{N} \hat{\mathbf{x}}_i d_i \tag{2.56}$$

Given these two correlation measurements, we then use the modified formula

$$\begin{aligned} \hat{\mathbf{w}}(N) &= \mathbf{R}_{z\hat{x}}^{-1} \mathbf{r}_{d\hat{x}} \\ &= \left(\sum_{i=1}^{N} \hat{\mathbf{x}}_i \mathbf{z}_i^T \right)^{-1} \left(\sum_{i=1}^{N} \hat{\mathbf{x}}_i d_i \right) \end{aligned} \tag{2.57}$$

for computing an estimate of the unknown parameter vector $\mathbf{w}$ (Young, 1984). Unlike the ML solution of Eq. (2.51), the modified formula of Eq. (2.57), based on the method of instrumental variables, provides an asymptotically unbiased estimate of the unknown parameter vector $\mathbf{w}$; see Problem 2.7.

In applying the method of instrumental variables, however, the key issue is how to obtain or generate variables that satisfy Properties 1 and 2. It turns out that in time-series analysis, the resolution of this issue is surprisingly straightforward (Young, 1984).

2.9 SUMMARY AND DISCUSSION

In this chapter, we studied the method of least squares for linear regression, which is well established in the statistics literature. The study was presented from two different, yet complementary, viewpoints:

- *Bayesian theory*, where the *maximum a posteriori estimate* of a set of unknown parameters is the objective of interest. This approach to parameter estimation requires knowledge of the prior distribution of the unknown parameters. The presentation was demonstrated for a Gaussian environment.
- *Regularization theory*, where the cost function to be minimized with respect to the unknown parameters consists of two components: the squared explanational errors summed over the training data, and a regularizing term defined in terms of the squared Euclidean norm of the parameter vector.

For the special case of an environment in which the prior distribution of the unknown parameters is Gaussian with zero mean and variance σ_w^2, it turns out that the regularization parameter λ is inversely proportional to σ_w^2. The implication of this statement is that when σ_w^2 is very large (i.e., the unknown parameters are uniformly distributed over a wide range), the formula for finding the estimate of the parameter vector $\mathbf{w}$ is defined by the *normal equation*

$$\hat{\mathbf{w}} = \hat{\mathbf{R}}_{xx}^{-1}\hat{\mathbf{r}}_{dx}$$

where $\hat{\mathbf{R}}_{xx}$ is the time-averaged correlation matrix of the input vector $\mathbf{x}$ and $\hat{\mathbf{r}}_{dx}$ is the corresponding time-averaged cross-correlation vector between the input vector $\mathbf{x}$ and the desired response d. Both correlation parameters are computed using the training sample $\{\mathbf{x}_i, d_i\}_{i=1}^N$ and are therefore dependent on its sample size N. Furthermore, this formula is identical to the solution obtained using the maximum-likelihood method that assumes a uniform distribution for the prior.

We also discussed three other important issues:

- The minimum-description-length (MDL) criterion for model-order selection (i.e., the size of the unknown parameter vector in a linear regression model).
- The bias–variance dilemma, which means that in parameter estimation (involving the use of a finite sample size) we have the inevitable task of trading off the variance of the estimate with the bias; the bias is defined as the difference between the expected value of the parameter estimate and the true value, and the variance is a measure of the "volatility" of the estimate around the expected value.
- The method of instrumental variables, the need for which arises when the observables in the training sample are noisy; such a situation is known to arise in practice.

NOTES AND REFERENCES

1. Regression models can be linear or nonlinear. *Linear regression models* are discussed in depth in the classic book by Rao (1973). *Nonlinear regression models* are discussed in Seber and Wild (1989).
2. For a highly readable account of Bayesian theory, see Robert (2001).
3. For a detailed discussion of the method of least squares, see Chapter 8 of Haykin (2002).

PROBLEMS

2.1. Discuss the basic differences between the maximum a posteriori and maximum-likelihood estimates of the parameter vector in a linear regression model.

2.2. Starting with the cost function of Eq. (2.36), $\mathscr{E}(\mathbf{w})$, derive the formula of Eq. (2.29) by minimizing the cost function with respect to the unknown parameter vector $\mathbf{w}$.

2.3. In this problem, we address properties of the least-squares estimator based on the linear regression model of Fig. 2.1:

Property 1. The least-squares estimate

$$\hat{\mathbf{w}} = \mathbf{R}_{xx}^{-1}\mathbf{r}_{dx}$$

is unbiased, provided that the expectational error ε in the linear regression model of Fig. 2.1 has zero mean.

Property 2. When the expectational error ε is drawn from a zero-mean white-noise process with variance σ^2, the covariance matrix of the least-squares estimate $\hat{\mathbf{w}}$ equals

$$\sigma^2\hat{\mathbf{R}}_{xx}^{-1}.$$

Property 3. The estimation error

$$e_o = d - \hat{\mathbf{w}}^T\mathbf{x}$$

produced by the optimized method of least squares is orthogonal to the estimate of the desired response, denoted by $\hat{d}$; this property is a corollary to the *principle of orthogonality.* If we were to use geometric representations of d, $\hat{d}$, and e_o, then we would find that the "vector" representing e_o, is perpendicular (i.e., normal) to that representing $\hat{d}$; indeed it is in light of this geometric representation that the formula

$$\hat{\mathbf{R}}_{xx}\hat{\mathbf{w}} = \hat{\mathbf{r}}_{dx}$$

is called the normal equation.

Starting with the normal equation, prove each of these three properties under the premise that $\hat{\mathbf{R}}_{xx}$ and $\hat{\mathbf{r}}_{dx}$ are time-averaged correlation functions.

2.4. Let $\mathbf{R}_{xx}$ denote the ensemble-averaged correlation function of the regressor $\mathbf{x}$, and let $\mathbf{r}_{dx}$ denote the corresponding ensemble-averaged cross-correlation vector between the regressor $\mathbf{x}$ and response d; that is,

$$\mathbf{R}_{xx} = \mathbb{E}[\mathbf{x}\mathbf{x}^T]$$
$$\mathbf{r}_{dx} = \mathbb{E}[d\mathbf{x}]$$

Referring to the linear regression model of Eq. (2.3), show that minimization of the mean-square error

$$\mathbf{J}(\mathbf{w}) = \mathbb{E}[\varepsilon^2]$$

leads to the *Wiener–Hopf equation*

$$\mathbf{R}_{xx}\mathbf{w} = \mathbf{r}_{dx}$$

where $\mathbf{w}$ is the parameter vector of the regression model. Compare this equation with the normal equation of Eq. (2.33).

2.5. Equation (2.47) expresses the natural measure of the effectiveness of the approximating function $F(\mathbf{x}, \hat{\mathbf{w}})$ as a predictor of the desired response d. This expression is made up of two components, one defining the squared bias and the other defining the variance. Derive this expression, starting from Eq. (2.46)

2.6. Elaborate on the following statement:

A network architecture, constrained through the incorporation of prior knowledge, addresses the bias–variance dilemma by reducing variance at the expense of increased bias.

2.7. The method of instrumental variables described in Eq. (2.57) provides an asymptotically unbiased estimate of the unknown parameter vector $\hat{\mathbf{w}}(N)$; that is,

$$\lim_{N\to\infty} \hat{\mathbf{w}}(N) = \mathbf{w}$$

Prove the validity of this statement, assuming joint ergodicity of the regressor $\mathbf{x}$ and response d.

COMPUTER EXPERIMENT

2.8. Repeat the pattern-classification experiment described in Section 2.5, this time setting the two moons at the very edge of linear separability, that is, $d = 0$. Comment on your results, and compare them with those obtained in Problem 1.6, involving the perceptron.

2.9. In performing the experiments in Section 2.5 and Problem 2.8, there was no regularization included in the method of least squares. Would the use of regularization have made a difference in the performance of the method of least squares?

To substantiate your response to this question, repeat the experiment of Problem 2.7, this time using the regularized least-squares algorithm.

CHAPTER 3

The Least-Mean-Square Algorithm

ORGANIZATION OF THE CHAPTER

In this chapter, we describe a highly popular on-line learning algorithm known as the least-mean-square (LMS) algorithm, which was developed by Widrow and Hoff in 1960.

The chapter is organized as follows:

1. Section 3.1 is introductory, followed by Section 3.2 that sets the stage for the rest of the chapter by describing a linear discrete-time filter of finite-duration impulse response.
2. Section 3.3 reviews two unconstrained optimization techniques: the method of steepest descent and Newton's method.
3. Section 3.4 formulates the Wiener filter, which is optimum in the mean-square-error sense. Traditionally, the average performance of the LMS algorithm is judged against the Wiener filter.
4. Section 3.5 presents the derivation of the LMS algorithm. Section 3.6 portrays a modified form of the LMS algorithm as a Markov model. Then, to prepare the way for studying the convergence behavior of the LMS algorithm, Section 3.7 introduces the Langevin equation, rooted in unstable thermodynamics. The other tool needed for convergence analysis of the algorithm is Kushner's method of direct averaging; this method is discussed in Section 3.8. Section 3.9 presents a detailed statistical analysis of the algorithm; most importantly, it shows that the statistical behavior of the algorithm (using a small learning-rate parameter) is, in fact, the discrete-time version of the Langevin equation.
5. Section 3.10 presents a computer experiment validating the small learning-rate theory of the LMS algorithm. Section 3.11 repeats the pattern-classification experiment of Section 1.5 on the perceptron, this time using the LMS algorithm.
6. Section 3.12 discusses the virtues and limitations of the LMS algorithm. Section 3.13 discusses the related issue of learning-rate annealing schedules.

Section 3.14 provides a summary and discussion that conclude the chapter.

3.1 INTRODUCTION

Rosenblatt's perceptron, discussed in Chapter 1, was the first learning algorithm for solving a linearly separable pattern-classification problem. The *least-mean-square (LMS) algorithm*, developed by Widrow and Hoff (1960), was the first linear adaptive-filtering

algorithm for solving problems such as prediction and communication-channel equalization. Development of the LMS algorithm was indeed inspired by the perceptron. Though different in applications, these two algorithms share a common feature: They both involve the use of a *linear combiner*, hence the designation "linear."

The amazing thing about the LMS algorithm is that it has established itself not only as the workhorse for adaptive-filtering applications, but also as the benchmark against which other adaptive-filtering algorithms are evaluated. The reasons behind this amazing track record are multifold:

- In terms of computational complexity, the LMS algorithm's complexity is *linear* with respect to adjustable parameters, which makes the algorithm *computationally efficient*, yet the algorithm is effective in performance.
- The algorithm is *simple* to code and therefore easy to build.
- Above all, the algorithm is *robust* with respect to external disturbances.

From an engineering perspective, these qualities are all highly desirable. It is therefore not surprising to see that the LMS algorithm has withstood the test of time.

In this chapter, we derive the LMS algorithm in its most basic form and discuss its virtues and limitations. Most importantly, the material presented herein sets the stage for the back-propagation algorithm to be discussed in the next chapter.

3.2 FILTERING STRUCTURE OF THE LMS ALGORITHM

Figure 3.1 shows the block diagram of an unknown dynamic system that is stimulated by an input vector consisting of the elements $x_1(i), x_2(i), \ldots, x_M(i)$, where i denotes the instant of time at which the stimulus (excitation) is applied to the system. The time index

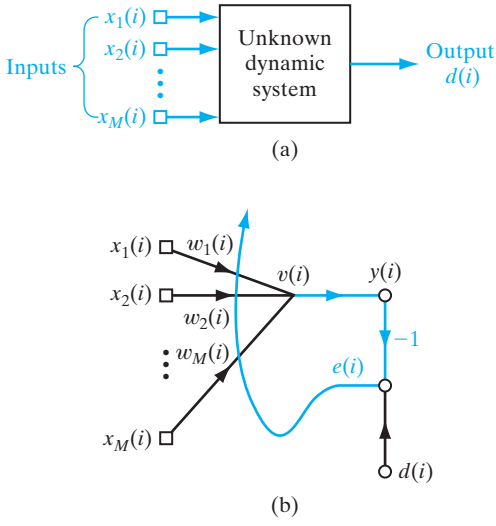


FIGURE 3.1 (a) Unknown dynamic system. (b) Signal-flow graph of adaptive model for the system; the graph embodies a feedback loop set in color.

$i = 1, 2, ..., n$. In response to this stimulus, the system produces an output denoted by $y(i)$. Thus, the external behavior of the system is described by the data set

$$\mathcal{T}: \{\mathbf{x}(i), d(i); i = 1, 2, ..., n, ...\} \tag{3.1}$$

where

$$\mathbf{x}(i) = [x_1(i), x_2(i), ..., x_M(i)]^T \tag{3.2}$$

The sample pairs composing $\mathcal{T}$ are identically distributed according to an unknown probability law. The dimension M pertaining to the input vector $\mathbf{x}(i)$ is referred to as the *dimensionality of the input space*, or simply as the *input dimensionality*.

The stimulus vector $\mathbf{x}(i)$ can arise in one of two fundamentally different ways, one spatial and the other temporal:

- The M elements of $\mathbf{x}(i)$ originate at different points in space; in this case, we speak of $\mathbf{x}(i)$ as a *snapshot* of data.
- The M elements of $\mathbf{x}(i)$ represent the set of present and $(M - 1)$ past values of some excitation that are *uniformly spaced in time*.

The problem we address is how to design a multiple-input–single-output *model* of the unknown dynamic system by building it around a *single linear neuron*. The neural model operates under the influence of an algorithm that *controls* necessary adjustments to the synaptic weights of the neuron, with the following points in mind:

- The algorithm starts from an *arbitrary setting* of the neuron's synaptic weights.
- Adjustments to the synaptic weights in response to statistical variations in the system's behavior are made on a *continuous* basis (i.e time is incorporated into the constitution of the algorithm).
- Computations of adjustments to the synaptic weights are completed inside an interval that is one sampling period long.

The neural model just described is referred to as an *adaptive filter*. Although the description is presented in the context of a task clearly recognized as one of *system identification*, the characterization of the adaptive filter is general enough to have wide application.

Figure 3.1b shows a signal-flow graph of the adaptive filter. Its operation consists of two continuous processes:

1. *Filtering process*, which involves the computation of two signals:
   - an output, denoted by $y(i)$, that is produced in response to the M elements of the stimulus vector $\mathbf{x}(i)$, namely, $x_1(i), x_2(i), ..., x_M(i)$;
   - an error signal, denoted by $e(i)$, that is obtained by comparing the output $y(i)$ with the corresponding output $d(i)$ produced by the unknown system. In effect, $d(i)$ acts as a *desired response*, or *target*, *signal*.
2. *Adaptive process*, which involves the automatic adjustment of the synaptic weights of the neuron in accordance with the error signal $e(i)$.

Thus, the combination of these two processes working together constitutes a *feedback loop* acting around the neuron, as shown in Fig. 3.1b.

Since the neuron is linear, the output $y(i)$ is exactly the same as the induced local field $v(i)$; that is,

$$y(i) = v(i) = \sum_{k=1}^{M} w_k(i)x_k(i) \tag{3.3}$$

where $w_1(i), w_2(i), ..., w_M(i)$ are the M synaptic weights of the neuron, measured at time i. In matrix form, we may express $y(i)$ as an inner product of the vectors $\mathbf{x}(i)$ and $\mathbf{w}(i)$ as

$$y(i) = \mathbf{x}^T(i)\mathbf{w}(i) \tag{3.4}$$

where

$$\mathbf{w}(i) = [w_1(i), w_2(i), ..., w_M(i)]^T$$

Note that the notation for a synaptic weight has been simplified here by *not* including an additional subscript to identify the neuron, since we have only a single neuron to deal with. This practice is followed throughout the book, whenever a single neuron is involved. The neuron's output $y(i)$ is compared with the corresponding output $d(i)$ received from the unknown system at time i. Typically, $y(i)$ is different from $d(i)$; hence, their comparison results in the error signal

$$e(i) = d(i) - y(i) \tag{3.5}$$

The manner in which the error signal $e(i)$ is used to control the adjustments to the neuron's synaptic weights is determined by the cost function used to derive the adaptive-filtering algorithm of interest. This issue is closely related to that of optimization. It is therefore appropriate to present a review of unconstrained-optimization methods. The material is applicable not only to linear adaptive filters, but also to neural networks in general.

3.3 UNCONSTRAINED OPTIMIZATION: A REVIEW

Consider a cost function $\mathscr{E}(\mathbf{w})$ that is a *continuously differentiable* function of some unknown weight (parameter) vector $\mathbf{w}$. The function $\mathscr{E}(\mathbf{w})$ maps the elements of $\mathbf{w}$ into real numbers. It is a measure of how to choose the weight (parameter) vector $\mathbf{w}$ of an adaptive-filtering algorithm so that it behaves in an optimum manner. We want to find an optimal solution $\mathbf{w}^*$ that satisfies the condition

$$\mathscr{E}(\mathbf{w}^*) \leq \mathscr{E}(\mathbf{w}) \tag{3.6}$$

That is, we need to solve an *unconstrained-optimization problem*, stated as follows:

Minimize the cost function $\mathscr{E}(\mathbf{w})$ with respect to the weight vector $\mathbf{w}$.

The necessary condition for optimality is

$$\nabla\mathscr{E}(\mathbf{w}^*) = \mathbf{0} \tag{3.7}$$

where ∇ is the *gradient operator*,

$$\nabla = \left[\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, ..., \frac{\partial}{\partial w_M}\right]^T \tag{3.8}$$

and $\nabla\mathscr{E}(\mathbf{w})$ is the *gradient vector* of the cost function,

$$\nabla\mathscr{E}(\mathbf{w}) = \left[\frac{\partial\mathscr{E}}{\partial w_1}, \frac{\partial\mathscr{E}}{\partial w_2}, ..., \frac{\partial\mathscr{E}}{\partial w_M}\right]^T \tag{3.9}$$

(Differentiation with respect to a vector is discussed in Note 1 at the end of this chapter.)

A class of unconstrained-optimization algorithms that is particularly well suited for the design of adaptive filters is based on the idea of local *iterative descent*:

> *Starting with an initial guess denoted by* $\mathbf{w}(0)$, *generate a sequence of weight vectors* $\mathbf{w}(1)$, $\mathbf{w}(2), \ldots$, *such that the cost function* $\mathscr{E}(\mathbf{w})$ *is reduced at each iteration of the algorithm, as shown by*
>
> $$\mathscr{E}(\mathbf{w}(n+1)) < \mathscr{E}(\mathbf{w}(n)) \tag{3.10}$$
>
> *where* $\mathbf{w}(n)$ *is the old value of the weight vector and* $\mathbf{w}(n+1)$ *is its updated value.*

We hope that the algorithm will eventually converge onto the optimal solution $\mathbf{w}^*$. We say "hope" because there is a distinct possibility that the algorithm will diverge (i.e., become unstable) unless special precautions are taken.

In this section, we describe three unconstrained-optimization methods that rely on the idea of iterative descent in one form or another (Bertsekas, 1995).

Method of Steepest Descent

In the method of steepest descent, the successive adjustments applied to the weight vector $\mathbf{w}$ are in the direction of steepest descent, that is, in a direction opposite to the gradient vector $\nabla\mathscr{E}(\mathbf{w})$. For convenience of presentation, we write

$$\mathbf{g} = \nabla\mathscr{E}(\mathbf{w}) \tag{3.11}$$

Accordingly, the steepest-descent algorithm is formally described by

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta\mathbf{g}(n) \tag{3.12}$$

where η is a positive constant called the *stepsize*, or *learning-rate*, *parameter*, and $\mathbf{g}(n)$ is the gradient vector evaluated at the point $\mathbf{w}(n)$. In going from iteration n to $n+1$, the algorithm applies the *correction*

$$\begin{aligned}\Delta\mathbf{w}(n) &= \mathbf{w}(n+1) - \mathbf{w}(n) \\ &= -\eta\mathbf{g}(n)\end{aligned} \tag{3.13}$$

Equation (3.13) is in fact a formal statement of the error-correction rule described in the introductory chapter.

To show that the formulation of the steepest-descent algorithm satisfies the condition of Eq. (3.10) for iterative descent, we use a first-order Taylor series expansion around $\mathbf{w}(n)$ to approximate $\mathscr{E}(\mathbf{w}(n+1))$ as

$$\mathscr{E}(\mathbf{w}(n+1)) \approx \mathscr{E}(\mathbf{w}(n)) + \mathbf{g}^T(n)\Delta\mathbf{w}(n)$$

the use of which is justified for small η. Substituting Eq. (3.13) into this approximate relation yields

$$\begin{aligned}\mathscr{E}(\mathbf{w}(n+1)) &\approx \mathscr{E}(\mathbf{w}(n)) - \eta\mathbf{g}^T(n)\mathbf{g}(n)\\ &= \mathscr{E}(\mathbf{w}(n)) - \eta\|\mathbf{g}(n)\|^2\end{aligned}$$

which shows that, for a positive learning-rate parameter η, the cost function is decreased as the algorithm progresses from one iteration to the next. The reasoning presented here is approximate in that this end result is true only for small enough learning rates.

The method of steepest descent converges to the optimal solution $\mathbf{w}^*$ slowly. Moreover, the learning-rate parameter η has a profound influence on its convergence behavior:

- When η is small, the transient response of the algorithm is *overdamped*, in that the trajectory traced by $\mathbf{w}(n)$ follows a smooth path in the $\mathscr{W}$-plane, as illustrated in Fig. 3.2a.
- When η is large, the transient response of the algorithm is *underdamped*, in that the trajectory of $\mathbf{w}(n)$ follows a zigzagging (oscillatory) path, as illustrated in Fig. 3.2b.
- When η exceeds a certain critical value, the algorithm becomes unstable (i.e., it diverges).

Newton's Method

For a more elaborate optimization technique, we may look to *Newton's method*, the basic idea of which is to minimize the quadratic approximation of the cost function $\mathscr{E}(\mathbf{w})$ around the current point $\mathbf{w}(n)$; this minimization is performed at each iteration of the algorithm. Specifically, using a *second-order* Taylor series expansion of the cost function around the point $\mathbf{w}(n)$, we may write

$$\begin{aligned}\Delta\mathscr{E}(\mathbf{w}(n)) &= \mathscr{E}(\mathbf{w}(n+1)) - \mathscr{E}(\mathbf{w}(n))\\ &\approx \mathbf{g}^T(n)\Delta\mathbf{w}(n) + \frac{1}{2}\Delta\mathbf{w}^T(n)\mathbf{H}(n)\Delta\mathbf{w}(n)\end{aligned} \tag{3.14}$$

As before, $\mathbf{g}(n)$ is the M-by-1 gradient vector of the cost function $\mathscr{E}(\mathbf{w})$ evaluated at the point $\mathbf{w}(n)$. The matrix $\mathbf{H}(n)$ is the m-by-m *Hessian* of $\mathscr{E}(\mathbf{w})$, also evaluated at $\mathbf{w}(n)$. The Hessian of $\mathscr{E}(\mathbf{w})$ is defined by

$$\begin{aligned}\mathbf{H} &= \nabla^2\mathscr{E}(\mathbf{w})\\ &= \begin{bmatrix} \dfrac{\partial^2\mathscr{E}}{\partial w_1^2} & \dfrac{\partial^2\mathscr{E}}{\partial w_1\partial w_2} & \cdots & \dfrac{\partial^2\mathscr{E}}{\partial w_1\partial w_M} \\ \dfrac{\partial^2\mathscr{E}}{\partial w_2\partial w_1} & \dfrac{\partial^2\mathscr{E}}{\partial w_2^2} & \cdots & \dfrac{\partial^2\mathscr{E}}{\partial w_2\partial w_M} \\ \vdots & \vdots & \vdots & \vdots \\ \dfrac{\partial^2\mathscr{E}}{\partial w_M\partial w_1} & \dfrac{\partial^2\mathscr{E}}{\partial w_M\partial w_2} & \cdots & \dfrac{\partial^2\mathscr{E}}{\partial w_M^2} \end{bmatrix}\end{aligned} \tag{3.15}$$

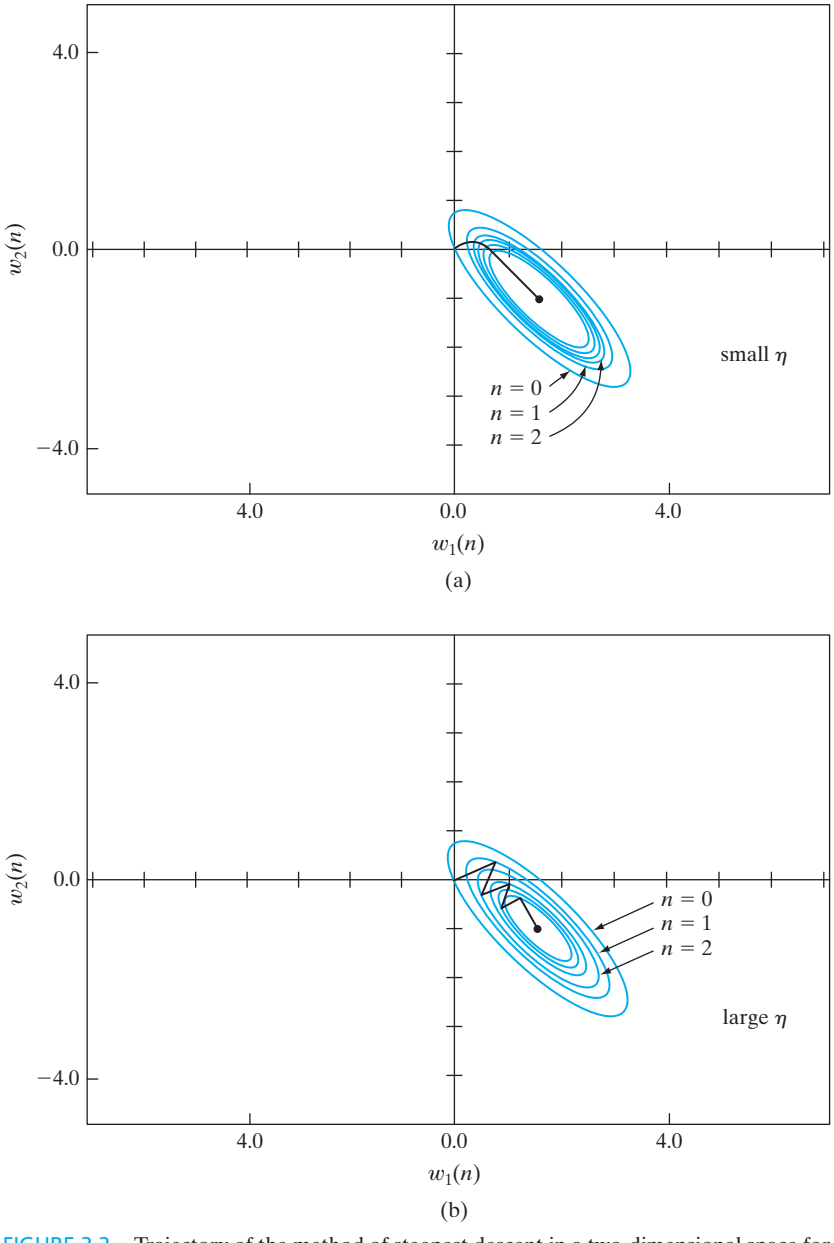


FIGURE 3.2 Trajectory of the method of steepest descent in a two-dimensional space for two different values of learning-rate parameter: (a) small η (b) large η. The coordinates w_1 and w_2 are elements of the weight vector $\mathbf{w}$; they both lie in the $\mathcal{W}$-plane.

Equation (3.15) requires the cost function $\mathscr{E}(\mathbf{w})$ to be twice continuously differentiable with respect to the elements of $\mathbf{w}$. Differentiating[1] Eq. (3.14) with respect to $\Delta\mathbf{w}$, we minimize the resulting change $\Delta\mathscr{E}(\mathbf{w})$ when

$$\mathbf{g}(n) + \mathbf{H}(n)\Delta\mathbf{w}(n) = \mathbf{0}$$

Solving this equation for $\Delta\mathbf{w}(n)$ yields

$$\Delta\mathbf{w}(n) = -\mathbf{H}^{-1}(n)\mathbf{g}(n)$$

That is,

$$\begin{aligned}\mathbf{w}(n+1) &= \mathbf{w}(n) + \Delta\mathbf{w}(n)\\ &= \mathbf{w}(n) - \mathbf{H}^{-1}(n)\mathbf{g}(n)\end{aligned} \tag{3.16}$$

where $\mathbf{H}^{-1}(n)$ is the inverse of the Hessian of $\mathscr{E}(\mathbf{w})$.

Generally speaking, Newton's method converges quickly asymptotically and does *not* exhibit the zigzagging behavior that sometimes characterizes the method of steepest descent. However, for Newton's method to work, the Hessian $\mathbf{H}(n)$ has to be a *positive definite matrix* for all n. Unfortunately, in general, there is no guarantee that $\mathbf{H}(n)$ is positive definite at every iteration of the algorithm. If the Hessian $\mathbf{H}(n)$ is not positive definite, modification of Newton's method is necessary (Powell, 1987; Bertsekas, 1995). In any event, a major limitation of Newton's method is its computational complexity.

Gauss–Newton Method

To deal with the computational complexity of Newton's method without seriously compromising its convergence behavior, we may use the *Gauss–Newton method*. To apply this method, we adopt a cost function that is expressed as the sum of error squares. Let

$$\mathscr{E}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n} e^2(i) \tag{3.17}$$

where the scaling factor $\frac{1}{2}$ is included to simplify matters in subsequent analysis. All the error terms in this formula are calculated on the basis of a weight vector $\mathbf{w}$ that is fixed over the entire observation interval $1 \le i \le n$.

The error signal $e(i)$ is a function of the adjustable weight vector $\mathbf{w}$. Given an operating point $\mathbf{w}(n)$, we linearize the dependence of $e(i)$ on $\mathbf{w}$ by introducing the new term

$$e'(i,\mathbf{w}) = e(i) + \left[\frac{\partial e(i)}{\partial \mathbf{w}}\right]^T_{\mathbf{w}=\mathbf{w}(n)} \times (\mathbf{w} - \mathbf{w}(n)), \qquad i = 1, 2, ..., n$$

Equivalently, by using matrix notation, we may write

$$\mathbf{e}'(n,\mathbf{w}) = \mathbf{e}(n) + \mathbf{J}(n)\,(\mathbf{w} - \mathbf{w}(n)) \tag{3.18}$$

where $\mathbf{e}(n)$ is the error vector

$$\mathbf{e}(n) = [e(1), e(2), ..., e(n)]^T$$

and $\mathbf{J}(n)$ is the n-by-m *Jacobian* of $\mathbf{e}(n)$:

$$\mathbf{J}(n) = \begin{bmatrix} \frac{\partial e(1)}{\partial w_1} & \frac{\partial e(1)}{\partial w_2} & \cdots & \frac{\partial e(1)}{\partial w_M} \\ \frac{\partial e(2)}{\partial w_1} & \frac{\partial e(2)}{\partial w_2} & \cdots & \frac{\partial e(2)}{\partial w_M} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e(n)}{\partial w_1} & \frac{\partial e(n)}{\partial w_2} & \cdots & \frac{\partial e(n)}{\partial w_M} \end{bmatrix}_{\mathbf{w}=\mathbf{w}(n)} \tag{3.19}$$

The Jacobian $\mathbf{J}(n)$ is the transpose of the m-by-n gradient matrix $\nabla\mathbf{e}(n)$, where

$$\nabla\mathbf{e}(n) = [\nabla e(1), \nabla e(2), ..., \nabla e(n)]$$

The updated weight vector $\mathbf{w}(n + 1)$ is now defined by

$$\mathbf{w}(n + 1) = \arg\min_{\mathbf{w}} \left\{ \frac{1}{2} \|\mathbf{e}'(n,\mathbf{w})\|^2 \right\} \tag{3.20}$$

Using Eq. (3.18) to evaluate the squared Euclidean norm of $\mathbf{e}'(n, \mathbf{w})$, we get

$$\begin{aligned} \frac{1}{2} \|\mathbf{e}'(n, \mathbf{w})\|^2 = \frac{1}{2} \|\mathbf{e}(n)\|^2 &+ \mathbf{e}^T(n)\mathbf{J}(n)(\mathbf{w} - \mathbf{w}(n)) \\ &+ \frac{1}{2} (\mathbf{w} - \mathbf{w}(n))^T \mathbf{J}^T(n)\mathbf{J}(n)(\mathbf{w} - \mathbf{w}(n)) \end{aligned}$$

Hence, differentiating this expression with respect to $\mathbf{w}$ and setting the result equal to zero, we obtain

$$\mathbf{J}^T(n)\mathbf{e}(n) + \mathbf{J}^T(n)\mathbf{J}(n)(\mathbf{w} - \mathbf{w}(n)) = \mathbf{0}$$

Solving this equation for $\mathbf{w}$, we may thus write, in light of Eq. 3.20,

$$\mathbf{w}(n + 1) = \mathbf{w}(n) - (\mathbf{J}^T(n)\mathbf{J}(n))^{-1}\mathbf{J}^T(n)\mathbf{e}(n) \tag{3.21}$$

which describes the *pure* form of the Gauss–Newton method.

Unlike Newton's method, which requires knowledge of the Hessian of the cost function $\mathscr{E}(n)$, the Gauss–Newton method requires only the Jacobian of the error vector $\mathbf{e}(n)$. However, for the Gauss–Newton iteration to be computable, the matrix product $\mathbf{J}^T(n)\mathbf{J}(n)$ must be nonsingular.

With regard to the latter point, we recognize that $\mathbf{J}^T(n)\mathbf{J}(n)$ is always nonnegative definite. To ensure that it is nonsingular, the Jacobian $\mathbf{J}(n)$ must have row *rank n*; that is, the n rows of $\mathbf{J}(n)$ in Eq. (3.19) must be linearly independent. Unfortunately, there is no guarantee that this condition will always hold. To guard against the possibility that $\mathbf{J}(n)$ is rank deficient, the customary practice is to add the diagonal matrix $\delta\mathbf{I}$ to the matrix $\mathbf{J}^T(n)\mathbf{J}(n)$, where $\mathbf{I}$ is the identity matrix. The parameter δ is a small positive constant chosen to ensure that

$$\mathbf{J}^T(n)\mathbf{J}(n) + \delta\mathbf{I} \text{ is positive definite for all } n$$

On this basis, the Gauss–Newton method is implemented in the slightly modified form

$$\mathbf{w}(n+1) = \mathbf{w}(n) - (\mathbf{J}^T(n)\mathbf{J}(n) + \delta\mathbf{I})^{-1}\mathbf{J}^T(n)\mathbf{e}(n) \tag{3.22}$$

The effect of the added term $\delta\mathbf{I}$ is progressively reduced as the number of iterations, n, is increased. Note also that the recursive equation (3.22) is the solution of the *modified* cost function

$$\mathscr{E}(\mathbf{w}) = \frac{1}{2}\left\{\sum_{i=1}^{n} e^2(i) + \delta\|\mathbf{w} - \mathbf{w}(n)\|^2\right\} \tag{3.23}$$

where $\mathbf{w}(n)$ is the *current value* of the weight vector $\mathbf{w}(i)$.

In the literature on signal processing, the addition of the term $\delta\mathbf{I}$ in Eq. (3.22) is referred to as *diagonal loading*. The addition of this term is accounted for by expanding the cost function $\mathscr{E}(\mathbf{w})$ in the manner described in Eq. (3.23), where we now have two terms (ignoring the scaling factor $\frac{1}{2}$):

- The first term, $\sum_{i=1}^{n} e^2(i)$, is the standard sum of squared errors, which depends on the training data.
- The second term contains the squared Euclidean norm, $\|\mathbf{w} - \mathbf{w}(n)\|^2$, which depends on the filter structure. In effect, this term acts as a *stabilizer*.

The scaling factor δ is commonly referred to as a *regularization parameter*, and the resulting modification of the cost function is correspondingly referred to as *structural regularization*. The issue of regularization is discussed in great detail in Chapter 7.

3.4 THE WIENER FILTER

The ordinary least-squares estimator was discussed in Chapter 2, where the traditional approach to minimization was used to find the least-squares solution from an observation model of the environment. To conform to the terminology adopted in this chapter, we will refer to it as the *least-squares filter*. Moreover, we will rederive the formula for this filter by using the Gauss–Newton method.

To proceed then, we use Eqs. (3.3) and (3.4) to define the error vector as

$$\begin{aligned}\mathbf{e}(n) &= \mathbf{d}(n) - [\mathbf{x}(1), \mathbf{x}(2), ..., \mathbf{x}(n)]^T\mathbf{w}(n)\\ &= \mathbf{d}(n) - \mathbf{X}(n)\mathbf{w}(n)\end{aligned} \tag{3.24}$$

where $\mathbf{d}(n)$ is the n-by-1 *desired response vector*,

$$\mathbf{d}(n) = [d(1), d(2), ..., d(n)]^T$$

and $\mathbf{X}(n)$ is the n-by-M *data matrix*,

$$\mathbf{X}(n) = [\mathbf{x}(n), \mathbf{x}(2), ..., \mathbf{x}(n)]^T$$

Differentiating the error vector $\mathbf{e}(n)$ with respect to $\mathbf{w}(n)$ yields the gradient matrix

$$\nabla\mathbf{e}(n) = -\mathbf{X}^T(n)$$

Correspondingly, the Jacobian of $\mathbf{e}(n)$ is

$$\mathbf{J}(n) = -\mathbf{X}(n) \tag{3.25}$$

Since the error equation (3.18) is already linear in the weight vector $\mathbf{w}(n)$, the Gauss–Newton method converges in a single iteration, as shown here. Substituting Eqs. (3.24) and (3.25) into (3.21) yields

$$\begin{aligned}\mathbf{w}(n+1) &= \mathbf{w}(n) + (\mathbf{X}^T(n)\mathbf{X}(n))^{-1}\mathbf{X}^T(n)(\mathbf{d}(n) - \mathbf{X}(n)\mathbf{w}(n)) \\ &= (\mathbf{X}^T(n)\mathbf{X}(n))^{-1}\mathbf{X}^T(n)\mathbf{d}(n)\end{aligned} \tag{3.26}$$

The term $(\mathbf{X}^T(n)\mathbf{X}(n))^{-1}\mathbf{X}^T(n)$ is called the *pseudoinverse* of the data matrix $\mathbf{X}(n)$; that is,[2]

$$\mathbf{X}^+(n) = (\mathbf{X}^T(n)\mathbf{X}(n))^{-1}\mathbf{X}^T(n) \tag{3.27}$$

Hence, we may rewrite Eq. (3.26) in the compact form

$$\mathbf{w}(n+1) = \mathbf{X}^+(n)\mathbf{d}(n) \tag{3.28}$$

This formula represents a convenient way of stating the following:

> *The weight vector $\mathbf{w}(n+1)$ solves the linear least-squares problem, defined over an observation interval of duration n, as the product of two terms: the pseudoinverse $\mathbf{X}^+(n)$ and the desired response vector $\mathbf{d}(n)$.*

Wiener Filter: Limiting Form of the Least-Squares Filter for an Ergodic Environment

Let $\mathbf{w}_o$ denote the limiting form of the least-squares filter as the number of observations, n, is allowed to approach infinity. We may then use Eq. (3.26) to write

$$\begin{aligned}\mathbf{w}_o &= \lim_{n\to\infty} \mathbf{w}(n+1) \\ &= \lim_{n\to\infty} (\mathbf{X}^T(n)\mathbf{X}(n))^{-1}\mathbf{X}^T(n)d(n) \\ &= \lim_{n\to\infty} \left(\frac{1}{n}\mathbf{X}^T(n)\mathbf{X}(n)\right)^{-1} \times \lim_{n\to\infty} \frac{1}{n}\mathbf{X}^T(n)\mathbf{d}(n)\end{aligned} \tag{3.29}$$

Suppose now the input vector $\mathbf{x}(i)$ and the corresponding desired response $d(i)$ are drawn from a jointly *ergodic environment* that is also stationary. We may then substitute time averages for ensemble averages. By definition, the ensemble-averaged form of the correlation matrix of the input vector $\mathbf{x}(i)$ is

$$\mathbf{R}_{xx} = \mathbb{E}[\mathbf{x}(i)\mathbf{x}^T(i)] \tag{3.30}$$

and, correspondingly, the ensemble-averaged form of the *cross-correlation vector* between the input vector $\mathbf{x}(i)$ and the desired response vector $d(i)$ is

$$\mathbf{r}_{dx} = \mathbb{E}[\mathbf{x}(i)d(i)] \tag{3.31}$$

where $\mathbb{E}$ is the expectation operator. Therefore, under the ergodicity assumption, we may now write

$$\mathbf{R}_{xx} = \lim_{n\to\infty} \frac{1}{n}\mathbf{X}(n)\mathbf{X}^T(n)$$

and

$$\mathbf{r}_{dx} = \lim_{n\to\infty} \mathbf{X}^T(n)\mathbf{d}(n)$$

Accordingly, we may recast Eq. (3.29) in terms of ensemble-averaged correlation parameters as

$$\mathbf{w}_o = \mathbf{R}_{xx}^{-1}\mathbf{r}_{dx} \tag{3.32}$$

where $\mathbf{R}_{xx}^{-1}$ is the inverse of the correlation matrix $\mathbf{R}_{xx}$. The formula of Eq. (3.32) is the ensemble-averaged version of the least-squares solution defined in Eq. (2.32).

The weight vector $\mathbf{w}_o$ is called the *Wiener solution* to the optimum linear filtering problem (Widrow and Stearns, 1985; Haykin, 2002). Accordingly, we may make the statement:

> *For an ergodic process, the least-squares filter asymptotically approaches the Wiener filter as the number of observations approaches infinity.*

Designing the Wiener filter requires knowledge of the second-order statistics: the correlation matrix $\mathbf{R}_{xx}$ of the input vector $\mathbf{x}(n)$, and the cross-correlation vector $\mathbf{r}_{xd}$ between $\mathbf{x}(n)$ and the desired response $d(n)$. However, this information is not available when the environment in which the filter operates is unknown. We may deal with such an environment by using a *linear adaptive filter*, adaptive in the sense that the filter is able to adjust its free parameters in response to statistical variations in the environment. A highly popular algorithm for doing this kind of adjustment on a continuing-time basis is the least-mean-square algorithm, discussed next.

3.5 THE LEAST-MEAN-SQUARE ALGORITHM

The *least-mean-square (LMS) algorithm* is configured to minimize the *instantaneous value* of the cost function,

$$\mathscr{E}(\hat{\mathbf{w}}) = \frac{1}{2}e^2(n) \tag{3.33}$$

where $e(n)$ is the error signal measured at time n. Differentiating $\mathscr{E}(\hat{\mathbf{w}})$ with respect to the weight vector $\hat{\mathbf{w}}$ yields

$$\frac{\partial \mathscr{E}(\hat{\mathbf{w}})}{\partial \hat{\mathbf{w}}} = e(n)\frac{\partial e(n)}{\partial \mathbf{w}} \tag{3.34}$$

As with the least-squares filter, the LMS algorithm operates with a linear neuron, so we may express the error signal as

$$e(n) = d(n) - \mathbf{x}^T(n)\hat{\mathbf{w}}(n) \tag{3.35}$$

Hence,

$$\frac{\partial e(n)}{\partial \hat{\mathbf{w}}(n)} = -\mathbf{x}(n)$$

and

$$\frac{\partial \mathscr{E}(\hat{\mathbf{w}})}{\partial \hat{\mathbf{w}}(n)} = -\mathbf{x}(n)e(n)$$

Using this latter result as the *instantaneous estimate* of the gradient vector, we may write

$$\hat{\mathbf{g}}(n) = -\mathbf{x}(n)e(n) \tag{3.36}$$

Finally, using Eq. (3.36) for the gradient vector in Eq. (3.12) for the method of steepest descent, we may formulate the LMS algorithm as follows:

$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + \eta\mathbf{x}(n)e(n) \tag{3.37}$$

It is also noteworthy that the inverse of the learning-rate parameter η acts as a measure of the *memory* of the LMS algorithm: The smaller we make η, the longer the memory span over which the LMS algorithm remembers past data will be. Consequently, when η is small, the LMS algorithm performs accurately, but the convergence rate of the algorithm is slow.

In deriving Eq. (3.37), we have used $\hat{\mathbf{w}}(n)$ in place of $\mathbf{w}(n)$ to emphasize the fact that the LMS algorithm produces an *instantaneous estimate* of the weight vector that would result from the use of the method of steepest-descent. As a consequence, in using the LMS algorithm we sacrifice a distinctive feature of the steepest-descent algorithm. In the steepest-descent algorithm, the weight vector $\mathbf{w}(n)$ follows a well-defined trajectory in the weight space $\mathscr{W}$ for a prescribed η. In contrast, in the LMS algorithm, the weight vector $\hat{\mathbf{w}}(n)$ traces a random trajectory. For this reason, the LMS algorithm is sometimes referred to as a "stochastic gradient algorithm." As the number of iterations in the LMS algorithm approaches infinity, $\hat{\mathbf{w}}(n)$ performs a random walk (Brownian motion) about the Wiener solution $\mathbf{w}_o$. The important point to note, however, is the fact that, unlike the method of steepest descent, the LMS algorithm does *not* require knowledge of the statistics of the environment. This feature of the LMS algorithm is important from a practical perspective.

A summary of the LMS algorithm, based on Eqs. (3.35) and (3.37), is presented in Table 3.1, which clearly illustrates the simplicity of the algorithm. As indicated in this table, *initialization* of the algorithm is done by simply setting the value of the weight vector $\hat{\mathbf{w}}(0) = \mathbf{0}$.

TABLE 3.1 Summary of the LMS Algorithm

Training Sample: Input signal vector = $\mathbf{x}(n)$
Desired response = $d(n)$

User-selected parameter: η

Initialization. Set $\hat{\mathbf{w}}(0) = \mathbf{0}$.

Computation. For $n = 1, 2, \ldots$, compute

$$e(n) = d(n) - \hat{\mathbf{w}}^T(n)\mathbf{x}(n)$$

$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + \eta\mathbf{x}(n)e(n)$$

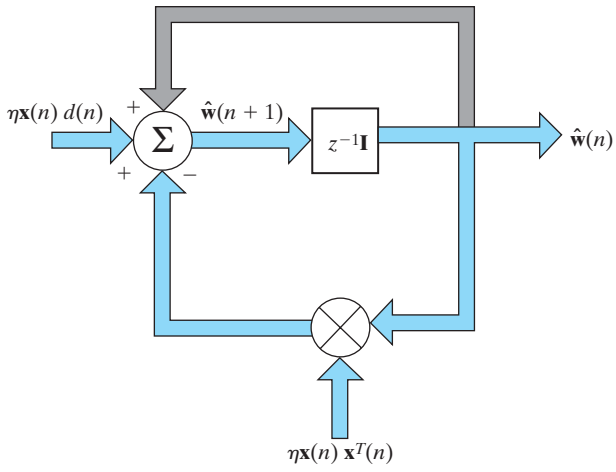


FIGURE 3.3 Signal-flow graph representation of the LMS algorithm. The graph embodies feedback depicted in color.

Signal-Flow Graph Representation of the LMS Algorithm

By combining Eqs. (3.35) and (3.37), we may express the evolution of the weight vector in the LMS algorithm as

$$\begin{aligned}\hat{\mathbf{w}}(n+1) &= \hat{\mathbf{w}}(n) + \eta\mathbf{x}(n)[d(n) - \mathbf{x}^T(n)\hat{\mathbf{w}}(n)] \\ &= [\mathbf{I} - \eta\mathbf{x}(n)\mathbf{x}^T(n)]\hat{\mathbf{w}}(n) + \eta\mathbf{x}(n)d(n)\end{aligned} \tag{3.38}$$

where $\mathbf{I}$ is the identity matrix. In using the LMS algorithm, we recognize that

$$\hat{\mathbf{w}}(n) = z^{-1}[\hat{\mathbf{w}}(n+1)] \tag{3.39}$$

where z^{-1} is the *unit-time delay operator*, implying storage. Using Eqs. (3.38) and (3.39), we may thus represent the LMS algorithm by the signal-flow graph depicted in Fig. 3.3. This signal-flow graph reveals that the LMS algorithm is an example of a *stochastic feedback system*. The presence of feedback has a profound impact on the convergence behavior of the LMS algorithm.

3.6 MARKOV MODEL PORTRAYING THE DEVIATION OF THE LMS ALGORITHM FROM THE WIENER FILTER

To perform a statistical analysis of the LMS algorithm, we find it more convenient to work with the *weight-error vector*, defined by

$$\boldsymbol{\varepsilon}(n) = \mathbf{w}_o - \hat{\mathbf{w}}(n) \tag{3.40}$$

where $\mathbf{w}_o$ is the optimum Wiener solution defined by Eq. (3.32) and $\hat{\mathbf{w}}(n)$ is the corresponding estimate of the weight vector computed by the LMS algorithm. Thus,

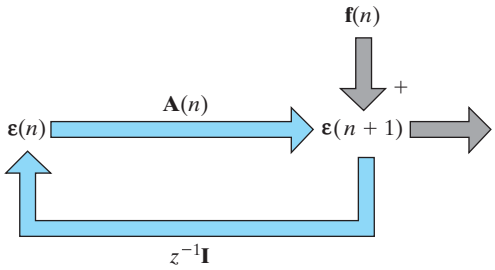


FIGURE 3.4 Signal-flow graph representation of the Markov model described in Eq. (3.41); the graph embodies feedback depicted in color.

in terms of $\boldsymbol{\varepsilon}(n)$, assuming the role of a *state*, we may rewrite Eq. (3.38) in the compact form

$$\boldsymbol{\varepsilon}(n+1) = \mathbf{A}(n)\boldsymbol{\varepsilon}(n) + \mathbf{f}(n) \tag{3.41}$$

Here, we have

$$\mathbf{A}(n) = \mathbf{I} - \eta\mathbf{x}(n)\mathbf{x}^T(n) \tag{3.42}$$

where $\mathbf{I}$ is the identity matrix. The additive noise term in the right-hand side of Eq. (3.41) is defined by

$$\mathbf{f}(n) = -\eta\mathbf{x}(n)e_o(n) \tag{3.43}$$

where

$$e_o(n) = d(n) - \mathbf{w}_o^T\mathbf{x}(n) \tag{3.44}$$

is the estimation error produced by the Wiener filter.

Equation (3.41) represents a *Markov model* of the LMS algorithm, with the model being characterized as follows:

- The *updated state* of the model, denoted by the vector $\boldsymbol{\varepsilon}(n+1)$, depends on the old state $\boldsymbol{\varepsilon}(n)$, with the dependence itself being defined by the *transition matrix* $\mathbf{A}(n)$.
- Evolution of the state over time n is perturbed by the intrinsically generated *noise* $\mathbf{f}(n)$, which acts as a "driving force".

Figure 3.4 shows a vector-valued signal-flow graph representation of this model. The branch labeled $z^{-1}\mathbf{I}$ represents the memory of the model, with z^{-1} acting as the *unit-time delay operator*, as shown by

$$z^{-1}[\boldsymbol{\varepsilon}(n+1)] = \boldsymbol{\varepsilon}(n) \tag{3.45}$$

This figure highlights the presence of feedback in the LMS algorithm in a more compact manner than that in Fig. 3.3.

The signal-flow graph of Fig. 3.4 and the accompanying equations provide the framework for the convergence analysis of the LMS algorithm under the assumption of a small learning-rate parameter η. However, before proceeding with this analysis, we will digress briefly to present two building blocks with that goal in mind: the Langevin equation, presented in Section 3.7, followed by Kushner's direct-averaging method, presented in Section 3.8. With those two building blocks in hand, we will then go on to study convergence analysis of the LMS algorithm in Section 3.9.

3.7 THE LANGEVIN EQUATION: CHARACTERIZATION OF BROWNIAN MOTION

Restating the remarks made towards the end of Section 3.5 in more precise terms insofar as stability or convergence is concerned, we may say that the LMS algorithm (for small enough η) never attains a perfectly stable or convergent condition. Rather, after a large number of iterations, n, the algorithm approaches a "pseudo-equilibrium" condition, which, in qualitative terms, is described by the algorithm executing Brownian motion around the Wiener solution. This kind of stochastic behavior is explained nicely by the Langevin equation of nonequilibrium thermodynamics.[3] So, we will make a brief digression to introduce this important equation.

Let $v(t)$ denote the velocity of a macroscopic particle of mass m immersed in a viscous fluid. It is assumed that the particle is small enough for its velocity due to thermal fluctuations deemed to be significant. Then, from the *equipartition law of thermodynamics*, the mean energy of the particle is given by

$$\frac{1}{2}\mathbb{E}[v^2(t)] = \frac{1}{2}k_B T \qquad \text{for all continuous time } t \tag{3.46}$$

where k_B is *Boltzmann's constant* and T is the *absolute temperature*. The total force exerted on the particle by the molecules in the viscous fluid is made up of two components:

(i) a continuous *damping force* equal to $-\alpha v(t)$ in accordance with *Stoke's law*, where α is the coefficient of friction;

(ii) a *fluctuating force* $F_f(t)$, whose properties are specified on the average.

The equation of motion of the particle in the absence of an external force is therefore given by

$$m\frac{dv}{dt} = -\alpha v(t) + F_f(t)$$

Dividing both sides of this equation by m, we get

$$\frac{dv}{dt} = -\gamma v(t) + \Gamma(t) \tag{3.47}$$

where

$$\gamma = \frac{\alpha}{m} \tag{3.48}$$

and

$$\Gamma(t) = \frac{F_f(t)}{m} \tag{3.49}$$

The term $\Gamma(t)$ is the *fluctuating force per unit mass*; it is a stochastic force because it depends on the positions of the incredibly large number of atoms constituting the particle, which are in a state of constant and irregular motion. Equation (3.47) is called the *Langevin equation*, and $\Gamma(t)$ is called the *Langevin force*. The Langevin equation, which describes the motion of the particle in the viscous fluid at all times (if its initial conditions are specified), was the first mathematical equation describing nonequilibrium thermodynamics.

In Section 3.9, we show that a transformed version of the LMS algorithm has the same mathematical form as the discrete-time version of the Langevin equation. But, before doing that, we need to describe our next building block.

3.8 KUSHNER'S DIRECT-AVERAGING METHOD

The Markov model of Eq. (3.41) is a *nonlinear stochastic difference equation*. This equation is nonlinear because the transition matrix $\mathbf{A}(n)$ depends on the outer product $\mathbf{x}(n)\mathbf{x}^T(n)$ of the input vector $\mathbf{x}(n)$. Hence, the dependence of the weight-error vector $\boldsymbol{\varepsilon}(n+1)$ on $\mathbf{x}(n)$ violates the principle of superposition, which is a requirement for linearity. Moreover, the equation is stochastic because the training sample $\{\mathbf{x}(n), d(n)\}$ is drawn from a stochastic environment. Given these two realities, we find that a rigorous statistical analysis of the LMS algorithm is indeed a very difficult task.

However, under certain conditions, the statistical analysis of the LMS algorithm can be simplified significantly by applying *Kushner's direct-averaging method* to the model of Eq. (3.41). For a formal statement of this method, we write the following (Kushner, 1984):

Consider a stochastic learning system described by the Markov model

$$\boldsymbol{\varepsilon}(n+1) = \mathbf{A}(n)\boldsymbol{\varepsilon}(n) + \mathbf{f}(n)$$

where, for some input vector $\mathbf{x}(n)$*, we have*

$$\mathbf{A}(n) = \mathbf{I} - \eta\mathbf{x}(n)\mathbf{x}^T(n)$$

and the additive noise $\mathbf{f}(n)$ *is linearly scaled by the learning-rate parameter* η*. Provided that*

- *the learning-rate parameter* η *is sufficiently small, and*
- *the additive noise* $\mathbf{f}(n)$ *is essentially independent of the state* $\boldsymbol{\varepsilon}(n)$*, the state evolution of a modified Markov model described by the two equations*

$$\boldsymbol{\varepsilon}_0(n+1) = \overline{\mathbf{A}}(n)\boldsymbol{\varepsilon}_0(n) + \mathbf{f}_0(n) \tag{3.50}$$

$$\overline{\mathbf{A}}(n) = \mathbf{I} - \eta\mathbb{E}[\mathbf{x}(n)\mathbf{x}^T(n)] \tag{3.51}$$

is practically the same as that of the original Markov model for all n.

The deterministic matrix $\overline{\mathbf{A}}(n)$ of Eq. (3.51) is the transition matrix of the modified Markov model. Note also that we have used the symbol $\boldsymbol{\varepsilon}_0(n)$ for the state of the modified Markov model to emphasize the fact that the evolution of this model over time is identically equal to that of the original Markov model *only* for the limiting case of a vanishingly small learning-rate parameter η.

A proof of the statement embodying Eqs. (3.50) and (3.51) is addressed in Problem 3.7, assuming ergodicity (i.e., substituting time averages for ensemble averages). For the discussion presented herein, it suffices to say the following:

1. As mentioned previously, when the learning-rate parameter η is small, the LMS algorithm has a *long memory*. Hence, the evolution of the updated state $\boldsymbol{\varepsilon}_0(n+1)$ can be traced in time, step by step, all the way back to the initial condition $\boldsymbol{\varepsilon}(0)$.
2. When η is small, we are justified in ignoring all second- and higher-order terms in η in the series expansion of $\boldsymbol{\varepsilon}_0(n+1)$.

3. Finally, the statement embodied in Eqs. (3.50) and (3.51) is obtained by invoking ergodicity, whereby ensemble averages are substituted for time agerages.

3.9 STATISTICAL LMS LEARNING THEORY FOR SMALL LEARNING-RATE PARAMETER

Now that we are equipped with Kushner's direct-averaging method, the stage is set for a principled statistical analysis of the LMS algorithm by making three justifiable assumptions:

Assumption I: The learning-rate parameter η is small

By making this assumption, we justify the application of Kushner's direct-averaging method—hence the adoption of the modified Markov model of Eqs. (3.50) and (3.51) as the basis for the statistical analysis of the LMS algorithm.

From a practical perspective, the choice of small η also makes sense. In particular, the LMS algorithm exhibits its most robust behavior with respect to external disturbances when η is small; the issue of robustness is discussed in Section 3.12.

Assumption II: The estimation error $e_o(n)$ produced by the Wiener filter is white.

This assumption is satisfied if the generation of the desired response is described by the *linear regression model*

$$d(n) = \mathbf{w}_o^T \mathbf{x}(n) + e_o(n) \tag{3.52}$$

Equation (3.52) is simply a rewrite of Eq. (3.44), which, in effect, implies that the weight vector of the Wiener filter is matched to the weight vector of the regression model describing the stochastic environment of interest.

Assumption III: The input vector x(n) and the desired response $d(n)$ are jointly Gaussian

Stochastic processes produced by physical phenomena are frequently mechanized such that a Gaussian model is appropriate—hence the justification for the third assumption.

No further assumptions are needed for the statistical analysis of the LMS algorithm (Haykin, 2002, 2006). In what follows, we present a condensed version of that analysis.

Natural Modes of the LMS Algorithm

Let $\mathbf{R}_{xx}$ denote the ensemble-averaged correlation matrix of the input vector $\mathbf{x}(n)$, drawn from a stationary process; that is,

$$\mathbf{R}_{xx} = \mathbb{E}[\mathbf{x}(n)\mathbf{x}^T(n)] \tag{3.53}$$

Correspondingly, we may express the averaged transition matrix in Eq. (3.51) pertaining to the modified Markov model as

$$\begin{aligned}\overline{\mathbf{A}} &= \mathbb{E}[\mathbf{I} - \eta\mathbf{x}(n)\mathbf{x}^T(n)] \\ &= [\mathbf{I} - \eta\mathbf{R}_{xx}]\end{aligned} \tag{3.54}$$

We may therefore expand Eq. (3.50) into the form

$$\boldsymbol{\varepsilon}_0(n+1) = (\mathbf{I} - \eta\mathbf{R}_{xx})\boldsymbol{\varepsilon}_0(n) + \mathbf{f}_0(n) \tag{3.55}$$

where $\mathbf{f}_0(n)$ is the addition noise. Henceforth, Eq. (3.55) is the equation on which the statistical analysis of the LMS algorithm is based.

Natural Modes of the LMS Algorithm

Applying the *orthogonality transformation* of matrix theory[4] to the correlation matrix $\mathbf{R}_{xx}$, we write

$$\mathbf{Q}^T\mathbf{R}_{xx}\mathbf{Q} = \boldsymbol{\Lambda} \tag{3.56}$$

where $\mathbf{Q}$ is an orthogonal matrix whose columns are the eigenvectors of $\mathbf{R}_{xx}$, and $\boldsymbol{\Lambda}$ is a diagonal matrix whose elements are the associated *eigenvalues*. Extending the application of this transformation to the difference equation Eq. (3.55) yields the corresponding *system of decoupled first-order equations* (Haykin, 2002, 2006)

$$v_k(n+1) = (1 - \eta\lambda_k)v_k(n) + \phi_k(n), \qquad k = 1, 2, ..., M \tag{3.57}$$

where M is the dimensionality of the weight vector $\hat{\mathbf{w}}(n)$. Moreover, $v_k(n)$ is the kth element of the transformed weight-error vector

$$\mathbf{v}(n) = \mathbf{Q}^T\boldsymbol{\varepsilon}_0(n) \tag{3.58}$$

and, correspondingly, $\phi_k(n)$ is the kth element of the transformed noise vector

$$\boldsymbol{\phi}(n) = \mathbf{Q}^T\mathbf{f}_0(n) \tag{3.59}$$

More specifically, $\phi_k(n)$ is the sample function of a white-noise process of zero mean and variance $\mu^2 J_{\min}\lambda_k$,where $J_{\min}$ is the minimum mean-square error produced by the Wiener filter. In effect, the variance of the zero-mean driving force for the kth difference equation Eq. (3.57) is proportional to the kth eigenvalue of the correlation matrix $\mathbf{R}_{xx}$, namely, λ_k.

Define the difference

$$\Delta v_k(n) = v_k(n+1) - v_k(n) \qquad \text{for } k = 1, 2, ..., M \tag{3.60}$$

We may then recast Eq. (3.57) in the form

$$\Delta v_k(n) = -\eta\lambda_k v_k(n) + \phi_k(n) \qquad \text{for } k = 1, 2, ..., M \tag{3.61}$$

The stochastic equation Eq. (3.61) is now recognized as the discrete-time version of the Langevin equation Eq. (3.47). In particular, as we compare these two equations, term by term, we construct the analogies listed in Table 3.2. In light of this table, we may now make the following important statement:

TABLE 3.2 Analogies between the Langevin equation (in continuous time) and the transformed LMS evolution (in discrete time)

Langevin equation Eq. (3.47)		LMS evolution Eq. (3.61)
$\frac{dv(t)}{dt}$	(acceleration)	$\Delta v_k(n)$
$\gamma v(t)$	(damping force)	$\eta\lambda_k v_k(n)$
$\Gamma(t)$	(stochastic driving force)	$\phi_k(n)$

The convergence behavior of the LMS filter resulting from application of the orthogonality transformation to the difference equation Eq. (3.55) is described by a system of M *decoupled Langevin equations whose kth component is characterized as follows*:

- damping force is defined by $\eta\lambda_k v_k(n)$;
- Langevin force $\phi_k(n)$ is described by a zero-mean white-noise process with the variance $\eta^2 J_{\min}\lambda_k$.

Most important, the Langevin force $\phi_k(n)$ is responsible for the nonequilibrium behavior of the LMS algorithm, which manifests itself in the form of *Brownian motion* performed by the algorithm around the optimum Wiener solution after a large enough number of iterations n. It must, however, be stressed that the findings summarized in Table 3.2 and the foregoing statement rest on the premise that the learning-rate parameter η is small.

Learning Curves of the LMS Algorithm

Following through the solution of the transformed difference equation Eq. (3.57), we arrive at the LMS learning curve described by Haykin, (2002, 2006),

$$J(n) = J_{\min} + \eta J_{\min}\sum_{k=1}^{M}\frac{\lambda_k}{2-\eta\lambda_k} + \sum_{k=1}^{M}\lambda_k\left(|v_k(0)|^2 - \frac{\eta J_{\min}}{2-\eta\lambda_k}\right)(1-\eta\lambda_k)^{2n} \quad (3.62)$$

where

$$J(n) = \mathbb{E}[|e(n)|^2]$$

is the mean-square error and $v_k(0)$ is the initial value of the kth element of the transformed vector $\mathbf{v}(n)$. Under the assumption that the learning-rate parameter η is small, Eq. (3.62) simplifies to

$$J(n) \approx J_{\min} + \frac{\eta J_{\min}}{2}\sum_{k=1}^{M}\lambda_k + \sum_{k=1}^{M}\lambda_k\left(|v_k(0)|^2 - \frac{\eta J_{\min}}{2}\right)(1-\eta\lambda_k)^{2n} \quad (3.63)$$

The practical validity of the small-learning-rate-parameter theory presented in this section is demonstrated in the computer experiment presented next.

3.10 COMPUTER EXPERIMENT I: LINEAR PREDICTION

The objective of this experiment is to verify the statistical learning theory of the LMS algorithm described in Section 3.9, assuming a small learning-rate parameter η.

For the experiment, we consider a generative model defined by

$$x(n) = ax(n-1) + \varepsilon(n) \tag{3.64}$$

which represents an *autoregressive (AR) process of order one*. The model being of first order, a is the only parameter of the model. The explanational error $\varepsilon(n)$ is drawn from a zero-mean white-noise process of variance σ_ε^2. The generative model is parameterized as follows:

$$\begin{aligned} a &= 0.99 \\ \sigma_\varepsilon^2 &= 0.02 \\ \sigma_x^2 &= 0.995 \end{aligned}$$

To estimate the model parameter a, we use the LMS algorithm characterized by the learning-rate parameter $\eta = 0.001$. Starting with the initial condition $\hat{w}(0) = 0$, we apply the scalar version of Eq. (3.35), where the estimation error

$$e(n) = x(n) - \hat{a}(n)x(n-1)$$

and where $\hat{a}(n)$ is the estimate of a produced by the LMS algorithm at time n. Then, performing 100 statistically independent application of the LMS algorithm, we plot the *ensemble-averaged learning curve* of the algorithm. The solid (randomly varying) curve plotted in Fig. 3.5 for 5,000 iterations is the result of this ensemble-averaging operation.

In Fig. 3.5, we have also included the result of computing the ensemble-averaged learning curve by using the theoretically derived formula of Eq. (3.63), assuming a small η. It is remarkable to see perfect agreement between theory and practice, as evidenced by

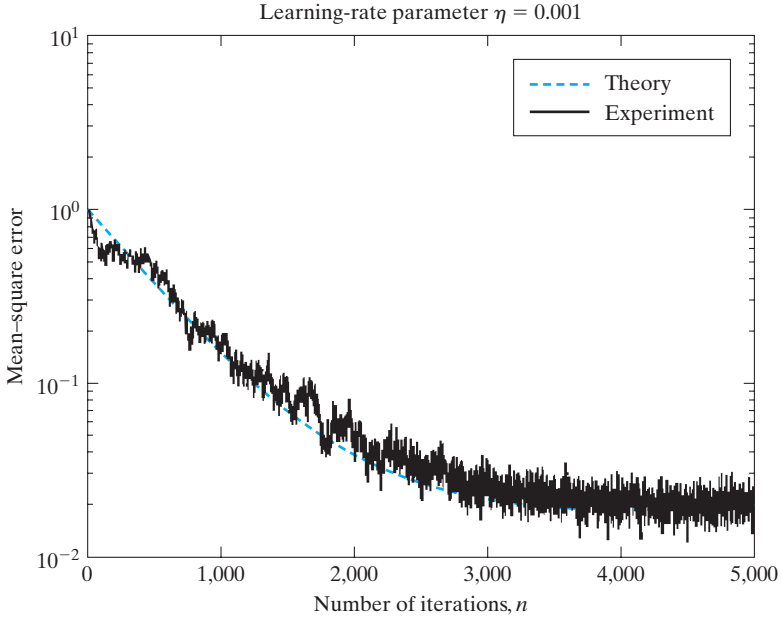


FIGURE 3.5 Experimental verification of the small-learning-rate-parameter theory of the LMS algorithm applied to an autoregressive process of order one.

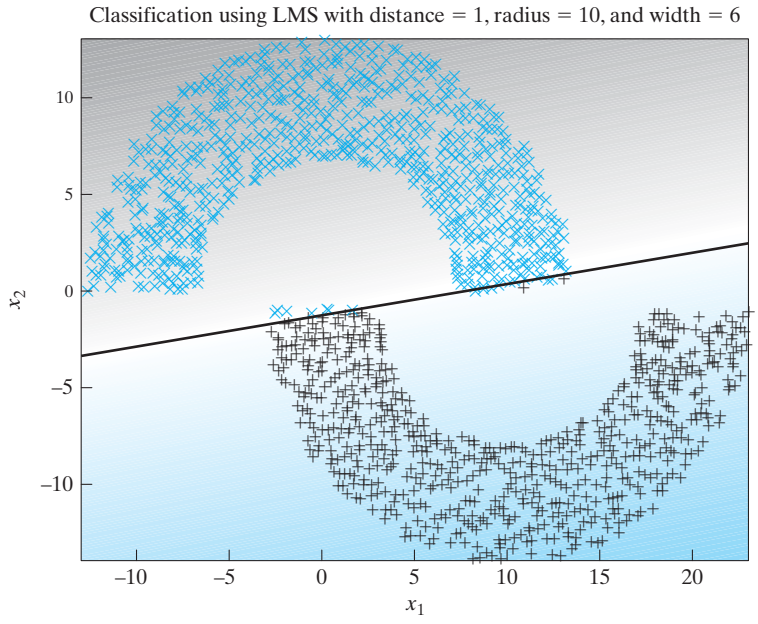


FIGURE 3.6 LMS classification with distance 1, based on the double-moon configuration of Fig. 1.8.

the results plotted in Fig. 3.6. Indeed, this remarkable agreement should be viewed as the confirmation of two important theoretical principles:

1. Kushner's method may be used to tackle the theoretical analysis of the LMS learning behavior under the assumption of a small learning-rate parameter.
2. The LMS algorithm's learning behavior may be explained as an instance of Langevin's equation.

3.11 COMPUTER EXPERIMENT II: PATTERN CLASSIFICATION

For the second experiment on the LMS algorithm, we study the algorithm's application to the double-moon configuration pictured in Fig. 1.8. To be more specific, the performance of the algorithm is evaluated for two settings of the double-moon configuration:

(i) $d = 1$, corresponding to linear separability;
(ii) $d = -4$, corresponding to nonlinear separability.

In doing so, in effect, we are repeating the experiment performed in Section 2.5 on the method of least squares, except this time we use the LMS algorithm.

The results of the experiment pertaining to these two values of d are presented in Figs. 3.6 and 3.7, respectively. Comparing these two figures with Figs. 2.2 and 2.3, we may make the following observations:

(a) Insofar as classification performance is concerned, the method of least squares and the LMS algorithm yield results that are identical for all practical purposes.

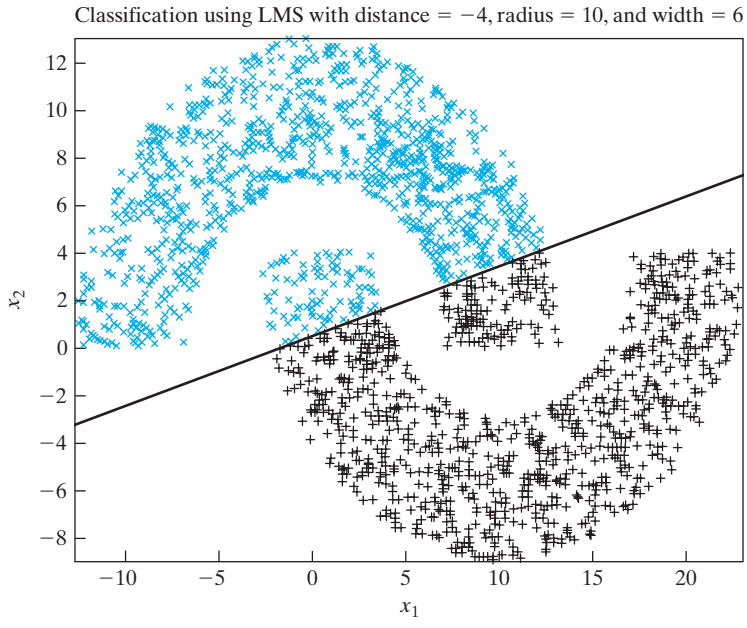


FIGURE 3.7 LMS classification with distance −4, based on the double-moon configuration of Fig. 1.8.

(b) In terms of convergence, the LMS algorithm is much slower than the method of least squares. This difference is attributed to the fact that the LMS algorithm is recursive, whereas the method of least squares operates in a batch mode that involves matrix inversion in one time-step.

As a matter of interest, in Chapter 5, we present a recursive implementation of the method of least squares. On account of using second-order information, recursive implementation of the method of least squares remains faster in its convergence behavior than the LMS algorithm.

3.12 VIRTUES AND LIMITATIONS OF THE LMS ALGORITHM

Computational Simplicity and Efficiency

Two virtues of the LMS algorithm are computational simplicity and efficiency, both of which are exemplified by the following summary of the algorithm presented in Table 3.1:

- Coding of the algorithm is composed of two or three lines, which is as simple as anyone could get.
- Computational complexity of the algorithm is linear in the number of adjustable parameters.

From a practical perspective, these are important virtues.

Robustness

Another important virtue of the LMS algorithm is that it is model independent and therefore *robust* with respect to disturbances. To explain what we mean by robustness, consider the situation depicted in Fig. 3.8, where a transfer operator $\mathcal{T}$ maps a couple of disturbances at its input into a "generic" estimation error at the output. Specifically, at the input, we have the following:

- An *initial weight-error vector* defined by

$$\delta\mathbf{w}(0) = \mathbf{w} - \hat{\mathbf{w}}(0) \tag{3.65}$$

  where $\mathbf{w}$ is an unknown parameter vector and $\hat{\mathbf{w}}(0)$ is its "proposed" initial estimate at time $n = 0$. In the LMS algorithm, we typically set $\hat{\mathbf{w}}(0) = \mathbf{0}$, which, in a way, is the worst possible initializing condition for the algorithm.
- An *explanational error* ε that traces back to the regression model of Eq. (2.3), reproduced here for convenience of presentation, where d is the model output produced in response to the regressor $\mathbf{x}$:

$$d = \mathbf{w}^T\mathbf{x} + \varepsilon \tag{3.66}$$

  Naturally, the operator $\mathcal{T}$ is a function of the strategy used to construct the estimate $\hat{\mathbf{w}}(n)$ (e.g., the LMS algorithm). We may now introduce the following definition:

  The energy gain of the estimator is defined as the ratio of the error energy at the output of the operator $\mathcal{T}$ to the total disturbance energy at the input.

To remove this dependence and thereby make the estimator "model independent," we consider the scenario where we have the *largest possible energy gain over all conceivable disturbance sequences* applied to the estimator input. In so doing, we will have defined the H^∞ *norm* of the transfer operator $\mathcal{T}$.

With this brief background, we may now formulate what the H^∞ norm of the transfer operator $\mathcal{T}$ is about:

Find a causal estimator that minimizes the H^∞ norm of $\mathcal{T}$, where $\mathcal{T}$ is a transfer operator that maps the disturbances to the estimation errors.

The optimal estimator designed in accordance with the H^∞ criterion is said to be of a *minimax* kind. More specifically, we may view the H^∞ optimal estimation problem as a "game-theoretic problem" in the following sense: Nature, acting as the "opponent," has access to the unknown disturbances, thereby maximizing the energy gain. On the other hand, the "designer" of the estimation strategy has the task of finding a causal algorithm for which the error energy is minimized. Note that in introducing the idea of the H^∞ criterion, we made no assumptions about the disturbances indicated at the input of Fig. 3.8. We may therefore say that an estimator designed in accordance with the H^∞ criterion is a *worst-case estimator*.

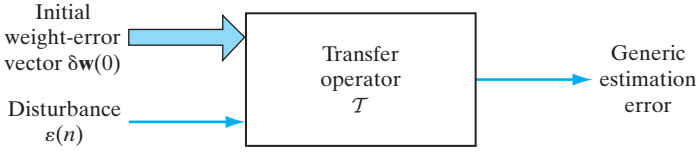


FIGURE 3.8 Formulation of the optimal H^∞ estimation problem. The generic estimation error at the transfer operator's output could be the weight-error vector, the explanational error, etc.

In precise mathematical terms, the LMS algorithm is optimal in accordance with the H^∞ (or minimax) criterion.[5] The basic philosophy of optimality in the H^∞ sense is to cater to the worst-case scenario:

If you do not know what you are up against, plan for the worst scenario and optimize.

For a long time, the LMS algorithm was regarded as an instantaneous approximation to the gradient-descent algorithm. However, the *H^∞ optimality of LMS algorithm* provides this widely used algorithm with a rigorous footing. Moreover, the H^∞ theory of the LMS algorithm shows that the most robust performance of the algorithm is attained when the learning-rate parameter η is assigned a small value.

The model-independent behavior of the LMS algorithm also explains the ability of the algorithm to work satisfactorily in both a stationary and a nonstationary environment. By a "nonstationary" environment, we mean an environment in which the statistics vary with time. In such an environment, the optimum Wiener solution takes on a time-varying form, and the LMS algorithm has the additional task of *tracking* variations in the minimum mean-square error of the Wiener filter.

Factors Limiting the LMS Performance

The primary limitations of the LMS algorithm are its slow rate of convergence and its sensitivity to variations in the eigenstructure of the input (Haykin, 2002). The LMS algorithm typically requires a number of iterations equal to about 10 times the dimensionality of the input data space for it to reach a steady-state condition. The slow rate of convergence of the LMS algorithm becomes particularly serious when the dimensionality of the input data space becomes high.

As for sensitivity to changes in environmental conditions, convergence behavior of the LMS algorithm is particularly sensitive to variations in the *condition number*, or *eigenvalue spread*, of the correlation matrix $\mathbf{R}_{xx}$ of the input vector $\mathbf{x}$. The condition number of $\mathbf{R}_{xx}$, denoted by $\chi(\mathbf{R})$, is defined by

$$\chi(\mathbf{R}) = \frac{\lambda_{\max}}{\lambda_{\min}} \tag{3.67}$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the maximum and minimum eigenvalues of the correlation matrix $\mathbf{R}_{xx}$, respectively. The sensitivity of the LMS algorithm to variations in the condition number $\chi(\mathbf{R})$ becomes particularly acute when the training sample to which the input vector $\mathbf{x}(n)$ belongs is *ill conditioned*—that is, when the condition number of the LMS algorithm is high.[6]

3.13 LEARNING-RATE ANNEALING SCHEDULES

The slow-rate convergence encountered with the LMS algorithm may be attributed to the fact that the learning-rate parameter is maintained constant at some value η_0 throughout the computation, as shown by

$$\eta(n) = \eta_0 \qquad \text{for all } n \tag{3.68}$$

This is the simplest possible form the learning-rate parameter can assume. In contrast, in *stochastic approximation*, which goes back to the classic paper by Robbins and Monro (1951), the learning-rate parameter is time varying. The particular time-varying form most commonly used in the stochastic approximation literature is described by

$$\eta(n) = \frac{c}{n} \tag{3.69}$$

where c is a constant. Such a choice is indeed sufficient to guarantee convergence of the stochastic approximation algorithm (Kushner and Clark, 1978). However, when the constant c is large, there is a danger of parameter blowup for small n.

As an alternative to Eqs. (3.68) and (3.69), we may use the *search-then-converge schedule*, described by Darken and Moody (1992), as

$$\eta(n) = \frac{\eta_0}{1 + (n/\tau)} \tag{3.70}$$

where η_0 and τ are user-selected constants. In the early stages of adaptation involving a number of iterations n that is small compared with the *search-time constant* τ, the learning-rate parameter $\eta(n)$ is approximately equal to η_0, and the algorithm operates essentially as the "conventional" LMS algorithm, as indicated in Fig. 3.9. Hence, by choosing a high value for η_0 within the permissible range, we hope that the adjustable weights of the filter will find and hover about a "good" set of values. Then, for a number n of iterations that is large compared with the search-time constant τ, the learning-rate parameter $\eta(n)$

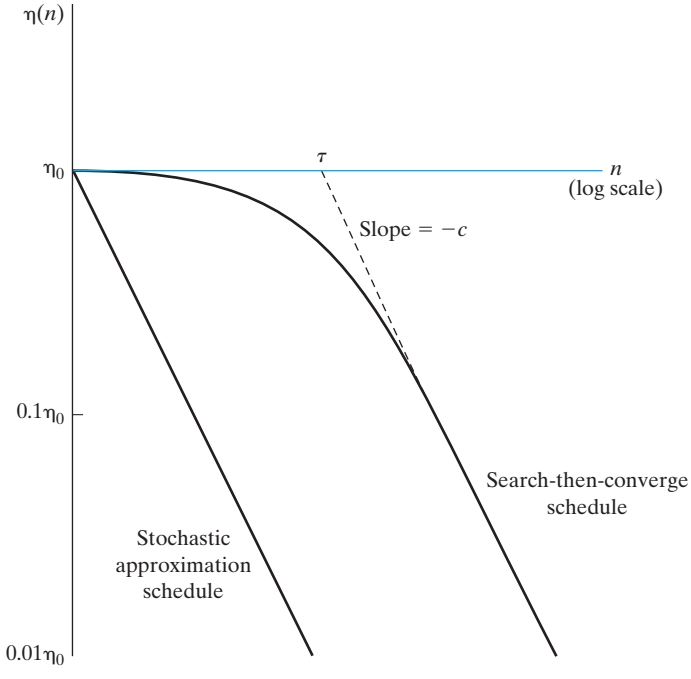


FIGURE 3.9 Learning-rate annealing schedules: The horizontal axis, printed in color, pertains to the standard LMS algorithm.

approximates as c/n, where $c = \tau\eta_0$, as illustrated in Fig. 3.9. The algorithm now operates as a traditional stochastic approximation algorithm, and the weights may converge to their optimum values. Thus, the *search-then-converge schedule* has the potential to combine the desirable features of the standard LMS algorithm with traditional stochastic approximation theory.

3.14 SUMMARY AND DISCUSSION

In this chapter, we studied the celebrated least-mean-square (LMS) algorithm, developed by Widrow and Hoff in 1960. Since its inception, this algorithm has withstood the test of time for a number of important practical reasons:

1. The algorithm is *simple* to formulate and just as simple to implement, be it in hardware or software form.
2. In spite of its simplicity, the algorithm is *effective* in performance.
3. Computationally speaking, the algorithm is *efficient* in that its complexity follows a *linear law* with respect to the number of adjustable parameters.
4. Last, but by no means least, the algorithm is model independent and therefore *robust* with respect to disturbances.

Under the assumption that the learning-rate parameter η is a small positive quantity, the convergence behavior of the LMS algorithm—usually difficult to analyze—becomes mathematically tractable, thanks to Kushner's *direct-averaging method*. The theoretical virtue of this method is that when η is small, the nonlinear "stochastic" difference equation, which describes the convergence behavior of the LMS algorithm, is replaced by a nonlinear "deterministic" version of the original equation. Moreover, through the clever use of eigendecomposition, the solution of the resulting nonlinear deterministic equation is replaced by a system of decoupled first-order difference equations. The important point to note here is that the first-order difference equation so derived is mathematically identical to the discrete-time version of the *Langevin equation* of nonequilibrium thermodynamics. This equivalence explains the Brownian motion executed by the LMS algorithm around the optimum Wiener solution after a large enough number of iterations. The computer experiment presented in Section 3.10 and other computer experiments presented in Haykin (2006) confirm the validity of Eq. (3.63), which describes the ensemble-averaged learning curve of the LMS algorithm.

It is also noteworthy that the LMS algorithm exhibits its most robust performance when the learning-rate parameter η is small. However, the price paid for this kind of practically important performance is a relatively slow rate of convergence. To some extent, this limitation of the LMS algorithm can be alleviated through the use of learning-rate annealing, as described in Section 3.13.

One last comment is in order. Throughout the chapter, we focused attention on the ordinary LMS algorithm. Needless to say, the algorithm has several variants, each of which offers a practical virtue of its own; for details, the interested reader is referred to (Haykin, 2002).

NOTES AND REFERENCES

1. **Differentiation with respect to a vector**
   Let $f(\mathbf{w})$ denote a real-valued function of parameter vector $\mathbf{w}$. The derivative of $f(\mathbf{w})$ with respect to $\mathbf{w}$ is defined by the vector

$$\frac{\partial f}{\partial \mathbf{w}} = \left[\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \ldots, \frac{\partial f}{\partial w_m}\right]^T$$

   where m is the dimension of vector $\mathbf{w}$. The following two cases are of special interest:

   Case 1 The function $f(\mathbf{w})$ is defined by the inner product:

$$\begin{aligned} f(\mathbf{w}) &= \mathbf{x}^T\mathbf{w} \\ &= \sum_{i=1}^{m} x_i w_i \end{aligned}$$

   Hence,

$$\frac{\partial f}{\partial w_i} = x_i, \qquad i = 1, 2, \ldots, m$$

   or, equivalently, in matrix form,

$$\frac{\partial f}{\partial \mathbf{w}} = \mathbf{x} \tag{3.71}$$

   Case 2 The function $f(\mathbf{w})$ is defined by the quadratic form:

$$\begin{aligned} f(\mathbf{w}) &= \mathbf{w}^T\mathbf{R}\mathbf{w} \\ &= \sum_{i=1}^{m}\sum_{j=1}^{m} w_i r_{ij} w_j \end{aligned}$$

   Here, r_{ij} is the ij-th element of the m-by-m matrix $\mathbf{R}$. Hence,

$$\frac{\partial f}{\partial w_i} = 2\sum_{j=1}^{m} r_{ij} w_j, \qquad i = 1, 2, \ldots, m$$

   or, equivalently, in matrix form,

$$\frac{\partial f}{\partial \mathbf{w}} = 2\mathbf{R}\mathbf{w} \tag{3.72}$$

   Equations (3.71) and (3.72) provide two useful rules for the differentiation of a real-valued function with respect to a vector.
2. The pseudoinverse of a rectangular matrix is discussed in Golub and Van Loan (1996); see also Chapter 8 of Haykin (2002).
3. The Langevin equation is discussed in Reif (1965). For a fascinating historical account of the Langevin equation, see the tutorial paper on noise by Cohen (2005).
4. The orthogonality transformation described in Eq. (3.56) follows from the *eigendecomposition* of a square matrix. This topic is described in detail in Chapter 8.

5. For an early (and perhaps the first) motivational treatment of H^∞ control, the reader is referred to Zames (1981).

   The first exposition of optimality of the LMS algorithm in the H^∞ sense was presented in Hassibi et al. (1993). Hassibi et al. (1999) treat the H^∞ theory from an estimation or adaptive-filtering perspective. Hassibi also presents a condensed treatment of robustness of the LMS algorithm in the H^∞ sense in Chapter 5 of Haykin and Widrow (2005).

   For books on H^∞ theory from a control perspective, the reader is referred to Zhou and Doyle (1998) and Green and Limebeer (1995).

6. Sensitivity of convergence behavior of the LMS algorithm to variations in the condition number of the correlation matrix $\mathbf{R}_{xx}$, denoted by $\chi(\mathbf{R})$, is demonstrated experimentally in Section 5.7 of the book by Haykin (2002). In Chapter 9 of Haykin (2002), which deals with recursive implementation of the method of least squares, it is also shown that convergence behavior of the resulting algorithm is essentially independent of the condition number $\chi(\mathbf{R})$.

PROBLEMS

3.1 **(a)** Let $\mathbf{m}(n)$ denote the mean weight vector of the LMS algorithm at iteration n; that is,

$$\mathbf{m}(n) = \mathbb{E}[\hat{\mathbf{w}}(n)]$$

Using the small-learning-rate parameter theory of Section 3.9, show that

$$\mathbf{m}(n) = (\mathbf{I} - \eta\mathbf{R}_{xx})^n[\mathbf{m}(0) - \mathbf{m}(\infty)] + \mathbf{m}(\infty)$$

where η is the learning-rate parameter, $\mathbf{R}_{xx}$ is the correlation matrix of the input vector $\mathbf{x}(n)$, and $\mathbf{m}(0)$ and $\mathbf{m}(\infty)$ are the initial and final values of $\mathbf{m}(n)$, respectively.

(b) Show that for *convergence of the LMS algorithm in the mean*, the learning-rate parameter η must satisfy the condition

$$O < \eta < \frac{2}{\lambda_{\max}}$$

where $\lambda_{\max}$ is the largest eigenvalue of the correlation matrix $\mathbf{R}_{xx}$.

3.2 Continuing from Problem 3.1, discuss why convergence of the LMS algorithm in the mean is not an adequate criterion for convergence in practice.

3.3 Consider the use of a white-noise sequence of zero mean and variance σ^2 as the input to the LMS algorithm. Determine the condition for convergence of the algorithm in the mean-square sense.

3.4 In a variant of the LMS algorithm called the *leaky LMS algorithm*, the cost function to be minimized is defined by

$$\mathscr{E}(n) = \frac{1}{2}|e(n)|^2 + \frac{1}{2}\lambda\|\mathbf{w}(n)\|^2$$

where $\mathbf{w}(n)$ is the parameter vector, $e(n)$ is the estimation error, and λ is a constant. As in the ordinary LMS algorithm, we have

$$e(n) = d(n) - \mathbf{w}^T(n)\mathbf{x}(n)$$

where $d(n)$ is the desired response corresponding to the input vector $\mathbf{x}(n)$.

(a) Show that the time update for the parameter vector of the leaky LMS algorithm is defined by

$$\hat{\mathbf{w}}(n+1) = (1 - \eta\lambda)\hat{\mathbf{w}}(n) + \eta\mathbf{x}(n)e(n)$$

which includes the ordinary LMS algorithm as a special case.

(b) Using the small learning-rate parameter theory of Section 3.9, show that

$$\lim_{x\to\infty} \mathbb{E}[\hat{\mathbf{w}}(n)] = (\mathbf{R}_{xx} + \lambda\mathbf{I})^{-1}\mathbf{r}_{dx}$$

where $\mathbf{R}_{xx}$ is the correlation matrix of $\mathbf{x}(n)$, $\mathbf{I}$ is the identity matrix, and $\mathbf{r}_{dx}$ is the cross-correlation vector between $\mathbf{x}(n)$ and $d(n)$.

3.5 Continuing from Problem 3.4, verify that the leaky LMS algorithm can be "simulated" by adding white noise to the input vector $\mathbf{x}(n)$.

(a) What should variance of this noise be for the condition in part (b) of Problem 3.4 to hold?

(b) When will the simulated algorithm take a form that is practically the same as the leaky LMS algorithm? Justify your answer.

3.6 An alternative to the mean-square error (MSE) formulation of the learning curve that we sometimes find in the literature is the *mean-square deviation* (MSD) *learning curve.* Define the weight-error vector

$$\boldsymbol{\varepsilon}(n) = \mathbf{w} - \hat{\mathbf{w}}(n)$$

where $\mathbf{w}$ is the parameter vector of the regression model supplying the desired response. This second learning curve is obtained by computing a plot of the MSD

$$D(n) = \mathbb{E}[\|\boldsymbol{\varepsilon}(n)\|^2]$$

versus the number of iterations n.

Using the small-learning-rate-parameter theory of Section 3.9, show that

$$\begin{aligned} D(\infty) &= \lim_{n\to\infty} D(n) \\ &= \frac{1}{2}\eta M J_{\min} \end{aligned}$$

where η is the learning-rate parameter, M is the size of the parameter vector $\hat{\mathbf{w}}$, and $J_{\min}$ is the minimum mean-square error of the LMS algorithm.

3.7 In this problem, we address a proof of the direct-averaging method, assuming ergodicity.

Start with Eq. (3.41), which defines the weight-error vector $\boldsymbol{\varepsilon}(n)$ in terms of the transition matrix $\mathbf{A}(n)$ and driving force $\mathbf{f}(n)$, which are themselves defined in terms of the input vector $\mathbf{x}(n)$ in Eqs. (3.42) and (3.43), respectively; then proceed as follows:

- Set $n = 0$, and evaluate $\boldsymbol{\varepsilon}(1)$.
- Set $n = 1$, and evaluate $\boldsymbol{\varepsilon}(2)$.
- Continue in this fashion for a few more iterations.

With these iterated values of $\boldsymbol{\varepsilon}(n)$ at hand, deduce a formula for the transition matrix $\mathbf{A}(n)$.

Next, assume that the learning-rate parameter η is small enough to justify retaining only the terms that are linear in η. Hence, show that

$$\overline{\mathbf{A}}(n) = \mathbf{I} - \eta\sum_{i=1}^{n}\mathbf{x}(i)\mathbf{x}^T(i)$$

which, assuming ergodicity, takes the form

$$\overline{\mathbf{A}}(n) = \mathbf{I} - \mu\mathbf{R}_{xx}$$

3.8 When the learning-rate parameter η is small, the LMS algorithm acts like a *low-pass filter with a small cutoff frequency.* Such a filter produces an output that is proportional to the *average* of the input signal.

Using Eq. (3.41), demonstrate this property of the LMS algorithm by considering the simple example of the algorithm using a single parameter.

3.9 Starting with Eq. (3.55) for a small learning-rate parameter, show that under steady-state conditions, the *Lyapunov equation*

$$\mathbf{R}\mathbf{P}_0(n) + \mathbf{P}_0(n)\mathbf{R} = \eta \sum_{i=0}^{\infty} J_{\min}^{(i)} \mathbf{R}^{(i)}$$

holds, where we have

$$J_{\min}^{(i)} = \mathbb{E}[e_o(n)e_o(n-i)]$$

and

$$\mathbf{R}^{(i)} = \mathbb{E}[\mathbf{x}(n)\mathbf{x}^T(n-i)]$$

for $i = 0, 1, 2, \ldots$. The matrix $\mathbf{P}_0$ is defined by $\mathbb{E}[\boldsymbol{\varepsilon}_o(n)\boldsymbol{\varepsilon}_o^T(n)]$, and $e_o(n)$ is the irreducible estimation error produced by the Wiener filter.

Computer Experiments

3.10 Repeat the computer experiment of Section 3.10 on linear prediction for the following values of the learning-rate parameter:

(i) $\eta = 0.002$;
(ii) $\eta = 0.01$;
(iii) $\eta = 0.02$.

Comment on your findings in the context of applicability of the small-learning-rate-parameter theory of the LMS algorithm for each value of η.

3.11 Repeat the computer experiment of Section 3.11 on pattern classification for the distance of separation between the two moons of Fig. 1.8 set at $d = 0$. Compare the results of your experiment with those in Problem 1.6 on the perceptron and Problem 2.7 on the method of least squares.

3.12 Plot the pattern-classification learning curves of the LMS algorithm applied to the double-moon configuration of Fig. 1.8 for the following values assigned to the distance of separation:

$d = 1$
$d = 0$
$d = -4$

Compare the results of the experiment with the corresponding ones obtained using Rosenblatt's perceptron in Chapter 1.

CHAPTER 4

Multilayer Perceptrons

ORGANIZATION OF THE CHAPTER

In this chapter, we study the many facets of the multilayer perceptron, which stands for a neural network with one or more hidden layers. After the introductory material presented in Section 4.1, the study proceeds as follows:

1. Sections 4.2 through 4.7 discuss matters relating to back-propagation learning. We begin with some preliminaries in Section 4.2 to pave the way for the derivation of the back-propagation algorithm. This section also includes a discussion of the credit-assignment problem. In Section 4.3, we describe two methods of learning: batch and on-line. In Section 4.4, we present a detailed derivation of the back-propagation algorithm, using the chain rule of calculus; we take a traditional approach in this derivation. In Section 4.5, we illustrate the use of the back-propagation algorithm by solving the XOR problem, an interesting problem that cannot be solved by Rosenblatt's perceptron. Section 4.6 presents some heuristics and practical guidelines for making the back-propagation algorithm perform better. Section 4.7 presents a pattern-classification experiment on the multilayer perceptron trained with the back-propagation algorithm.
2. Sections 4.8 and 4.9 deal with the error surface. In Section 4.8, we discuss the fundamental role of back-propagation learning in computing partial derivatives of a network-approximating function. We then discuss computational issues relating to the Hessian of the error surface in Section 4.9. In Section 4.10, we discuss two issues: how to fulfill optimal annealing and how to make the learning-rate parameter adaptive.
3. Sections 4.11 through 4.14 focus on various matters relating to the performance of a multilayer perceptron trained with the back-propagation algorithm. In Section 4.11, we discuss the issue of generalization—the very essence of learning. Section 4.12 addresses the approximation of continuous functions by means of multiplayer perceptrons. The use of cross-validation as a statistical design tool is discussed in Section 4.13. In Section 4.14, we discuss the issue of complexity regularization, as well as network-pruning techniques.
4. Section 4.15, summarizes the advantages and limitations of back-propagation learning.
5. Having completed the study of back-propagation learning, we next take a different perspective on learning in Section 4.16 by viewing supervised learning as an optimization problem.

6. Section 4.17 describes an important neural network structure: the *convolutional multilayer perceptron*. This network has been successfully used in the solution of difficult pattern-recognition problems.
7. Section 4.18 deals with nonlinear filtering, where time plays a key role. The discussion begins with short-term memory structures, setting the stage for the universal myopic mapping theorem.
8. Section 4.19 discusses the issue of small-scale versus large-scale learning problems.

The chapter concludes with summary and discussion in Section 4.20.

4.1 INTRODUCTION

In Chapter 1, we studied Rosenblatt's perceptron, which is basically a single-layer neural network. Therein, we showed that this network is limited to the classification of linearly separable patterns. Then we studied adaptive filtering in Chapter 3, using Widrow and Hoff's LMS algorithm. This algorithm is also based on a single linear neuron with adjustable weights, which limits the computing power of the algorithm. To overcome the practical limitations of the perceptron and the LMS algorithm, we look to a neural network structure known as the *multilayer perceptron*.

The following three points highlight the basic features of multilayer perceptrons:

- The model of each neuron in the network includes a nonlinear activation function that is *differentiable*.
- The network contains one or more layers that are *hidden* from both the input and output nodes.
- The network exhibits a high degree of *connectivity*, the extent of which is determined by synaptic weights of the network.

These same characteristics, however, are also responsible for the deficiencies in our knowledge on the behavior of the network. First, the presence of a distributed form of nonlinearity and the high connectivity of the network make the theoretical analysis of a multilayer perceptron difficult to undertake. Second, the use of hidden neurons makes the learning process harder to visualize. In an implicit sense, the learning process must decide which features of the input pattern should be represented by the hidden neurons. The learning process is therefore made more difficult because the search has to be conducted in a much larger space of possible functions, and a choice has to be made between alternative representations of the input pattern.

A popular method for the training of multilayer perceptrons is the back-propagation algorithm, which includes the LMS algorithm as a special case. The training proceeds in two phases:

1. In the *forward phase*, the synaptic weights of the network are fixed and the input signal is propagated through the network, layer by layer, until it reaches the output. Thus, in this phase, changes are confined to the activation potentials and outputs of the neurons in the network.

2. In the *backward phase*, an error signal is produced by comparing the output of the network with a desired response. The resulting error signal is propagated through the network, again layer by layer, but this time the propagation is performed in the backward direction. In this second phase, successive adjustments are made to the synaptic weights of the network. Calculation of the adjustments for the output layer is straightforward, but it is much more challenging for the hidden layers.

Usage of the term "back propagation" appears to have evolved after 1985, when the term was popularized through the publication of the seminal book entitled *Parallel Distributed Processing* (Rumelhart and McClelland, 1986).

The development of the back-propagation algorithm in the mid-1980s represented a landmark in neural networks in that it provided a *computationally efficient* method for the training of multilayer perceptrons, putting to rest the pessimism about learning in multilayer perceptrons that may have been inferred from the book by Minsky and Papert (1969).

4.2 SOME PRELIMINARIES

Figure 4.1 shows the architectural graph of a multiplayer perceptron with two hidden layers and an output layer. To set the stage for a description of the multilayer perceptron in its general form, the network shown here is *fully connected*. This means that a neuron in any layer of the network is connected to all the neurons (nodes) in the previous layer. Signal flow through the network progresses in a forward direction, from left to right and on a layer-by-layer basis.

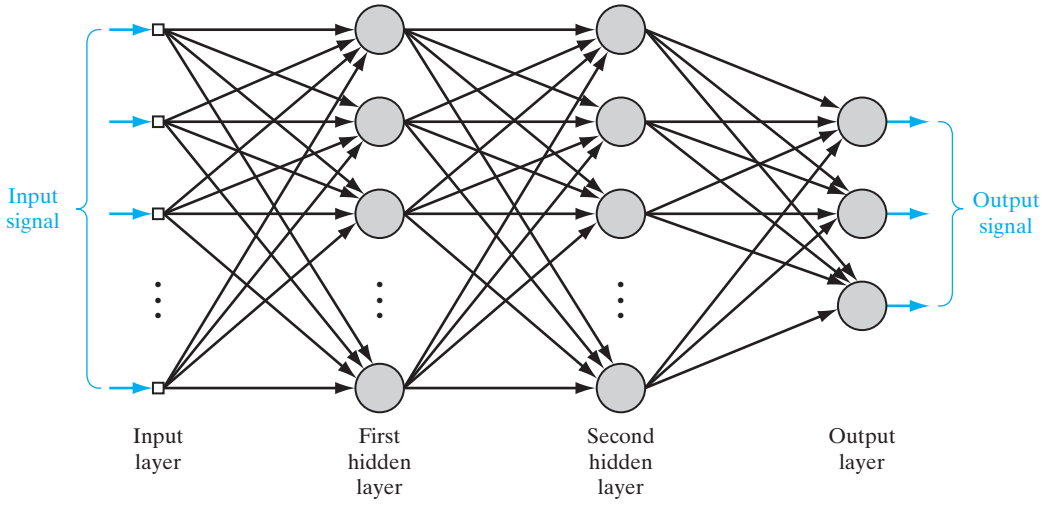


FIGURE 4.1 Architectural graph of a multilayer perceptron with two hidden layers.

Figure 4.2 depicts a portion of the multilayer perceptron. Two kinds of signals are identified in this network:

1. *Function Signals.* A function signal is an input signal (stimulus) that comes in at the input end of the network, propagates forward (neuron by neuron) through the network, and emerges at the output end of the network as an output signal. We refer to such a signal as a "function signal" for two reasons. First, it is presumed to perform a useful function at the output of the network. Second, at each neuron of the network through which a function signal passes, the signal is calculated as a function of the inputs and associated weights applied to that neuron. The function signal is also referred to as the input signal.
2. *Error Signals.* An error signal originates at an output neuron of the network and propagates backward (layer by layer) through the network. We refer to it as an "error signal" because its computation by every neuron of the network involves an error-dependent function in one form or another.

The output neurons constitute the output layer of the network. The remaining neurons constitute hidden layers of the network. Thus, the hidden units are not part of the output or input of the network—hence their designation as "hidden." The first hidden layer is fed from the input layer made up of sensory units (source nodes); the resulting outputs of the first hidden layer are in turn applied to the next hidden layer; and so on for the rest of the network.

Each hidden or output neuron of a multilayer perceptron is designed to perform two computations:

1. the computation of the function signal appearing at the output of each neuron, which is expressed as a continuous nonlinear function of the input signal and synaptic weights associated with that neuron;
2. the computation of an estimate of the gradient vector (i.e., the gradients of the error surface with respect to the weights connected to the inputs of a neuron), which is needed for the backward pass through the network.

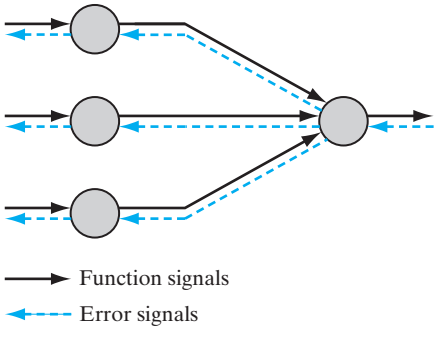


FIGURE 4.2 Illustration of the directions of two basic signal flows in a multilayer perceptron: forward propagation of function signals and back propagation of error signals.

Function of the Hidden Neurons

The hidden neurons act as *feature detectors*; as such, they play a critical role in the operation of a multilayer perceptron. As the learning process progresses across the multilayer perceptron, the hidden neurons begin to gradually "discover" the salient features that characterize the training data. They do so by performing a nonlinear transformation on the input data into a new space called the *feature space*. In this new space, the classes of interest in a pattern-classification task, for example, may be more easily separated from each other than could be the case in the original input data space. Indeed, it is the formation of this feature space through supervised learning that distinguishes the multilayer perceptron from Rosenblatt's perceptron.

Credit-Assignment Problem

When studying learning algorithms for distributed systems, exemplified by the multilayer perceptron of Figure 4.1, it is instructive to pay attention to the notion of *credit assignment.* Basically, the credit-assignment problem is the problem of assigning *credit* or *blame* for overall outcomes to each of the *internal decisions* made by the hidden computational units of the distributed learning system, recognizing that those decisions are responsible for the overall outcomes in the first place.

In a multilayer perceptron using *error-correlation learning*, the credit-assignment problem arises because the operation of each hidden neuron and of each output neuron in the network is important to the network's correct overall action on a learning task of interest. That is, in order to solve the prescribed task, the network must assign certain forms of behavior to all of its neurons through a specification of the error-correction learning algorithm. With this background, consider the multilayer perceptron depicted in Fig. 4.1. Since each output neuron is visible to the outside world, it is possible to supply a desired response to guide the behavior of such a neuron. Thus, as far as output neurons are concerned, it is a straightforward matter to adjust the synaptic weights of each output neuron in accordance with the error-correction algorithm. But how do we assign credit or blame for the action of the hidden neurons when the error-correction learning algorithm is used to adjust the respective synaptic weights of these neurons? The answer to this fundamental question requires more detailed attention than in the case of output neurons.

In what follows in this chapter, we show that the back-propagation algorithm, basic to the training of a multilayer perceptron, solves the credit-assignment problem in an elegant manner. But before proceeding to do that, we describe two basic methods of supervised learning in the next section.

4.3 BATCH LEARNING AND ON-LINE LEARNING

Consider a multilayer perceptron with an input layer of source nodes, one or more hidden layers, and an output layer consisting of one or more neurons; as illustrated in Fig. 4.1. Let

$$\mathcal{T} = \{\mathbf{x}(n), \mathbf{d}(n)\}_{n=1}^{N} \tag{4.1}$$

denote the *training sample* used to train the network in a supervised manner. Let $y_j(n)$ denote the function signal produced at the output of neuron j in the output layer by the stimulus $\mathbf{x}(n)$ applied to the input layer. Correspondingly, the *error signal* produced at the output of neuron j is defined by

$$e_j(n) = d_j(n) - y_j(n) \tag{4.2}$$

where $d_j(n)$ is the ith element of the desired-response vector $\mathbf{d}(n)$. Following the terminology of the LMS algorithm studied in Chapter 3, the *instantaneous error energy* of neuron j is defined by

$$\mathscr{E}_j(n) = \frac{1}{2}e_j^2(n) \tag{4.3}$$

Summing the error-energy contributions of all the neurons in the output layer, we express the *total instantaneous error energy* of the whole network as

$$\begin{aligned}\mathscr{E}(n) &= \sum_{j\in C}\mathscr{E}_j(n)\\ &= \frac{1}{2}\sum_{j\in C}e_j^2(n)\end{aligned} \tag{4.4}$$

where the set C includes all the neurons in the output layer. With the training sample consisting of N examples, the *error energy averaged over the training sample*, or the *empirical risk*, is defined by

$$\begin{aligned}\mathscr{E}_{\text{av}}(N) &= \frac{1}{N}\sum_{n=1}^{N}\mathscr{E}(n)\\ &= \frac{1}{2N}\sum_{n=1}^{N}\sum_{j\in C}e_j^2(n)\end{aligned} \tag{4.5}$$

Naturally, the instantaneous error energy, and therefore the average error energy, are both functions of all the adjustable synaptic weights (i.e., free parameters) of the multilayer perceptron. This functional dependence has not been included in the formulas for $\mathscr{E}(n)$ and $\mathscr{E}_{\text{av}}(N)$, merely to simplify the terminology.

Depending on how the supervised learning of the multilayer perceptron is actually performed, we may identify two different methods—namely, batch learning and on-line learning, as discussed next in the context of gradient descent.

Batch Learning

In the batch method of supervised learning, adjustments to the synaptic weights of the multilayer perceptron are performed *after* the presentation of *all* the N examples in the training sample $\mathcal{T}$ that constitute one *epoch* of training. In other words, the cost function for batch learning is defined by the average error energy $\mathscr{E}_{\text{av}}$. Adjustments to the synaptic weights of the multilayer perceptron are made on an *epoch-by-epoch basis*. Correspondingly, one realization of the learning curve is obtained by plotting $\mathscr{E}_{\text{av}}$ versus the number

of epochs, where, for each epoch of training, the examples in the training sample $\mathcal{T}$ are *randomly shuffled*. The learning curve is then computed by *ensemble averaging* a large enough number of such realizations, where each realization is performed for a *different set of initial conditions* chosen at random.

With the method of gradient descent used to perform the training, the advantages of batch learning include the following:

- *accurate estimation* of the gradient vector (i.e., the derivative of the cost function $\mathscr{E}_{\text{av}}$ with respect to the weight vector $\mathbf{w}$), thereby guaranteeing, under simple conditions, convergence of the method of steepest descent to a local minimum;
- *parallelization* of the learning process.

However, from a practical perspective, batch learning is rather demanding in terms of *storage requirements*.

In a statistical context, batch learning may be viewed as a form of *statistical inference*. It is therefore well suited for solving *nonlinear regression problems*.

On-line Learning

In the on-line method of supervised learning, adjustments to the synaptic weights of the multilayer perceptron are performed on an *example-by-example basis*. The cost function to be minimized is therefore the total instantaneous error energy $\mathscr{E}(n)$.

Consider an epoch of N training examples arranged in the order $\{\mathbf{x}(1), \mathbf{d}(1)\}, \{\mathbf{x}(2), \mathbf{d}(2)\}, ..., \{\mathbf{x}(N), \mathbf{d}(N)\}$. The first example pair $\{\mathbf{x}(1), \mathbf{d}(1)\}$ in the epoch is presented to the network, and the weight adjustments are performed using the method of gradient descent. Then the second example $\{\mathbf{x}(2), \mathbf{d}(2)\}$ in the epoch is presented to the network, which leads to further adjustments to weights in the network. This procedure is continued until the last example $\{\mathbf{x}(N), \mathbf{d}(N)\}$ is accounted for. Unfortunately, such a procedure works against the parallalization of on-line learning.

For a given set of initial conditions, a single realization of the learning curve is obtained by plotting the final value $\mathscr{E}(N)$ versus the number of epochs used in the training session, where, as before, the training examples are randomly shuffled after each epoch. As with batch learning, the learning curve for on-line learning is computed by ensemble averaging such realizations over a large enough number of initial conditions chosen at random. Naturally, for a given network structure, the learning curve obtained under on-line learning will be quite different from that under batch learning.

Given that the training examples are presented to the network in a random manner, the use of on-line learning makes the search in the multidimensional weight space *stochastic* in nature; it is for this reason that the method of on-line learning is sometimes referred to as a *stochastic method*. This stochasticity has the desirable effect of making it less likely for the learning process to be trapped in a local minimum, which is a definite advantage of on-line learning over batch learning. Another advantage of on-line learning is the fact that it requires much less storage than batch learning.

Moreover, when the training data are *redundant* (i.e., the training sample $\mathcal{T}$ contains several copies of the same example), we find that, unlike batch learning, on-line

learning is able to take advantage of this redundancy because the examples are presented one at a time.

Another useful property of on-line learning is its ability to *track small changes* in the training data, particularly when the environment responsible for generating the data is nonstationary.

To summarize, despite the disadvantages of on-line learning, it is highly popular for solving *pattern-classification problems* for two important practical reasons:

- On-line learning is simple to implement.
- It provides effective solutions to large-scale and difficult pattern-classification problems.

It is for these two reasons that much of the material presented in this chapter is devoted to on-line learning.

4.4 THE BACK-PROPAGATION ALGORITHM

The popularity of on-line learning for the supervised training of multilayer perceptrons has been further enhanced by the development of the back-propagation algorithm. To describe this algorithm, consider Fig. 4.3, which depicts neuron j being fed by a set of

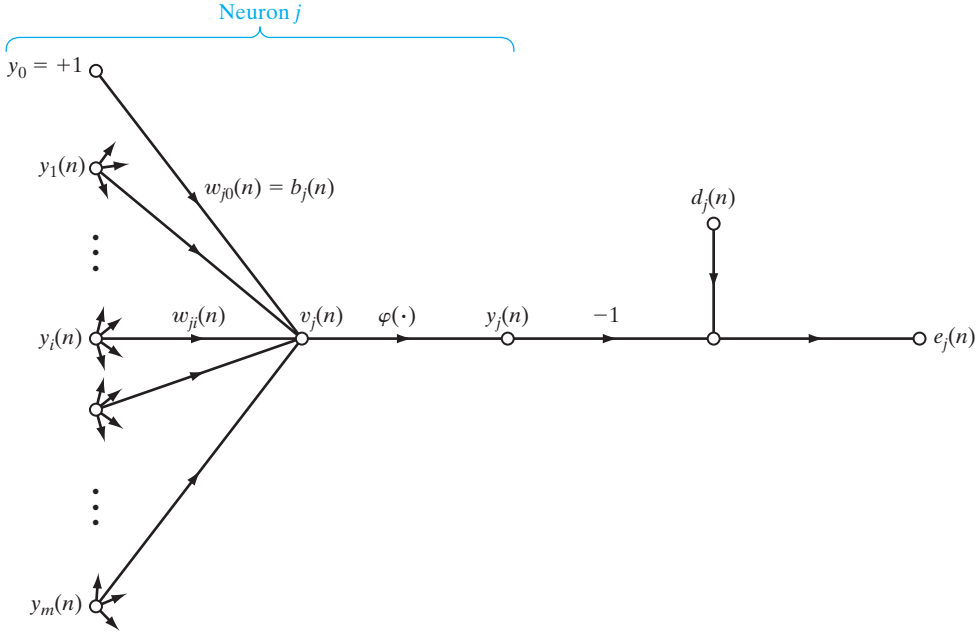


FIGURE 4.3 Signal-flow graph highlighting the details of output neuron j.

function signals produced by a layer of neurons to its left. The induced local field $v_j(n)$ produced at the input of the activation function associated with neuron j is therefore

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{4.6}$$

where m is the total number of inputs (excluding the bias) applied to neuron j. The synaptic weight w_{j0} (corresponding to the fixed input $y_0 = +1$) equals the bias b_j applied to neuron j. Hence, the function signal $y_j(n)$ appearing at the output of neuron j at iteration n is

$$y_j(n) = \varphi_j(v_j(n)) \tag{4.7}$$

In a manner similar to the LMS algorithm studied in Chapter 3, the back-propagation algorithm applies a correction $\Delta w_{ji}(n)$ to the synaptic weight $w_{ji}(n)$, which is proportional to the partial derivative $\partial \mathscr{E}(n)/\partial w_{ji}(n)$. According to the *chain rule* of calculus, we may express this gradient as

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{4.8}$$

The partial derivative $\partial \mathscr{E}(n)/\partial w_{ji}(n)$ represents a *sensitivity factor*, determining the direction of search in weight space for the synaptic weight w_{ji}.

Differentiating both sides of Eq. (4.4) with respect to $e_j(n)$, we get

$$\frac{\partial \mathscr{E}(n)}{\partial e_j(n)} = e_j(n) \tag{4.9}$$

Differentiating both sides of Eq. (4.2) with respect to $y_j(n)$, we get

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{4.10}$$

Next, differentiating Eq. (4.7) with respect to $v_j(n)$, we get

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)) \tag{4.11}$$

where the use of prime (on the right-hand side) signifies differentiation with respect to the argument. Finally, differentiating Eq. (4.6) with respect to $w_{ji}(n)$ yields

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n) \tag{4.12}$$

The use of Eqs. (4.9) to (4.12) in Eq. (4.8) yields

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n) \tag{4.13}$$

The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ is defined by the *delta rule*, or

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} \tag{4.14}$$

where η is the *learning-rate parameter* of the back-propagation algorithm. The use of the minus sign in Eq. (4.14) accounts for *gradient descent* in weight space (i.e., seeking a direction for weight change that reduces the value of $\mathscr{E}(n)$). Accordingly, the use of Eq. (4.13) in Eq. (4.14) yields

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) \tag{4.15}$$

where the *local gradient* $\delta_j(n)$ is defined by

$$\begin{aligned} \delta_j(n) &= \frac{\partial \mathscr{E}(n)}{\partial v_j(n)} \\ &= \frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= e_j(n) \varphi_j'(v_j(n)) \end{aligned} \tag{4.16}$$

The local gradient points to required changes in synaptic weights. According to Eq. (4.16), the local gradient $\delta_j(n)$ for output neuron j is equal to the product of the corresponding error signal $e_j(n)$ for that neuron and the derivative $\varphi_j'(v_j(n))$ of the associated activation function.

From Eqs. (4.15) and (4.16), we note that a key factor involved in the calculation of the weight adjustment $\Delta w_{ji}(n)$ is the error signal $e_j(n)$ at the output of neuron j. In this context, we may identify two distinct cases, depending on where in the network neuron j is located. In case 1, neuron j is an output node. This case is simple to handle because each output node of the network is supplied with a desired response of its own, making it a straightforward matter to calculate the associated error signal. In case 2, neuron j is a hidden node. Even though hidden neurons are not directly accessible, they share responsibility for any error made at the output of the network. The question, however, is to know how to penalize or reward hidden neurons for their share of the responsibility. This problem is the *credit-assignment problem* considered in Section 4.2.

Case 1 Neuron *j* Is an Output Node

When neuron j is located in the output layer of the network, it is supplied with a desired response of its own. We may use Eq. (4.2) to compute the error signal $e_j(n)$ associated with this neuron; see Fig. 4.3. Having determined $e_j(n)$, we find it a straightforward matter to compute the local gradient $\delta_j(n)$ by using Eq. (4.16).

Case 2 Neuron *j* Is a Hidden Node

When neuron j is located in a hidden layer of the network, there is no specified desired response for that neuron. Accordingly, the error signal for a hidden neuron would have to be determined recursively and working backwards in terms of the error signals of all the neurons to which that hidden neuron is directly connected; this is where the

development of the back-propagation algorithm gets complicated. Consider the situation in Fig. 4.4, which depicts neuron j as a hidden node of the network. According to Eq. (4.16), we may redefine the local gradient $\delta_j(n)$ for hidden neuron j as

$$\begin{aligned}\delta_j(n) &= -\frac{\partial \mathscr{E}(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)} \\ &= -\frac{\partial \mathscr{E}(n)}{\partial y_j(n)}\varphi_j'(v_j(n)), \qquad \text{neuron } j \text{ is hidden}\end{aligned} \tag{4.17}$$

where in the second line we have used Eq. (4.11). To calculate the partial derivative $\partial \mathscr{E}(n)/\partial y_j(n)$, we may proceed as follows: From Fig. 4.4, we see that

$$\mathscr{E}(n) = \frac{1}{2}\sum_{k \in C} e_k^2(n), \qquad \text{neuron } k \text{ is an output node} \tag{4.18}$$

which is Eq. (4.4) with index k used in place of index j. We have made this substitution in order to avoid confusion with the use of index j that refers to a hidden neuron under case 2. Differentiating Eq. (4.18) with respect to the function signal $y_j(n)$, we get

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} \tag{4.19}$$

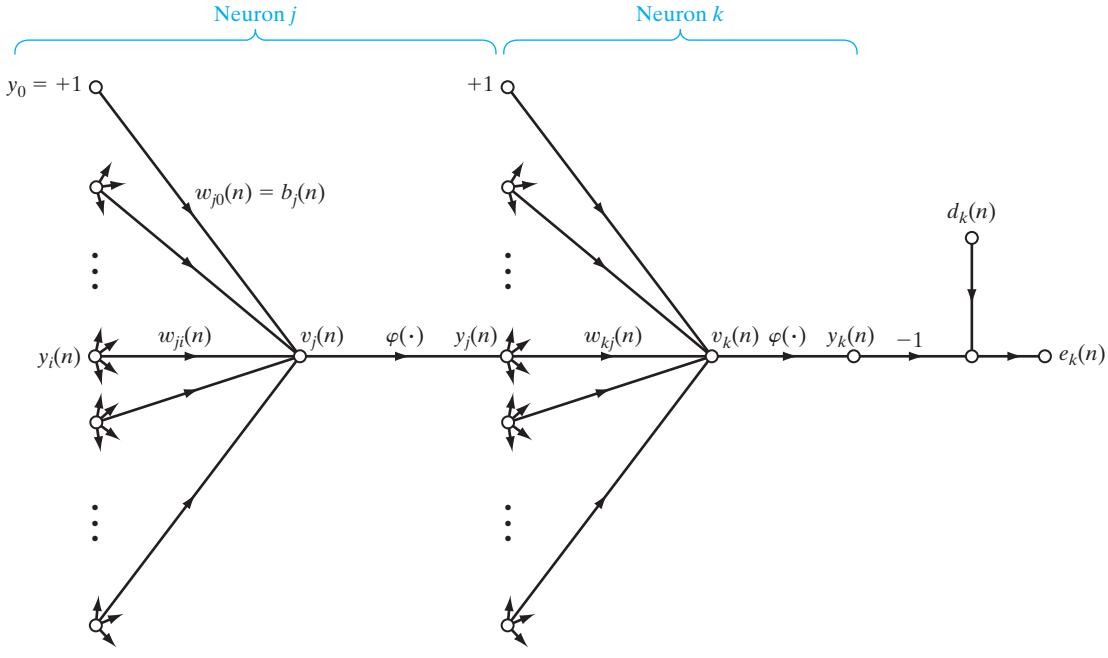


FIGURE 4.4 Signal-flow graph highlighting the details of output neuron k connected to hidden neuron j.

Next we use the chain rule for the partial derivative $\partial e_k(n)/\partial y_j(n)$ and rewrite Eq. (4.19) in the equivalent form

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \tag{4.20}$$

However, from Fig. 4.4, we note that

$$\begin{aligned} e_k(n) &= d_k(n) - y_k(n) \\ &= d_k(n) - \varphi_k(v_k(n)), \qquad \text{neuron } k \text{ is an output node} \end{aligned} \tag{4.21}$$

Hence,

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'(v_k(n)) \tag{4.22}$$

We also note from Fig. 4.4 that for neuron k, the induced local field is

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \tag{4.23}$$

where m is the total number of inputs (excluding the bias) applied to neuron k. Here again, the synaptic weight $w_{k0}(n)$ is equal to the bias $b_k(n)$ applied to neuron k, and the corresponding input is fixed at the value +1. Differentiating Eq. (4.23) with respect to $y_j(n)$ yields

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \tag{4.24}$$

By using Eqs. (4.22) and (4.24) in Eq. (4.20), we get the desired partial derivative

$$\begin{aligned} \frac{\partial \mathscr{E}(n)}{\partial y_j(n)} &= -\sum_k e_k(n) \varphi_k'(v_k(n)) w_{kj}(n) \\ &= -\sum_k \delta_k(n) w_{kj}(n) \end{aligned} \tag{4.25}$$

where, in the second line, we have used the definition of the local gradient $\delta_k(n)$ given in Eq. (4.16), with the index k substituted for j.

Finally, using Eq. (4.25) in Eq. (4.17), we get the *back-propagation formula* for the local gradient $\delta_j(n)$, described by

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n), \qquad \text{neuron } j \text{ is hidden} \tag{4.26}$$

Figure 4.5 shows the signal-flow graph representation of Eq. (4.26), assuming that the output layer consists of m_L neurons.

The outside factor $\varphi_j'(v_j(n))$ involved in the computation of the local gradient $\delta_j(n)$ in Eq. (4.26) depends solely on the activation function associated with hidden neuron j. The remaining factor involved in this computation—namely, the summation over k—depends on two sets of terms. The first set of terms, the $\delta_k(n)$, requires knowledge of the error

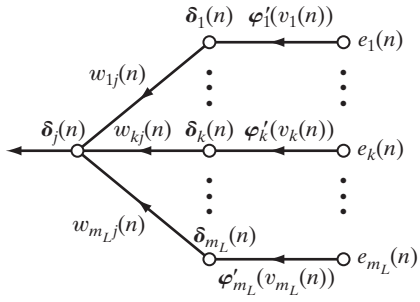


FIGURE 4.5 Signal-flow graph of a part of the adjoint system pertaining to back-propagation of error signals.

signals $e_k(n)$ for all neurons that lie in the layer to the immediate right of hidden neuron j and that are directly connected to neuron j; see Fig. 4.4. The second set of terms, the $w_{kj}(n)$, consists of the synaptic weights associated with these connections.

We now summarize the relations that we have derived for the back-propagation algorithm. First, the correction $\Delta w_{ji}(n)$ applied to the synaptic weight connecting neuron i to neuron j is defined by the delta rule:

$$\begin{pmatrix} \textit{Weight} \\ \textit{correction} \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} \textit{learning-} \\ \textit{rate parameter} \\ \eta \end{pmatrix} \times \begin{pmatrix} \textit{local} \\ \textit{gradient} \\ \delta_j(n) \end{pmatrix} \times \begin{pmatrix} \textit{input signal} \\ \textit{of neuron } j, \\ y_i(n) \end{pmatrix} \tag{4.27}$$

Second, the local gradient $\delta_j(n)$ depends on whether neuron j is an output node or a hidden node:

1. If neuron j is an output node, $\delta_j(n)$ equals the product of the derivative $\varphi_j'(v_j(n))$ and the error signal $e_j(n)$, both of which are associated with neuron j; see Eq. (4.16).
2. If neuron j is a hidden node, $\delta_j(n)$ equals the product of the associated derivative $\varphi_j'(v_j(n))$ and the weighted sum of the δs computed for the neurons in the next hidden or output layer that are connected to neuron j; see Eq. (4.26).

The Two Passes of Computation

In the application of the back-propagation algorithm, two different passes of computation are distinguished. The first pass is referred to as the forward pass, and the second is referred to as the backward pass.

In the *forward pass*, the synaptic weights remain unaltered throughout the network, and the function signals of the network are computed on a neuron-by-neuron basis. The function signal appearing at the output of neuron j is computed as

$$y_j(n) = \varphi(v_j(n)) \tag{4.28}$$

where $v_j(n)$ is the induced local field of neuron j, defined by

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{4.29}$$

where m is the total number of inputs (excluding the bias) applied to neuron j; $w_{ji}(n)$ is the synaptic weight connecting neuron i to neuron j; and $y_i(n)$ is an input signal of neuron j, or, equivalently, the function signal appearing at the output of neuron i. If neuron j is in the first hidden layer of the network, then $m = m_0$ and the index i refers to the ith input terminal of the network, for which we write

$$y_i(n) = x_i(n) \tag{4.30}$$

where $x_i(n)$ is the ith element of the input vector (pattern). If, on the other hand, neuron j is in the output layer of the network, then $m = m_L$ and the index j refers to the jth output terminal of the network, for which we write

$$y_j(n) = o_j(n) \tag{4.31}$$

where $o_j(n)$ is the jth element of the output vector of the multilayer perceptron. This output is compared with the desired response $d_j(n)$, obtaining the error signal $e_j(n)$ for the jth output neuron. Thus, the forward phase of computation begins at the first hidden layer by presenting it with the input vector and terminates at the output layer by computing the error signal for each neuron of this layer.

The backward pass, on the other hand, starts at the output layer by passing the error signals leftward through the network, layer by layer, and recursively computing the δ (i.e., the local gradient) for each neuron. This recursive process permits the synaptic weights of the network to undergo changes in accordance with the delta rule of Eq. (4.27). For a neuron located in the output layer, the δ is simply equal to the error signal of that neuron multiplied by the first derivative of its nonlinearity. Hence, we use Eq. (4.27) to compute the changes to the weights of all the connections feeding into the output layer. Given the δs for the neurons of the output layer, we next use Eq. (4.26) to compute the δs for all the neurons in the penultimate layer and therefore the changes to the weights of all connections feeding into it. The recursive computation is continued, layer by layer, by propagating the changes to all synaptic weights in the network.

Note that for the presentation of each training example, the input pattern is fixed—that is, "clamped" throughout the round-trip process, which encompasses the forward pass followed by the backward pass.

Activation Function

The computation of the δ for each neuron of the multilayer perceptron requires knowledge of the derivative of the activation function $\varphi(\cdot)$ associated with that neuron. For this derivative to exist, we require the function $\varphi(\cdot)$ to be continuous. In basic terms, *differentiability* is the only requirement that an activation function has to satisfy. An example of a continuously differentiable nonlinear activation function commonly used in multilayer perceptrons is *sigmoidal nonlinearity*,[1] two forms of which are described here:

1. *Logistic Function.* This form of sigmoidal nonlinearity, in its general form, is defined by

$$\varphi_j(v_j(n)) = \frac{1}{1 + \exp(-av_j(n))}, \qquad a > 0 \tag{4.32}$$

where $v_j(n)$ is the induced local field of neuron j and a is an adjustable positive parameter. According to this nonlinearity, the amplitude of the output lies inside the range $0 \leq y_j \leq 1$. Differentiating Eq. (4.32) with respect to $v_j(n)$, we get

$$\varphi_j'(v_j(n)) = \frac{a \exp(-av_j(n))}{[1 + \exp(-av_j(n))]^2} \tag{4.33}$$

With $y_j(n) = \varphi_j(v_j(n))$, we may eliminate the exponential term $\exp(-av_j(n))$ from Eq. (4.33) and consequently express the derivative $\varphi_j'(v_j(n))$ as

$$\varphi_j'(v_j(n)) = ay_j(n)[1 - y_j(n)] \tag{4.34}$$

For a neuron j located in the output layer, $y_j(n) = o_j(n)$. Hence, we may express the local gradient for neuron j as

$$\begin{aligned} \delta_j(n) &= e_j(n)\varphi_j'(v_j(n)) \\ &= a[d_j(n) - o_j(n)]o_j(n)[1 - o_j(n)], \qquad \text{neuron } j \text{ is an output node} \end{aligned} \tag{4.35}$$

where $o_j(n)$ is the function signal at the output of neuron j, and $d_j(n)$ is the desired response for it. On the other hand, for an arbitrary hidden neuron j, we may express the local gradient as

$$\begin{aligned} \delta_j(n) &= \varphi_j'(v_j(n)) \sum_k \delta_k(n)w_{kj}(n) \\ &= ay_j(n)[1 - y_j(n)] \sum_k \delta_k(n)w_{kj}(n), \qquad \text{neuron } j \text{ is hidden} \end{aligned} \tag{4.36}$$

Note from Eq. (4.34) that the derivative $\varphi_j'(v_j(n))$ attains its maximum value at $y_j(n) = 0.5$ and its minimum value (zero) at $y_j(n) = 0$, or $y_j(n) = 1.0$. Since the amount of change in a synaptic weight of the network is proportional to the derivative $\varphi_j'(v_j(n))$, it follows that for a sigmoid activation function, the synaptic weights are changed the most for those neurons in the network where the function signals are in their midrange. According to Rumelhart et al. (1986a), it is this feature of back-propagation learning that contributes to its stability as a learning algorithm.

2. *Hyperbolic tangent function.* Another commonly used form of sigmoidal nonlinearity is the hyperbolic tangent function, which, in its most general form, is defined by

$$\varphi_j(v_j(n)) = a \tanh(bv_j(n)) \tag{4.37}$$

where a and b are positive constants. In reality, the hyperbolic tangent function is just the logistic function rescaled and biased. Its derivative with respect to $v_j(n)$ is given by

$$\begin{aligned} \varphi_j'(v_j(n)) &= ab \operatorname{sech}^2(bv_j(n)) \\ &= ab(1 - \tanh^2(bv_j(n))) \\ &= \frac{b}{a}[a - y_j(n)][a + y_j(n)] \end{aligned} \tag{4.38}$$

For a neuron j located in the output layer, the local gradient is

$$\begin{aligned}\delta_j(n) &= e_j(n)\varphi_j'(v_j(n)) \\ &= \frac{b}{a}[d_j(n) - o_j(n)][a - o_j(n)][a + o_j(n)]\end{aligned} \tag{4.39}$$

For a neuron j in a hidden layer, we have

$$\begin{aligned}\delta_j(n) &= \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \\ &= \frac{b}{a}[a - y_j(n)][a + y_j(n)] \sum_k \delta_k(n) w_{kj}(n), \qquad \text{neuron } j \text{ is hidden}\end{aligned} \tag{4.40}$$

By using Eqs. (4.35) and (4.36) for the logistic function and Eqs. (4.39) and (4.40) for the hyperbolic tangent function, we may calculate the local gradient δ_j without requiring explicit knowledge of the activation function.

Rate of Learning

The back-propagation algorithm provides an "approximation" to the trajectory in weight space computed by the method of steepest descent. The smaller we make the learning-rate parameter η, the smaller the changes to the synaptic weights in the network will be from one iteration to the next, and the smoother will be the trajectory in weight space. This improvement, however, is attained at the cost of a slower rate of learning. If, on the other hand, we make the learning-rate parameter η too large in order to speed up the rate of learning, the resulting large changes in the synaptic weights assume such a form that the network may become unstable (i.e., oscillatory). A simple method of increasing the rate of learning while avoiding the danger of instability is to modify the delta rule of Eq. (4.15) by including a momentum term, as shown by

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n) \tag{4.41}$$

where α is usually a positive number called the *momentum constant*. It controls the feedback loop acting around $\Delta w_{ji}(n)$, as illustrated in Fig. 4.6, where z^{-1} is the unit-time delay operator. Equation (4.41) is called the *generalized delta rule*[2]; it includes the delta rule of Eq. (4.15) as a special case (i.e., $\alpha = 0$).

In order to see the effect of the sequence of pattern presentations on the synaptic weights due to the momentum constant α, we rewrite Eq. (4.41) as a time series with index t. The index t goes from the initial time 0 to the current time n. Equation (4.41)

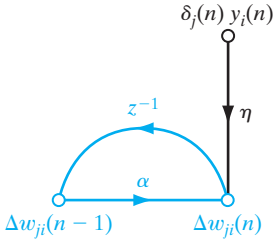


FIGURE 4.6 Signal-flow graph illustrating the effect of momentum constant α, which lies inside the feedback loop.

may be viewed as a first-order difference equation in the weight correction $\Delta w_{ji}(n)$. Solving this equation for $\Delta w_{ji}(n)$, we have

$$\Delta w_{ji}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t) \tag{4.42}$$

which represents a time series of length $n + 1$. From Eqs. (4.13) and (4.16), we note that the product $\delta_j(n)y_i(n)$ is equal to $-\partial \mathscr{E}(n)/\partial w_{ji}(n)$. Accordingly, we may rewrite Eq. (4.42) in the equivalent form

$$\Delta w_{ji}(n) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial \mathscr{E}(t)}{\partial w_{ji}(t)} \tag{4.43}$$

Based on this relation, we may make the following insightful observations:

1. The current adjustment $\Delta w_{ji}(n)$ represents the sum of an exponentially weighted time series. For the time series to be *convergent*, the momentum constant must be restricted to the range $0 \leq |\alpha| < 1$. When α is zero, the back-propagation algorithm operates without momentum. Also, the momentum constant α can be positive or negative, although it is unlikely that a negative α would be used in practice.

2. When the partial derivative $\partial \mathscr{E}(t)/\partial w_{ji}(t)$ has the same algebraic sign on consecutive iterations, the exponentially weighted sum $\Delta w_{ji}(n)$ grows in magnitude, and consequently the weight $w_{ji}(n)$ is adjusted by a large amount. The inclusion of momentum in the back-propagation algorithm tends to *accelerate descent* in steady downhill directions.

3. When the partial derivative $\partial \mathscr{E}(t)/\partial w_{ji}(t)$ has opposite signs on consecutive iterations, the exponentially weighted sum $\Delta w_{ji}(n)$ shrinks in magnitude, and consequently the weight $w_{ji}(n)$ is adjusted by a small amount. The inclusion of momentum in the back-propagation algorithm has a *stabilizing effect* in directions that oscillate in sign.

The incorporation of momentum in the back-propagation algorithm represents a minor modification to the weight update; however, it may have some beneficial effects on the learning behavior of the algorithm. The momentum term may also have the benefit of preventing the learning process from terminating in a shallow local minimum on the error surface.

In deriving the back-propagation algorithm, it was assumed that the learning-rate parameter is a constant denoted by η. In reality, however, it should be defined as η_{ji}; that is, the learning-rate parameter should be *connection dependent.* Indeed, many interesting things can be done by making the learning-rate parameter different for different parts of the network. We provide more detail on this issue in subsequent sections.

It is also noteworthy that in the application of the back-propagation algorithm, we may choose all the synaptic weights in the network to be adjustable, or we may constrain any number of weights in the network to remain fixed during the adaptation process. In the latter case, the error signals are back propagated through the network in the usual manner; however, the fixed synaptic weights are left unaltered. This can be done simply by making the learning-rate parameter η_{ji} for synaptic weight w_{ji} equal to zero.

Stopping Criteria

In general, the back-propagation algorithm cannot be shown to converge, and there are no well-defined criteria for stopping its operation. Rather, there are some reasonable criteria, each with its own practical merit, that may be used to terminate the weight adjustments. To formulate such a criterion, it is logical to think in terms of the unique properties of a *local* or *global minimum* of the error surface.[3] Let the weight vector $\mathbf{w}^*$ denote a minimum, be it local or global. A necessary condition for $\mathbf{w}^*$ to be a minimum is that the gradient vector $\mathbf{g}(\mathbf{w})$ (i.e., first-order partial derivative) of the error surface with respect to the weight vector $\mathbf{w}$ must be zero at $\mathbf{w} = \mathbf{w}^*$. Accordingly, we may formulate a sensible convergence criterion for back-propagation learning as follows (Kramer and Sangiovanni-Vincentelli, 1989):

> *The back-propagation algorithm is considered to have converged when the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold.*

The drawback of this convergence criterion is that, for successful trials, learning times may be long. Also, it requires the computation of the gradient vector $\mathbf{g}(\mathbf{w})$.

Another unique property of a minimum that we can use is the fact that the cost function $\mathscr{E}_{\text{av}}(\mathbf{w})$ is stationary at the point $\mathbf{w} = \mathbf{w}^*$. We may therefore suggest a different criterion of convergence:

> *The back-propagation algorithm is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small.*

The rate of change in the average squared error is typically considered to be small enough if it lies in the range of 0.1 to 1 percent per epoch. Sometimes a value as small as 0.01 percent per epoch is used. Unfortunately, this criterion may result in a premature termination of the learning process.

There is another useful, and theoretically supported, criterion for convergence: After each learning iteration, the network is tested for its generalization performance. The learning process is stopped when the generalization performance is adequate or when it is apparent that the generalization performance has peaked; see Section 4.13 for more details.

Summary of the Back-Propagation Algorithm

Figure 4.1 presents the architectural layout of a multilayer perceptron. The corresponding signal-flow graph for back-propagation learning, incorporating both the forward and backward phases of the computations involved in the learning process, is presented in Fig. 4.7 for the case of $L = 2$ and $m_0 = m_1 = m_2 = 3$. The top part of the signal-flow graph accounts for the forward pass. The lower part of the signal-flow graph accounts for the backward pass, which is referred to as a *sensitivity graph* for computing the local gradients in the back-propagation algorithm (Narendra and Parthasarathy, 1990).

Earlier, we mentioned that the sequential updating of weights is the preferred method for on-line implementation of the back-propagation algorithm. For this mode

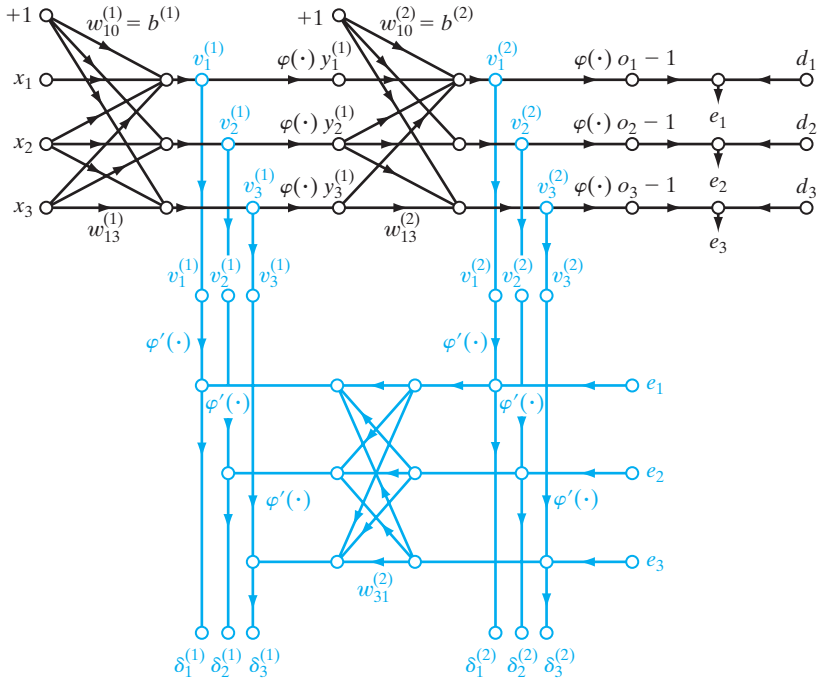


FIGURE 4.7 Signal-flow graphical summary of back-propagation learning. Top part of the graph: forward pass. Bottom part of the graph: backward pass.

of operation, the algorithm cycles through the training sample $\{(\mathbf{x}(n), \mathbf{d}(n))\}_{n=1}^{N}$ as follows:

1. *Initialization.* Assuming that no prior information is available, pick the synaptic weights and thresholds from a uniform distribution whose mean is zero and whose variance is chosen to make the standard deviation of the induced local fields of the neurons lie at the transition between the linear and standards parts of the sigmoid activation function.

2. *Presentations of Training Examples.* Present the network an epoch of training examples. For each example in the sample, ordered in some fashion, perform the sequence of forward and backward computations described under points 3 and 4, respectively.

3. *Forward Computation.* Let a training example in the epoch be denoted by $(\mathbf{x}(n), \mathbf{d}(n))$, with the input vector $\mathbf{x}(n)$ applied to the input layer of sensory nodes and the desired response vector $\mathbf{d}(n)$ presented to the output layer of computation nodes. Compute the induced local fields and function signals of the network by proceeding forward through the network, layer by layer. The induced local field $v_j^{(l)}(n)$ for neuron j in layer l is

$$v_j^{(l)}(n) = \sum_i w_{ji}^{(l)}(n) y_i^{(l-1)}(n) \tag{4.44}$$

where $y_i^{(l-1)}(n)$ is the output (function) signal of neuron i in the previous layer $l-1$ at iteration n, and $w_{ji}^{(l)}(n)$ is the synaptic weight of neuron j in layer l that is fed from neuron i in layer $l-1$. For $i = 0$, we have $y_0^{(l-1)}(n) = +1$, and $w_{j0}^{(l)}(n) = b_j^{(l)}(n)$ is the bias applied to neuron j in layer l. Assuming the use of a sigmoid function, the output signal of neuron j in layer l is

$$y_j^{(l)} = \varphi_j(v_j(n))$$

If neuron j is in the first hidden layer (i.e., $l = 1$), set

$$y_j^{(0)}(n) = x_j(n)$$

where $x_j(n)$ is the jth element of the input vector $\mathbf{x}(n)$. If neuron j is in the output layer (i.e., $l = L$, where L is referred to as the *depth* of the network), set

$$y_j^{(L)} = o_j(n)$$

Compute the error signal

$$e_j(n) = d_j(n) - o_j(n) \tag{4.45}$$

where $d_j(n)$ is the jth element of the desired response vector $\mathbf{d}(n)$.

4. *Backward Computation.* Compute the δs (i.e., local gradients) of the network, defined by

$$\delta_j^{(l)}(n) = \begin{cases} e_j^{(L)}(n)\varphi_j'(v_j^{(L)}(n)) & \text{for neuron } j \text{ in output layer } L \\ \varphi_j'(v_j^{(l)}(n)) \sum_k \delta_k^{(l+1)}(n) w^{(l+1)}_{kj}(n) & \text{for neuron } j \text{ in hidden layer } l \end{cases} \tag{4.46}$$

where the prime in $\varphi_j'(\cdot)$ denotes differentiation with respect to the argument. Adjust the synaptic weights of the network in layer l according to the generalized delta rule

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha[\Delta w_{ji}^{(l)}(n-1)] + \eta\delta_j^{(l)}(n) y_i^{(l-1)}(n) \tag{4.47}$$

where η is the learning-rate parameter and α is the momentum constant.

5. *Iteration.* Iterate the forward and backward computations under points 3 and 4 by presenting new epochs of training examples to the network until the chosen stopping criterion is met.

Notes: The order of presentation of training examples should be randomized from epoch to epoch. The momentum and learning-rate parameter are typically adjusted (and usually decreased) as the number of training iterations increases. Justification for these points will be presented later.

4.5 XOR PROBLEM

In Rosenblatt's single-layer perceptron, there are no hidden neurons. Consequently, it cannot classify input patterns that are not linearly separable. However, nonlinearly separable patterns commonly occur. For example, this situation arises in the *exclusive-OR (XOR) problem*, which may be viewed as a special case of a more general problem, namely, that of classifying points in the *unit hypercube*. Each point in the hypercube is in either class 0 or class 1. However, in the special case of the XOR problem, we need

consider only the four corners of a *unit square* that correspond to the input patterns (0,0), (0,1), (1,1), and (1,0), where a single bit (i.e., binary digit) changes as we move from one corner to the next. The first and third input patterns are in class 0, as shown by

$$0 \oplus 0 = 0$$

and

$$1 \oplus 1 = 0$$

where $\oplus$ denotes the exclusive-OR Boolean function operator. The input patterns (0,0) and (1,1) are at opposite corners of the unit square, yet they produce the identical output 0. On the other hand, the input patterns (0,1) and (1,0) are also at opposite corners of the square, but they are in class 1, as shown by

$$0 \oplus 1 = 1$$

and

$$1 \oplus 0 = 1$$

We first recognize that the use of a single neuron with two inputs results in a straight line for a decision boundary in the input space. For all points on one side of this line, the neuron outputs 1; for all points on the other side of the line, it outputs 0. The position and orientation of the line in the input space are determined by the synaptic weights of the neuron connected to the input nodes and the bias applied to the neuron. With the input patterns (0,0) and (1,1) located on opposite corners of the unit square, and likewise for the other two input patterns (0,1) and (1,0), it is clear that we cannot construct a straight line for a decision boundary so that (0,0) and (0,1) lie in one decision region and (0,1) and (1,0) lie in the other decision region. In other words, the single-layer perceptron cannot solve the XOR problem.

However, we may solve the XOR problem by using a single hidden layer with two neurons, as in Fig. 4.8a (Touretzky and Pomerleau, 1989). The signal-flow graph of the network is shown in Fig. 4.8b. The following assumptions are made here:

- Each neuron is represented by a McCulloch–Pitts model, which uses a threshold function for its activation function.
- Bits 0 and 1 are represented by the levels 0 and +1, respectively.

The top neuron, labeled as "Neuron 1" in the hidden layer, is characterized as

$$w_{11} = w_{12} = +1$$
$$b_1 = -\frac{3}{2}$$

The slope of the decision boundary constructed by this hidden neuron is equal to −1 and positioned as in Fig. 4.9a. The bottom neuron, labeled as "Neuron 2" in the hidden layer, is characterized as

$$w_{21} = w_{22} = +1$$
$$b_2 = -\frac{1}{2}$$

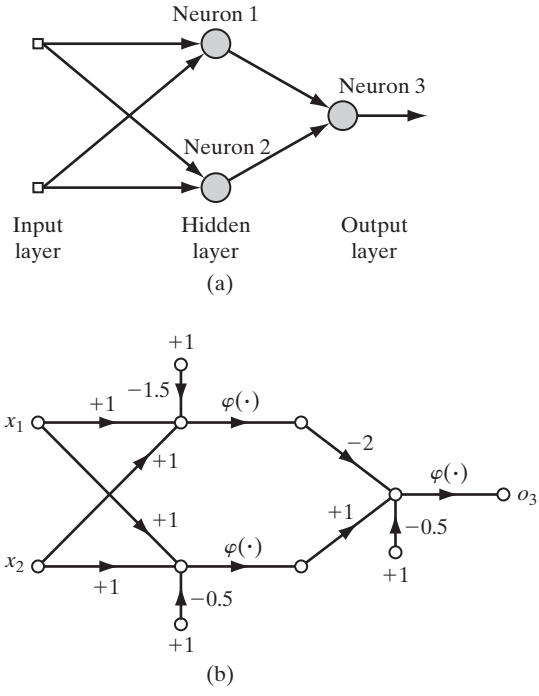


FIGURE 4.8 (a) Architectural graph of network for solving the XOR problem. (b) Signal-flow graph of the network.

The orientation and position of the decision boundary constructed by this second hidden neuron are as shown in Fig. 4.9b.

The output neuron, labeled as "Neuron 3" in Fig. 4.8a, is characterized as

$$w_{31} = -2$$
$$w_{32} = +1$$
$$b_3 = -\frac{1}{2}$$

The function of the output neuron is to construct a linear combination of the decision boundaries formed by the two hidden neurons. The result of this computation is shown in Fig. 4.9c. The bottom hidden neuron has an excitatory (positive) connection to the output neuron, whereas the top hidden neuron has an inhibitory (negative) connection to the output neuron. When both hidden neurons are off, which occurs when the input pattern is (0,0), the output neuron remains off. When both hidden neurons are on, which occurs when the input pattern is (1,1), the output neuron is switched off again because the inhibitory effect of the larger negative weight connected to the top hidden neuron overpowers the excitatory effect of the positive weight connected to the bottom hidden neuron. When the top hidden neuron is off and the bottom hidden neuron is on, which occurs when the input pattern is (0,1) or (1,0), the output neuron is switched on because of the excitatory effect of the positive weight connected to the bottom hidden neuron. Thus, the network of Fig. 4.8a does indeed solve the XOR problem.

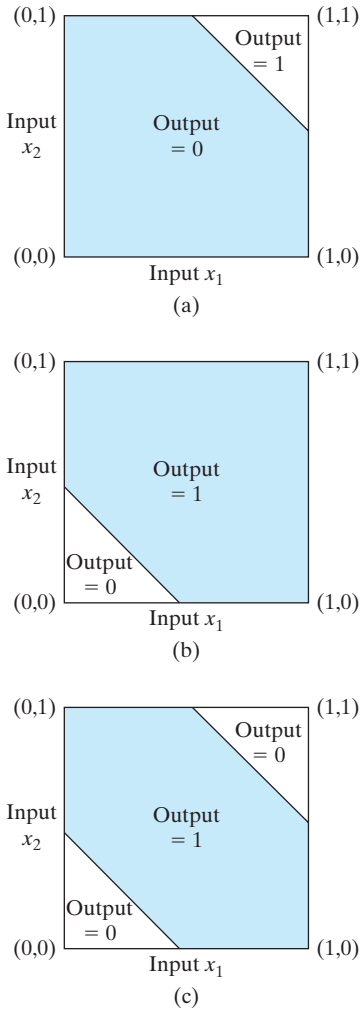


FIGURE 4.9 (a) Decision boundary constructed by hidden neuron 1 of the network in Fig. 4.8. (b) Decision boundary constructed by hidden neuron 2 of the network. (c) Decision boundaries constructed by the complete network.

4.6 HEURISTICS FOR MAKING THE BACK-PROPAGATION ALGORITHM PERFORM BETTER

It is often said that the design of a neural network using the back-propagation algorithm is more of an art than a science, in the sense that many of the factors involved in the design are the results of one's own personal experience. There is some truth in this statement. Nevertheless, there are methods that will significantly improve the back-propagation algorithm's performance, as described here:

1. *Stochastic versus batch update.* As mentioned previously, the stochastic (sequential) mode of back-propagation learning (involving pattern-by-pattern updating) is computationally faster than the batch mode. This is especially true when the

training data sample is large and highly redundant. (Highly redundant data pose computational problems for the estimation of the Jacobian required for the batch update.)

2. *Maximizing information content.* As a general rule, every training example presented to the back-propagation algorithm should be chosen on the basis that its information content is the largest possible for the task at hand (LeCun, 1993). Two ways of realizing this choice are as follows:

- Use an example that results in the largest training error.
- Use an example that is radically different from all those previously used.

These two heuristics are motivated by a desire to search more of the weight space.

In pattern-classification tasks using sequential back-propagation learning, a simple and commonly used technique is to randomize (i.e., shuffle) the order in which the examples are presented to the multilayer perceptron from one epoch to the next. Ideally, the randomization ensure that successive examples in an epoch presented to the network rarely belong to the same class.

3. *Activation function.* Insofar as the speed of learning is concerned, the preferred choice is to use a sigmoid activation function that is an *odd function of its argument*, as shown by

$$\varphi(-v) = -\varphi(v)$$

This condition is satisfied by the hyperbolic function

$$\varphi(v) = a \tanh(bv)$$

as shown in Fig. 4.10, but not the logistic function. Suitable values for the constraints a and b in the formula for $\varphi(v)$ are as follows (LeCun, 1993):

$$a = 1.7159$$

and

$$b = \frac{2}{3}$$

The hyperbolic tangent function $\varphi(v)$ of Fig. 4.10 has the following useful properties:

- $\varphi(1) = 1$ and $\varphi(-1) = -1$.
- At the origin, the slope (i.e., effective gain) of the activation function is close to unity, as shown by

$$\begin{aligned}\varphi(0) &= ab \\ &= 1.7159\left(\frac{2}{3}\right) \\ &= 1.1424\end{aligned}$$

- The second derivative of $\varphi(v)$ attains its maximum value at $v = 1$.

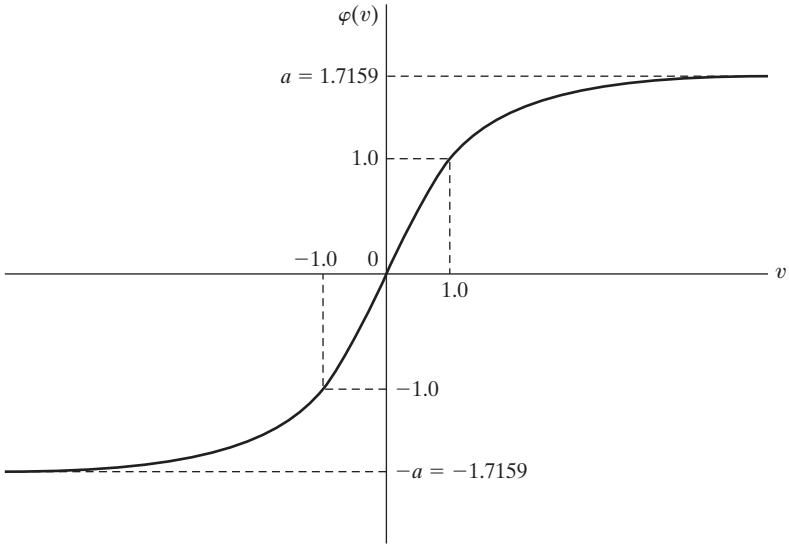


FIGURE 4.10 Graph of the hyperbolic tangent function $\varphi(v) = a\tanh(bv)$ for $a = 1.7159$ and $b = 2/3$. The recommended target values are $+1$ and -1.

4. *Target values.* It is important that the target values (desired response) be chosen within the range of the sigmoid activation function. More specifically, the desired response d_j for neuron j in the output layer of the multilayer perceptron should be *offset* by some amount ε away from the limiting value of the sigmoid activation function, depending on whether the limiting value is positive or negative. Otherwise, the back-propagation algorithm tends to drive the free parameters of the network to infinity and thereby slow down the learning process by driving the hidden neurons into saturation. To be specific, consider the hyperbolic tangent function of Fig. 4.10. For the limiting value $+a$, we set

$$d_j = a - \varepsilon$$

and for the limiting value of $-a$, we set

$$d_j = -a + \varepsilon$$

where ε is an appropriate positive constant. For the choice of $a = 1.7159$ used in Fig. 4.10, we may set $\varepsilon = 0.7159$, in which case the target value (desired response) d_j can be conveniently chosen as ± 1, as indicated in the figure.

5. *Normalizing the inputs.* Each input variable should be *preprocessed* so that its mean value, averaged over the entire training sample, is close to zero, or else it will be small compared to its standard deviation (LeCun, 1993). To appreciate the practical significance of this rule, consider the extreme case where the input variables are consistently positive. In this situation, the synaptic weights of a neuron in the first hidden layer can only increase together or decrease together. Accordingly, if the weight vector of that

neuron is to change direction, it can do so only by zigzagging its way through the error surface, which is typically slow and should therefore be avoided.

In order to accelerate the back-propagation learning process, the normalization of the inputs should also include two other measures (LeCun, 1993):

- The input variables contained in the training set should be *uncorrelated*; this can be done by using principal-components analysis, to be discussed in Chapter 8.
- The decorrelated input variables should be scaled so that their *covariances are approximately equal*, thereby ensuring that the different synaptic weights in the network learn at approximately the same speed.

Figure 4.11 illustrates the results of three normalization steps: mean removal, decorrelation, and covariance equalization, applied in that order.

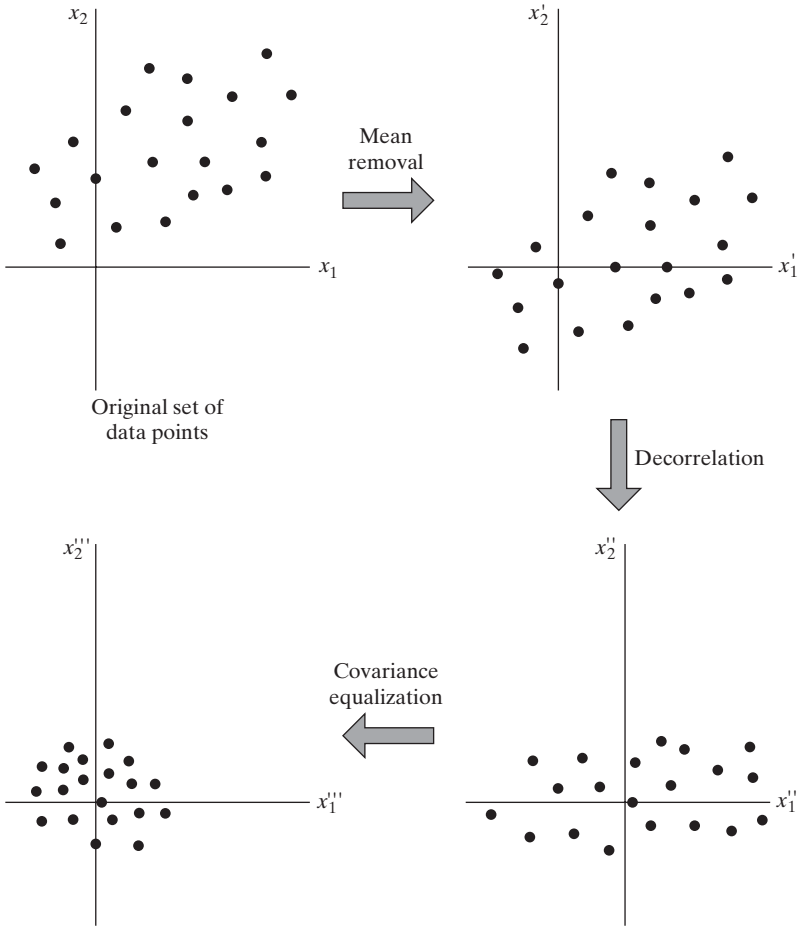


FIGURE 4.11 Illustrating the operation of mean removal, decorrelation, and covariance equalization for a two-dimensional input space.

It is also of interest to note that when the inputs are transformed in the manner illustrated in Fig. 4.11 and used in conjunction with the hyperbolic tangent function specified in Fig. 4.10, the variance of the individual neural outputs in the multilayer perceptron will be close to unity (Orr and Müller, 1998). The rationale for this statement is that the effective gain of the sigmoid function over its useful range is roughly unity.

6. *Initialization.* A good choice for the initial values of the synaptic weights and thresholds of the network can be of tremendous help in a successful network design. The key question is: What is a good choice?

When the synaptic weights are assigned large initial values, it is highly likely that the neurons in the network will be driven into saturation. If this happens, the local gradients in the back-propagation algorithm assume small values, which in turn will cause the learning process to slow down. However, if the synaptic weights are assigned small initial values, the back-propagation algorithm may operate on a very flat area around the origin of the error surface; this is particularly true in the case of sigmoid functions such as the hyperbolic tangent function. Unfortunately, the origin is a *saddle point*, which refers to a stationary point where the curvature of the error surface across the saddle is negative and the curvature along the saddle is positive. For these reasons, the use of both large and small values for initializing the synaptic weights should be avoided. The proper choice of initialization lies somewhere between these two extreme cases.

To be specific, consider a multilayer perceptron using the hyperbolic tangent function for its activation functions. Let the bias applied to each neuron in the network be set to zero. We may then express the induced local field of neuron j as

$$v_j = \sum_{i=1}^{m} w_{ji} y_i$$

Let it be assumed that the inputs applied to each neuron in the network have zero mean and unit variance, as shown by

$$\mu_y = \mathbb{E}[y_i] = 0 \qquad \text{for all } i$$

and

$$\sigma_y^2 = \mathbb{E}[(y_i - \mu_i)^2] = \mathbb{E}[y_i^2] = 1 \qquad \text{for all } i$$

Let it be further assumed that the inputs are uncorrelated, as shown by

$$\mathbb{E}[y_i y_k] = \begin{cases} 1 & \text{for } k = i \\ 0 & \text{for } k \neq i \end{cases}$$

and that the synaptic weights are drawn from a uniformly distributed set of numbers with zero mean, that is,

$$\mu_w = \mathbb{E}[w_{ji}] = 0 \qquad \text{for all } (j, i) \text{ pairs}$$

and variance

$$\sigma_w^2 = \mathbb{E}[(w_{ji} - \mu_w)^2] = \mathbb{E}[w_{ji}^2] \qquad \text{for all } (j, i) \text{ pairs}$$

Accordingly, we may express the mean and variance of the induced local field v_j as

$$\mu_v = \mathbb{E}[v_j] = \mathbb{E}\left[\sum_{i=1}^{m} w_{ji} y_i\right] = \sum_{i=1}^{m} \mathbb{E}[w_{ji}]\mathbb{E}[y_i] = 0$$

and

$$\begin{aligned}
\sigma_v^2 &= \mathbb{E}[(v_j - \mu_v)^2] = \mathbb{E}[v_j^2] \\
&= \mathbb{E}\left[\sum_{i=1}^{m}\sum_{k=1}^{m} w_{ji} w_{jk} y_i y_k\right] \\
&= \sum_{i=1}^{m}\sum_{k=1}^{m} \mathbb{E}[w_{ji} w_{jk}]\mathbb{E}[y_i y_k] \\
&= \sum_{i=1}^{m} \mathbb{E}[w_{ji}^2] \\
&= m\sigma_w^2
\end{aligned}$$

where m is the number of synaptic connections of a neuron.

In light of this result, we may now describe a good strategy for initializing the synaptic weights so that the standard deviation of the induced local field of a neuron lies in the transition area between the linear and saturated parts of its sigmoid activation function. For example, for the case of a hyperbolic tangent function with parameters a and b used in Fig. 4.10, this objective is satisfied by setting $\sigma_v = 1$ in the previous equation, in which case we obtain the following (LeCun, 1993):

$$\sigma_w = m^{-1/2} \tag{4.48}$$

Thus, it is desirable for the uniform distribution, from which the synaptic weights are selected, to have a mean of zero and a variance equal to the reciprocal of the number of synaptic connections of a neuron.

7. *Learning from hints.* Learning from a sample of training examples deals with an unknown input–output mapping function $f(\cdot)$. In effect, the learning process exploits the information contained in the examples about the function $f(\cdot)$ to *infer* an approximate implementation of it. The process of learning from examples may be generalized to *include learning from hints*, which is achieved by allowing prior information that we may have about the function $f(\cdot)$ to be included in the learning process (Abu-Mostafa, 1995). Such information may include invariance properties, symmetries, or any other knowledge about the function $f(\cdot)$ that may be used to accelerate the search for its approximate realization and, more importantly, to improve the quality of the final estimate. The use of Eq. (4.48) is an example of how this is achieved.

8. *Learning rates.* All neurons in the multilayer perceptron should ideally learn at the same rate. The last layers usually have larger local gradients than the layers at the front end of the network. Hence, the learning-rate parameter η should be assigned a

smaller value in the last layers than in the front layers of the multilayer perceptron. Neurons with many inputs should have a smaller learning-rate parameter than neurons with few inputs so as to maintain a similar learning time for all neurons in the network. In LeCun (1993), it is suggested that for a given neuron, the learning rate should be inversely proportional to the square root of synaptic connections made to that neuron.

4.7 COMPUTER EXPERIMENT: PATTERN CLASSIFICATION

In this computer experiment, we resume the sequence of pattern-classification experiments performed first in Chapter 1 using Rosenblatt's perceptron and then in Chapter 2 using the method of least squares. For both experiments, we used training and test data generated by randomly sampling the *double-moon* structure pictured in Fig. 1.8. In each of the experiments, we considered two cases, one employing linearly separable patterns and the other employing nonlinearly separable patterns. The perceptron worked perfectly fine for the linearly separable setting of $d = 1$, but the method of least squares required a larger separation between the two moons for perfect classification. In any event, they both failed the nonlinearly separable setting of $d = -4$.

The objective of the computer experiment presented herein is twofold:

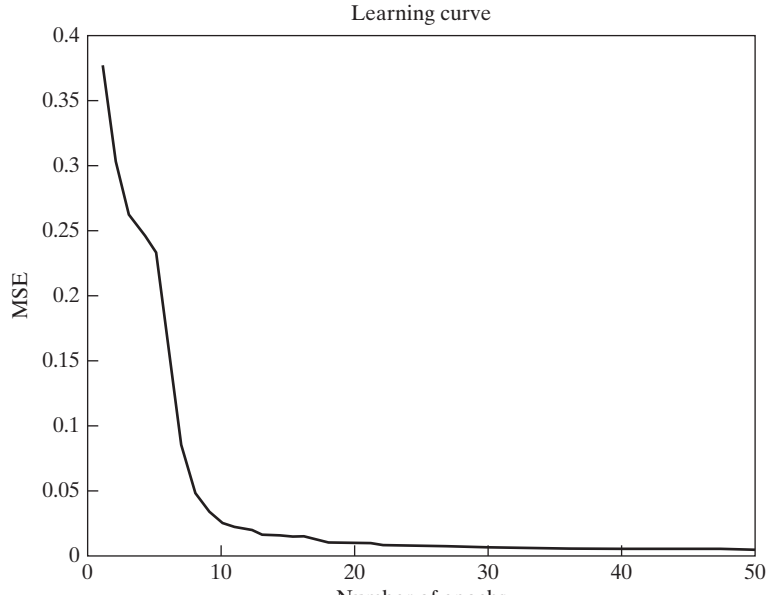
1. to demonstrate that the multilayer perceptron, trained with the back-propagation algorithm, is capable of classifying nonlinearly separable test data;
2. to find a more difficult case of nonlinearly separable patterns for which the multilayer perceptron fails the double-moon classification test.

The specifications of the multilayer perceptron used in the experiment are as follows:

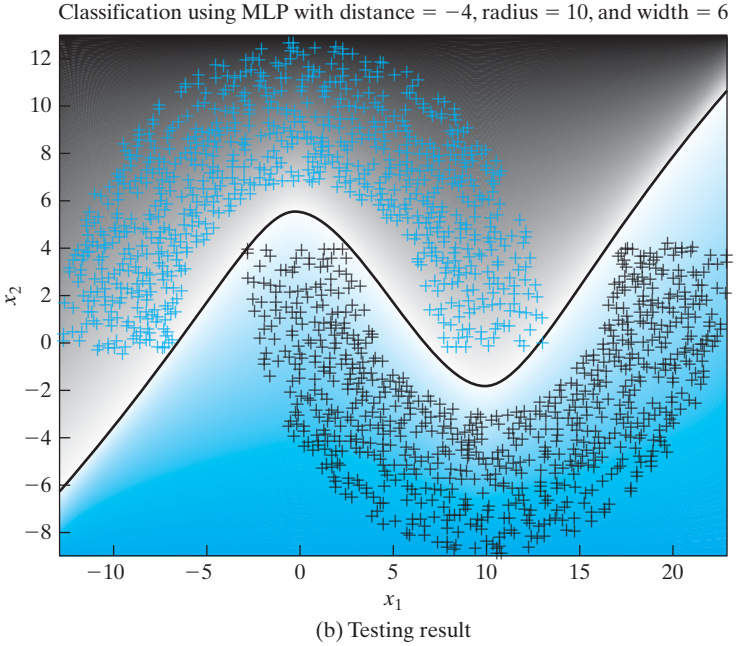
Size of the input layer: $m_0 = 2$
Size of the (only) hidden layer: $m_1 = 20$
Size of the output layer: $m_2 = 1$
Activation function: hyperbolic tangent function $\varphi(v) = \dfrac{1 - \exp(-2v)}{1 + \exp(-2v)}$
Threshold setting: zero
Learning-rate parameter η: annealed linearly from 10^{-1} down to 10^{-5}

The experiment is carried out in two parts, one corresponding to the vertical separation $d = -4$, and the other corresponding to $d = -5$:

(a) *Vertical separation $d = -4$.*
Figure 4.12 presents the results of the MLP experiment for the length of separation between the two moons of $d = -4$. Part (a) of the figure displays the learning curve resulting from the training session. We see that the learning curve reached convergence effectively in about 15 epochs of training. Part (b) of the figure displays the optimal nonlinear decision boundary computed by the MLP. Most important, perfect classification of the two patterns was achieved, with no classification errors. This perfect performance is attributed to the hidden layer of the MLP.

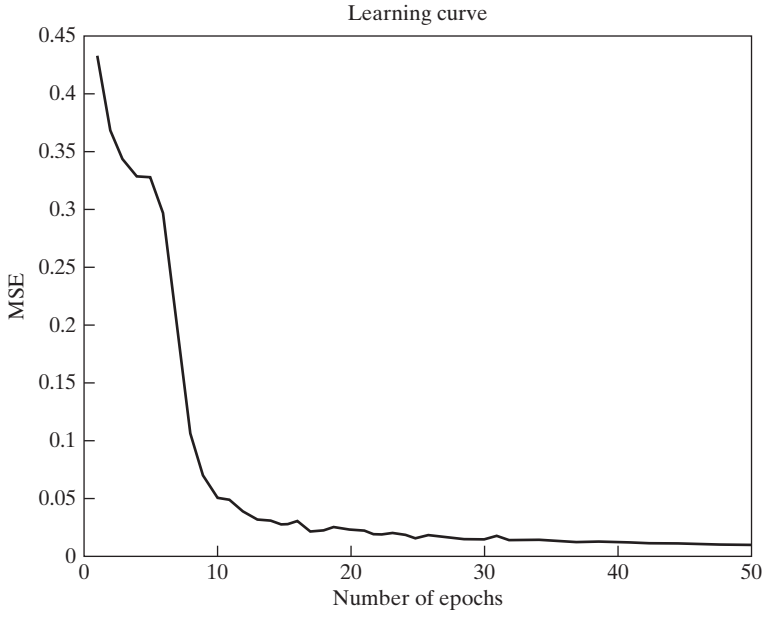


(a) Learning curve

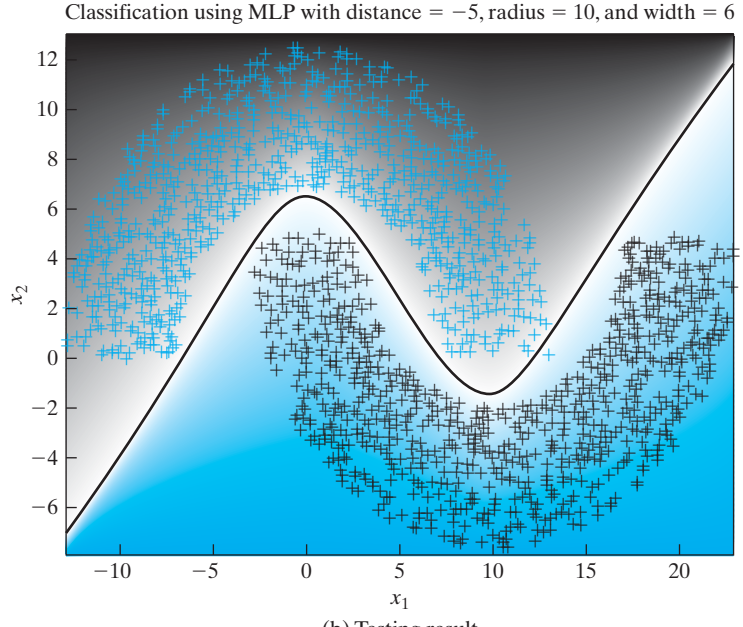


(b) Testing result

FIGURE 4.12 Results of the computer experiment on the back-propagation algorithm applied to the MLP with distance $d = -4$. MSE stands for mean-square error.



(a) Learning curve



(b) Testing result

FIGURE 4.13 Results of the computer experiment on the back-propagation algorithm applied to the MLP with distance $d = -5$.

(b) *Vertical separation $d = -5$.*
To challenge the multilayer perceptron with a more difficult pattern-classification task, we reduced the vertical separation between the two moons to $d = -5$. The results of this second part of the experiment are presented in Fig. 4.13. The learning curve of the back-propagation algorithm, plotted in part (a) of the figure, shows a slower rate of convergence, roughly three times that for the easier case of $d = -4$. Moreover, the testing results plotted in part (b) of the figure reveal three classification errors in a testing set of 2,000 data points, representing an error rate of 0.15 percent.

The decision boundary is computed by finding the coordinates x_1 and x_2 pertaining to the input vector $\mathbf{x}$, for which the response of the output neuron is zero on the premise that the two classes of the experiment are equally likely. Accordingly, when a threshold of zero is exceeded, a decision is made in favor of one class; otherwise, the decision is made in favor of the other class. This procedure is followed on all the double-moon classification experiments reported in the book.

4.8 BACK PROPAGATION AND DIFFERENTIATION

Back propagation is a specific technique for implementing *gradient descent* in weight space for a multilayer perceptron. The basic idea is to efficiently compute *partial derivatives* of an approximating function $F(\mathbf{w}, \mathbf{x})$ realized by the network with respect to all the elements of the adjustable weight vector $\mathbf{w}$ for a given value of input vector $\mathbf{x}$. Herein lies the computational power of the back-propagation algorithm.[4]

To be specific, consider a multilayer perceptron with an input layer of m_0 nodes, two hidden layers, and a single output neuron, as depicted in Fig. 4.14. The elements of the weight vector $\mathbf{w}$ are ordered by layer (starting from the first hidden layer), then by neurons in a layer, and then by the number of a synapse within a neuron. Let $w_{ji}^{(l)}$ denote the synaptic weight from neuron i to neuron j in layer $l = 1, 2, \ldots$. For $l = 1$, corresponding to the first hidden layer, the index i refers to a source node rather than to a

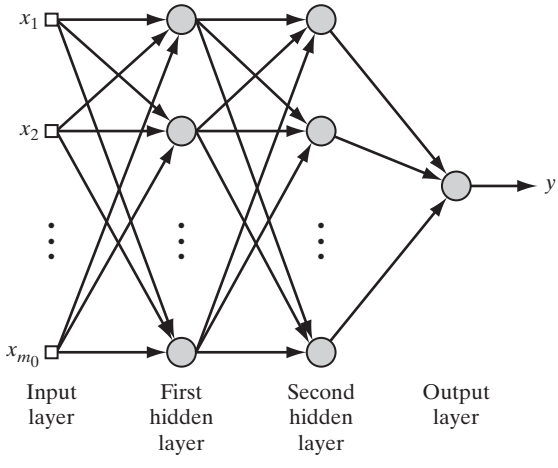


FIGURE 4.14 Multilayer perceptron with two hidden layers and one output neuron.

neuron. For $l = 3$, corresponding to the output layer in Fig. 4.14, we have $j = 1$. We wish to evaluate the derivatives of the function $F(\mathbf{w}, \mathbf{x})$ with respect to all the elements of the weight vector $\mathbf{w}$ for a specified input vector $\mathbf{x} = [x_1, x_2, ..., x_{m_0}]^T$. We have included the weight vector $\mathbf{w}$ as an argument of the function F in order to focus attention on it. For example, for $l = 2$ (i.e., a single hidden layer and a linear output layer), we have

$$F(\mathbf{w}, \mathbf{x}) = \sum_{j=0}^{m_1} w_{oj}\, \varphi\left(\sum_{i=0}^{m_0} w_{ji} x_i \right) \tag{4.49}$$

where $\mathbf{w}$ is the ordered weight vector and $\mathbf{x}$ is the input vector.

The multilayer perceptron of Fig. 4.14 is parameterized by an *architecture* $\mathscr{A}$ (representing a discrete parameter) and a *weight vector* $\mathbf{w}$ (made up of continuous elements). Let $\mathscr{A}_j^{(l)}$ denote that part of the architecture extending from the input layer ($l = 0$) to node j in layer $l = 1, 2, 3$. Accordingly, we may write

$$F(\mathbf{w}, \mathbf{x}) = \varphi(\mathscr{A}_1^{(3)}) \tag{4.50}$$

where φ is the activation function. However, $\mathscr{A}_1^{(3)}$ is to be interpreted merely as an architectural symbol rather than a variable. Thus, adapting Eqs. (4.2), (4.4), (4.13), and (4.25) for use in this new situation, we obtain the formulas

$$\frac{\partial F(\mathbf{w}, \mathbf{x})}{\partial w_{1k}^{(3)}} = \varphi'(\mathscr{A}_1^{(3)})\varphi(\mathscr{A}_k^{(2)}) \tag{4.51}$$

$$\frac{\partial F(\mathbf{w}, \mathbf{x})}{\partial w_{kj}^{(2)}} = \varphi'(\mathscr{A}_1^{(3)})\varphi'(\mathscr{A}_k^{(2)})\varphi(\mathscr{A}_j^{(1)})w_{1k}^{(3)} \tag{4.52}$$

$$\frac{\partial F(\mathbf{w}, \mathbf{x})}{\partial w_{ji}^{(1)}} = \varphi'(\mathscr{A}_1^{(3)})\varphi'(\mathscr{A}_j^{(1)})x_i \left[\sum_k w_{1k}^{(3)}\varphi'(\mathscr{A}_k^{(2)})w_{kj}^{(2)} \right] \tag{4.53}$$

where φ' is the partial derivative of the nonlinearity φ with respect to its argument and x_i is the ith element of the input vector $\mathbf{x}$. In a similar way, we may derive the equations for the partial derivatives of a general network with more hidden layers and more neurons in the output layer.

Equations (4.51) through (4.53) provide the basis for calculating the sensitivity of the network function $F(\mathbf{w}, \mathbf{x})$ with respect to variations in the elements of the weight vector $\mathbf{w}$. Let ω denote an element of the weight vector $\mathbf{w}$. The *sensitivity* of $F(\mathbf{w}, \mathbf{x})$ with respect to ω, is formally defined by

$$S_\omega^F = \frac{\partial F/F}{\partial \omega/\omega}$$

It is for this reason that we refer to the lower part of the signal-flow graph in Fig. 4.7 as a "sensitivity graph."

The Jacobian

Let W denote the total number of free parameters (i.e., synaptic weights and biases) of a multilayer perceptron, which are ordered in a manner described to form the weight vector $\mathbf{w}$. Let N denote the total number of examples used to train the network. Using back

propagation, we may compute a set of W partial derivatives of the approximating function $F[\mathbf{w}, \mathbf{x}(n)]$ with respect to the elements of the weight vector $\mathbf{w}$ for a specific example $\mathbf{x}(n)$ in the training sample. Repeating these computations for $n = 1, 2, \ldots, N$, we end up with an N-by-W matrix of partial derivatives. This matrix is called the *Jacobian* $\mathbf{J}$ of the multilayer perceptron evaluated at $\mathbf{x}(n)$. Each row of the Jacobian corresponds to a particular example in the training sample.

There is experimental evidence to suggest that many neural network training problems are intrinsically *ill conditioned*, leading to a Jacobian $\mathbf{J}$ that is almost rank deficient (Saarinen et al., 1991). The *rank* of a matrix is equal to the number of linearly independent columns or rows in the matrix, whichever one is smallest. The Jacobian $\mathbf{J}$ is said to be *rank deficient* if its rank is less than min (N, W). Any rank deficiency in the Jacobian causes the back-propagation algorithm to obtain only partial information of the possible search directions. Rank deficiency also causes training times to be long.

4.9 THE HESSIAN AND ITS ROLE IN ON-LINE LEARNING

The *Hessian matrix*, or simply the *Hessian*, of the cost function $\mathscr{E}_{\text{av}}(\mathbf{w})$, denoted by $\mathbf{H}$, is defined as the second derivative of $\mathscr{E}_{\text{av}}(\mathbf{w})$ with respect to the weight vector $\mathbf{w}$, as shown by

$$\mathbf{H} = \frac{\partial^2 \mathscr{E}_{\text{av}}(\mathbf{w})}{\partial \mathbf{w}^2} \tag{4.54}$$

The Hessian plays an important role in the study of neural networks; specifically, we mention the following points[5]:

1. The eigenvalues of the Hessian have a profound influence on the dynamics of back-propagation learning.
2. The inverse of the Hessian provides a basis for pruning (i.e., deleting) insignificant synaptic weights from a multilayer perceptron; this issue will be discussed in Section 4.14.
3. The Hessian is basic to the formulation of second-order optimization methods as an alternative to back-propagation learning, to be discussed in Section 4.16.

In this section, we confine our attention to point 1.

In Chapter 3, we indicated that the eigenstructure of the Hessian has a profound influence on the convergence properties of the LMS algorithm. So it is also with the back-propagation algorithm, but in a much more complicated way. Typically, the Hessian of the error surface pertaining to a multilayer perceptron trained with the back-propagation algorithm has the following composition of eigenvalues (LeCun et al., 1998):

- a small number of small eigenvalues,
- a large number of medium-sized eigenvalues, and
- a small number of large eigenvalues.

There is therefore a wide spread in the eigenvalues of the Hessian.

The factors affecting the composition of the eigenvalues may be grouped as follows:

- nonzero-mean input signals or nonzero-mean induced neural output signals;
- correlations between the elements of the input signal vector and correlations between induced neural output signals;
- wide variations in the second-order derivatives of the cost function with respect to synaptic weights of neurons in the network as we proceed from one layer to the next. These derivatives are often smaller in the lower layers, with the synaptic weights in the first hidden layer learning slowly and those in the last layers learning quickly.

Avoidance of Nonzero-mean Inputs

From Chapter 3, we recall that the *learning time* of the LMS algorithm is sensitive to variations in the condition number $\lambda_{max}/\lambda_{min}$, where λ_{max} is the largest eigenvalue of the Hessian and λ_{min} is its smallest nonzero eigenvalue. Experimental results show that a similar situation holds for the back-propagation algorithm, which is a generalization of the LMS algorithm. For inputs with nonzero mean, the ratio $\lambda_{max}/\lambda_{min}$ is larger than its corresponding value for zero-mean inputs: The larger the mean of the inputs, the larger the ratio $\lambda_{max}/\lambda_{min}$ will be. This observation has a serious implication for the dynamics of back-propagation learning.

For the learning time to be minimized, the use of nonzero-mean inputs should be avoided. Now, insofar as the signal vector $\mathbf{x}$ applied to a neuron in the first hidden layer of a multilayer perceptron (i.e., the signal vector applied to the input layer) is concerned, it is easy to remove the mean from each element of $\mathbf{x}$ before its application to the network. But what about the signals applied to the neurons in the remaining hidden and output layers of the network? The answer to this question lies in the type of activation function used in the network. In the case of the logistic function, the output of each neuron is restricted to the interval $[0, 1]$. Such a choice acts as a source of *systematic bias* for those neurons located beyond the first hidden layer of the network. To overcome this problem, we need to use the hyperbolic tangent function that is odd symmetric. With this latter choice, the output of each neuron is permitted to assume both positive and negative values in the interval $[-1, 1]$, in which case it is likely for its mean to be zero. If the network connectivity is large, back-propagation learning with odd-symmetric activation functions can yield faster convergence than a similar process with nonsymmetric activation functions. This condition provides the justification for heuristic 3 described in Section 4.6.

Asymptotic Behavior of On-line Learning

For a good understanding of on-line learning, we need to know how the ensemble-averaged learning curve evolves across time. Unlike the LMS algorithm, this calculation is unfortunately much too difficult to perform. Generally speaking, the error-performance surface may have exponentially many local minima and multiple global minima because of symmetry properties of the network. Surprisingly, this characteristic of the error-performance surface may turn out to be a useful feature in the following sense: Given that an early-stopping method is used for network training (see Section 4.13) or the

network is regularized (see Section 4.14), we may nearly always find ourselves "close" to a local minimum.

In any event, due to the complicated nature of the error-performance surface, we find that in the literature, statistical analysis of the learning curve is confined to its asymptotic behavior in the neighborhood of a local minimum. In this context, we may highlight some important aspects of this asymptotic behavior, assuming a fixed learning-rate parameter, as follows (Murata, 1998):

(i) The learning curve consists of three terms:
- *minimal loss*, determined by the optimal parameter $\mathbf{w}^*$, which pertains to a local or global minimum;
- *additional loss*, caused by fluctuations in evolution of the weight-vector estimator $w(n)$ around the mean

$$\lim_{n\to\infty} \mathbb{E}[\hat{\mathbf{w}}(n)] = \mathbf{w}^*$$

- *a time-dependent term*, describing the effect of decreasing speed of error convergence on algorithmic performance.

(ii) To ensure stability of the on-line learning algorithm, the learning-rate parameter η must be assigned a value smaller than the reciprocal of the largest eigenvalue of the Hessian, $1/\lambda_{max}$. On the other hand, the speed of convergence of the algorithm is dominated by the smallest eigenvalue of the Hessian, λ_{min}.

(iii) Roughly speaking, if the learning-rate parameter η is assigned a large value, then the speed of convergence is fast, but there will be large fluctuations around the local or global minimum, even if the number of iterations, n, approaches infinity. Conversely, if η is small, then the extent of fluctuations is small, but the speed of convergence will be slow.

4.10 OPTIMAL ANNEALING AND ADAPTIVE CONTROL OF THE LEARNING RATE

In Section 4.2, we emphasized the popularity of the on-line learning algorithm for two main reasons:

(i) The algorithm is simple, in that its implementation requires a minimal amount of memory, which is used merely to store the old value of the estimated weight vector from one iteration to the next.

(ii) With each example $\{\mathbf{x}, \mathbf{d}\}$ being used only once at every time-step, the learning rate assumes a more important role in on-line learning than in batch learning, in that the on-line learning algorithm has the built-in ability to *track* statistical variations in the environment responsible for generating the training set of examples.

In Amari (1967) and, more recently, Opper (1996), it is shown that *optimally annealed* on-line learning is enabled to operate as fast as batch learning in an *asymptotic sense*. This issue is explored in what follows.

Optimal Annealing of the Learning Rate

Let $\mathbf{w}$ denote the vector of synaptic weights in the network, stacked up on top of each other in some orderly fashion. With $\hat{\mathbf{w}}(n)$ denoting the *old estimate* of the weight vector $\mathbf{w}$ at time-step n, let $\hat{\mathbf{w}}(n+1)$ denote the *updated estimate* of $\mathbf{w}$ on receipt of the "input-desired response" example $\{\mathbf{x}(n+1), \mathbf{d}(n+1)\}$. Correspondingly, let $\mathbf{F}(\mathbf{x}(n+1); \hat{\mathbf{w}}(n))$ denote the vector-valued output of the network produced in response to the input $\mathbf{x}(n+1)$; naturally the dimension of the function $\mathbf{F}$ must be the same as that of the desired response vector $\mathbf{d}(n)$. Following the defining equation of Eq. (4.3), we may express the instantaneous energy as the squared Euclidean norm of the estimation error, as shown by

$$\mathscr{E}(\mathbf{x}(n), \mathbf{d}(n); \mathbf{w}) = \frac{1}{2}\|\mathbf{d}(n) - \mathbf{F}(\mathbf{x}(n); \mathbf{w})\|^2 \tag{4.55}$$

The *mean-square error*, or *expected risk*, of the on-line learning problem is defined by

$$J(\mathbf{w}) = \mathbb{E}_{\mathbf{x},\mathbf{d}}[\mathscr{E}(\mathbf{x}, \mathbf{d}; \mathbf{w})] \tag{4.56}$$

where $\mathbb{E}_{\mathbf{x},\mathbf{d}}$ is the expectation operator performed with respect to the example $\{\mathbf{x}, \mathbf{d}\}$. The solution

$$\mathbf{w}^* = \arg\min_{\mathbf{w}}[J(\mathbf{w})] \tag{4.57}$$

defines the *optimal parameter vector*.

The instantaneous *gradient vector* of the learning process is defined by

$$\begin{aligned}\mathbf{g}(\mathbf{x}(n), \mathbf{d}(n); \mathbf{w}) &= \frac{\partial}{\partial \mathbf{w}}\mathscr{E}(\mathbf{x}(n), \mathbf{d}(n); \mathbf{w}) \\ &= -(\mathbf{d}(n) - \mathbf{F}(\mathbf{x}(n); \mathbf{w})\mathbf{F}'(\mathbf{x}(n); \mathbf{w})\end{aligned} \tag{4.58}$$

where

$$\mathbf{F}'(\mathbf{x}; \mathbf{w}) = \frac{\partial}{\partial \mathbf{w}}\mathbf{F}(\mathbf{x}; \mathbf{w}) \tag{4.59}$$

With the definition of the gradient vector just presented, we may now express the on-line learning algorithm as

$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) - \eta(n)\mathbf{g}(\mathbf{x}(n+1), \mathbf{d}(n+1); \hat{\mathbf{w}}(n)) \tag{4.60}$$

or, equivalently,

$$\underbrace{\hat{\mathbf{w}}(n+1)}_{\substack{\text{Updated}\\ \text{estimate}}} = \underbrace{\hat{\mathbf{w}}(n)}_{\substack{\text{Old}\\ \text{estimate}}} + \underbrace{\eta(n)}_{\substack{\text{Learning-}\\ \text{rate}\\ \text{parameter}}}\underbrace{[\mathbf{d}(n+1) - \mathbf{F}(\mathbf{x}(n+1); \hat{\mathbf{w}}(n))]}_{\text{Error signal}}\underbrace{\mathbf{F}'(\mathbf{x}(n+1); \hat{\mathbf{w}}(n))}_{\substack{\text{Partial derivative of}\\ \text{the network function } \mathbf{F}}} \tag{4.61}$$

Given this difference equation, we may go on to describe the *ensemble-averaged dynamics* of the weight vector $\mathbf{w}$ in the neighborhood of the optimal parameter $\mathbf{w}^*$ by the continuous differential equation

$$\frac{d}{dt}\hat{\mathbf{w}}(t) = -\eta(t)\mathbb{E}_{\mathbf{x},\mathbf{d}}[\mathbf{g}(\mathbf{x}(t), \mathbf{d}(t); \hat{\mathbf{w}}(t))] \tag{4.62}$$

where t denotes continuous time. Following Murata (1998), the expected value of the gradient vector is approximated by

$$\mathbb{E}_{\mathbf{x},\mathbf{d}}[\mathbf{g}(\mathbf{x}, \mathbf{d}; \hat{\mathbf{w}}(t))] \approx -\mathbf{K}^*(\mathbf{w}^* - \hat{\mathbf{w}}(t)) \tag{4.63}$$

where the ensembled-averaged matrix $\mathbf{K}^*$ is itself defined by

$$\begin{aligned}\mathbf{K}^* &= \mathbb{E}_{\mathbf{x},\mathbf{d}}\left[\frac{\partial}{\partial \mathbf{w}}\mathbf{g}(\mathbf{x}, \mathbf{d}; \mathbf{w})\right] \\ &= \mathbb{E}_{\mathbf{x},\mathbf{d}}\left[\frac{\partial^2}{\partial \mathbf{w}^2}\mathscr{E}(\mathbf{x}, \mathbf{d}; \mathbf{w})\right]\end{aligned} \tag{4.64}$$

The new Hessian $\mathbf{K}^*$ is a positive-definite matrix defined differently from the Hessian $\mathbf{H}$ of Eq. (4.54). However, if the environment responsible for generating the training examples $\{\mathbf{x}, \mathbf{d}\}$ is *ergodic*, we may then substitute the Hessian $\mathbf{H}$, based on time averaging, for the Hessian $\mathbf{K}^*$, based on ensemble-averaging. In any event, using Eq. (4.63) in Eq. (4.62), we find that the continuous differential equation describing the evolution of the estimator $\hat{\mathbf{w}}(t)$ may be approximated as

$$\frac{d}{dt}\hat{\mathbf{w}}(t) \approx -\eta(t)\mathbf{K}^*(\mathbf{w}^* - \hat{\mathbf{w}}(t)) \tag{4.65}$$

Let the vector $\mathbf{q}$ denote an eigenvector of the matrix $\mathbf{K}^*$, as shown by the defining equation

$$\mathbf{K}^*\mathbf{q} = \lambda\mathbf{q} \tag{4.66}$$

where λ is the eigenvalue associated with the eigenvector $\mathbf{q}$. We may then introduce the new function

$$\xi(t) = \mathbb{E}_{\mathbf{x},\mathbf{d}}[\mathbf{q}^T\mathbf{g}(\mathbf{x}, \mathbf{d}; \hat{\mathbf{w}}(t))] \tag{4.67}$$

which, in light of Eq. (4.63), may itself be approximated as

$$\begin{aligned}\xi(t) &\approx -\mathbf{q}^T\mathbf{K}^*(\mathbf{w}^* - \hat{\mathbf{w}}(t)) \\ &= -\lambda\mathbf{q}^T(\mathbf{w}^* - \hat{\mathbf{w}}(t))\end{aligned} \tag{4.68}$$

At each instant of time t, the function $\xi(t)$ takes on a scalar value, which may be viewed as an approximate measure of the *Euclidean distance* between two projections onto the eigenvector $\mathbf{q}$, one due to the optimal parameter $\mathbf{w}^*$ and the other due to the estimator $\hat{\mathbf{w}}(t)$. The value of $\xi(t)$ is therefore reduced to zero if, and when, the estimator $\hat{\mathbf{w}}(t)$ converges to $\mathbf{w}^*$.

From Eqs. (4.65), (4.66), and (4.68), we find that the function $\xi(t)$ is related to the time-varying learning-rate parameter $\eta(t)$ as follows:

$$\frac{d}{dt}\xi(t) = -\lambda\eta(t)\xi(t) \tag{4.69}$$

This differential equation may be solved to yield

$$\xi(t) = c\exp(-\lambda\textstyle\int\eta(t)dt) \tag{4.70}$$

where c is a positive integration constant.

Following the annealing schedule due to Darken and Moody (1991) that was discussed in Chapter 3 on the LMS algorithm, let the formula

$$\eta(t) = \frac{\tau}{t + \tau}\,\eta_0 \tag{4.71}$$

account for dependence of the learning-rate on time t, where τ and η_0 are positive *tuning parameters*. Then, substituting this formula into Eq. (4.70), we find that the corresponding formula tor the function $\xi(t)$ is

$$\xi(t) = c(t + \tau)^{-\lambda\tau\eta_0} \tag{4.72}$$

For $\xi(t)$ to vanish as time t approaches infinity, we require that the product term $\lambda\tau\eta_0$ in the exponent be large compared with unity, which may be satisfied by setting $\eta_0 = \alpha/\lambda$ for positive α.

Now, there remains only the issue of how to choose the eigenvector $\mathbf{q}$. From the previous section, we recall that the convergence speed of the learning curve is dominated by the smallest eigenvalue $\lambda_{\min}$ of the Hessian $\mathbf{H}$. With this Hessian and the new Hessian $\mathbf{K}^*$ tending to behave similarly, a clever choice is to hypothesize that for a sufficiently large number of iterations, the evolution of the estimator $\hat{\mathbf{w}}(t)$ over time t may be considered as a one-dimensional process, running "almost parallel" to the eigenvector of the Hessian $\mathbf{K}^*$ associated with the smallest eigenvalue $\lambda_{\min}$, as illustrated in Fig. 4.15. We may thus set

$$\mathbf{q} = \frac{\mathbb{E}_{\mathbf{x},\mathbf{d}}[\mathbf{g}(\mathbf{x}, \mathbf{d}; \hat{\mathbf{w}})]}{\|\mathbb{E}_{\mathbf{x},\mathbf{d}}[\mathbf{g}(\mathbf{x}, \mathbf{d}; \hat{\mathbf{w}})]\|} \tag{4.73}$$

where the normalization is introduced to make the eigenvector $\mathbf{q}$ assume unit Euclidean length. Correspondingly, the use of this formula in Eq. (4.67) yields

$$\xi(t) = \|\mathbb{E}_{\mathbf{x},\mathbf{d}}[\mathbf{g}(\mathbf{x}, \mathbf{d}; \hat{\mathbf{w}}(t))]\| \tag{4.74}$$

We may now summarize the results of the discussion presented in this section by making the following statements:

1. The choice of the annealing schedule described in Eq. (4.71) satisfies the two conditions

$$\sum_t \eta(t) \to \infty \text{ and } \sum_t \eta^2(t) > \infty, \text{ as } t \to \infty \tag{4.75}$$

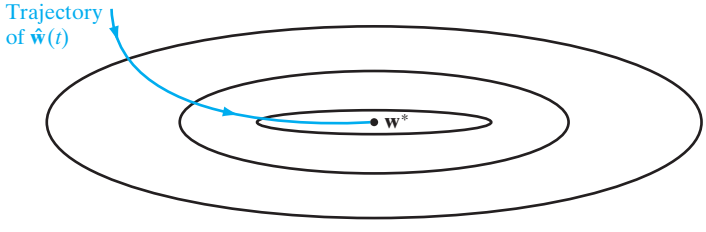


FIGURE 4.15 The evolution of the estimator $\hat{\mathbf{w}}(t)$ over time t. The ellipses represent contours of the expected risk for varying values of $\mathbf{w}$, assumed to be two-dimensional.

In other words, $\eta(t)$ satisfies the requirements of *stochastic approximation theory* (Robbins and Monro, 1951).

2. As time t approaches infinity, the function $\xi(t)$ approaches zero asymptotically. In accordance with Eq. (4.68), it follows that the estimator $\hat{\mathbf{w}}(t)$ approaches the optimal estimator $\mathbf{w}^*$ as t approaches infinity.
3. The ensemble-averaged trajectory of the estimator $\hat{\mathbf{w}}(t)$ is almost parallel to the eigenvector of the Hessian $\mathbf{K}^*$ associated with the smallest eigenvalue $\lambda_{\min}$ after a large enough number of iterations.
4. The optimally annealed on-line learning algorithm for a network characterized by the weight vector $\mathbf{w}$ is collectively described by the following set of three equations:

$$\left.\begin{aligned} &\underbrace{\hat{\mathbf{w}}(n+1)}_{\text{Updated estimate}} = \underbrace{\hat{\mathbf{w}}(n)}_{\text{Old estimate}} + \underbrace{\eta(n)}_{\text{Learning-rate parameter}}\underbrace{(\mathbf{d}(n+1) - \mathbf{F}(\mathbf{x}(n)+1;\ \hat{\mathbf{w}}(n))}_{\text{Error signal}}\underbrace{\mathbf{F}'(\mathbf{x}(n+1);\ \hat{\mathbf{w}}(n))}_{\text{Partial derivative of the network function } \mathbf{F}} \\ &\eta(n) = \frac{n_{\text{switch}}}{n + n_{\text{switch}}}\eta_0 \\ &\eta_0 = \frac{\alpha}{\lambda_{\min}}, \qquad \alpha = \text{positive constant} \end{aligned}\right\} \tag{4.76}$$

Here, it is assumed that the environment responsible for generating the training examples $\{\mathbf{x}, \mathbf{d}\}$ is ergodic, so that the ensemble-averaged Hessian $\mathbf{K}^*$ assumes the same value as the time-averaged Hessian $\mathbf{H}$.

5. When the learning-rate parameter η_0 is *fixed* in on-line learning based on stochastic gradient descent, stability of the algorithm requires that we choose $\eta_0 < 1/\lambda_{\max}$, where $\lambda_{\max}$ is the largest eigenvalue of the Hessian $\mathbf{H}$. On the other hand, in the case of optimally annealed stochastic gradient descent, according to the third line of Eq. (4.76), the choice is $\eta_0 < 1/\lambda_{\min}$, where $\lambda_{\min}$ is the smallest eigenvalue of $\mathbf{H}$.
6. The *time constant* n_{switch}, a positive integer, defines the *transition* from a regime of fixed η_0 to the annealing regime, where the time-varying learning-rate parameter $\eta(n)$ assumes the desired form c/n, where c is a constant, in accordance with stochastic approximation theory.

Adaptive Control of the Learning Rate

The optimal annealing schedule, described in the second line of Eq. (4.76), provides an important step in improved utilization of on-line learning. However, a practical limitation of this annealing schedule is the requirement that we know the time constant η_{switch} a priori. A practical issue of concern, then, is the fact that when the application of interest builds on the use of on-line learning in a nonstationary environment where the statistics of the training sequence change from one example to the next, the use of a prescribed time constant n_{switch} may no longer be a realistic option. In situations of this kind, which occur frequently in practice, the on-line learning algorithm needs to be equipped with a built-in mechanism for the *adaptive control* of the learning rate. Such

a mechanism was first described in the literature by Murata (1998), in which the so-called *learning of the learning algorithm* (Sompolinsky et al., 1995) was appropriately modified.

The adaptive algorithm due to Murata is configured to achieve two objectives:

1. *automatic adjustment* of the learning rate, which accounts for statistical variations in the environment responsible for generation of the training sequence of examples;
2. *generalization* of the on-line learning algorithm so that its applicability is broadened by avoiding the need for a prescribed cost function.

To be specific, the ensemble-averaged dynamics of the weight vector $\mathbf{w}$, defined in Eq. (4.62), is now rewritten as[6]

$$\frac{d}{dt}\hat{\mathbf{w}}(t) = -\eta(t)\mathbb{E}_{\mathbf{x},\mathbf{d}}[\mathbf{f}(\mathbf{x}(t), \mathbf{d}(t); \hat{\mathbf{w}}(t))] \tag{4.77}$$

where the vector-valued function $\mathbf{f}(\cdot,\cdot;\cdot)$ denotes *flow* that determines the change applied to the estimator $\hat{\mathbf{w}}(t)$ in response to the incoming example $\{\mathbf{x}(t), \mathbf{d}(t)\}$. The flow $\mathbf{f}$ is required to satisfy the condition

$$\mathbb{E}_{\mathbf{x},\mathbf{d}}[\mathbf{f}(\mathbf{x}, \mathbf{d}; \mathbf{w}^*)] = \mathbf{0} \tag{4.78}$$

where $\mathbf{w}^*$ is the optimal value of the weight vector $\mathbf{w}$, as previously defined in Eq. (4.57). In other words, the flow $\mathbf{f}$ must asymptotically converge to the optimal parameter $\mathbf{w}^*$ across time t. Moreover, for stability, we also require that the gradient of $\mathbf{f}$ should be a positive-definite matrix. The flow $\mathbf{f}$ includes the gradient vector $\mathbf{g}$ in Eq. (4.62) as a special case.

The previously defined equations of Eqs. (4.63) through (4.69) apply equally well to Murata's algorithm. Thereafter, however, the assumption made is that the evolution of the learning rate $\eta(t)$ across time t is governed by a dynamic system that comprises the pair of differential equations

$$\frac{d}{dt}\xi(t) = -\lambda\eta(t)\xi(t) \tag{4.79}$$

and

$$\frac{d}{dt}\eta(t) = \alpha\eta(t)(\beta\xi(t) - \eta(t)) \tag{4.80}$$

where it should be noted that $\xi(t)$ is always positive and α and β are positive constants. The first equation of this dynamic system is a repeat of Eq. (4.69). The second equation of the system is motivated by the corresponding differential equation in the learning of the learning algorithm described in Sompolinsky et al. (1995).[7]

As before, the λ in Eq. (4.79) is the eigenvalue associated with the eigenvector $\mathbf{q}$ of the Hessian $\mathbf{K}^*$. Moreover, it is hypothesized that $\mathbf{q}$ is chosen as the particular eigenvector associated with the smallest eigenvalue $\lambda_{\min}$. This, in turn, means that the ensemble-averaged flow $\mathbf{f}$ converges to the optimal parameter $\mathbf{w}^*$ in a manner similar to that previously described, as depicted in Fig. 4.15.

The *asymptotic behavior* of the dynamic system described in Eqs. (4.79) and (4.80) is given by the corresponding pair of equations

$$\xi(t) = \frac{1}{\beta}\left(\frac{1}{\lambda} - \frac{1}{\alpha}\right)\frac{1}{t}, \qquad \alpha > \lambda \tag{4.81}$$

and

$$\eta(t) = \frac{c}{t}, \qquad c = \lambda^{-1} \tag{4.82}$$

The important point to note here is that this new dynamic system exhibits the desired annealing of the learning rate $\eta(t)$—namely, c/t for large t—which is optimal for any estimator $\hat{\mathbf{w}}(t)$ converging to $\mathbf{w}^*$, as previously discussed.

In light of the considerations just presented, we may now formally describe the *Murata adaptive algorithm* for on-line learning in discrete time as follows (Murata, 1998; Müller et al., 1998):

$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) - \eta(n)\mathbf{f}(\mathbf{x}(n+1), \mathbf{d}(n+1); \hat{\mathbf{w}}(n)) \tag{4.83}$$

$$\mathbf{r}(n+1) = \mathbf{r}(n) + \delta\mathbf{f}(\mathbf{x}(n+1), \mathbf{d}(n+1); \hat{\mathbf{w}}(n)), \qquad 0 < \delta < 1 \tag{4.84}$$

$$\eta(n+1) = \eta(n) + \alpha\eta(n)(\beta\|\mathbf{r}(n+1)\| - \eta(n)) \tag{4.85}$$

The following points are noteworthy in the formulation of this discrete-time system of equations:

- Equation (4.83) is simply the instantaneous discrete-time version of the differential equation of Eq. (4.77).
- Equation (4.84) includes an *auxiliary* vector $\mathbf{r}(n)$, which has been introduced to account for the continuous-time function $\xi_\chi(t)$. Moreover, this second equation of the Murata adaptive algorithm includes a *leakage factor* whose value δ controls the running average of the flow $\mathbf{f}$.
- Equation (4.85) is a discrete-time version of the differential equation Eq. (4.80). The updated auxiliary vector $\mathbf{r}(n+1)$ included in Eq. (4.85) links it to Eq. (4.84); in so doing, allowance is made for the linkage between the continuous-time functions $\xi(t)$ and $\eta(t)$ previously defined in Eqs. (4.79) and (4.80).

Unlike the continuous-time dynamic system described in Eqs. (4.79) and (4.80), the asymptotic behavior of the learning-rate parameter $\eta(t)$ in Eq. (4.85) does not converge to zero as the number of iterations, n, approaches infinity, thereby violating the requirement for optimal annealing. Accordingly, in the neighborhood of the optimal parameter $\mathbf{w}^*$, we now find that for the Murata adaptive algorithm:

$$\lim_{n\to\infty} \hat{\mathbf{w}}(n) \neq \mathbf{w}^* \tag{4.86}$$

This asymptotic behavior is different from that of the optimally annealed on-line learning algorithm of Eq. (4.76). Basically, the deviation from optimal annealing is attributed to the use of a running average of the flow in Eq. (4.77), the inclusion of which was motivated by the need to account for the algorithm not having access to a prescribed cost

function, as was the case in deriving the optimally annealed on-line learning algorithm of Eq. (4.76).

The learning of the learning rule is useful when the optimal $\hat{\mathbf{w}}^*$ varies with time n slowly (i.e., the environment responsible for generating the examples is nonstationary) or it changes suddenly. On the other hand, the $1/n$ rule is not a good choice in such an environment, because η_n becomes very small for large n, causing the $1/n$ rule to lose its learning capability. Basically, the difference between the optimally annealed on-learning algorithm of Eq. (4.76) and the on-line learning algorithm described in Eqs. (4.83) to (4.85) is that the latter has a built-in mechanism for adaptive control of the learning rate—hence its ability to *track* variations in the optimal $\hat{\mathbf{w}}^*$.

A final comment is in order: Although the Murata adaptive algorithm is indeed *suboptimal* insofar as annealing of the learning-rate parameter is concerned, its important virtue is the broadened applicability of on-line learning in a practically implementable manner.

4.11 GENERALIZATION

In back-propagation learning, we typically start with a training sample and use the back-propagation algorithm to compute the synaptic weights of a multilayer perceptron by loading (encoding) as many of the training examples as possible into the network. The hope is that the neural network so designed will generalize well. A network is said to *generalize* well when the input–output mapping computed by the network is correct (or nearly so) for test data never used in creating or training the network; the term "generalization" is borrowed from psychology. Here, it is assumed that the test data are drawn from the same population used to generate the training data.

The learning process (i.e., training of a neural network) may be viewed as a "curve-fitting" problem. The network itself may be considered simply as a nonlinear input–output mapping. Such a viewpoint then permits us to look at generalization not as a mystical property of neural networks, but rather simply as the effect of a good nonlinear interpolation of the input data. The network performs useful interpolation primarily because multilayer perceptrons with continuous activation functions lead to output functions that are also continuous.

Figure 4.16a illustrates how generalization may occur in a hypothetical network. The nonlinear input–output mapping represented by the curve depicted in this figure is computed by the network as a result of learning the points labeled as "training data." The point marked in red on the curve as "generalization" is thus seen as the result of interpolation performed by the network.

A neural network that is designed to generalize well will produce a correct input–output mapping even when the input is slightly different from the examples used to train the network, as illustrated in the figure. When, however, a neural network learns too many input–output examples, the network may end up memorizing the training data. It may do so by finding a feature (due to noise, for example) that is present in the training data, but not true of the underlying function that is to be modeled. Such a phenomenon is referred to as *overfitting* or *overtraining*. When the network is overtrained, it loses the ability to generalize between similar input–output patterns.

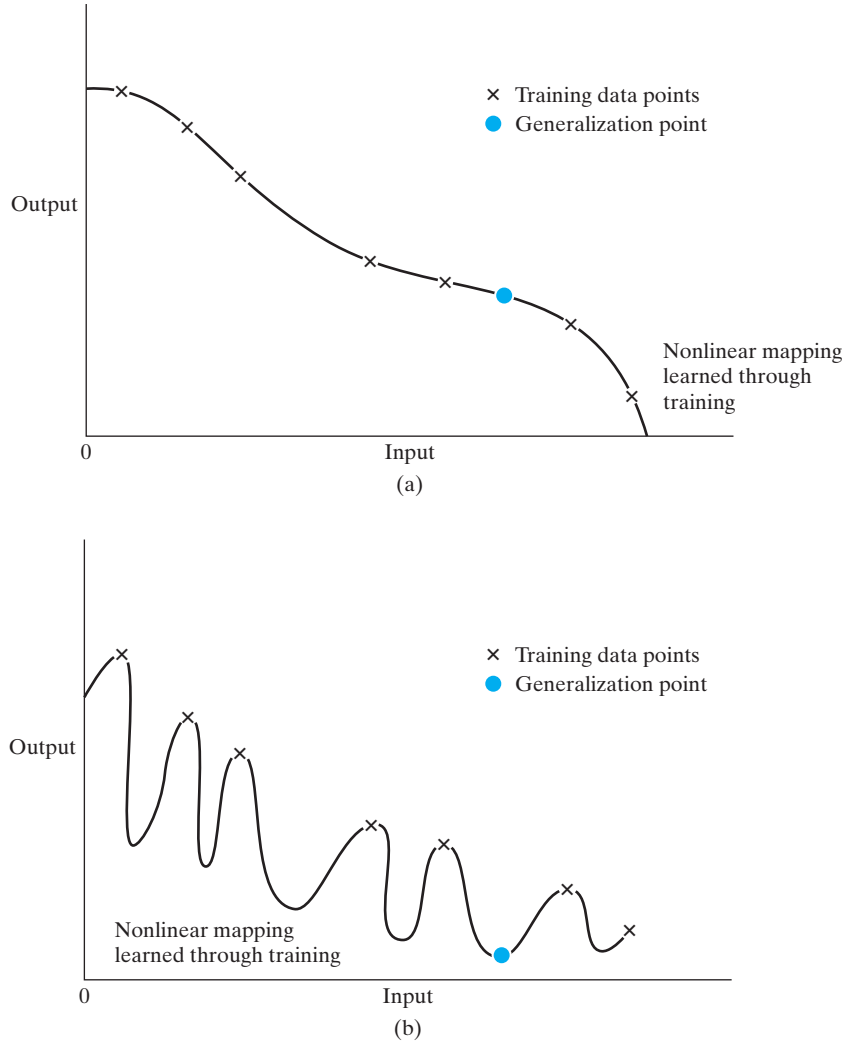


FIGURE 4.16 (a) Properly fitted nonlinear mapping with good generalization. (b) Overfitted nonlinear mapping with poor generalization.

Ordinarily, loading data into a multilayer perceptron in this way requires the use of more hidden neurons than are actually necessary, with the result that undesired contributions in the input space due to noise are stored in synaptic weights of the network. An example of how poor generalization due to memorization in a neural network may occur is illustrated in Fig. 4.16b for the same data as depicted in Fig. 4.16a. "Memorization" is essentially a "look-up table," which implies that the input–output mapping computed by the neural network is not smooth. As pointed out in Poggio and Girosi (1990a), smoothness of input–output mapping is closely related to such model-selection criteria as *Occam's*

razor, the essence of which is to select the "simplest" function in the absence of any prior knowledge to the contrary. In the context of our present discussion, the simplest function means the smoothest function that approximates the mapping for a given error criterion, because such a choice generally demands the fewest computational resources. Smoothness is also natural in many applications, depending on the scale of the phenomenon being studied. It is therefore important to seek a smooth nonlinear mapping for ill-posed input–output relationships, so that the network is able to classify novel patterns correctly with respect to the training patterns (Wieland and Leighton, 1987).

Sufficient Training-Sample Size for a Valid Generalization

Generalization is influenced by three factors: (1) the size of the training sample and how representative the training sample is of the environment of interest, (2) the architecture of the neural network, and (3) the physical complexity of the problem at hand. Clearly, we have no control over the lattermost factor. In the context of the other two factors, we may view the issue of generalization from two different perspectives:

- The architecture of the network is fixed (hopefully in accordance with the physical complexity of the underlying problem), and the issue to be resolved is that of determining the size of the training sample needed for a good generalization to occur.
- The size of the training sample is fixed, and the issue of interest is that of determining the best architecture of network for achieving good generalization.

Both of these viewpoints are valid in their own individual ways.

In practice, it seems that all we really need for a good generalization is to have the size of the training sample, N, satisfy the condition

$$N = O\left(\frac{W}{\varepsilon}\right) \tag{4.87}$$

where W is the total number of free parameters (i.e., synaptic weights and biases) in the network, ε denotes the fraction of classification errors permitted on test data (as in pattern classification), and $O(\cdot)$ denotes the order of quantity enclosed within. For example, with an error of 10 percent, the number of training examples needed should be about 10 times the number of free parameters in the network.

Equation (4.87) is in accordance with *Widrow's rule of thumb* for the LMS algorithm, which states that the settling time for adaptation in linear adaptive temporal filtering is approximately equal to the memory span of an adaptive tapped-delay-line filter divided by the misadjustment (Widrow and Stearns, 1985; Haykin, 2002). The misadjustment in the LMS algorithm plays a role somewhat analogous to the error ε in Eq. (4.87). Further justification for this empirical rule is presented in the next section.

4.12 APPROXIMATIONS OF FUNCTIONS

A multilayer perceptron trained with the back-propagation algorithm may be viewed as a practical vehicle for performing a *nonlinear input–output mapping* of a general nature. To be specific, let m_0 denote the number of input (source) nodes of a multilayer

perceptron, and let $M = m_L$ denote the number of neurons in the output layer of the network. The input–output relationship of the network defines a mapping from an m_0-dimensional Euclidean input space to an M-dimensional Euclidean output space, which is infinitely continuously differentiable when the activation function is likewise. In assessing the capability of the multilayer perceptron from this viewpoint of input–output mapping, the following fundamental question arises:

What is the minimum number of hidden layers in a multilayer perceptron with an input–output mapping that provides an approximate realization of any continuous mapping?

Universal Approximation Theorem

The answer to this question is embodied in the *universal approximation theorem*[8] for a nonlinear input–output mapping, which may be stated as follows:

Let $\varphi(\cdot)$ be a nonconstant, bounded, and monotone-increasing continuous function. Let I_{m_0} denote the m_0-dimensional unit hypercube $[0, 1]^{m_0}$. The space of continuous functions on I_{m_0} is denoted by $C(I_{m_0})$. Then, given any function $f \ni C(I_{m_0})$ and $\varepsilon > 0$, there exist an integer m_1 and sets of real constants α_i, b_i, and w_{ij}, where $i = 1, ..., m_1$ and $j = 1, ..., m_0$ such that we may define

$$F(x_1, ..., x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \varphi\left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i \right) \tag{4.88}$$

as an approximate realization of the function $f(\cdot)$; that is,

$$|F(x_1, ..., x_{m_0}) - f(x_1, ..., x_{m_0})| < \varepsilon$$

for all $x_1, x_2, ..., x_{m_0}$ that lie in the input space.

The universal approximation theorem is directly applicable to multilayer perceptrons. We first note, for example, that the hyperbolic tangent function used as the nonlinearity in a neural model for the construction of a multilayer perceptron is indeed a nonconstant, bounded, and monotone-increasing function; it therefore satisfies the conditions imposed on the function $\varphi(\cdot)$ Next, we note that Eq. (4.88) represents the output of a multilayer perceptron described as follows:

1. The network has m_0 input nodes and a single hidden layer consisting of m_1 neurons; the inputs are denoted by $x_1, ..., x_{m_0}$.
2. Hidden neuron i has synaptic weights $w_{i_1}, ..., w_{m_0}$, and bias b_i.
3. The network output is a linear combination of the outputs of the hidden neurons, with $\alpha_1, ..., \alpha_{m_1}$ defining the synaptic weights of the output layer.

The universal approximation theorem is an *existence theorem* in the sense that it provides the mathematical justification for the approximation of an arbitrary continuous function as opposed to exact representation. Equation (4.88), which is the backbone of the theorem, merely generalizes approximations by finite Fourier series. In effect, the theorem states that *a single hidden layer is sufficient for a multilayer perceptron to compute a uniform ε approximation to a given training set represented by the set of inputs* $x_1, ..., x_{m_0}$ and a desired (target) output $f(x_1, ..., x_{m_0})$. However, the theorem

does not say that a single hidden layer is optimum in the sense of learning time, ease of implementation, or (more importantly) generalization.

Bounds on Approximation Errors

Barron (1993) has established the approximation properties of a multilayer perceptron, assuming that the network has a single layer of hidden neurons using sigmoid functions and a linear output neuron. The network is trained using the back-propagation algorithm and then tested with new data. During training, the network learns specific points of a target function f in accordance with the training data and thereby produces the approximating function F defined in Eq. (4.88). When the network is exposed to test data that have not been seen before, the network function F acts as an "estimator" of new points of the target function; that is, $F = \hat{f}$.

A smoothness property of the target function f is expressed in terms of its Fourier representation. In particular, the average of the norm of the frequency vector weighted by the Fourier magnitude distribution is used as a measure for the extent to which the function f oscillates. Let $\tilde{f}(\boldsymbol{\omega})$ denote the multidimensional Fourier transform of the function $f(\mathbf{x})$, $\mathbf{x} \in \mathbb{R}^{m_0}$; the m_0-by-1 vector $\boldsymbol{\omega}$ is the frequency vector. The function $f(x)$ is defined in terms of its Fourier transform $\tilde{f}(\boldsymbol{\omega})$ by the inverse formula

$$f(x) = \int_{\mathbb{R}^{m_0}} \tilde{f}(\boldsymbol{\omega}) \exp(j\boldsymbol{\omega}^T\mathbf{x})\, d\boldsymbol{\omega} \tag{4.89}$$

where $j = \sqrt{-1}$. For the complex-valued function $\tilde{f}(\boldsymbol{\omega})$ for which $\boldsymbol{\omega}\tilde{f}(\boldsymbol{\omega})$ is integrable, we define the *first absolute moment* of the Fourier magnitude distribution of the function f as

$$C_f = \int_{\mathbb{R}^{m_0}} |\tilde{f}(\boldsymbol{\omega})| \times \|\boldsymbol{\omega}\|^{1/2}\, d\boldsymbol{\omega} \tag{4.90}$$

where $\|\boldsymbol{\omega}\|$ is the Euclidean norm of $\boldsymbol{\omega}$ and $|\tilde{f}(\boldsymbol{\omega})|$ is the absolute value of $\tilde{f}(\boldsymbol{\omega})$. The first absolute moment C_f quantifies the *smoothness* of the function f.

The first absolute moment C_f provides the basis for a *bound* on the error that results from the use of a multilayer perceptron represented by the input–output mapping function $F(\mathbf{x})$ of Eq. (4.88) to approximate $f(\mathbf{x})$. The approximation error is measured by the *integrated squared error* with respect to an arbitrary probability measure μ on the ball $B_r = \{\mathbf{x}: \|\mathbf{x}\| \leq r\}$ of radius $r > 0$. On this basis, we may state the following proposition for a bound on the approximation error given by Barron (1993):

> *For every continuous function $f(\mathbf{x})$ with finite first moment C_f and every $m_1 \geq 1$, there exists a linear combination of sigmoid-based functions $F(\mathbf{x})$ of the form defined in Eq. (4.88) such that when the function $f(\mathbf{x})$ is observed at a set of values of the input vector $\mathbf{x}$ denoted by $\{\mathbf{x}_i\}_{i=1}^N$ that are restricted to lie inside the prescribed ball of radius r, the result provides the following bound on the empirical risk:*
>
> $$\mathscr{E}_{\text{av}}(N) = \frac{1}{N}\sum_{i=1}^{N} (f(\mathbf{x}_i) - F(\mathbf{x}_i))^2 \leq \frac{C_f'}{m_1} \tag{4.91}$$
>
> where $C_f' = (2rC_f)^2$.

In Barron (1992), the approximation result of Eq. (4.91) is used to express the bound on the risk $\mathscr{E}_{\text{av}}(N)$ resulting from the use of a multilayer perceptron with m_0 input nodes and m_1 hidden neurons as follows:

$$\mathscr{E}_{\text{av}}(N) \leq O\left(\frac{C_f^2}{m_1}\right) + O\left(\frac{m_0 m_1}{N} \log N\right) \tag{4.92}$$

The two terms in the bound on the risk $\mathscr{E}_{\text{av}}(N)$ express the tradeoff between two conflicting requirements on the size of the hidden layer:

1. *Accuracy of best approximation.* For this requirement to be satisfied, the size of the hidden layer, m_1, must be large in accordance with the universal approximation theorem.
2. *Accuracy of empirical fit to the approximation.* To satisfy this second requirement, we must use a small ratio m_1/N. For a fixed size of training sample, N, the size of the hidden layer, m_1, should be kept small, which is in conflict with the first requirement.

The bound on the risk $\mathscr{E}_{\text{av}}(N)$ described in Eq. (4.92) has other interesting implications. Specifically, we see that an exponentially large sample size, large in the dimensionality m_0 of the input space, is *not* required to get an accurate estimate of the target function, provided that the first absolute moment C_f remains finite. This result makes multilayer perceptrons as universal approximators even more important in practical terms.

The error between the empirical fit and the best approximation may be viewed as an *estimation error*. Let ε_0 denote the mean-square value of this estimation error. Then, ignoring the logarithmic factor $\log N$ in the second term of the bound in Eq. (4.92), we may infer that the size N of the training sample needed for a good generalization is about $m_0 m_1/\varepsilon_0$. This result has a mathematical structure similar to the empirical rule of Eq. (4.87), bearing in mind that $m_0 m_1$ is equal to the total number of free parameters W in the network. In other words, we may generally say that for good generalization, the number N of training examples should be larger than the ratio of the total number of free parameters in the network to the mean-square value of the estimation error.

Curse of Dimensionality

Another interesting result that emerges from the bounds described in (4.92) is that when the size of the hidden layer is optimized (i.e., the risk $\mathscr{E}_{\text{av}}(N)$ is minimized with respect to N) by setting

$$m_1 \simeq C_f \left(\frac{N}{m_0 \log N}\right)^{1/2}$$

then the risk $\mathscr{E}_{\text{av}}(N)$ is bounded by $O(C_f \sqrt{m_0 (\log N / N)})$. A surprising aspect of this result is that in terms of the first-order behavior of the risk $\mathscr{E}_{\text{av}}(N)$, the rate of convergence expressed as a function of the training-sample size N is of order $(1/N)^{1/2}$ (times a logarithmic factor). In contrast, for traditional smooth functions (e.g., polynomials and trigonometric

functions), we have a different behavior. Let s denote a measure of *smoothness*, defined as the number of continuous derivatives of a function of interest. Then, for traditional smooth functions, we find that the minimax rate of convergence of the total risk $\mathscr{E}_{\text{av}}(N)$ is of order $(1/N)^{2s/(2s+mo)}$. The dependence of this rate on the dimensionality of the input space, m_0, is responsible for the *curse of dimensionality*, which severely restricts the practical application of these functions. The use of a multilayer perceptron for function approximation appears to offer an advantage over the use of traditional smooth functions. This advantage is, however, subject to the condition that the first absolute moment C_f remains finite; this is a smoothness constraint.

The curse of dimensionality was introduced by Richard Bellman in his studies of adaptive control processes (Bellman, 1961). For a geometric interpretation of this notion, let $\mathbf{x}$ denote an m_0-dimensional input vector and $\{(\mathbf{x}_i, d_i)\}$, $i = 1, 2, \ldots, N$, denote the training sample. The *sampling density* is proportional to N^{1/m_0}. Let a function $f(\mathbf{x})$ represent a surface lying in the m_0-dimensional input space that passes near the data points $\{(\mathbf{x}_i, d_i)\}_{i=1}^{N}$. Now, if the function $f(\mathbf{x})$ is arbitrarily complex and (for the most part) completely unknown, we need *dense* sample (data) points to learn it well. Unfortunately, dense samples are hard to find in "high dimensions"—hence the curse of dimensionality. In particular, there is an *exponential* growth in complexity as a result of an increase in dimensionality, which, in turn, leads to the deterioration of the space-filling properties for uniformly randomly distributed points in higher-dimension spaces. The basic reason for the curse of dimensionality is as follows (Friedman, 1995):

> *A function defined in high-dimensional space is likely to be much more complex than a function defined in a lower-dimensional space, and those complications are harder to discern.*

Basically, there are only two ways of mitigating the curse-of-dimensionality problem:

1. Incorporate *prior knowledge* about the unknown function to be approximated. This knowledge is provided over and above the training data. Naturally, the acquisition of knowledge is problem dependent. In pattern classification, for example, knowledge may be acquired from understanding the pertinent classes (categories) of the input data.
2. Design the network so as to provide increasing *smoothness* of the unknown function with increasing input dimensionality.

Practical Considerations

The universal approximation theorem is important from a theoretical viewpoint because it provides the *necessary mathematical tool* for the viability of feedforward networks with a single hidden layer as a class of approximate solutions. Without such a theorem, we could conceivably be searching for a solution that cannot exist. However, the theorem is not constructive; that is, it does not actually specify how to determine a multilayer perceptron with the stated approximation properties.

The universal approximation theorem assumes that the continuous function to be approximated is given and that a hidden layer of unlimited size is available for the

approximation. Both of these assumptions are violated in most practical applications of multilayer perceptrons.

The problem with multilayer perceptrons using a single hidden layer is that the neurons therein tend to interact with each other globally. In complex situations, this interaction makes it difficult to improve the approximation at one point without worsening it at some other point. On the other hand, with two hidden layers, the approximation (curve-fitting) process becomes more manageable. In particular, we may proceed as follows (Funahashi, 1989; Chester, 1990):

1. *Local features* are extracted in the first hidden layer. Specifically, some neurons in the first hidden layer are used to partition the input space into regions, and other neurons in that layer learn the local features characterizing those regions.
2. *Global features* are extracted in the second hidden layer. Specifically, a neuron in the second hidden layer combines the outputs of neurons in the first hidden layer operating on a particular region of the input space and thereby learns the global features for that region and outputs zero elsewhere.

Further justification for the use of two hidden layers is presented in Sontag (1992) in the context of *inverse problems*.

4.13 CROSS-VALIDATION

The essence of back-propagation learning is to encode an input–output mapping (represented by a set of labeled examples) into the synaptic weights and thresholds of a multilayer perceptron. The hope is that the network becomes well trained so that it learns enough about the past to generalize to the future. From such a perspective, the learning process amounts to a choice of network parameterization for a given set of data. More specifically, we may view the network selection problem as choosing, within a set of candidate model structures (parameterizations), the "best" one according to a certain criterion.

In this context, a standard tool in statistics, known as *cross-validation*, provides an appealing guiding principle[9] (Stone, 1974, 1978). First the available data set is randomly partitioned into a training sample and a test set. The training sample is further partitioned into two disjoint subsets:

- *an estimation subset*, used to select the model;
- *a validation subset*, used to test or validate the model.

The motivation here is to validate the model on a data set different from the one used for parameter estimation. In this way, we may use the training sample to assess the performance of various candidate models and thereby choose the "best" one. There is, however, a distinct possibility that the model with the best-performing parameter values so selected may end up overfitting the validation subset. To guard against this possibility, the generalization performance of the selected model is measured on the test set, which is different from the validation subset.

The use of cross-validation is appealing particularly when we have to design a large neural network with good generalization as the goal. For example, we may use

cross-validation to determine the multilayer perceptron with the best number of hidden neurons and to figure out when it is best to stop training, as described in the next two subsections.

Model Selection

To expand on the idea of selecting a model in accordance with cross-validation, consider a nested *structure* of Boolean function classes denoted by

$$\begin{aligned} &\mathscr{F}_1 \subset \mathscr{F}_2 \subset \cdots \subset \mathscr{F}_n \\ &\mathscr{F}_k = \{F_k\} \\ &\quad = \{F(\mathbf{x}, \mathbf{w}); \mathbf{w} \in \mathscr{W}_k\}, \qquad k = 1, 2, ..., n \end{aligned} \tag{4.93}$$

In words, the kth function class $\mathscr{F}_k$ encompasses a family of multilayer perceptrons with similar architecture and weight vectors $\mathbf{w}$ drawn from a multidimensional weight space $\mathscr{W}_k$. A member of this class, characterized by the function or hypothesis $F_k = F(\mathbf{x}, \mathbf{w})$, $\mathbf{w} \in \mathscr{W}_k$, maps the input vector $\mathbf{x}$ into $\{0, 1\}$, where $\mathbf{x}$ is drawn from an input space $\mathscr{X}$ with some unknown probability P. Each multilayer perceptron in the structure described is trained with the back-propagation algorithm, which takes care of training the parameters of the multilayer perceptron. The model-selection problem is essentially that of choosing the multilayer perceptron with the best value of $\mathbf{w}$, the number of free parameters (i.e., synaptic weights and biases). More precisely, given that the scalar desired response for an input vector $\mathbf{x}$ is $d = \{0, 1\}$, we define the generalization error as the probability

$$\varepsilon_g(F) = P(F(\mathbf{x}) \neq d) \qquad \text{for } \mathbf{x} \in \mathscr{X}$$

We are given a training sample of labeled examples

$$\mathscr{T} = \{(\mathbf{x}_i, d_i)\}_{i=1}^{N}$$

The objective is to select the particular hypothesis $F(\mathbf{x}, \mathbf{w})$ that minimizes the generalization error $\varepsilon_g(F)$, which results when it is given inputs from the test set.

In what follows, we assume that the structure described by Eq. (4.93) has the property that, for any sample size N, we can always find a multilayer perceptron with a large enough number of free parameters $W_{\max}(N)$ such that the training sample $\mathscr{T}$ can be fitted adequately. This assumption is merely restating the universal approximation theorem of Section 4.12. We refer to $W_{\max}(N)$ as the *fitting number*. The significance of $W_{\max}(N)$ is that a reasonable model-selection procedure would choose a hypothesis $F(\mathbf{x}, \mathbf{w})$ that requires $W \leq W_{\max}(N)$; otherwise, the network complexity would be increased.

Let a parameter r, lying in the range between 0 and 1, determine the split of the training sample $\mathscr{T}$ between the estimation subset and validation subset. With $\mathscr{T}$ consisting of N examples, $(1 - r)N$ examples are allotted to the estimation subset, and the remaining rN examples are allotted to the validation subset. The estimation subset, denoted by $\mathscr{T}'$, is used to train a nested sequence of multilayer perceptrons, resulting in the hypotheses $\mathscr{F}_1, \mathscr{F}_2, ..., \mathscr{F}_n$ of increasing complexity. With $\mathscr{T}'$ made up of $(1-r)N$ examples, we consider values of W smaller than or equal to the corresponding fitting number $W_{\max}((1 - r)N)$.

The use of cross-validation results in the choice

$$\mathcal{F}_{cv} = \min_{k=1,2,\ldots,v} \{e_t''(\mathcal{F}_k)\} \tag{4.94}$$

where v corresponds to $W_v \leq W_{\max}((1-r)N)$, and $e_t''(\mathcal{F}_k)$ is the classification error produced by hypothesis $\mathcal{F}_k$ when it is tested on the validation subset $\mathcal{T}''$, consisting of rN examples.

The key issue is how to specify the parameter r that determines the split of the training sample $\mathcal{T}$ between the estimation subset $\mathcal{T}'$ and validation subset $\mathcal{T}''$. In a study described in Kearns (1996) involving an analytic treatment of this issue and supported with detailed computer simulations, several qualitative properties of the optimum r are identified:

- When the complexity of the target function, which defines the desired response d in terms of the input vector $\mathbf{x}$, is small compared with the sample size N, the performance of cross-validation is relatively insensitive to the choice of r.
- As the target function becomes more complex relative to the sample size N, the choice of optimum r has a more pronounced effect on cross-validation performance, and the value of the target function itself decreases.
- A single *fixed* value of r works *nearly* optimally for a wide range of target-function complexity.

On the basis of the results reported in Kearns (1996), a fixed value of r equal to 0.2 appears to be a sensible choice, which means that 80 percent of the training sample $\mathcal{T}$ is assigned to the estimation subset and the remaining 20 percent is assigned to the validation subset.

Early-Stopping Method of Training

Ordinarily, a multilayer perceptron trained with the back-propagation algorithm learns in stages, moving from the realization of fairly simple to more complex mapping functions as the training session progresses. This process is exemplified by the fact that in a typical situation, the mean-square error decreases with an increasing number of epochs used for training: It starts off at a large value, decreases rapidly, and then continues to decrease slowly as the network makes its way to a local minimum on the error surface. With good generalization as the goal, it is very difficult to figure out when it is best to stop training if we were to look at the learning curve for training all by itself. In particular, in light of what was said in Section 4.11 on generalization, it is possible for the network to end up overfitting the training data if the training session is not stopped at the right point.

We may identify the onset of overfitting through the use of cross-validation, for which the training data are split into an estimation subset and a validation subset. The estimation subset of examples is used to train the network in the usual way, except for a minor modification: The training session is stopped periodically (i.e., every so many epochs), and the network is tested on the validation subset after each period of training. More specifically, the periodic "estimation-followed-by-validation process" proceeds as follows:

- After a period of estimation (training)—every five epochs, for example—the synaptic weights and bias levels of the multilayer perceptron are all fixed, and the

network is operated in its forward mode. The validation error is thus measured for each example in the validation subset.

- When the validation phase is completed, the estimation (training) is resumed for another period, and the process is repeated.

This procedure is referred to as the *early-stopping method of training*, which is simple to understand and therefore widely used in practice.

Figure 4.17 shows conceptualized forms of two learning curves, one pertaining to measurements on the estimation subset and the other pertaining to the validation subset. Typically, the model does not do as well on the validation subset as it does on the estimation subset, on which its design was based. The *estimation learning curve* decreases monotonically for an increasing number of epochs in the usual manner. In contrast, the *validation learning curve* decreases monotonically to a minimum and then starts to increase as the training continues. When we look at the estimation learning curve, it may appear that we could do better by going beyond the minimum point on the validation learning curve. In reality, however, what the network is learning beyond this point is essentially noise contained in the training data. This heuristic suggests that the minimum point on the validation learning curve be used as a sensible criterion for stopping the training session.

However, a word of caution is in order here. In reality, the validation-sample error does *not* evolve over the number of epochs used for training as smoothly as the idealized curve shown in Fig. 4.17. Rather, the validation-sample error may exhibit few local minima of its own before it starts to increase with an increasing number of epochs. In such situations, a stopping criterion must be selected in some systematic manner. An empirical investigation on multilayer perceptrons carried out by Prechelt (1998) demonstrates experimentally that there is, in fact, a tradeoff between training time and generalization performance. Based on experimental results obtained therein on 1,296 training sessions, 12 different problems, and 24 different network architectures, it is concluded that, in the presence of two or more local minima, the selection of a "slower" stopping criterion (i.e., a criterion that stops later than other criteria) permits the attainment of a small improvement in generalization performance (typically, about 4 percent, on average) at the cost of a much longer training time (about a factor of four, on average).

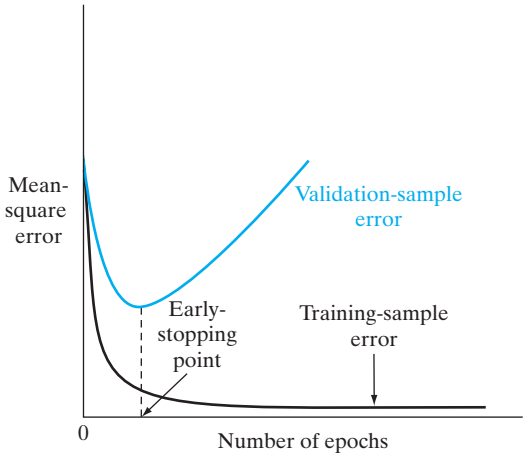


FIGURE 4.17 Illustration of the early-stopping rule based on cross-validation.

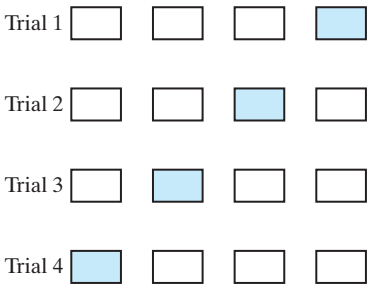


FIGURE 4.18 Illustration of the multifold method of cross-validation. For a given trial, the subset of data shaded in red is used to validate the model trained on the remaining data.

Variants of Cross-Validation

The approach to cross-validation just described is also referred to as the *holdout method.* There are other variants of cross-validation that find their own uses in practice, particularly when there is a scarcity of labeled examples. In such a situation, we may use *multifold cross-validation* by dividing the available set of N examples into K subsets, where $K > 1$; this procedure assumes that K is divisible into N. The model is trained on all the subsets except for one, and the validation error is measured by testing it on the subset that is left out. This procedure is repeated for a total of K trials, each time using a different subset for validation, as illustrated in Fig. 4.18 for $K = 4$. The performance of the model is assessed by averaging the squared error under validation over all the trials of the experiment. There is a disadvantage to multifold cross-validation: It may require an excessive amount of computation, since the model has to be trained K times, where $1 < K \leq N$.

When the available number of labeled examples, N, is severely limited, we may use the extreme form of multifold cross-validation known as the *leave-one-out method.* In this case, $N-1$ examples are used to train the model, and the model is validated by testing it on the example that is left out. The experiment is repeated for a total of N times, each time leaving out a different example for validation. The squared error under validation is then averaged over the N trials of the experiment.

4.14 COMPLEXITY REGULARIZATION AND NETWORK PRUNING

In designing a multilayer perceptron by whatever method, we are in effect building a nonlinear *model* of the physical phenomenon responsible for the generation of the input–output examples used to train the network. Insofar as the network design is statistical in nature, we need an appropriate tradeoff between reliability of the training data and goodness of the model (i.e., a method for solving the bias–variance dilemma discussed in Chapter 2). In the context of back-propagation learning, or any other supervised learning procedure for that matter, we may realize this tradeoff by minimizing the *total risk*, expressed as a function of the parameter vector $\mathbf{w}$, as follows:

$$R(\mathbf{w}) = \mathscr{E}_{\text{av}}(\mathbf{w}) + \lambda \mathscr{E}_c(\mathbf{w}) \tag{4.95}$$

The first term, $\mathscr{E}_{\text{av}}(\mathbf{w})$, is the standard *performance metric*, which depends on both the network (model) and the input data. In back-propagation learning, it is typically defined

as a mean-square error whose evaluation extends over the output neurons of the network and is carried out for all the training examples on an epoch-by-epoch basis, see Eq. (4.5). The second term, $\mathscr{E}_c(\mathbf{w})$, is the *complexity penalty*, where the notion of complexity is measured in terms of the network (weights) alone; its inclusion imposes on the solution prior knowledge that we may have on the models being considered. For the present discussion, it suffices to think of λ as a *regularization parameter*, which represents the relative importance of the complexity-penalty term with respect to the performance-metric term. When λ is zero, the back-propagation learning process is unconstrained, with the network being completely determined from the training examples. When λ is made infinitely large, on the other hand, the implication is that the constraint imposed by the complexity penalty is by itself sufficient to specify the network, which is another way of saying that the training examples are unreliable. In practical applications of complexity regularization, the regularization parameter λ is assigned a value somewhere between these two limiting cases. The subject of regularization theory is discussed in great detail in Chapter 7.

Weight-Decay Procedure

In a simplified, yet effective, form of complex regularization called the *weight-decay procedure* (Hinton, 1989), the complexity penalty term is defined as the squared norm of the weight vector $\mathbf{w}$ (i.e., all the free parameters) in the network, as shown by

$$\begin{aligned}\mathscr{E}_c(\mathbf{w}) &= \|\mathbf{w}\|^2 \\ &= \sum_{i \in \mathscr{C}_{\text{total}}} w_i^2\end{aligned} \tag{4.96}$$

where the set $\mathscr{C}_{\text{total}}$ refers to all the synaptic weights in the network. This procedure operates by forcing some of the synaptic weights in the network to take values close to zero, while permitting other weights to retain their relatively large values. Accordingly, the weights of the network are grouped roughly into two categories:

(i) weights that have a significant influence on the network's performance;

(ii) weights that have practically little or no influence on the network's performance.

The weights in the latter category are referred to as *excess weights*. In the absence of complexity regularization, these weights result in poor generalization by virtue of their high likelihood of taking on completely arbitrary values or causing the network to overfit the data in order to produce a slight reduction in the training error (Hush and Horne, 1993). The use of complexity regularization encourages the excess weights to assume values close to zero and thereby improve generalization.

Hessian-Based Network Pruning: Optimal Brain Surgeon

The basic idea of an analytic approach to network pruning is to use information on second-order derivatives of the error surface in order to make a trade-off between network complexity and training-error performance. In particular, a local model of the error surface is constructed for analytically predicting the effect of perturbations in synaptic weights. The starting point in the construction of such a model is the local

approximation of the cost function $\mathscr{E}_{\text{av}}$ by using a *Taylor series* about the operating point, described as

$$\mathscr{E}_{\text{av}}(\mathbf{w} + \Delta\mathbf{w}) = \mathscr{E}_{\text{av}}(\mathbf{w}) + \mathbf{g}^T(\mathbf{w})\Delta\mathbf{w} + \frac{1}{2}\Delta\mathbf{w}^T\mathbf{H}\Delta\mathbf{w} + O(\|\Delta\mathbf{w}\|^3) \tag{4.97}$$

where $\Delta\mathbf{w}$ is a perturbation applied to the operating point $\mathbf{w}$ and $\mathbf{g}(\mathbf{w})$ is the gradient vector evaluated at $\mathbf{w}$. The Hessian is also evaluated at the point $\mathbf{w}$, and therefore, to be correct, we should denote it by $\mathbf{H}(\mathbf{w})$. We have not done so in Eq. (4.97) merely to simplify the notation.

The requirement is to identify a set of parameters whose deletion from the multilayer perceptron will cause the least increase in the value of the cost function $\mathscr{E}_{\text{av}}$. To solve this problem in practical terms, we make the following approximations:

1. *Extremal Approximation.* We assume that parameters are deleted from the network only after the training process has converged (i.e., the network is fully trained). The implication of this assumption is that the parameters have a set of values corresponding to a local minimum or global minimum of the error surface. In such a case, the gradient vector $\mathbf{g}$ may be set equal to zero, and the term $\mathbf{g}^T\Delta\mathbf{w}$ on the right-hand side of Eq. (4.97) may therefore be ignored; otherwise, the saliency measures (defined later) will be invalid for the problem at hand.

2. *Quadratic Approximation.* We assume that the error surface around a local minimum or global minimum is "nearly quadratic." Hence, the higher-order terms in Eq. (4.97) may also be neglected.

Under these two assumptions, Eq. (4.97) is simplified as

$$\begin{aligned}\Delta\mathscr{E}_{\text{av}} &= \mathscr{E}(\mathbf{w} + \Delta\mathbf{w}) - \mathscr{E}(\mathbf{w}) \\ &= \frac{1}{2}\Delta\mathbf{w}^T\mathbf{H}\Delta\mathbf{w}\end{aligned} \tag{4.98}$$

Equation (4.98) provides the basis for the pruning procedure called *optimal brain surgeon* (OBS), which is due to Hassibi and Stork (1993).

The goal of OBS is to set one of the synaptic weights to zero in order to minimize the incremental increase in $\mathscr{E}_{\text{av}}$ given in Eq. (4.98). Let $w_i(n)$ denote this particular synaptic weight. The elimination of this weight is equivalent to the condition

$$\mathbf{1}_i^T\Delta\mathbf{w} + w_i = 0 \tag{4.99}$$

where $\mathbf{1}_i$ is the *unit vector* whose elements are all zero, except for the ith element, which is equal to unity. We may now restate the goal of OBS as follows:

> *Minimize the quadratic form $\frac{1}{2}\Delta\mathbf{w}^T\mathbf{H}\Delta\mathbf{w}$ with respect to the incremental change in the weight vector, $\Delta\mathbf{w}$, subject to the constraint that $\mathbf{1}_i^T\Delta\mathbf{w} + w_i$ is zero, and then minimize the result with respect to the index i.*

There are two levels of minimization going on here. One minimization is over the synaptic-weight vectors that remain after the ith weight vector is set equal to zero. The second minimization is over which particular vector is pruned.

To solve this constrained-optimization problem, we first construct the *Lagrangian*

$$S = \frac{1}{2}\Delta\mathbf{w}^T\mathbf{H}\Delta\mathbf{w} - \lambda(\mathbf{1}_i^T\Delta\mathbf{w} + w_i) \tag{4.100}$$

where λ is the *Lagrange multiplier*. Then, taking the derivative of the Lagrangian S with respect to $\Delta\mathbf{w}$, applying the constraint of Eq. (4.99), and using matrix inversion, we find that the optimum change in the weight vector $\mathbf{w}$ is given by

$$\Delta\mathbf{w} = -\frac{w_i}{[\mathbf{H}^{-1}]_{i,i}}\mathbf{H}^{-1}\mathbf{1}_i \tag{4.101}$$

and the corresponding optimum value of the Lagrangian S for element w_i is

$$S_i = \frac{w_i^2}{2[\mathbf{H}^{-1}]_{i,i}} \tag{4.102}$$

where $\mathbf{H}^{-1}$ is the inverse of the Hessian $\mathbf{H}$, and $[\mathbf{H}^{-1}]_{i,i}$ is the ii-th element of this inverse matrix. The Lagrangian S_i optimized with respect to $\Delta\mathbf{w}$, subject to the constraint that the ith synaptic weight w_i be eliminated, is called the *saliency* of w_i. In effect, the saliency S_i represents the increase in the mean-square error (performance measure) that results from the deletion of w_i. Note that the saliency S_i, is proportional to w_i^2. Thus, small weights have a small effect on the mean-square error. However, from Eq. (4.102), we see that the saliency S_i, is also inversely proportional to the diagonal elements of the inverse Hessian. Thus, if $[\mathbf{H}^{-1}]_{i,i}$ is small, then even small weights may have a substantial effect on the mean-square error.

In the OBS procedure, the weight corresponding to the smallest saliency is the one selected for deletion. Moreover, the corresponding optimal changes in the remainder of the weights are given in Eq. (4.101), which show that they should be updated along the direction of the i-th column of the inverse of the Hessian.

According to Hassibi and coworkers commenting on some benchmark problems, the OBS procedure resulted in smaller networks than those obtained using the weight-decay procedure. It is also reported that as a result of applying the OBS procedure to the NETtalk multilayer perceptron, involving a single hidden layer and well over 18,000 weights, the network was pruned to a mere 1,560 weights, a dramatic reduction in the size of the network. NETtalk, due to Sejnowski and Rosenberg (1987), is described in Section 4.18.

Computing the inverse Hessian. The inverse Hessian $\mathbf{H}^{-1}$ is fundamental to the formulation of the OBS procedure. When the number of free parameters, W, in the network is large, the problem of computing $\mathbf{H}^{-1}$ may be intractable. In what follows, we describe a manageable procedure for computing $\mathbf{H}^{-1}$, assuming that the multilayer perceptron is fully trained to a local minimum on the error surface (Hassibi and Stork, 1993).

To simplify the presentation, suppose that the multilayer perceptron has a single output neuron. Then, for a given training sample, we may redefine the cost function of Eq. (4.5) as

$$\mathscr{E}_{\text{av}}(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N}(d(n) - o(n))^2$$

where $o(n)$ is the actual output of the network on the presentation of the nth example, $d(n)$ is the corresponding desired response, and N is the total number of examples in the training sample. The output $o(n)$ may itself be expressed as

$$o(n) = F(\mathbf{w}, \mathbf{x})$$

where F is the input–output mapping function realized by the multilayer perceptron, $\mathbf{x}$ is the input vector, and $\mathbf{w}$ is the synaptic-weight vector of the network. The first derivative of $\mathscr{E}_{\text{av}}$ with respect to $\mathbf{w}$ is therefore

$$\frac{\partial \mathscr{E}_{\text{av}}}{\partial \mathbf{w}} = -\frac{1}{N}\sum_{n=1}^{N}\frac{\partial F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}}(d(n) - o(n)) \tag{4.103}$$

and the second derivative of $\mathscr{E}_{\text{av}}$ with respect to $\mathbf{w}$ or the Hessian is

$$\begin{aligned}\mathbf{H}(N) &= \frac{\partial^2 \mathscr{E}_{\text{av}}}{\partial \mathbf{w}^2} \\ &= \frac{1}{N}\sum_{n=1}^{N}\left\{\left(\frac{\partial F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}}\right)\left(\frac{\partial F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}}\right)^T \right. \\ &\quad \left. - \frac{\partial^2 F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}^2}(d(n) - o(n))\right\}\end{aligned} \tag{4.104}$$

where we have emphasized the dependence of the Hessian on the size of the training sample, N.

Under the assumption that the network is fully trained—that is, the cost function $\mathscr{E}_{\text{av}}$ has been adjusted to a local minimum on the error surface—it is reasonable to say that $o(n)$ is close to $d(n)$. Under this condition, we may ignore the second term and approximate Eq. (4.104) as

$$\mathbf{H}(N) \approx \frac{1}{N}\sum_{n=1}^{N}\left(\frac{\partial F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}}\right)\left(\frac{\partial F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}}\right)^T \tag{4.105}$$

To simplify the notation, define the W-by-1 vector

$$\boldsymbol{\xi}(n) = \frac{1}{\sqrt{N}}\frac{\partial F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}} \tag{4.106}$$

which may be computed using the procedure described in Section 4.8. We may then rewrite Eq. (4.105) in the form of a recursion as follows:

$$\begin{aligned}\mathbf{H}(n) &= \sum_{k=1}^{n}\boldsymbol{\xi}(k)\boldsymbol{\xi}^T(k) \\ &= \mathbf{H}(n-1) + \boldsymbol{\xi}(n)\boldsymbol{\xi}^T(n), \qquad n = 1, 2, ..., N\end{aligned} \tag{4.107}$$

This recursion is in the right form for application of the so-called *matrix inversion lemma*, also known as *Woodbury's equality*.

Let $\mathbf{A}$ and $\mathbf{B}$ denote two positive-definite matrices related by

$$\mathbf{A} = \mathbf{B}^{-1} + \mathbf{CDC}^T$$

where **C** and **D** are two other matrices. According to the matrix inversion lemma, the inverse of matrix **A** is defined by

$$\mathbf{A}^{-1} = \mathbf{B} - \mathbf{BC}(\mathbf{D} + \mathbf{C}^T\mathbf{BC})^{-1}\mathbf{C}^T\mathbf{B}$$

For the problem described in Eq. (4.107) we have

$$\begin{aligned}\mathbf{A} &= \mathbf{H}(n)\\ \mathbf{B}^{-1} &= \mathbf{H}(n-1)\\ \mathbf{C} &= \boldsymbol{\xi}(n)\\ \mathbf{D} &= 1\end{aligned}$$

Application of the matrix inversion lemma therefore yields the desired formula for recursive computation of the inverse Hessian:

$$\mathbf{H}^{-1}(n) = \mathbf{H}^{-1}(n-1) - \frac{\mathbf{H}^{-1}(n-1)\boldsymbol{\xi}(n)\boldsymbol{\xi}^T(n)\mathbf{H}^{-1}(n-1)}{1 + \boldsymbol{\xi}^T(n)\mathbf{H}^{-1}(n-1)\boldsymbol{\xi}(n)} \tag{4.108}$$

Note that the denominator in Eq. (4.108) is a scalar; it is therefore straightforward to calculate its reciprocal. Thus, given the past value of the inverse Hessian, $\mathbf{H}^{-1}(n-1)$, we may compute its updated value $\mathbf{H}^{-1}(n)$ on the presentation of the nth example, represented by the vector $\boldsymbol{\xi}(n)$. This recursive computation is continued until the entire set of N examples has been accounted for. To initialize the algorithm, we need to make $\mathbf{H}^{-1}(0)$ large, since it is being constantly reduced according to Eq. (4.108). This requirement is satisfied by setting

$$\mathbf{H}^{-1}(0) = \delta^{-1}\mathbf{I}$$

where δ is a small positive number and **I** is the identity matrix. This form of initialization assures that $\mathbf{H}^{-1}(n)$ is always positive definite. The effect of δ becomes progressively smaller as more and more examples are presented to the network.

A summary of the optimal-brain-surgeon algorithm is presented in Table 4.1.

4.15 VIRTUES AND LIMITATIONS OF BACK-PROPAGATION LEARNING

First and foremost, it should be understood that the back-propagation algorithm is *not* an algorithm intended for the optimum design of a multilayer perceptron. Rather, the correct way to describe it is to say:

> *The back-propagation algorithm is a computationally efficient technique for computing the gradients (i.e., first-order derivatives) of the cost function $\mathscr{E}(w)$, expressed as a function of the adjustable parameters (synaptic weights and bias terms) that characterize the multilayer perceptron.*

The computational power of the algorithm is derived from two distinct properties:

1. The back-propagation algorithm is *simple to compute locally.*
2. It performs *stochastic gradient descent* in weight space, when the algorithm is implemented in its on-line (sequential) mode of learning.

TABLE 4.1 Summary of the Optimal-Brain-Surgeon Algorithm

1. Train the given multilayer perceptron to minimum mean-square error.
2. Use the procedure described in Section 4.8 to compute the vector

$$\boldsymbol{\xi}(n) = \frac{1}{\sqrt{N}} \frac{\partial F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}}$$

where $F(\mathbf{w}, \mathbf{x}(n))$ is the input–output mapping realized by the multilayer perceptron with an overall weight vector $\mathbf{w}$, and $\mathbf{x}(n)$ is the input vector.

3. Use the recursion in Eq. (4.108) to compute the inverse Hessian $\mathbf{H}^{-1}$.
4. Find the i that corresponds to the smallest saliency

$$S_i = \frac{w_i^2}{2[\mathbf{H}^{-1}]_{i,i}}$$

where $[\mathbf{H}^{-1}]_{i,i}$ is the (i, i)th element of $\mathbf{H}^{-1}$. If the saliency S_i is much smaller than the mean-square error $\mathscr{E}_{\text{av}}$, then delete the synaptic weight w_i and proceed to step 5. Otherwise, go to step 6.

5. Update all the synaptic weights in the network by applying the adjustment

$$\Delta \mathbf{w} = -\frac{w_i}{[\mathbf{H}^{-1}]_{i,i}} \mathbf{H}^{-1} \mathbf{1}_i$$

Go to step 2.

6. Stop the computation when no more weights can be deleted from the network without a large increase in the mean-square error. (It may be desirable to retrain the network at this point).

Connectionism

The back-propagation algorithm is an example of a *connectionist paradigm* that relies on local computations to discover the information-processing capabilities of neural networks. This form of computational restriction is referred to as the *locality constraint*, in the sense that the computation performed by each neuron in the network is influenced solely by those other neurons that are in physical contact with it. The use of local computations in the design of (artificial) neural networks is usually advocated for three principal reasons:

1. Neural networks that perform local computations are often held up as *metaphors* for biological neural networks.
2. The use of local computations permits a graceful degradation in performance caused by hardware errors and therefore provides the basis for a *fault-tolerant* network design.
3. Local computations favor the use of *parallel architectures* as an efficient method for the implementation of neural networks.

Replicator (Identity) Mapping

The hidden neurons of a multilayer perceptron trained with the back-propagation algorithm play a critical role as feature detectors. A novel way in which this important property of the multilayer perceptron can be exploited is in its use as a *replicator* or *identity map* (Rumelhart et al., 1986b; Cottrel et al., 1987). Figure 4.19 illustrates

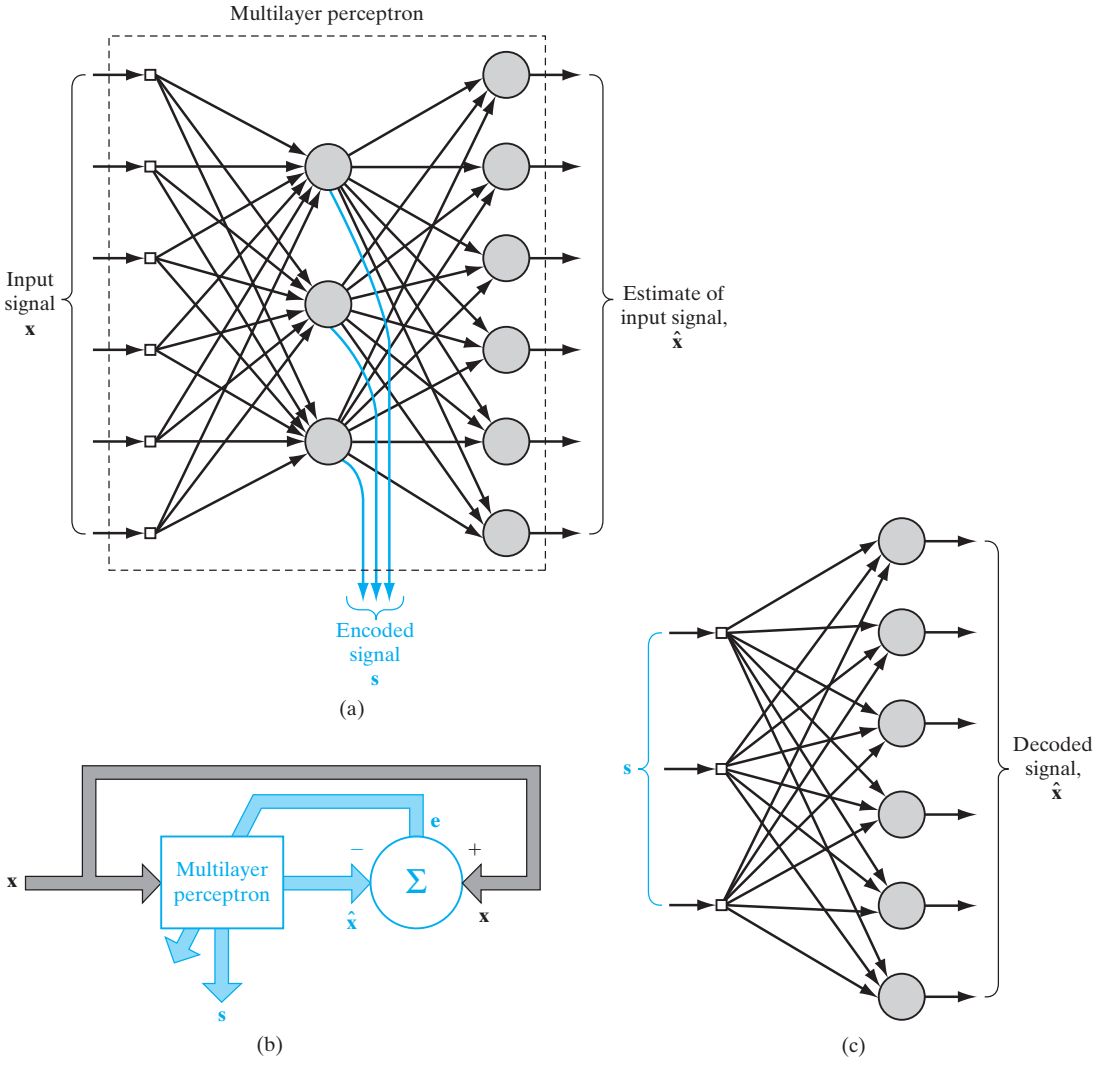


FIGURE 4.19 (a) Replicator network (identity map) with a single hidden layer used as an encoder. (b) Block diagram for the supervised training of the replicator network. (c) Part of the replicator network used as a decoder.

how this can be accomplished for the case of a multilayer perceptron using a single hidden layer. The network layout satisfies the following structural requirements, as illustrated in Fig. 4.19a:

- The input and output layers have the same size, m.
- The size of the hidden layer, M, is smaller than m.
- The network is fully connected.

A given pattern $\mathbf{x}$ is simultaneously applied to the input layer as the stimulus and to the output layer as the desired response. The actual response of the output layer, $\hat{\mathbf{x}}$, is

intended to be an "estimate" of $\mathbf{x}$. The network is trained using the back-propagation algorithm in the usual way, with the estimation error vector $(\mathbf{x} - \hat{\mathbf{x}})$ treated as the error signal, as illustrated in Fig. 4.19b. The training is performed in an *unsupervised* manner (i.e., without the need for a teacher). By virtue of the special structure built into the design of the multilayer perceptron, the network is *constrained* to perform identity mapping through its hidden layer. An *encoded* version of the input pattern, denoted by $\mathbf{s}$, is produced at the output of the hidden layer, as indicated in Fig. 4.19a. In effect, the fully trained multilayer perceptron performs the role of an "encoder." To reconstruct an estimate $\hat{\mathbf{x}}$ of the original input pattern $\mathbf{x}$ (i.e., to perform *decoding*), we apply the encoded signal to the hidden layer of the replicator network, as illustrated in Fig. 4.19c. In effect, this latter network performs the role of a "decoder." The smaller we make the size M of the hidden layer compared with the size m of the input–output layer, the more effective the configuration of Fig. 4.19a will be as a *data-compression system*.[10]

Function Approximation

A multilayer perceptron trained with the back-propagation algorithm manifests itself as a *nested sigmoidal structure*, written for the case of a single output in the compact form

$$F(\mathbf{x}, \mathbf{w}) = \varphi\left(\sum_k w_{ok}\varphi\left(\sum_j w_{kj}\varphi\left(\cdots\varphi\left(\sum_i w_{li}x_i\right)\right)\right)\right) \tag{4.109}$$

where $\varphi(\cdot)$ is a sigmoid activation function; w_{ok} is the synaptic weight from neuron k in the last hidden layer to the single output neuron o, and so on for the other synaptic weights; and x_i is the ith element of the input vector $\mathbf{x}$. The weight vector $\mathbf{w}$ denotes the entire set of synaptic weights ordered by layer, then neurons in a layer, and then synapses in a neuron. The scheme of nested nonlinear functions described in Eq. (4.109) is unusual in classical approximation theory. It is a *universal approximator*, as discussed in Section 4.12.

Computational Efficiency

The *computational complexity* of an algorithm is usually measured in terms of the number of multiplications, additions, and storage requirement involved in its implementation. A learning algorithm is said to be *computationally efficient* when its computational complexity is *polynomial* in the number of adjustable parameters that are to be updated from one iteration to the next. On this basis, it can be said that the back-propagation algorithm is computationally efficient, as stated in the summarizing description at the beginning of this section. Specifically, in using the algorithm to train a multilayer perceptron containing a total of W synaptic weights (including biases), its computational complexity is linear in W. This important property of the back-propagation algorithm can be readily verified by examining the computations involved in performing the forward and backward passes summarized in Section 4.4. In the forward pass, the only computations involving the synaptic weights are those that pertain to the induced local fields of the various neurons in the network. Here, we see from Eq. (4.44) that these computations are all linear in the synaptic weights of the network. In the backward pass, the only computations involving the synaptic weights are those that pertain to (1) the local gradients of the hidden neurons, and (2) the updating of the synaptic weights themselves, as shown in Eqs. (4.46) and (4.47), respectively. Here again, we also see that these computations are

all linear in the synaptic weights of the network. The conclusion is therefore that the computational complexity of the back-propagation algorithm is *linear* in W; that is, it is $O(W)$.

Sensitivity Analysis

Another computational benefit gained from the use of back-propagation learning is the efficient manner in which we can carry out a sensitivity analysis of the input–output mapping realized by the algorithm. The *sensitivity* of an input–output mapping function F with respect to a parameter of the function, denoted by ω, is defined by

$$S_{\omega}^{F} = \frac{\partial F/F}{\partial \omega/\omega} \tag{4.110}$$

Consider then a multilayer perceptron trained with the back-propagation algorithm. Let the function $F(\mathbf{w})$ be the input–output mapping realized by this network; $\mathbf{w}$ denotes the vector of all synaptic weights (including biases) contained in the network. In Section 4.8, we showed that the partial derivatives of the function $F(\mathbf{w})$ with respect to all the elements of the weight vector $\mathbf{w}$ can be computed efficiently. In particular, we see that the complexity involved in computing each of these partial derivatives is linear in W, the total number of weights contained in the network. This linearity holds regardless of where the synaptic weight in question appears in the chain of computations.

Robustness

In Chapter 3, we pointed out that the LMS algorithm is robust in the sense that disturbances with small energy can give rise only to small estimation errors. If the underlying observation model is linear, the LMS algorithm is an H^{∞}-optimal filter (Hassibi et al., 1993, 1996). What this means is that the LMS algorithm minimizes the *maximum energy gain* from the disturbances to the estimation errors.

If, on the other hand, the underlying observation model is nonlinear, Hassibi and Kailath (1995) have shown that the back-propagation algorithm is a *locally* H^{∞}-optimal filter. The term "local" means that the initial value of the weight vector used in the back-propagation algorithm is sufficiently close to the optimum value $\mathbf{w}^*$ of the weight vector to ensure that the algorithm does not get trapped in a poor local minimum. In conceptual terms, it is satisfying to see that the LMS and back-propagation algorithms belong to the same class of H^{∞}-optimal filters.

Convergence

The back-propagation algorithm uses an "instantaneous estimate" for the gradient of the error surface in weight space. The algorithm is therefore *stochastic* in nature; that is, it has a tendency to zigzag its way about the true direction to a minimum on the error surface. Indeed, back-propagation learning is an application of a statistical method known as *stochastic approximation* that was originally proposed by Robbins and Monro (1951). Consequently, it tends to converge slowly. We may identify two fundamental causes for this property (Jacobs, 1988):

1. The error surface is fairly flat along a weight dimension, which means that the derivative of the error surface with respect to that weight is small in magnitude. In such

a situation, the adjustment applied to the weight is small, and consequently many iterations of the algorithm may be required to produce a significant reduction in the error performance of the network. Alternatively, the error surface is highly curved along a weight dimension, in which case the derivative of the error surface with respect to that weight is large in magnitude. In this second situation, the adjustment applied to the weight is large, which may cause the algorithm to overshoot the minimum of the error surface.

2. The direction of the negative gradient vector (i.e., the negative derivative of the cost function with respect to the vector of weights) may point away from the minimum of the error surface: hence, the adjustments applied to the weights may induce the algorithm to move in the wrong direction.

To avoid the slow rate of convergence of the back-propagation algorithm used to train a multilayer perceptron, we may opt for the optimally annealed on-line learning algorithm described in Section 4.10.

Local Minima

Another peculiarity of the error surface that affects the performance of the back-propagation algorithm is the presence of *local minima* (i.e., isolated valleys) in addition to global minima; in general, it is difficult to determine the numbers of local and global minima. Since back-propagation learning is basically a hill-climbing technique, it runs the risk of being trapped in a local minimum where every small change in synaptic weights increases the cost function. But somewhere else in the weight space, there exists another set of synaptic weights for which the cost function is smaller than the local minimum in which the network is stuck. It is clearly undesirable to have the learning process terminate at a local minimum, especially if it is located far above a global minimum.

Scaling

In principle, neural networks such as multilayer perceptrons trained with the back-propagation algorithm have the potential to be universal computing machines. However, for that potential to be fully realized, we have to overcome the *scaling problem*, which addresses the issue of how well the network behaves (e.g., as measured by the time required for training or the best generalization performance attainable) as the computational task increases in size and complexity. Among the many possible ways of measuring the size or complexity of a computational task, the predicate order defined by Minsky and Papert (1969, 1988) provides the most useful and important measure.

To explain what we mean by a predicate, let $\psi(X)$ denote a function that can have only two values. Ordinarily, we think of the two values of $\psi(X)$ as 0 and 1. But by taking the values to be FALSE or TRUE, we may think of $\psi(X)$ as a *predicate*—that is, a variable statement whose falsity or truth depends on the choice of argument X. For example, we may write

$$\psi_{\text{CIRCLE}}(X) = \begin{cases} 1 & \text{if the figure } X \text{ is a circle} \\ 0 & \text{if the figure } X \text{ is not a circle} \end{cases}$$

Using the idea of a predicate, Tesauro and Janssens (1988) performed an empirical study involving the use of a multilayer perceptron trained with the back-propagation

algorithm to learn to compute the parity function. The *parity function* is a Boolean predicate defined by

$$\psi_{\text{PARITY}}(X) = \begin{cases} 1 & \text{if } |X| \text{ is an odd number} \\ 0 & \text{otherwise} \end{cases}$$

and whose order is equal to the number of inputs. The experiments performed by Tesauro and Janssens appear to show that the time required for the network to learn to compute the parity function scales exponentially with the number of inputs (i.e., the predicate order of the computation), and that projections of the use of the back-propagation algorithm to learn arbitrarily complicated functions may be overly optimistic.

It is generally agreed that it is inadvisable for a multilayer perceptron to be fully connected. In this context, we may therefore raise the following question: Given that a multilayer perceptron should not be fully connected, how should the synaptic connections of the network be allocated? This question is of no major concern in the case of small-scale applications, but it is certainly crucial to the successful application of back-propagation learning for solving large-scale, real-world problems.

One effective method of alleviating the scaling problem is to develop insight into the problem at hand (possibly through neurobiological analogy) and use it to put ingenuity into the architectural design of the multilayer perceptron. Specifically, the network architecture and the constraints imposed on synaptic weights of the network should be designed so as to incorporate prior information about the task into the makeup of the network. This design strategy is illustrated in Section 4.17 for the optical character recognition problem.

4.16 SUPERVISED LEARNING VIEWED AS AN OPTIMIZATION PROBLEM

In this section, we take a viewpoint on supervised learning that is quite different from that pursued in previous sections of the chapter. Specifically, we view the supervised training of a multilayer perceptron as a problem in *numerical optimization*. In this context, we first point out that the error surface of a multilayer perceptron with supervised learning is a nonlinear function of a weight vector $\mathbf{w}$; in the case of a multilayer perceptron, $\mathbf{w}$ represents the synaptic weight of the network arranged in some orderly fashion. Let $\mathscr{E}_{\text{av}}(\mathbf{w})$ denote the cost function, averaged over the training sample. Using the Taylor series, we may expand $\mathscr{E}_{\text{av}}(\mathbf{w})$ about the current operating point on the error surface as in Eq. (4.97), reproduced here in the form:

$$\begin{aligned}\mathscr{E}_{\text{av}}(\mathbf{w}(n) + \Delta\mathbf{w}(n)) = \mathscr{E}_{\text{av}}(\mathbf{w}(n)) + \mathbf{g}^T(n)\Delta\mathbf{w}(n) + \frac{1}{2}\Delta\mathbf{w}^T(n)\mathbf{H}(n)\Delta\mathbf{w}(n) \\ + \text{(third- and higher-order terms)}\end{aligned} \tag{4.111}$$

where $\mathbf{g}(n)$ is the local *gradient vector*, defined by

$$\mathbf{g}(n) = \left.\frac{\partial \mathscr{E}_{\text{av}}(\mathbf{w})}{\partial \mathbf{w}}\right|_{\mathbf{w}=\mathbf{w}(n)} \tag{4.112}$$

The matrix $\mathbf{H}(n)$ is the local *Hessian* representing "curvature" of the error performance surface, defined by

$$\mathbf{H}(n) = \left.\frac{\partial^2 \mathscr{E}_{\text{av}}(\mathbf{w})}{\partial \mathbf{w}^2}\right|_{\mathbf{w}=\mathbf{w}(n)} \tag{4.113}$$

The use of an ensemble-averaged cost function $\mathscr{E}_{\text{av}}(\mathbf{w})$ presumes a *batch* mode of learning.

In the steepest-descent method, exemplified by the back-propagation algorithm, the adjustment $\Delta\mathbf{w}(n)$ applied to the synaptic weight vector $\mathbf{w}(n)$ is defined by

$$\Delta\mathbf{w}(n) = -\eta\mathbf{g}(n) \tag{4.114}$$

where η is a fixed learning-rate parameter. In effect, the steepest-descent method operates on the basis of a *liner approximation* of the cost function in the local neighborhood of the operating point $\mathbf{w}(n)$. In so doing, it relies on the gradient vector $\mathbf{g}(n)$ as the only source of local *first-order* information about the error surface. This restriction has a beneficial effect: simplicity of implementation. Unfortunately, it also has a detrimental effect: a slow rate of convergence, which can be excruciating, particularly in the case of large-scale problems. The inclusion of the momentum term in the update equation for the synaptic weight vector is a crude attempt at using second-order information about the error surface, which is of some help. However, its use makes the training process more delicate to manage by adding one more item to the list of parameters that have to be "tuned" by the designer.

In order to produce a significant improvement in the convergence performance of a multilayer perceptron (compared with back-propagation learning), we have to use *higher-order information* in the training process. We may do so by invoking a *quadratic approximation* of the error surface around the current point $\mathbf{w}(n)$. We then find from Eq. (4.111) that the optimum value of the adjustment $\Delta\mathbf{w}(n)$ applied to the synaptic weight vector $\mathbf{w}(n)$ is given by

$$\Delta\mathbf{w}^*(n) = \mathbf{H}^{-1}(n)\mathbf{g}(n) \tag{4.115}$$

where $\mathbf{H}^{-1}(n)$ is the inverse of the Hessian $\mathbf{H}(n)$, assuming that it exists. Equation (4.115) is the essence of *Newton's method*. If the cost function $\mathscr{E}_{\text{av}}(\mathbf{w})$ is quadratic (i.e., the third- and higher-order terms in Eq. (4.109) are zero), Newton's method converges to the optimum solution in one iteration. However, the practical application of Newton's method to the supervised training of a multilayer perceptron is handicapped by three factors:

- **(i)** Newton's method requires calculation of the inverse Hessian $\mathbf{H}^{-1}(n)$, which can be computationally expensive.
- **(ii)** For $\mathbf{H}^{-1}(n)$ to be computable, $\mathbf{H}(n)$ has to be nonsingular. In the case where $\mathbf{H}(n)$ is positive definite, the error surface around the current point $\mathbf{w}(n)$ is describable by a "convex bowl." Unfortunately, there is no guarantee that the Hessian of the error surface of a multilayer perceptron will always fit this description. Moreover, there is the potential problem of the Hessian being rank deficient (i.e., not all the

columns of **H** are linearly independent), which results from the intrinsically ill-conditioned nature of supervised-learning problems (Saarinen et al., 1992); this factor only makes the computational task more difficult.

(iii) When the cost function $\mathscr{E}_{av}(\mathbf{w})$ is nonquadratic, there is no guarantee for convergence of Newton's method, which makes it unsuitable for the training of a multilayer perceptron.

To overcome some of these difficulties, we may use a *quasi-Newton method*, which requires only an estimate of the gradient vector **g**. This modification of Newton's method maintains a positive-definite estimate of the inverse matrix $\mathbf{H}^{-1}$ directly without matrix inversion. By using such an estimate, a quasi-Newton method is assured of going downhill on the error surface. However, we still have a computational complexity that is $O(W^2)$, where W is the size of weight vector **w**. Quasi-Newton methods are therefore computationally impractical, except for in the training of very small-scale neural networks. A description of quasi-Newton methods is presented later in the section.

Another class of second-order optimization methods includes the conjugate-gradient method, which may be regarded as being somewhat intermediate between the method of steepest descent and Newton's method. Use of the conjugate-gradient method is motivated by the desire to accelerate the typically slow rate of convergence experienced with the method of steepest descent, while avoiding the computational requirements associated with the evaluation, storage, and inversion of the Hessian in Newton's method.

Conjugate-Gradient Method[11]

The conjugate-gradient method belongs to a class of second-order optimization methods known collectively as *conjugate-direction methods*. We begin the discussion of these methods by considering the minimization of the *quadratic function*

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} + c \tag{4.116}$$

where **x** is a W-by-1 parameter vector; **A** is a W-by-W symmetric, positive-definite matrix; **b** is a W-by-1 vector; and c is a scalar. Minimization of the quadratic function $f(\mathbf{x})$ is achieved by assigning to **x** the unique value

$$\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b} \tag{4.117}$$

Thus, minimizing $f(\mathbf{x})$ and solving the linear system of equations $\mathbf{A}\mathbf{x}^* = \mathbf{b}$ are equivalent problems.

Given the matrix **A**, we say that a set of nonzero vectors $\mathbf{s}(0), \mathbf{s}(1), \ldots, \mathbf{s}(W-1)$ is **A**-*conjugate* (i.e., noninterfering with each other in the context of matrix **A**) if the following condition is satisfied:

$$\mathbf{s}^T(n)\mathbf{A}\mathbf{s}(j) = 0 \qquad \text{for all } n \text{ and } j \text{ such that } n \neq j \tag{4.118}$$

If **A** is equal to the identity matrix, conjugacy is equivalent to the usual notion of orthogonality.

EXAMPLE 1 Interpretation of A-conjugate vectors

For an interpretation of **A**-conjugate vectors, consider the situation described in Fig. 4.20a, pertaining to a two-dimensional problem. The elliptic locus shown in this figure corresponds to a plot of Eq. (4.116) for

$$\mathbf{x} = [x_0, x_1]^T$$

at some constant value assigned to the quadratic function $f(\mathbf{x})$. Figure 4.20a also includes a pair of direction vectors that are conjugate with respect to the matrix **A**. Suppose that we define a new parameter vector **v** related to **x** by the transformation

$$\mathbf{v} = \mathbf{A}^{1/2}\mathbf{x}$$

where $\mathbf{A}^{1/2}$ is the square root of **A**. Then the elliptic locus of Fig. 4.20a is transformed into a circular locus, as shown in Fig. 4.20b. Correspondingly, the pair of **A**-conjugate direction vectors in Fig. 4.20a is transformed into a pair of orthogonal direction vectors in Fig. 4.20b. ■

An important property of **A**-conjugate vectors is that they are *linearly independent.* We prove this property by contradiction. Let one of these vectors—say, $\mathbf{s}(0)$—be expressed as a linear combination of the remaining $W-1$ vectors as follows:

$$\mathbf{s}(0) = \sum_{j=1}^{W-1} \alpha_j \mathbf{s}(j)$$

Multiplying by **A** and then taking the inner product of $\mathbf{As}(0)$ with $\mathbf{s}(0)$ yields

$$\mathbf{s}^T(0)\mathbf{As}(0) = \sum_{j=1}^{W-1} \alpha_j \mathbf{s}^T(0)\mathbf{As}(j) = 0$$

However, it is impossible for the quadratic form $\mathbf{s}^T(0)\mathbf{As}(0)$ to be zero, for two reasons: The matrix **A** is positive definite by assumption, and the vector $\mathbf{s}(0)$ is nonzero by definition. It follows therefore that the **A**-conjugate vectors $\mathbf{s}(0), \mathbf{s}(1), \ldots, \mathbf{s}(W-1)$ cannot be linearly dependent; that is, they must be linearly independent.

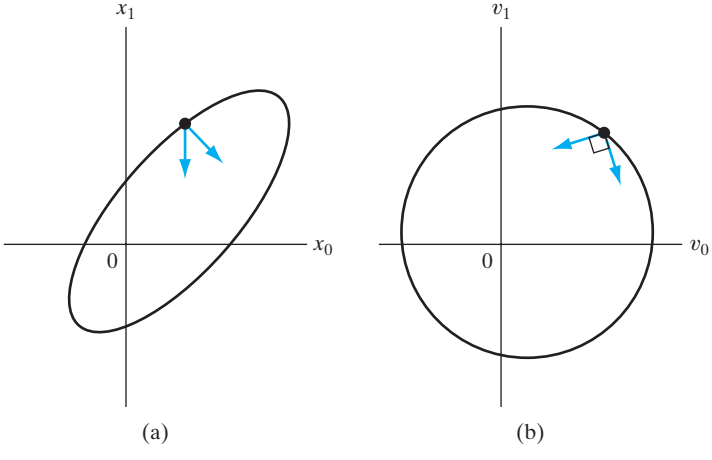


FIGURE 4.20 Interpretation of **A**-conjugate vectors. (a) Elliptic locus in two-dimensional weight space. (b) Transformation of the elliptic locus into a circular locus.

For a given set of **A**-conjugate vectors $\mathbf{s}(0), \mathbf{s}(1), ..., \mathbf{s}(W-1)$, the corresponding *conjugate-direction method* for unconstrained minimization of the quadratic error function $f(\mathbf{x})$ is defined by

$$\mathbf{x}(n+1) = \mathbf{x}(n) + \eta(n)\mathbf{s}(n), \qquad n = 0, 1, ..., W-1 \tag{4.119}$$

where $\mathbf{x}(0)$ is an arbitrary starting vector and $\eta(n)$ is a scalar defined by

$$f(\mathbf{x}(n) + \eta(n)\mathbf{s}(n)) = \min_{\eta} f(\mathbf{x}(n) + \eta\mathbf{s}(n)) \tag{4.120}$$

(Fletcher, 1987; Bertsekas, 1995). The procedure of choosing η so as to minimize the function $f(\mathbf{x}(n) + \eta\mathbf{s}(n))$ for some fixed n is referred to as a *line search*, which represents a one-dimensional minimization problem.

In light of Eqs. (4.118), (4.119) and (4.120), we now offer some observations:

1. Since the **A**-conjugate vectors $\mathbf{s}(0), \mathbf{s}(1), ..., \mathbf{s}(W-1)$ are linearly independent, they form a basis that spans the vector space of $\mathbf{w}$.
2. The update equation (4.119) and the line minimization of Eq. (4.120) lead to the same formula for the learning-rate parameter, namely,

$$\eta(n) = -\frac{\mathbf{s}^T(n)\mathbf{A}\mathbf{e}(n)}{\mathbf{s}^T(n)\mathbf{A}\mathbf{s}(n)}, \qquad n = 0, 1, ..., W-1 \tag{4.121}$$

   where $\mathbf{e}(n)$ is the *error vector* defined by

$$\mathbf{e}(n) = \mathbf{x}(n) - \mathbf{x}^* \tag{4.122}$$

3. Starting from an arbitrary point $\mathbf{x}(0)$, the conjugate-direction method is guaranteed to find the optimum solution $\mathbf{x}^*$ of the quadratic equation $f(\mathbf{x}) = 0$ in at most W iterations.

The principal property of the conjugate-direction method is described in the following statement (Fletcher, 1987; Bertsekas, 1995):

> *At successive iterations, the conjugate-direction method minimizes the quadratic function $f(\mathbf{x})$ over a progressively expanding linear vector space that eventually includes the global minimum of $f(\mathbf{x})$.*

In particular, for each iteration n, the iterate $\mathbf{x}(n+1)$ minimizes the function $f(\mathbf{x})$ over a linear vector space $\mathcal{D}_n$ that passes through some arbitrary point $\mathbf{x}(0)$ and is spanned by the **A**-conjugate vectors $\mathbf{s}(0), \mathbf{s}(1), ..., \mathbf{s}(n)$, as shown by

$$\mathbf{x}(n+1) = \arg\min_{\mathbf{x} \in \mathcal{D}_n} f(\mathbf{x}) \tag{4.123}$$

where the space $\mathcal{D}_n$ is defined by

$$\mathcal{D}_n = \left\{ \mathbf{x}(n) \,|\, \mathbf{x}(n) = \mathbf{x}(0) + \sum_{j=0}^{n} \eta(j)\mathbf{s}(j) \right\} \tag{4.124}$$

For the conjugate-direction method to work, we require the availability of a set of **A**-conjugate vectors $\mathbf{s}(0), \mathbf{s}(1), ..., \mathbf{s}(W-1)$. In a special form of this method known as the *scaled conjugate-gradient method*,[12] the successive direction vectors are generated as **A**-conjugate versions of the successive gradient vectors of the quadratic function $f(\mathbf{x})$ as the method progresses—hence the name of the method. Thus, except for $n = 0$, the set of direction vectors $\{\mathbf{s}(n)\}$ is not specified beforehand, but rather it is determined in a sequential manner at successive steps of the method.

First, we define the *residual* as the steepest-descent direction:

$$\mathbf{r}(n) = \mathbf{b} - \mathbf{A}\mathbf{x}(n) \tag{4.125}$$

Then, to proceed, we use a linear combination of $\mathbf{r}(n)$ and $\mathbf{s}(n-1)$, as shown by

$$\mathbf{s}(n) = \mathbf{r}(n) + \beta(n)\mathbf{s}(n-1), \qquad n = 1, 2, ..., W-1 \tag{4.126}$$

where $\beta(n)$ is a scaling factor to be determined. Multiplying this equation by **A**, taking the inner product of the resulting expression with $\mathbf{s}(n-1)$, invoking the **A**-conjugate property of the direction vectors, and then solving the resulting expression for $\beta(n)$, we get

$$\beta(n) = -\frac{\mathbf{s}^T(n-1)\mathbf{A}\mathbf{r}(n)}{\mathbf{s}^T(n-1)\mathbf{A}\mathbf{s}(n-1)} \tag{4.127}$$

Using Eqs. (4.126) and (4.127), we find that the vectors $\mathbf{s}(0), \mathbf{s}(1), ..., \mathbf{s}(W-1)$ so generated are indeed **A**-conjugate.

Generation of the direction vectors in accordance with the recursive equation (4.126) depends on the coefficient $\beta(n)$. The formula of Eq. (4.127) for evaluating $\beta(n)$, as it presently stands, requires knowledge of matrix **A**. For computational reasons, it would be desirable to evaluate $\beta(n)$ without explicit knowledge of **A**. This evaluation can be achieved by using one of two formulas (Fletcher, 1987):

1. *the Polak–Ribière formula*, for which $\beta(n)$ is defined by

$$\beta(n) = \frac{\mathbf{r}^T(n)(\mathbf{r}(n) - \mathbf{r}(n-1))}{\mathbf{r}^T(n-1)\mathbf{r}(n-1)} \tag{4.128}$$

2. *the Fletcher–Reeves formula*, for which $\beta(n)$ is defined by

$$\beta(n) = \frac{\mathbf{r}^T(n)\mathbf{r}(n)}{\mathbf{r}^T(n-1)\mathbf{r}(n-1)} \tag{4.129}$$

To use the conjugate-gradient method to attack the unconstrained minimization of the cost function $\mathscr{E}_{\text{av}}(\mathbf{w})$ pertaining to the unsupervised training of multilayer perceptron, we do two things:

- Approximate the cost function $\mathscr{E}_{\text{av}}(\mathbf{w})$ by a quadratic function. That is, the third- and higher-order terms in Eq. (4.111) are ignored, which means that we are operating close to a local minimum on the error surface. On this basis, comparing Eqs. (4.111) and (4.116), we can make the associations indicated in Table 4.2.
- Formulate the computation of coefficients $\beta(n)$ and $\eta(n)$ in the conjugate-gradient algorithm so as to require only gradient information.

TABLE 4.2 Correspondence Between $f(\mathbf{x})$ and $\mathscr{E}_{av}(\mathbf{w})$

Quadratic function $f(\mathbf{x})$	Cost function $\mathscr{E}_{av}(\mathbf{w})$
Parameter vector $\mathbf{x}(n)$	Synaptic weight vector $\mathbf{w}(n)$
Gradient vector $\partial f(\mathbf{x})/\partial \mathbf{x}$	Gradient vector $\mathbf{g} = \partial \mathscr{E}_{av}/\partial \mathbf{w}$
Matrix $\mathbf{A}$	Hessian matrix $\mathbf{H}$

The latter point is particularly important in the context of multilayer perceptrons because it avoids using the Hessian $\mathbf{H}(n)$, the evaluation of which is plagued with computational difficulties.

To compute the coefficient $\beta(n)$ that determines the search direction $\mathbf{s}(n)$ without explicit knowledge of the Hessian $\mathbf{H}(n)$, we can use the Polak–Ribière formula of Eq. (4.128) or the Fletcher–Reeves formula of Eq. (4.129). Both of these formulas involve the use of residuals only. In the linear form of the conjugate-gradient method, assuming a quadratic function, the Polak–Ribière and Fletcher–Reeves formulas are equivalent. On the other hand, in the case of a nonquadratic cost function, they are not.

For nonquadratic optimization problems, the Polak–Ribière form of the conjugate-gradient algorithm is typically superior to the Fletcher–Reeves form of the algorithm, for which we offer the following heuristic explanation (Bertsekas, 1995): Due to the presence of third- and higher-order terms in the cost function $\mathscr{E}_{av}(\mathbf{w})$ and possible inaccuracies in the line search, conjugacy of the generated search directions is progressively lost. This condition may in turn cause the algorithm to "jam" in the sense that the generated direction vector $\mathbf{s}(n)$ is nearly orthogonal to the residual $\mathbf{r}(n)$. When this phenomenon occurs, we have $\mathbf{r}(n) = \mathbf{r}(n-1)$, in which case the scalar $\beta(n)$ will be nearly zero. Correspondingly, the direction vector $\mathbf{s}(n)$ will be close to $\mathbf{r}(n)$, thereby breaking the jam. In contrast, when the Fletcher–Reeves formula is used, the conjugate-gradient algorithm typically continues to jam under similar conditions.

In rare cases, however, the Polak–Ribière method can cycle indefinitely without converging. Fortunately, convergence of the Polak–Ribière method can be guaranteed by choosing

$$\beta = \max\{\beta_{PR}, 0\} \tag{4.130}$$

where β_{PR} is the value defined by the Polak–Ribière formula of Eq. (4.128) (Shewchuk, 1994). Using the value of β defined in Eq. (4.130) is equivalent to restarting the conjugate gradient algorithm if $\beta_{PR} < 0$. To restart the algorithm is equivalent to forgetting the last search direction and starting it anew in the direction of steepest descent.

Consider next the issue of computing the parameter $\eta(n)$, which determines the learning rate of the conjugate-gradient algorithm. As with $\beta(n)$, the preferred method for computing $\eta(n)$ is one that avoids having to use the Hessian $\mathbf{H}(n)$. We recall that the line minimization based on Eq. (4.120) leads to the same formula for $\eta(n)$ as that derived from the update equation Eq. (4.119). We therefore need a *line search*,[12] the purpose of which is to minimize the function $\mathscr{E}_{av}(\mathbf{w} + \eta\mathbf{s})$ with respect to η. That is, given fixed values of the vectors $\mathbf{w}$ and $\mathbf{s}$, the problem is to vary η such that this function is minimized. As η varies, the argument $\mathbf{w} + \eta\mathbf{s}$ traces a line in the W-dimensional

vector space of $\mathbf{w}$—hence the name "line search." A *line-search algorithm* is an iterative procedure that generates a sequence of estimates $\{\eta(n)\}$ for each iteration of the conjugate-gradient algorithm. The line search is terminated when a satisfactory solution is found. The computation of a line search must be performed along each search direction.

Several line-search algorithms have been proposed in the literature, and a good choice is important because it has a profound impact on the performance of the conjugate-gradient algorithm in which it is embedded. There are two phases to any line-search algorithm (Fletcher, 1987):

- *the bracketing phase*, which searches for a *bracket* (that is, a nontrivial interval that is known to contain a minimum), and
- *the sectioning phase*, in which the bracket is *sectioned* (i.e., divided), thereby generating a sequence of brackets whose length is progressively reduced.

We now describe a *curve-fitting procedure* that takes care of these two phases in a straightforward manner.

Let $\mathscr{E}_{\text{av}}(\eta)$ denote the cost function of the multilayer perceptron, expressed as a function of η. It is assumed that $\mathscr{E}_{\text{av}}(\eta)$ is strictly *unimodal* (i.e., it has a single minimum in the neighborhood of the current point $\mathbf{w}(n)$) and is twice continuously differentiable. We initiate the search procedure by searching along the line until we find three points η_1, η_2, and η_3 such that the following condition is satisfied, as illustrated in Fig. 4.21:

$$\mathscr{E}_{\text{av}}(\eta_1) \geq \mathscr{E}_{\text{av}}(\eta_3) \geq \mathscr{E}_{\text{av}}(\eta_2) \qquad \text{for } \eta_1 < \eta_2 < \eta_3 \tag{4.131}$$

Since $\mathscr{E}_{\text{av}}(\eta)$ is a continuous function of η, the choice described in Eq. (4.131) ensures that the bracket $[\eta_1, \eta_3]$ contains a minimum of the function $\mathscr{E}_{\text{av}}(\eta)$. Provided that the function $\mathscr{E}_{\text{av}}(\eta)$ is sufficiently smooth, we may consider this function to be parabolic in the immediate neighborhood of the minimum. Accordingly, we may use *inverse parabolic interpolation* to do the sectioning (Press et al., 1988). Specifically, a parabolic function is fitted through the three original points η_1, η_2, and η_3, as illustrated in Fig. 4.22, where the solid line corresponds to $\mathscr{E}_{\text{av}}(\eta)$ and the dashed line corresponds to the first iteration

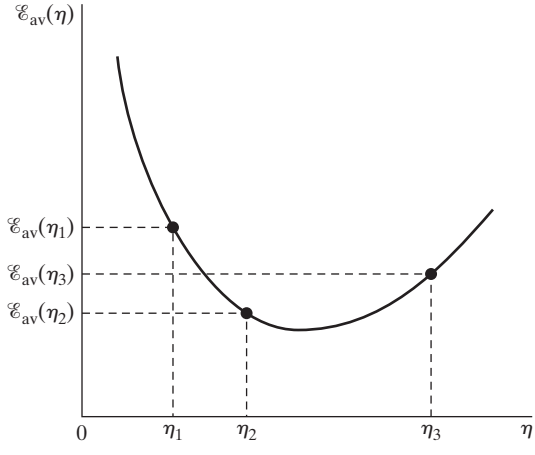


FIGURE 4.21 Illustration of the line search.

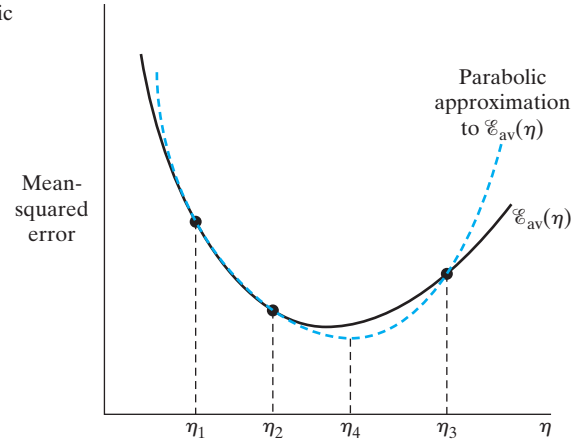


FIGURE 4.22 Inverse parabolic interpolation.

of the sectioning procedure. Let the minimum of the parabola passing through the three points η_1, η_2, and η_3 be denoted by η_4. In the example illustrated in Fig. 4.22, we have $\mathscr{E}_{av}(\eta_4) < \mathscr{E}_{av}(\eta_2)$ and $\mathscr{E}_{av}(\eta_4) < \mathscr{E}_{av}(\eta_1)$. Point η_3 is replaced in favor of η_4, making $[\eta_1, \eta_4]$ the new bracket. The process is repeated by constructing a new parabola through the points η_1, η_2, and η_4. The bracketing-followed-by-sectioning procedure, as illustrated in Fig. 4.22, is repeated several times until a point close enough to the minimum of $\mathscr{E}_{av}(\eta)$ is located, at which time the line search is terminated.

Brent's method constitutes a highly refined version of the three-point curve-fitting procedure just described (Press et al., 1988). At any particular stage of the computation, Brent's method keeps track of six points on the function $\mathscr{E}_{av}(\eta)$, which may not all be necessarily distinct. As before, parabolic interpolation is attempted through three of these points. For the interpolation to be acceptable, certain criteria involving the remaining three points must be satisfied. The net result is a robust line-search algorithm.

Summary of the Nonlinear Conjugate-Gradient Algorithm

All the ingredients we need to formally describe the nonlinear (nonquadratic) form of the conjugate-gradient algorithm for the supervised training of a multilayer perceptron are now in place. A summary of the algorithm is presented in Table 4.3.

Quasi-Newton Methods

Resuming the discussion on quasi-Newton methods, we find that these are basically gradient methods described by the update equation

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(n)\mathbf{s}(n) \tag{4.132}$$

where the direction vector $\mathbf{s}(n)$ is defined in terms of the gradient vector $\mathbf{g}(n)$ by

$$\mathbf{s}(n) = -\mathbf{S}(n)\mathbf{g}(n) \tag{4.133}$$

TABLE 4.3 Summary of the Nonlinear Conjugate-Gradient Algorithm for the Supervised Training of a Multilayer Perceptron

Initialization

Unless prior knowledge on the weight vector $\mathbf{w}$ is available, choose the initial value $\mathbf{w}(0)$ by using a procedure similar to that described for the back-propagation algorithm.

Computation

1. For $\mathbf{w}(0)$, use back propagation to compute the gradient vector $\mathbf{g}(0)$.

2. Set $\mathbf{s}(0) = \mathbf{r}(0) = -\mathbf{g}(0)$.

3. At time-step n, use a line search to find $\eta(n)$ that minimizes $\mathscr{E}_{\text{av}}(\eta)$ sufficiently, representing the cost function $\mathscr{E}_{\text{av}}$ expressed as a function of η for fixed values of $\mathbf{w}$ and $\mathbf{s}$.

4. Test to determine whether the Euclidean norm of the residual $\mathbf{r}(n)$ has fallen below a specified value, that is, a small fraction of the initial value $\|\mathbf{r}(0)\|$.

5. Update the weight vector:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(n)\mathbf{s}(n)$$

6. For $\mathbf{w}(n+1)$, use back propagation to compute the updated gradient vector $\mathbf{g}(n+1)$.

7. Set $\mathbf{r}(n+1) = -\mathbf{g}(n+1)$.

8. Use the Polak–Ribière method to calculate:

$$\beta(n+1) = \max\left\{\frac{\mathbf{r}^T(n+1)(\mathbf{r}(n+1) - \mathbf{r}(n))}{\mathbf{r}^T(n)\mathbf{r}(n)}, 0\right\}$$

9. Update the direction vector:

$$\mathbf{s}(n+1) = \mathbf{r}(n+1) + \beta(n+1)\mathbf{s}(n)$$

10. Set $n = n + 1$, and go back to step 3.

Stopping criterion. Terminate the algorithm when the condition

$$\|\mathbf{r}(n)\| \leq \varepsilon\|\mathbf{r}(0)\|$$

is satisfied, where ε is a prescribed small number.

The matrix $\mathbf{S}(n)$ is a positive-definite matrix that is adjusted from one iteration to the next. This is done in order to make the direction vector $\mathbf{s}(n)$ approximate the *Newton direction*, namely,

$$-(\partial^2\mathscr{E}_{\text{av}}/\partial\mathbf{w}^2)^{-1}\,(\partial\mathscr{E}_{\text{av}}/\partial\mathbf{w})$$

Quasi-Newton methods use second-order (curvature) information about the error surface without actually requiring knowledge of the Hessian. They do so by using two successive iterates $\mathbf{w}(n)$ and $\mathbf{w}(n+1)$, together with the respective gradient vectors $\mathbf{g}(n)$ and $\mathbf{g}(n+1)$. Let

$$\mathbf{q}(n) = \mathbf{g}(n+1) - \mathbf{g}(n) \tag{4.134}$$

and

$$\Delta\mathbf{w}(n) = \mathbf{w}(n+1) - \mathbf{w}(n) \tag{4.135}$$

We may then derive curvature information by using the approximate formula

$$\mathbf{q}(n) \simeq \left(\frac{\partial}{\partial \mathbf{w}} \mathbf{g}(n) \right) \Delta \mathbf{w}(n) \tag{4.136}$$

In particular, given W linearly independent weight increments $\Delta\mathbf{w}(0), \Delta\mathbf{w}(1), ..., \Delta\mathbf{w}(W-1)$ and the respective gradient increments $\mathbf{q}(0), \mathbf{q}(1), ..., \mathbf{q}(W-1)$, we may approximate the Hessian as

$$\mathbf{H} \simeq [\mathbf{q}(0), \mathbf{q}(1), ..., \mathbf{q}(W-1)] [\Delta\mathbf{w}(0), \Delta\mathbf{w}(1), ..., \Delta\mathbf{w}(W-1)]^{-1} \tag{4.137}$$

We may also approximate the inverse Hessian as follows[13]:

$$\mathbf{H}^{-1} \simeq [\Delta\mathbf{w}(0), \Delta\mathbf{w}(1), ..., \Delta\mathbf{w}(W-1)] [\mathbf{q}(0), \mathbf{q}(1), ..., \mathbf{q}(W-1)]^{-1} \tag{4.138}$$

When the cost function $\mathscr{E}_{av}(\mathbf{w})$ is quadratic, Eqs. (4.137) and (4.138) are exact.

In the most popular class of quasi-Newton methods, the updated matrix $\mathbf{S}(n+1)$ is obtained from its previous value $\mathbf{S}(n)$, the vectors $\Delta\mathbf{w}(n)$ and $\mathbf{q}(n)$, by using the following recursion (Fletcher, 1987; Bertsekas, 1995):

$$\begin{aligned} \mathbf{S}(n+1) = \mathbf{S}(n) &+ \frac{\Delta\mathbf{w}(n)\Delta\mathbf{w}^T(n)}{\mathbf{q}^T(n)\mathbf{q}(n)} - \frac{\mathbf{S}(n)\mathbf{q}(n)\mathbf{q}^T(n)\mathbf{S}(n)}{\mathbf{q}^T(n)\mathbf{S}(n)\mathbf{q}(n)} \\ &+ \xi(n)[\mathbf{q}^T(n)\mathbf{S}(n)\mathbf{q}(n)]\,[\mathbf{v}(n)\mathbf{v}^T(n)] \end{aligned} \tag{4.139}$$

where

$$\mathbf{v}(n) = \frac{\Delta\mathbf{w}(n)}{\Delta\mathbf{w}^T(n)\Delta\mathbf{w}(n)} - \frac{\mathbf{S}(n)\mathbf{q}(n)}{\mathbf{q}^T(n)\mathbf{S}(n)\mathbf{q}(n)} \tag{4.140}$$

and

$$0 \le \xi(n) \le 1 \qquad \text{for all } n \tag{4.141}$$

The algorithm is initiated with some arbitrary positive- definite matrix $\mathbf{S}(0)$. The particular form of the quasi-Newton method is parameterized by how the scalar $\xi(n)$ is defined, as indicated by the following two points (Fletcher, 1987):

1. For $\xi(n) = 0$ for all n, we obtain the *Davidon–Fletcher–Powell (DFP) algorithm*, which is historically the first quasi-Newton method.
2. For $\xi(n) = 1$ for all n, we obtain the *Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm*, which is considered to be the best form of quasi-Newton methods currently known.

Comparison of Quasi-Newton Methods with Conjugate-Gradient Methods

We conclude this brief discussion of quasi-Newton methods by comparing them with conjugate-gradient methods in the context of nonquadratic optimization problems (Bertsekas, 1995):

- Both quasi-Newton and conjugate-gradient methods avoid the need to use the Hessian. However, quasi-Newton methods go one step further by generating an

approximation to the inverse Hessian. Accordingly, when the line search is accurate and we are in close proximity to a local minimum with a positive-definite Hessian, a quasi-Newton method tends to approximate Newton's method, thereby attaining faster convergence than would be possible with the conjugate-gradient method.

- Quasi-Newton methods are not as sensitive to accuracy in the line-search stage of the optimization as the conjugate-gradient method.
- Quasi-Newton methods require storage of the matrix $\mathbf{S}(n)$, in addition to the matrix-vector multiplication overhead associated with the computation of the direction vector $\mathbf{s}(n)$. The net result is that the computational complexity of quasi-Newton methods is $O(W^2)$. In contrast, the computational complexity of the conjugate-gradient method is $O(W)$. Thus, when the dimension W (i.e., size of the weight vector $\mathbf{w}$) is large, conjugate-gradient methods are preferable to quasi-Newton methods in computational terms.

It is because of the lattermost point that the use of quasi-Newton methods is restricted, in practice, to the design of small-scale neural networks.

Levenberg–Marquardt Method

The Levenberg–Marquardt method, due to Levenberg (1994) and Marquardt (1963), is a compromise between the following two methods:

- Newton's method, which converges rapidly near a local or global minimum, but may also diverge;
- Gradient descent, which is assured of convergence through a proper selection of the step-size parameter, but converges slowly.

To be specific, consider the optimization of a second-order function $F(\mathbf{w})$, and let $\mathbf{g}$ be its gradient vector and $\mathbf{H}$ be its Hessian. According to the Levenberg–Marquardt method, the optimum adjustment $\Delta\mathbf{w}$ applied to the parameter vector $\mathbf{w}$ is defined by

$$\Delta\mathbf{w} = [\mathbf{H} + \lambda\mathbf{I}]^{-1}\mathbf{g} \tag{4.142}$$

where $\mathbf{I}$ is the identity matrix of the same dimensions as $\mathbf{H}$ and λ is a *regularizing*, or *loading*, *parameter* that forces the sum matrix $(\mathbf{H} + \lambda\mathbf{I})$ to be positive definite and safely well conditioned throughout the computation. Note also that the adjustment $\Delta\mathbf{w}$ of Eq. (4.142) is a minor modification of the formula defined in Eq. (4.115).

With this background, consider a multilayer perceptron with a single output neuron. The network is trained by minimizing the cost function

$$\mathscr{E}_{\text{av}}(\mathbf{w}) = \frac{1}{2N}\sum_{i=1}^{N}[d(i) - F(\mathbf{x}(i); \mathbf{w})]^2 \tag{4.143}$$

where $\{\mathbf{x}(i), d(i)\}_{i=1}^{N}$ is the training sample and $F(\mathbf{x}(i); \mathbf{w})$ is the approximating function realized by the network; the synaptic weights of the network are arranged in some orderly manner to form the weight vector $\mathbf{w}$. The gradient and the Hessian of the cost function $\mathscr{E}_{\text{av}}(\mathbf{w})$ are respectively defined by

$$\mathbf{g}(\mathbf{w}) = \frac{\partial \mathscr{E}_{\text{av}}(\mathbf{w})}{\partial \mathbf{w}} = -\frac{1}{N}\sum_{i=1}^{N}[d(i) - F(\mathbf{x}(i); \mathbf{w})]\frac{\partial F(\mathbf{x}(i); \mathbf{w})}{\partial \mathbf{w}} \tag{4.144}$$

and

$$\mathbf{H}(\mathbf{w}) = \frac{\partial^2 \mathscr{E}_{\text{av}}(\mathbf{w})}{\partial \mathbf{w}^2} = \frac{1}{N}\sum_{i=1}^{N}\left[\frac{\partial F(\mathbf{x}(i); \mathbf{w})}{\partial \mathbf{w}}\right]\left[\frac{\partial F(\mathbf{x}(i); \mathbf{w})}{\partial \mathbf{w}}\right]^T - \frac{1}{N}\sum_{i=1}^{N}[d(i) - F(\mathbf{x}(i); \mathbf{w})]\frac{\partial^2 F(\mathbf{x}(i); \mathbf{w})}{\partial \mathbf{w}^2} \tag{4.145}$$

Thus, substituting Eqs. (4.144) and (4.145) into Eq. (4.142), the desired adjustment $\Delta\mathbf{w}$ is computed for each iteration of the Levenberg-Marquardt algorithm.

However, from a practical perspective, the computational complexity of Eq. (4.145) can be demanding, particularly when the dimensionality of the weight vector $\mathbf{w}$ is high; the computational difficulty is attributed to the complex nature of the Hessian $\mathbf{H}(\mathbf{w})$. To mitigate this difficulty, the recommended procedure is to ignore the second term on the right-hand side of Eq. (4.145), thereby approximating the Hessian simply as

$$\mathbf{H}(\mathbf{w}) \approx \frac{1}{N}\sum_{i=1}^{N}\left[\frac{\partial F(\mathbf{x}(i); \mathbf{w})}{\partial \mathbf{w}}\right]\left[\frac{\partial F(\mathbf{x}(i); \mathbf{w})}{\partial \mathbf{w}}\right]^T \tag{4.146}$$

This approximation is recognized as the outer product of the partial derivative $\partial F(\mathbf{w}, \mathbf{x}(i))/\partial \mathbf{w}$ with itself, averaged over the training sample; accordingly, it is referred to as the *outer-product approximation* of the Hessian. The use of this approximation is justified when the Levenberg-Marquardt algorithm is operating in the neighborhood of a local or global minimum.

Clearly, the approximate version of the Levenberg–Marquardt algorithm, based on the gradient vector of Eq. (4.144) and the Hessian of Eq. (4.146), is a first-order method of optimization that is well suited for nonlinear least-squares estimation problems. Moreover, because of the fact that both of these equations involve averaging over the training sample, the algorithm is of a batch form.

The regularizing parameter λ plays a critical role in the way the Levenberg-Marquardt algorithm functions. If we set λ equal to zero, then the formula of Eq. (4.142) reduces to Newton's method. On the other hand, if we assign a large value to λ such that $\lambda\mathbf{I}$ overpowers the Hessian $\mathbf{H}$, the Levenberg-Marquardt algorithm functions effectively as a gradient descent method. From these two observations, it follows that at each iteration of the algorithm, the value assigned to λ should be just large enough to maintain the sum matrix $(\mathbf{H} + \lambda\mathbf{I})$ in its positive-definite form. In specific terms, the recommended *Marquardt recipe* for the selection of λ is as follows (Press et al.,) 1988:

1. Compute $\mathscr{E}_{\text{av}}(\mathbf{w})$ at iteration $n - 1$.
2. Choose a modest value for λ, say $\lambda = 10^{-3}$.