Résumé de la déclaration 44110 **Généré le: 2023/09/21 20:08**

IA

Identification du déclarant

- Prénom: **Ghania Sarah**
- Nom: **FARHI**
- Courriel: **ghania.farhi@usherbrooke.ca**
- Date de création: **2023/09/21**

Identification du cours

- Code du cours: **GEI895**
- Titre du cours: **Conception de systèmes intelligents**
- Session: **Automne 2023**
- Prénom du professeur: **Jean-Baptiste**
- Nom du professeur: **Michaud**
- Unité administrative: **1808-FGÉN Électrique et informatique**

Titre du document: IA
Type de reproduction: Intranet (accès limité par mot de passe)
Nombre d'étudiants: 30

Ce document contient des extraits d'oeuvres protégées par la Loi sur le droit d'auteur.

Ouvrage(s) déclaré(s): **1**

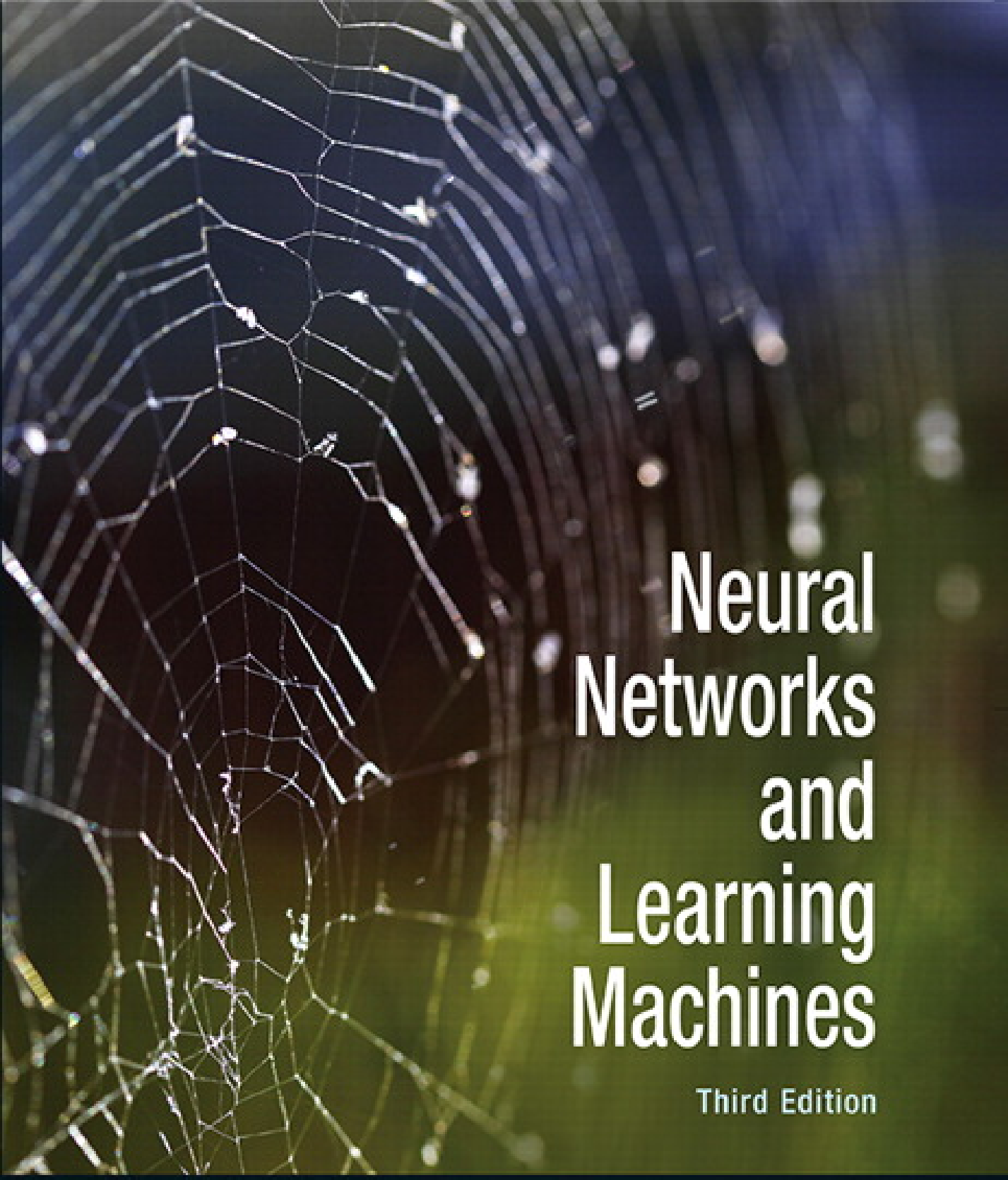
Li Titre: **Neural Networks and Learning Machines** Emprunt: **154 pages**

- ISBN: **978-0-13-147139-9**
- Titre: **Neural Networks and Learning Machines**
- Auteur(s): **Simon Haykin**
- Éditeur: **Prentice Hall Press**
- Provenance: **Canada**
- Date de publication: **2008**
- Nombre total de pages de l'ouvrage: **936**
- Domaine public: **Non**
- Cahier d'exercice: **Non**
- Type de support: **Imprimé**

Détail des pages empruntées :

- Chapitre: **0,pages 1-46, total 46 pages , gratuit**
- Chapitre: **4,pages 122-229, total 108 pages , 0.00$**

Cet emprunt est couvert par l'entente avec Copibec



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

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 $\mathbf{w}$.

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

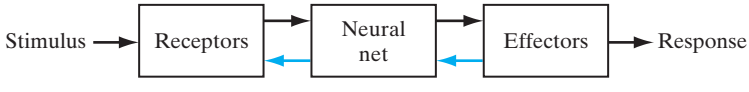


FIGURE 1 Block diagram representation of nervous system.

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

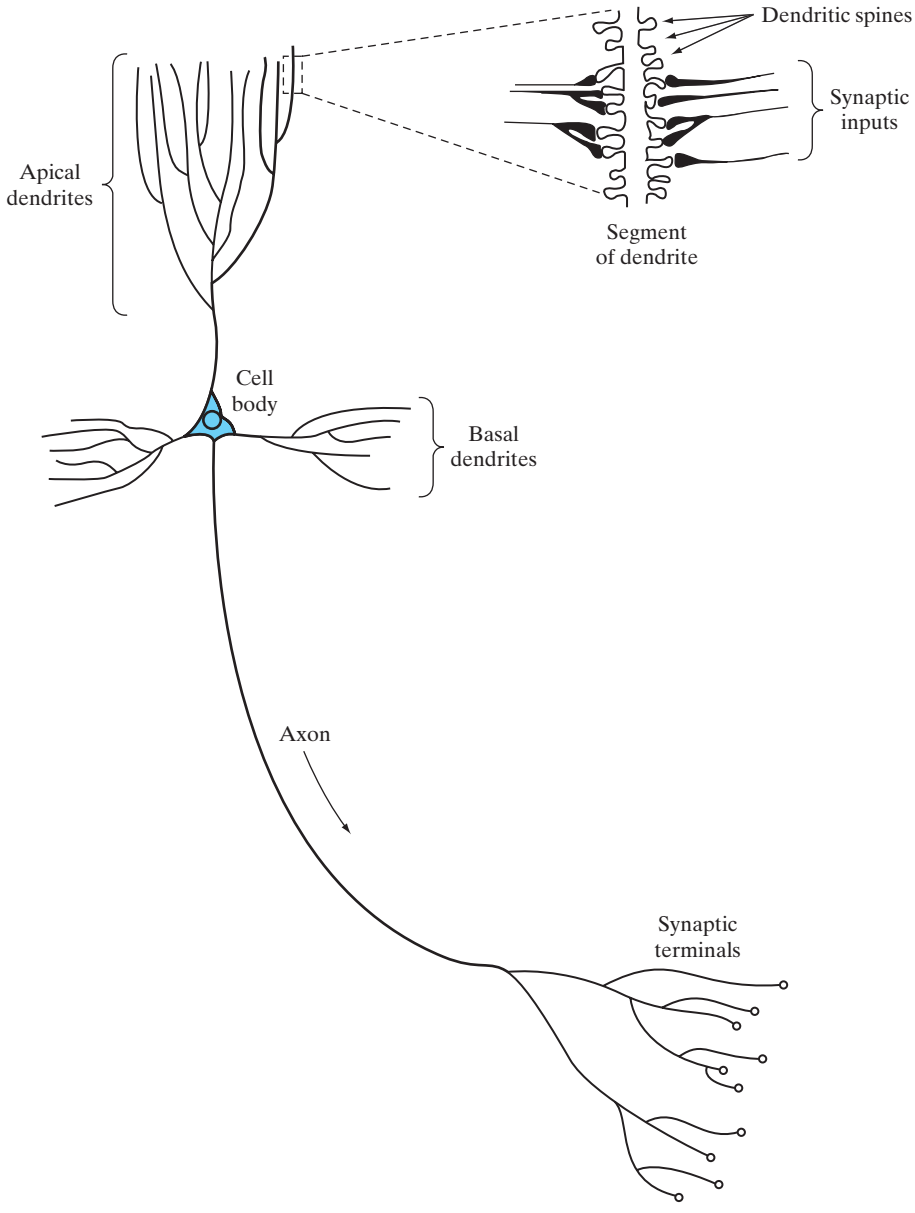


FIGURE 2 The pyramidal cell.

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,

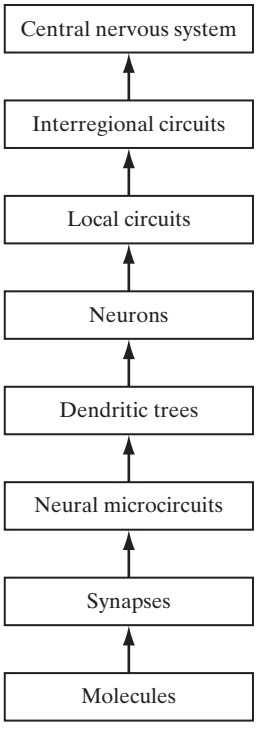


FIGURE 3 Structural organization of levels in the brain.

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.

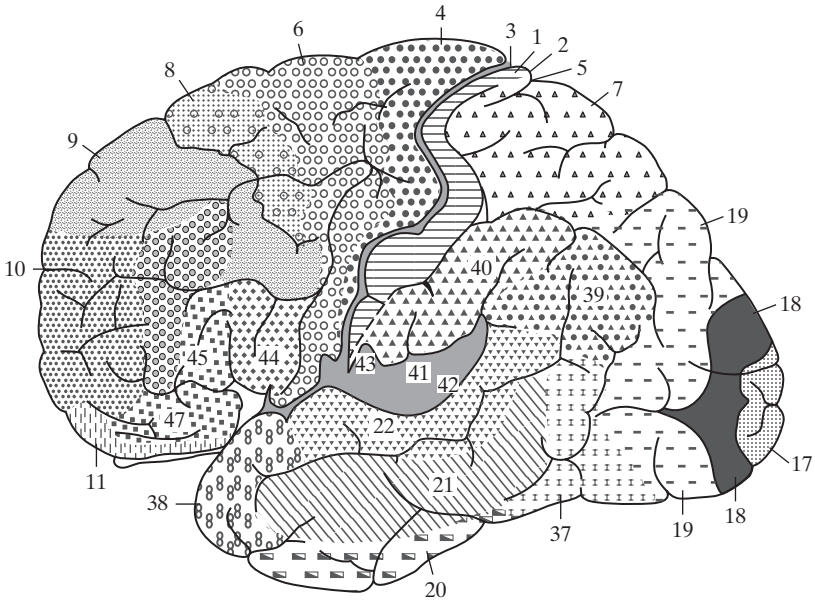


FIGURE 4 Cytoarchitectural map of the cerebral cortex. The different areas are identified by the thickness of their layers and types of cells within them. Some of the key sensory areas are as follows: Motor cortex: motor strip, area 4; premotor area, area 6; frontal eye fields, area 8. Somatosensory 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.

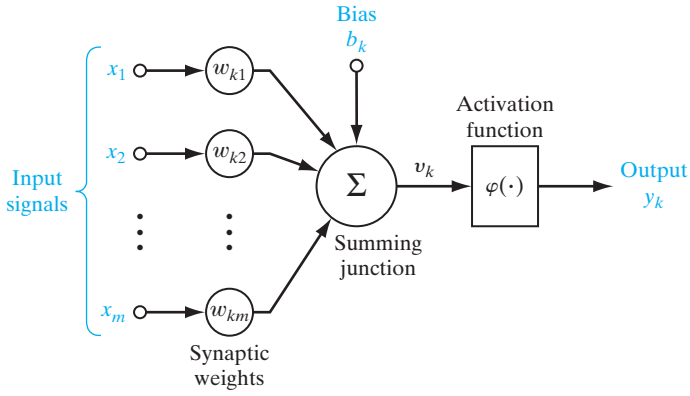


FIGURE 5 Nonlinear model of a neuron, labeled k.

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:

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

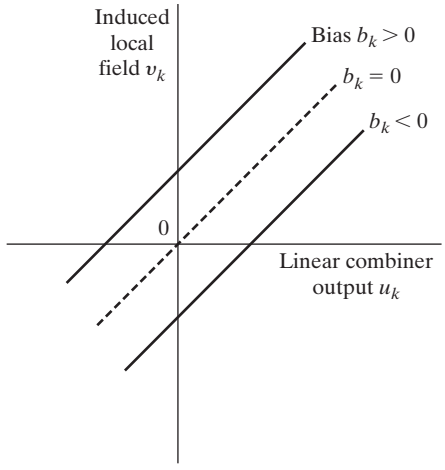


FIGURE 6 Affine transformation produced by the presence of a bias; note that $v_k = b_k$ at $u_k = 0$.

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.

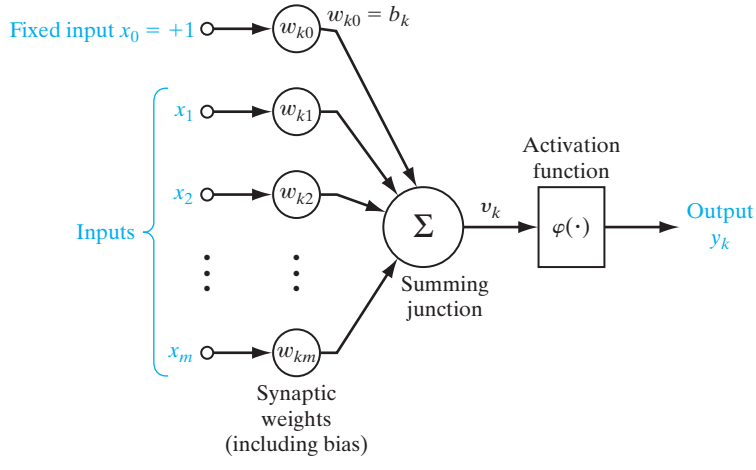


FIGURE 7 Another nonlinear model of a neuron; w_{k0} accounts for the bias b_k.

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,

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

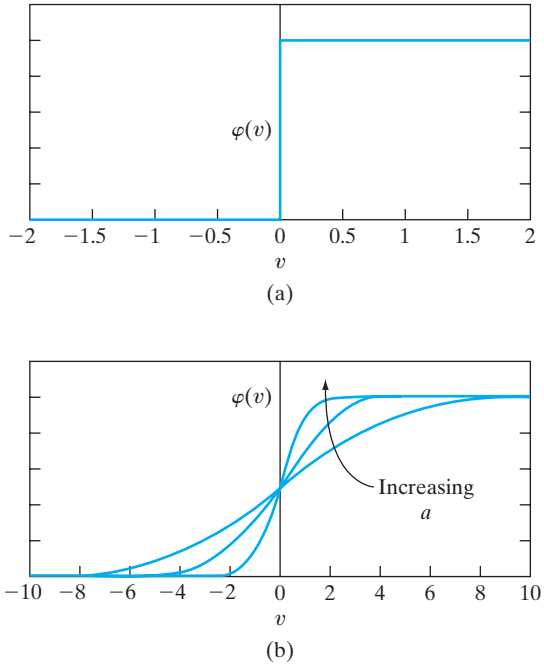


FIGURE 8 (a) Threshold function. (b) Sigmoid function for varying slope parameter a.

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.

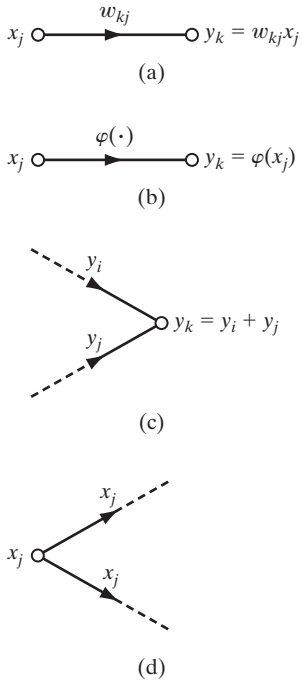


FIGURE 9 Illustrating basic rules for the construction of signal-flow graphs.

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.*

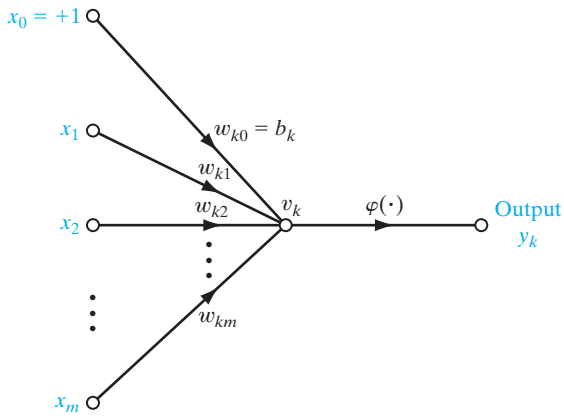


FIGURE 10 Signal-flow graph of a neuron.

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.

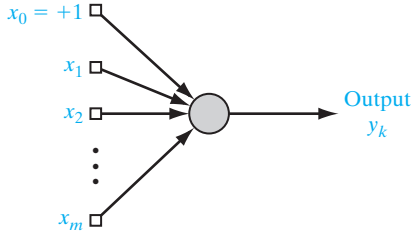


FIGURE 11 Architectural graph of a neuron.

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

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

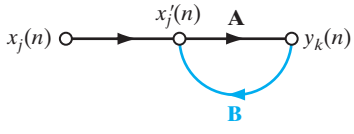


FIGURE 12 Signal-flow graph of a single-loop feedback system.

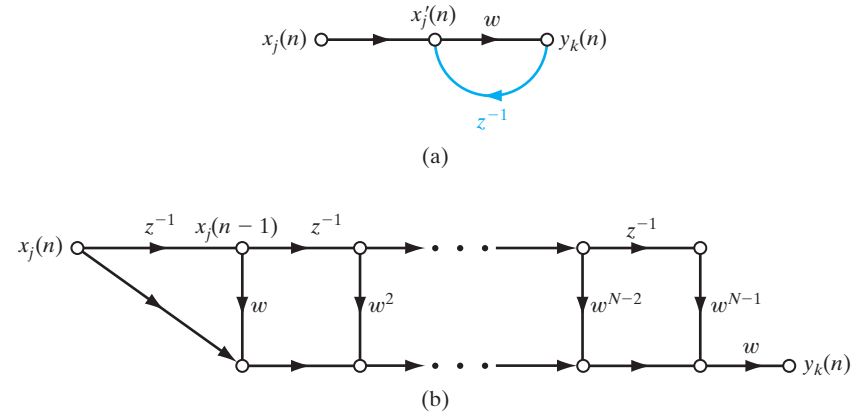


FIGURE 13 (a) Signal-flow graph of a first-order, infinite-duration impulse response (IIR) filter. (b) Feedforward approximation of part (a) of the figure, obtained by truncating Eq. (20).

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.

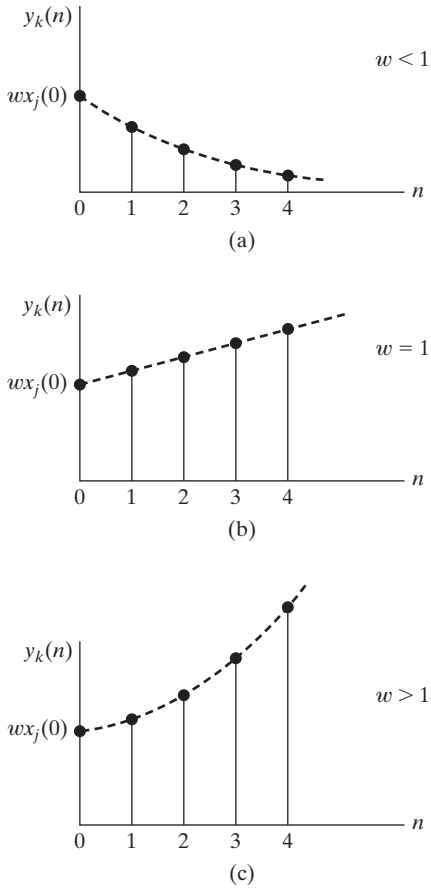


FIGURE 14 Time response of Fig. 13 for three different values of feedforward weight w.
(a) Stable.
(b) Linear divergence.
(c) Exponential divergence.

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.

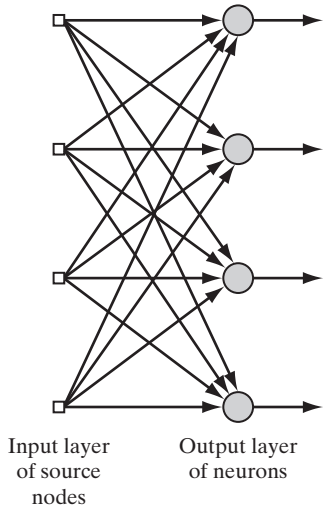


FIGURE 15 Feedforward network with a single layer of neurons.

(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.

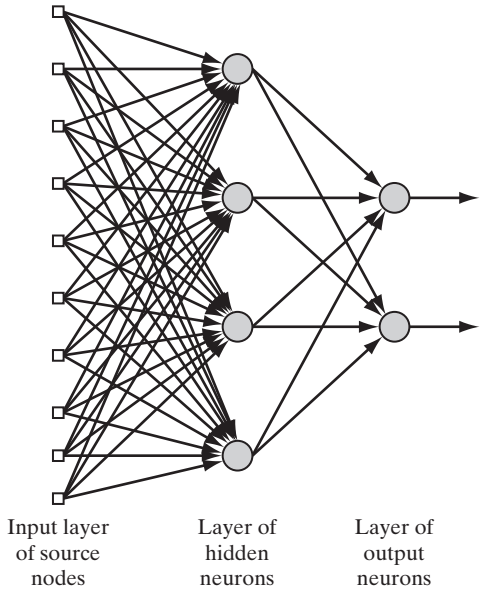


FIGURE 16 Fully connected feedforward network with one hidden layer and one output layer.

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.

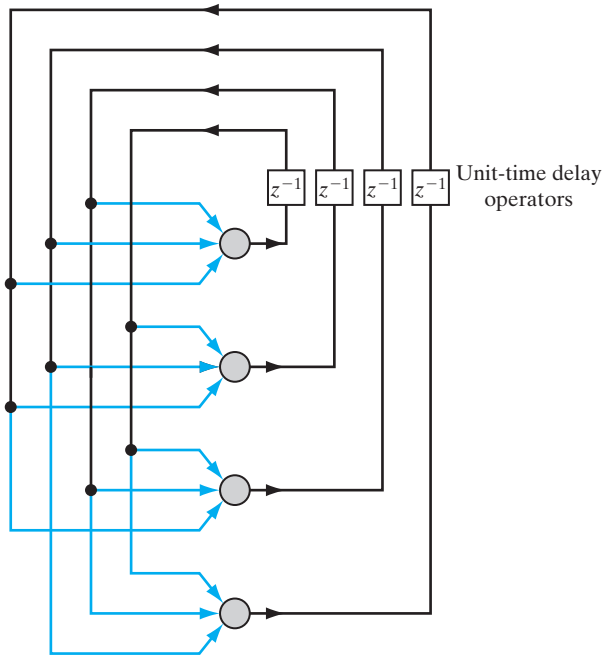


FIGURE 17 Recurrent network with no self-feedback loops and no hidden neurons.

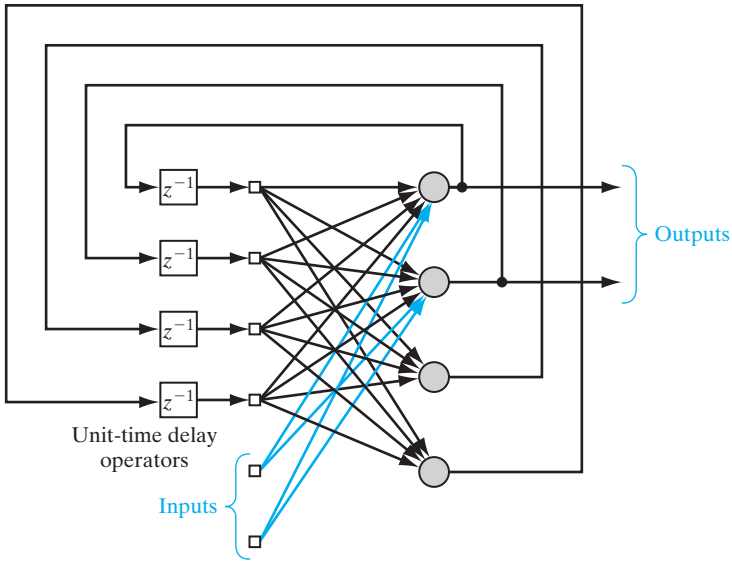


FIGURE 18 Recurrent network with hidden neurons.

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

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

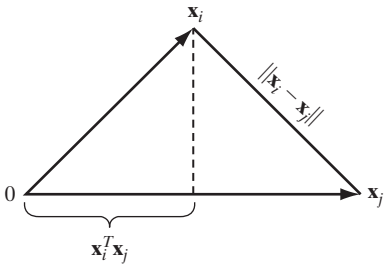


FIGURE 19 Illustrating the relationship between inner product and Euclidean distance as measures of similiarity between patterns.

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

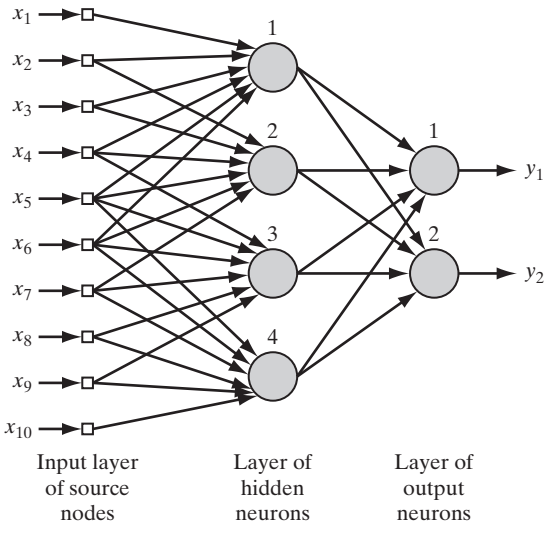


FIGURE 20 Illustrating the combined use of a receptive field and weight sharing. All four hidden neurons share the same set of weights exactly for their six synaptic connections.

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.

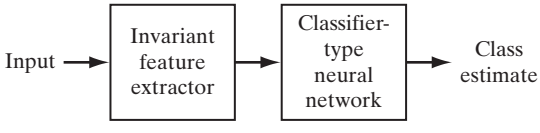


FIGURE 21 Block diagram of an invariant-feature-space type of system.

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

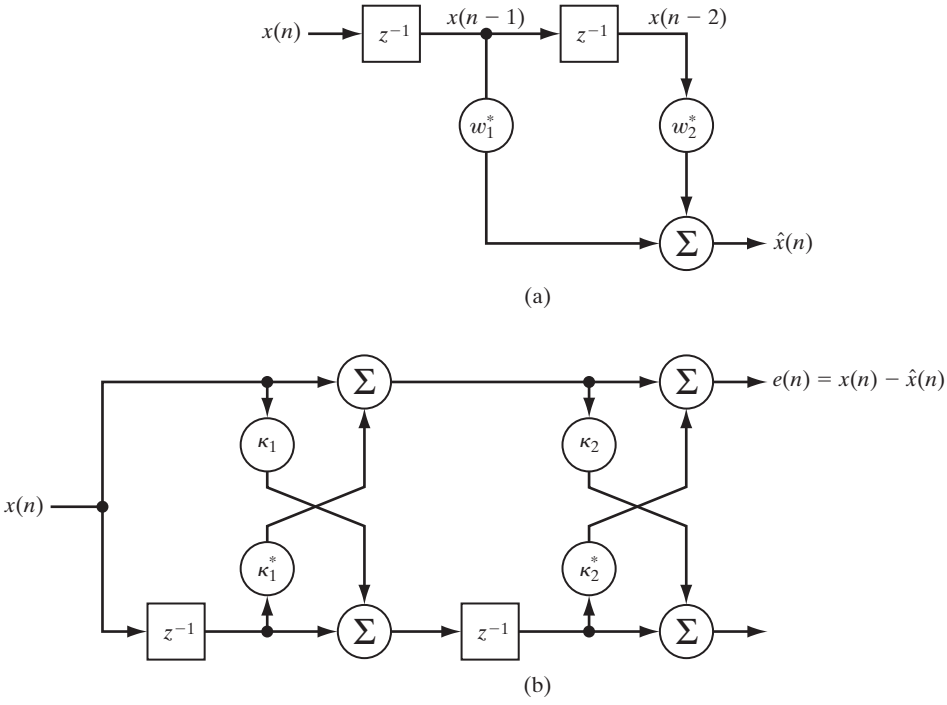


FIGURE 22 Autoregressive model of order 2: (a) tapped-delay-line model; (b) lattice-filter model. (The asterisk denotes complex conjugation.)

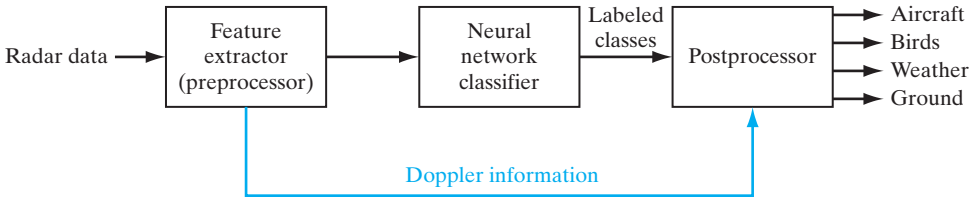


FIGURE 23 Doppler-shift-invariant classifier of radar signals.

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

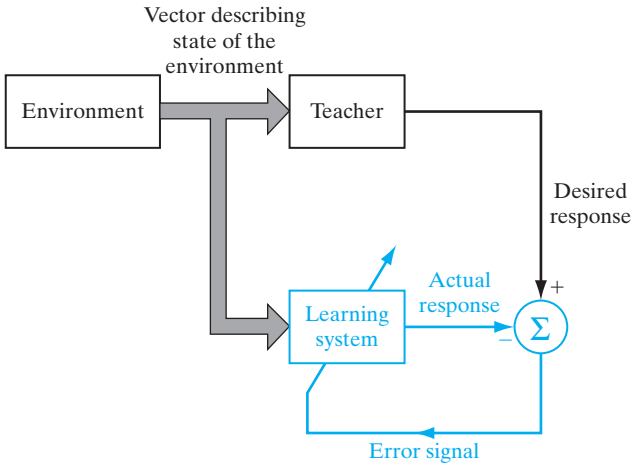


FIGURE 24 Block diagram of learning with a teacher; the part of the figure printed in red constitutes a feedback loop.

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.

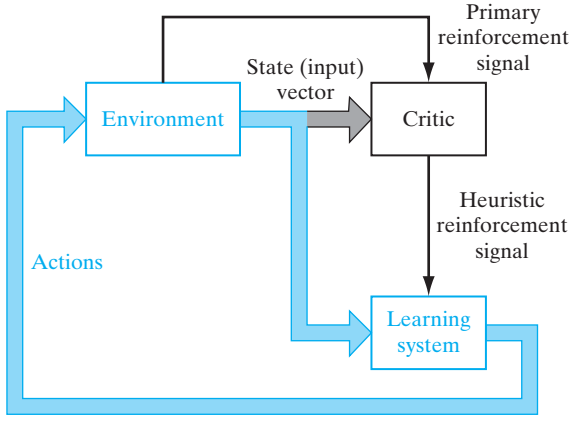


FIGURE 25 Block diagram of reinforcement learning; the learning system and the environment are both inside the feedback loop.

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.

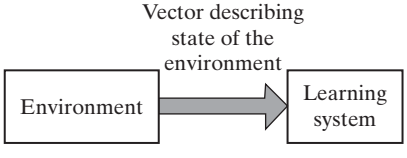


FIGURE 26 Block diagram of unsupervised learning.

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.*

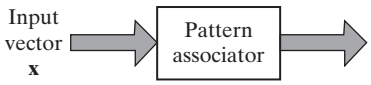


FIGURE 27 Input–output relation of pattern associator.

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.*

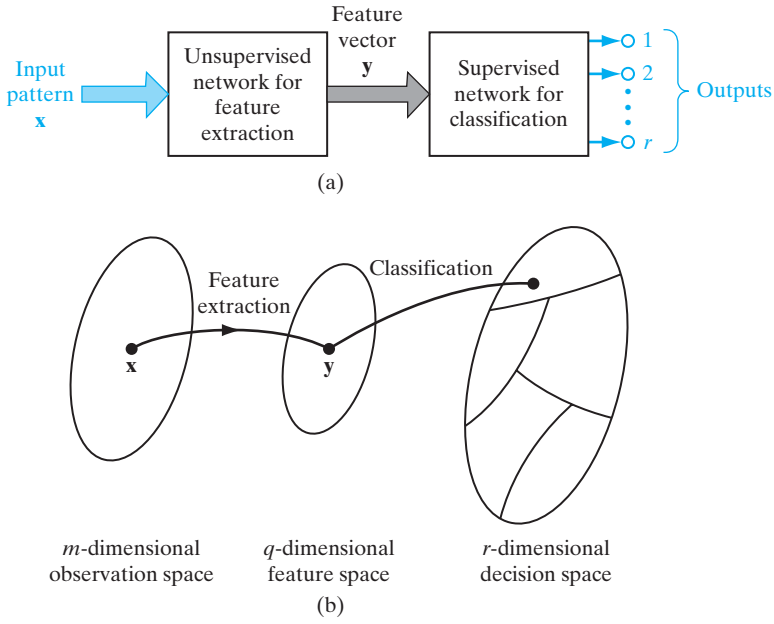


FIGURE 28 Illustration of the classical approach to pattern classification.

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

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

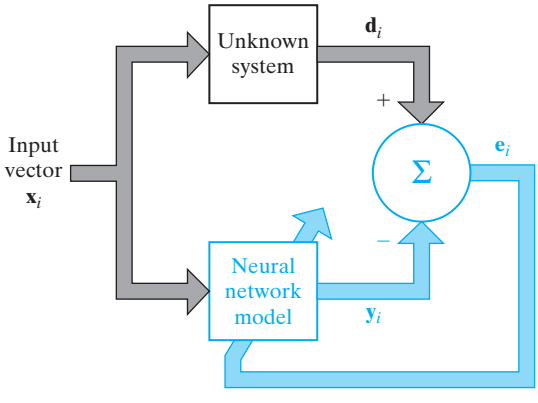


FIGURE 29 Block diagram of system identification: The neural network, doing the identification, is part of the feedback loop.

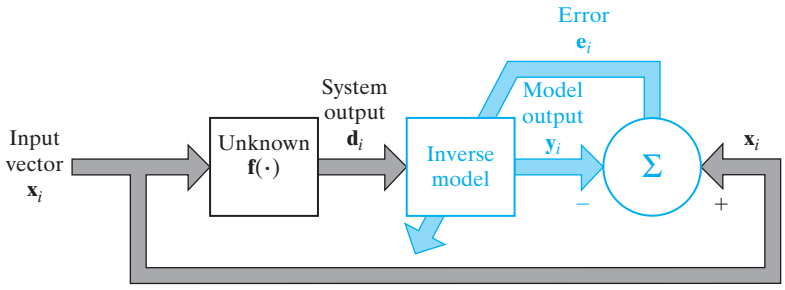


FIGURE 30 Block diagram of inverse system modeling. The neural network, acting as the inverse model, is part of the feedback loop.

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

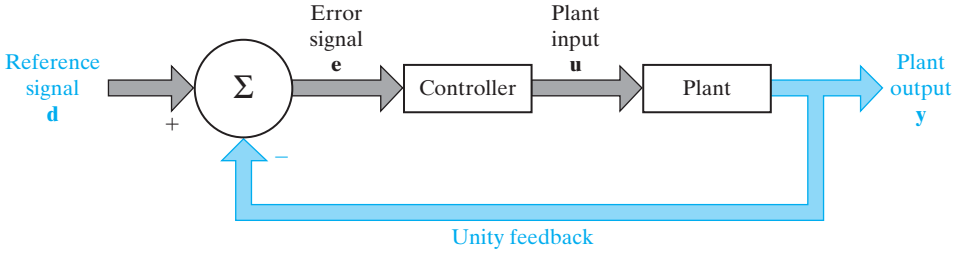


FIGURE 31 Block diagram of feedback control system.

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.

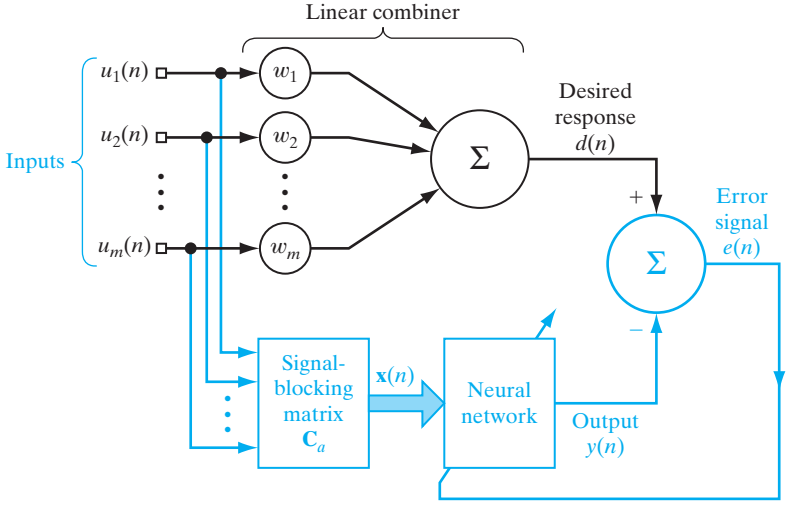


FIGURE 32 Block diagram of generalized sidelobe canceller.

- 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 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.

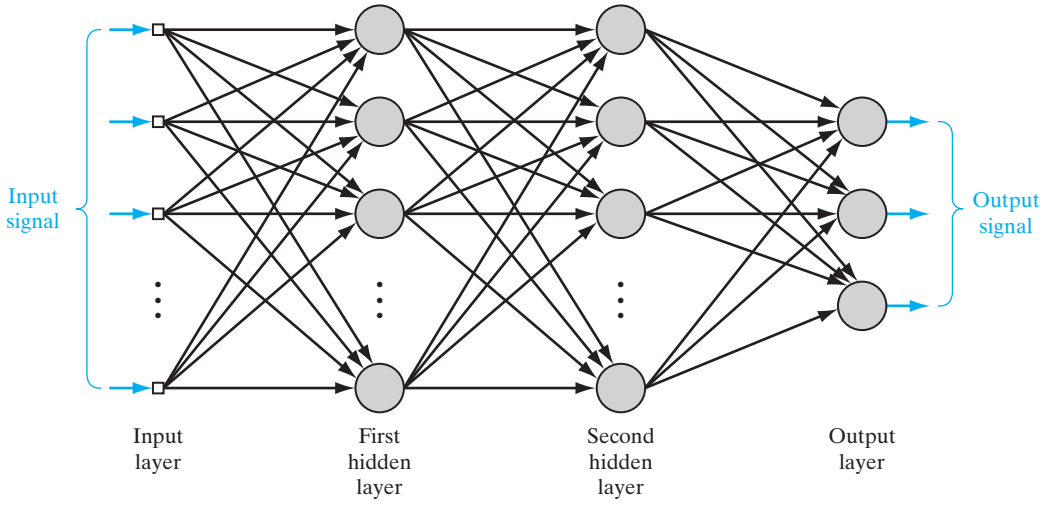


FIGURE 4.1 Architectural graph of a multilayer perceptron with two hidden layers.

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.

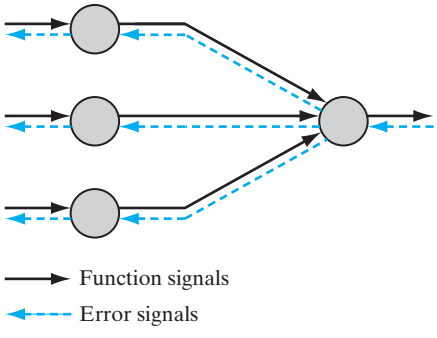


FIGURE 4.2 Illustration of the directions of two basic signal flows in a multilayer perceptron: forward propagation of function signals and back propagation of error signals.

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

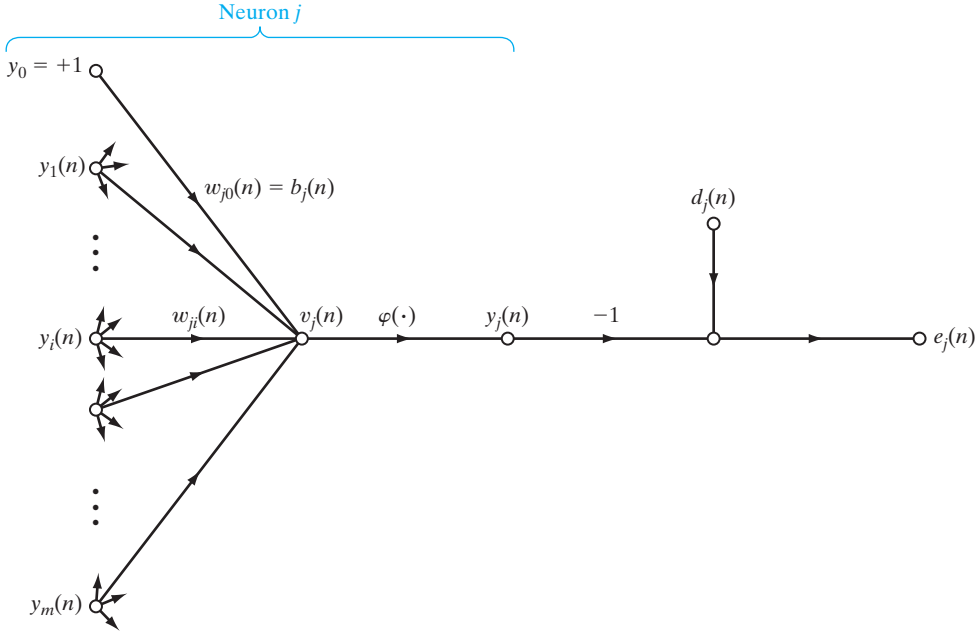


FIGURE 4.3 Signal-flow graph highlighting the details of output neuron j.

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

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} \tag{4.19}$$

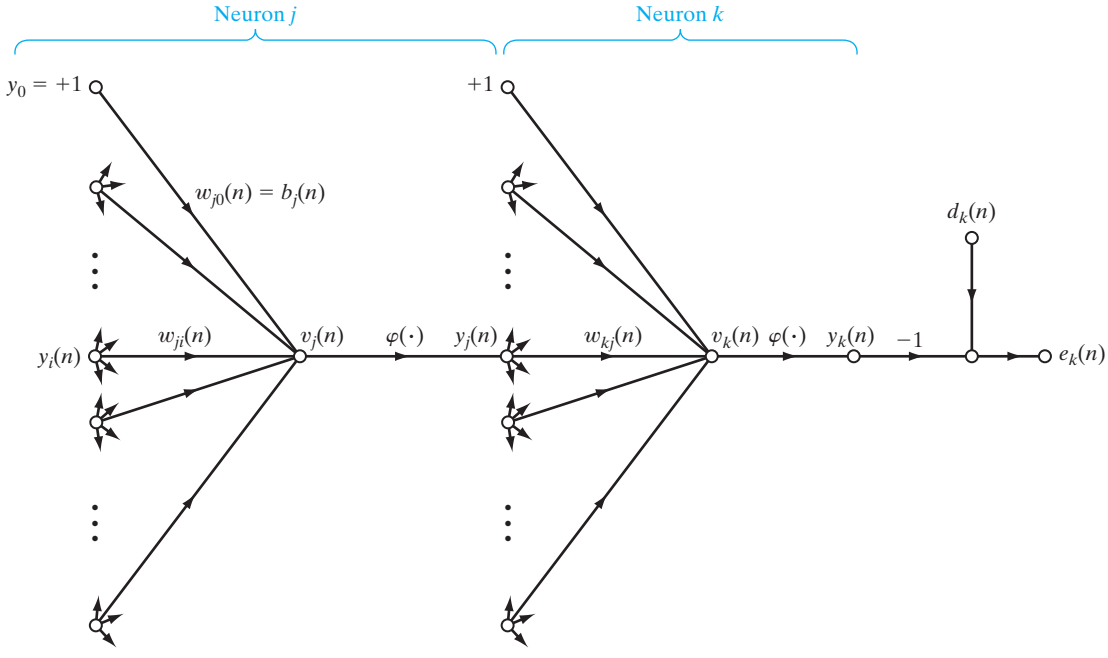


FIGURE 4.4 Signal-flow graph highlighting the details of output neuron k connected to hidden neuron j.

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

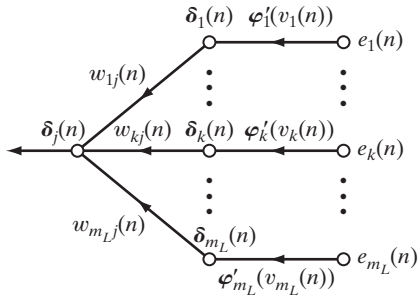


FIGURE 4.5 Signal-flow graph of a part of the adjoint system pertaining to back-propagation of error signals.

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)

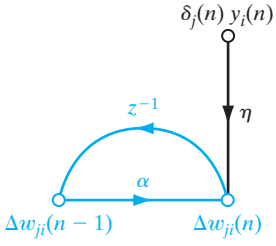


FIGURE 4.6 Signal-flow graph illustrating the effect of momentum constant α, which lies inside the feedback loop.

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

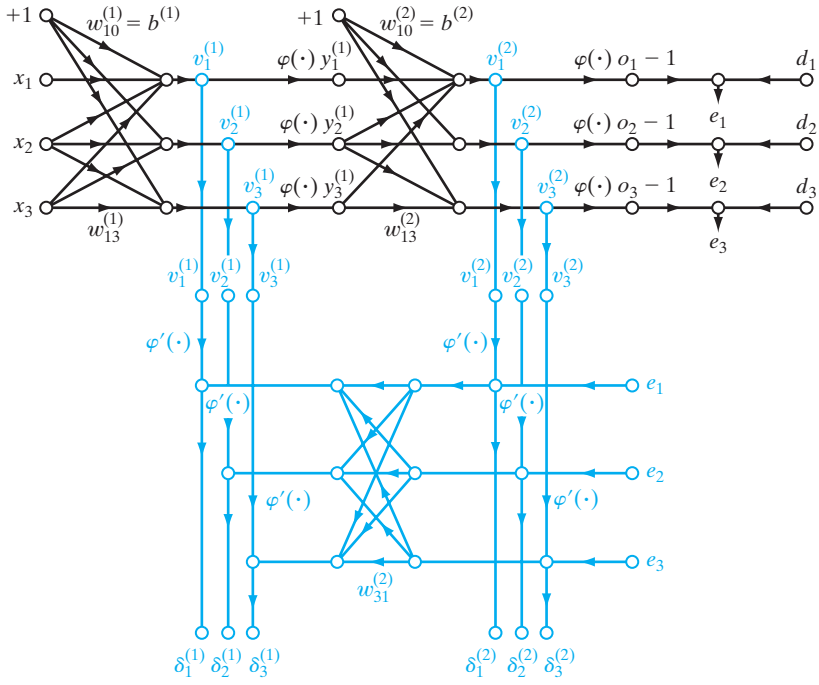


FIGURE 4.7 Signal-flow graphical summary of back-propagation learning. Top part of the graph: forward pass. Bottom part of the graph: backward pass.

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

$$w_{21} = w_{22} = +1$$
$$b_2 = -\frac{1}{2}$$

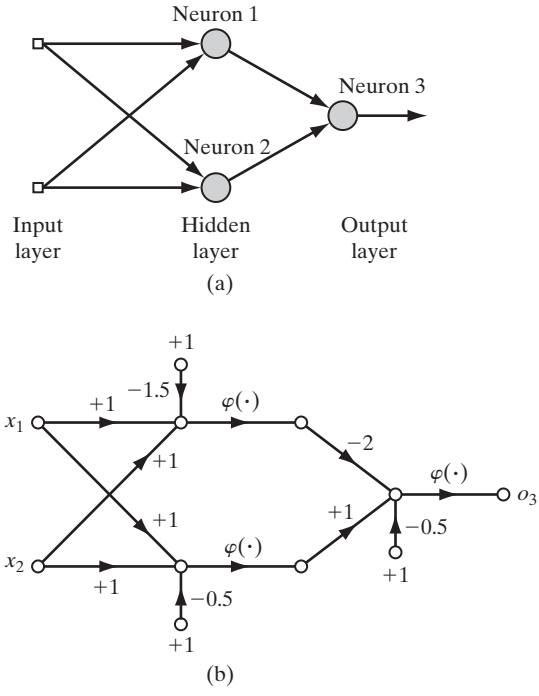


FIGURE 4.8 (a) Architectural graph of network for solving the XOR problem. (b) Signal-flow graph of the network.

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.

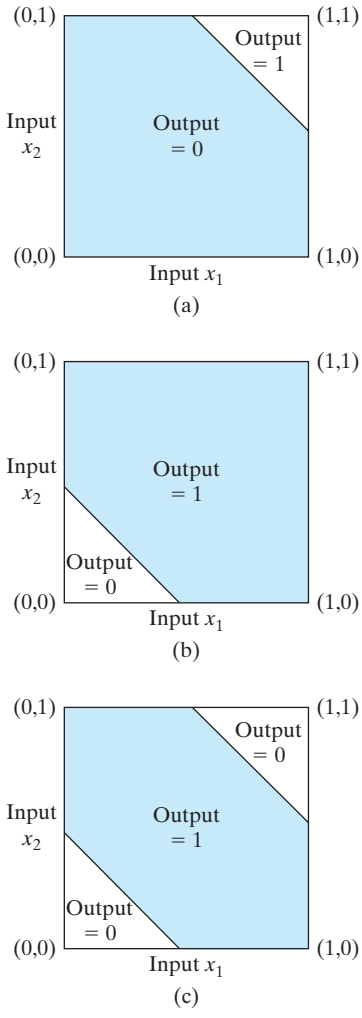


FIGURE 4.9 (a) Decision boundary constructed by hidden neuron 1 of the network in Fig. 4.8. (b) Decision boundary constructed by hidden neuron 2 of the network. (c) Decision boundaries constructed by the complete network.

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 $v = 1$.

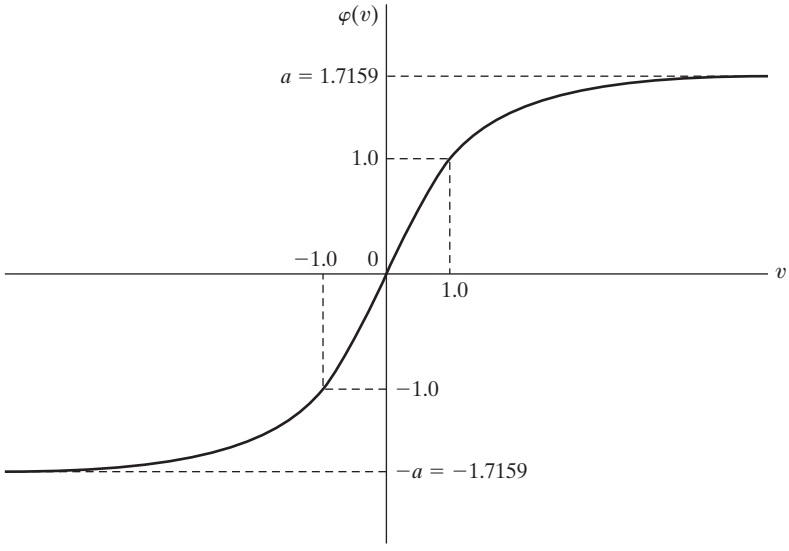


FIGURE 4.10 Graph of the hyperbolic tangent function $\varphi(v) = a\tanh(bv)$ for $a = 1.7159$ and $b = 2/3$. The recommended target values are $+1$ and -1.

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.

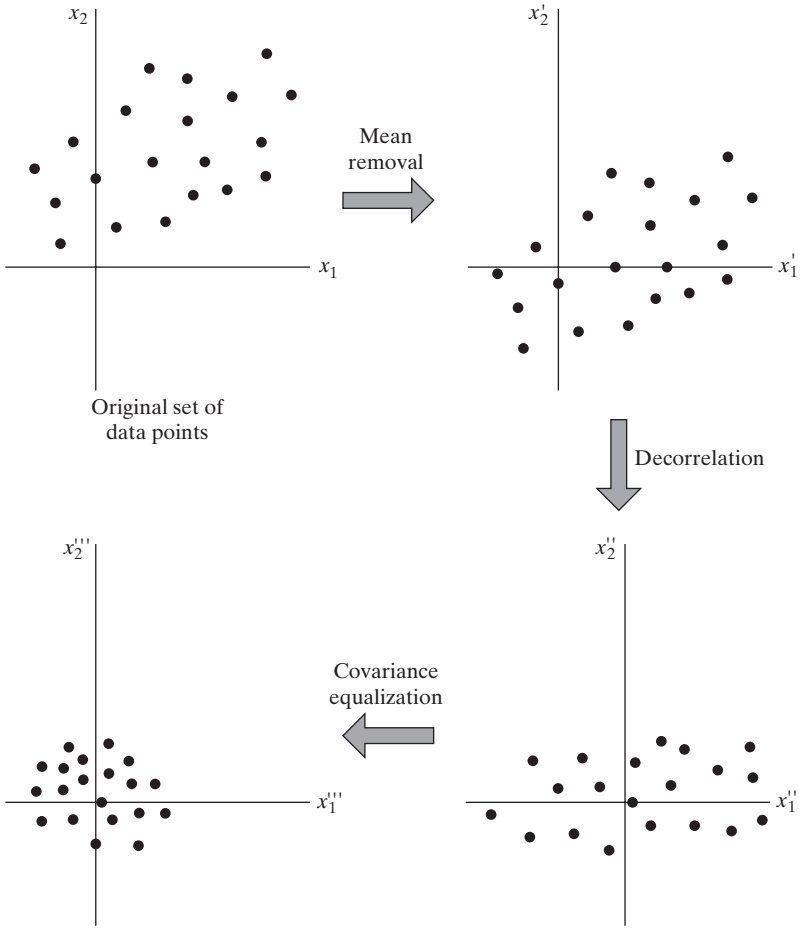


FIGURE 4.11 Illustrating the operation of mean removal, decorrelation, and covariance equalization for a two-dimensional input space.

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:

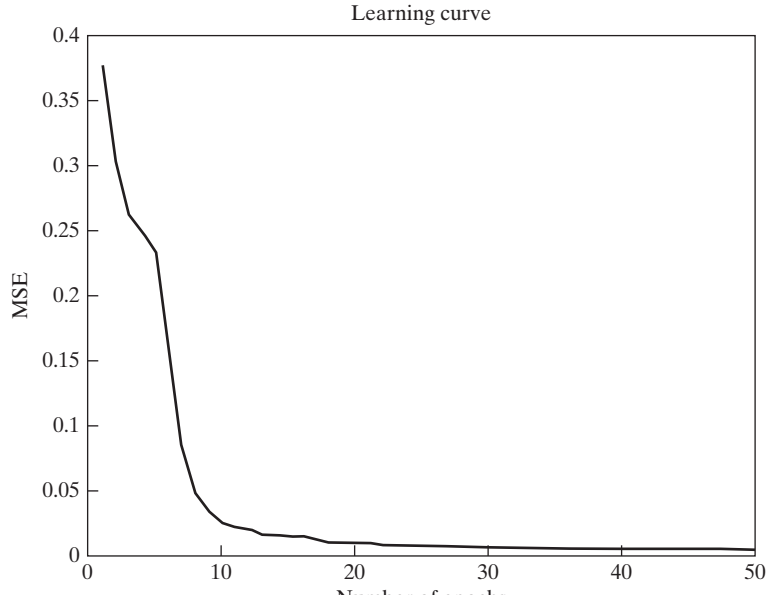
1. to demonstrate that the multilayer perceptron, trained with the back-propagation algorithm, is capable of classifying nonlinearly separable test data;
2. to find a more difficult case of nonlinearly separable patterns for which the multilayer perceptron fails the double-moon classification test.

The specifications of the multilayer perceptron used in the experiment are as follows:

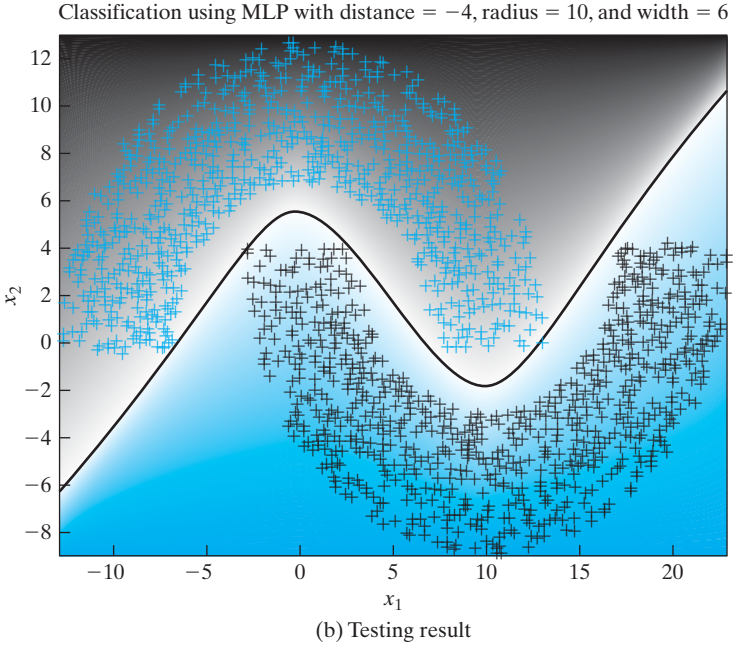
Size of the input layer: $m_0 = 2$
Size of the (only) hidden layer: $m_1 = 20$
Size of the output layer: $m_2 = 1$
Activation function: hyperbolic tangent function $\varphi(v) = \dfrac{1 - \exp(-2v)}{1 + \exp(-2v)}$
Threshold setting: zero
Learning-rate parameter η: annealed linearly from 10^{-1} down 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.

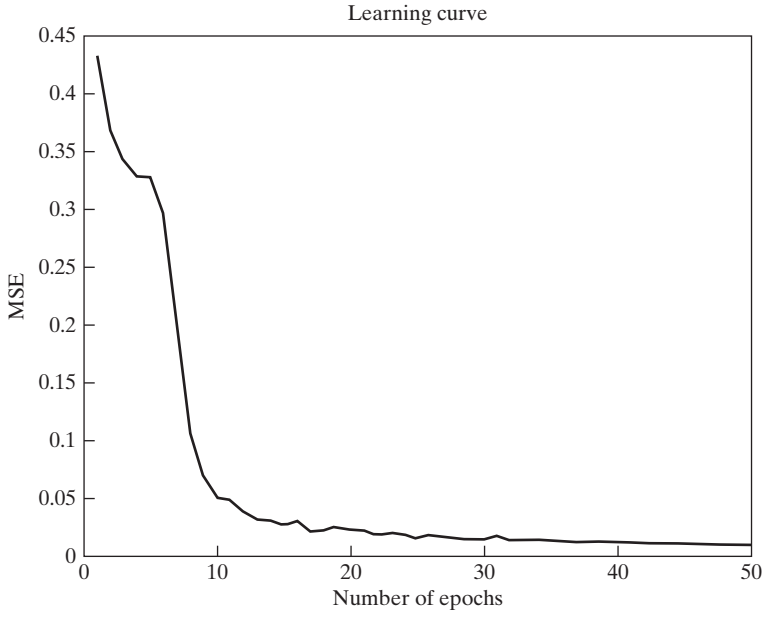


(a) Learning curve

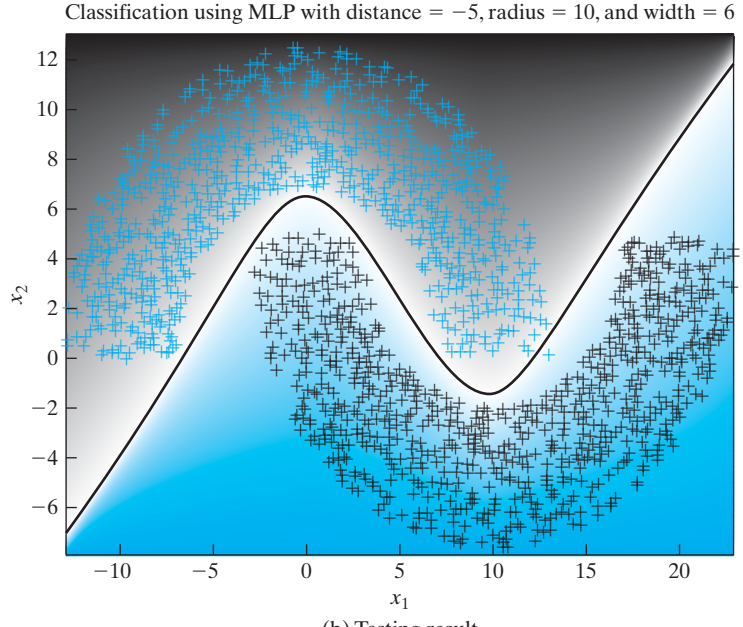


(b) Testing result

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

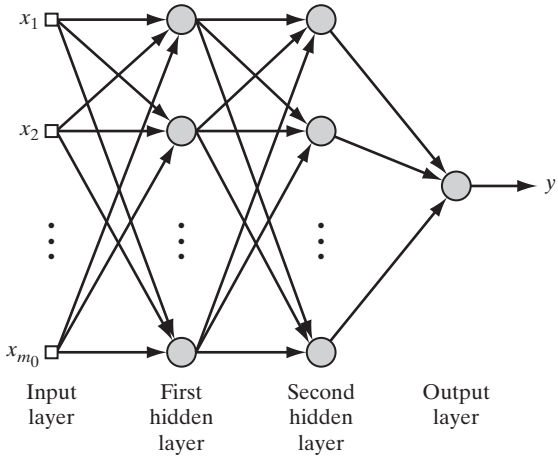


FIGURE 4.14 Multilayer perceptron with two hidden layers and one output neuron.

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

$$\sum_t \eta(t) \to \infty \text{ and } \sum_t \eta^2(t) > \infty, \text{ as } t \to \infty \tag{4.75}$$

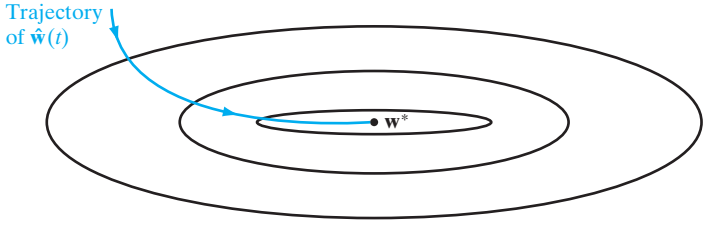


FIGURE 4.15 The evolution of the estimator $\hat{\mathbf{w}}(t)$ over time t. The ellipses represent contours of the expected risk for varying values of $\mathbf{w}$, assumed to be two-dimensional.

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.

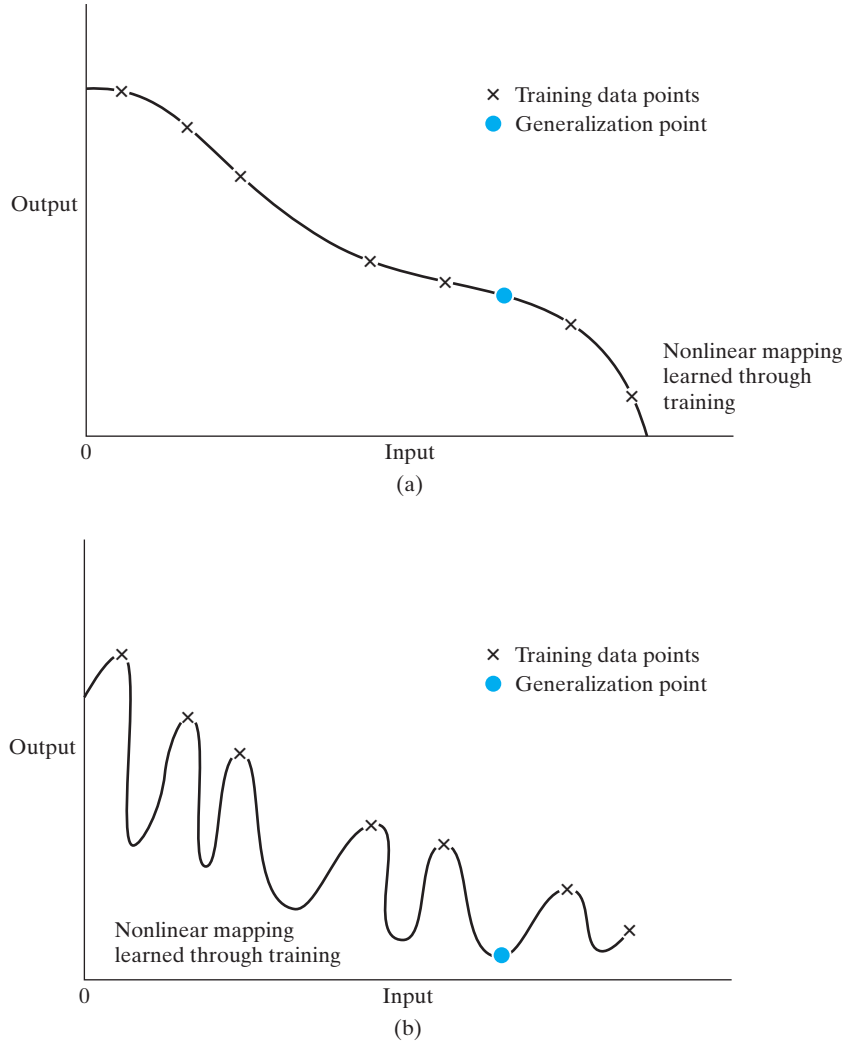


FIGURE 4.16 (a) Properly fitted nonlinear mapping with good generalization. (b) Overfitted nonlinear mapping with poor generalization.

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).

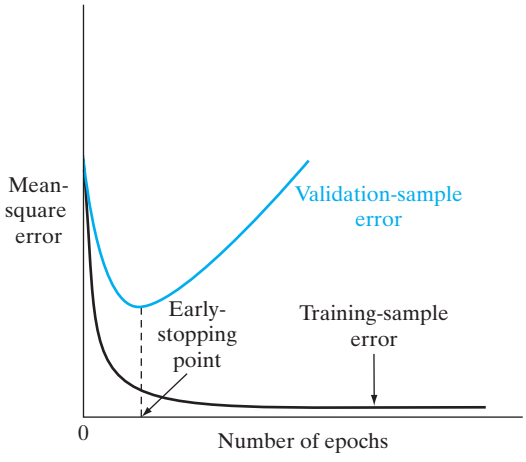


FIGURE 4.17 Illustration of the early-stopping rule based on cross-validation.

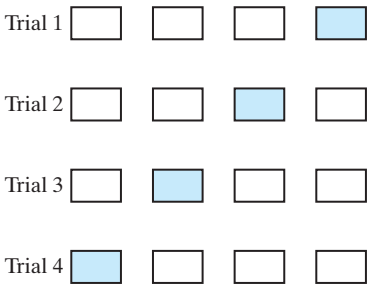


FIGURE 4.18 Illustration of the multifold method of cross-validation. For a given trial, the subset of data shaded in red is used to validate the model trained on the remaining data.

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

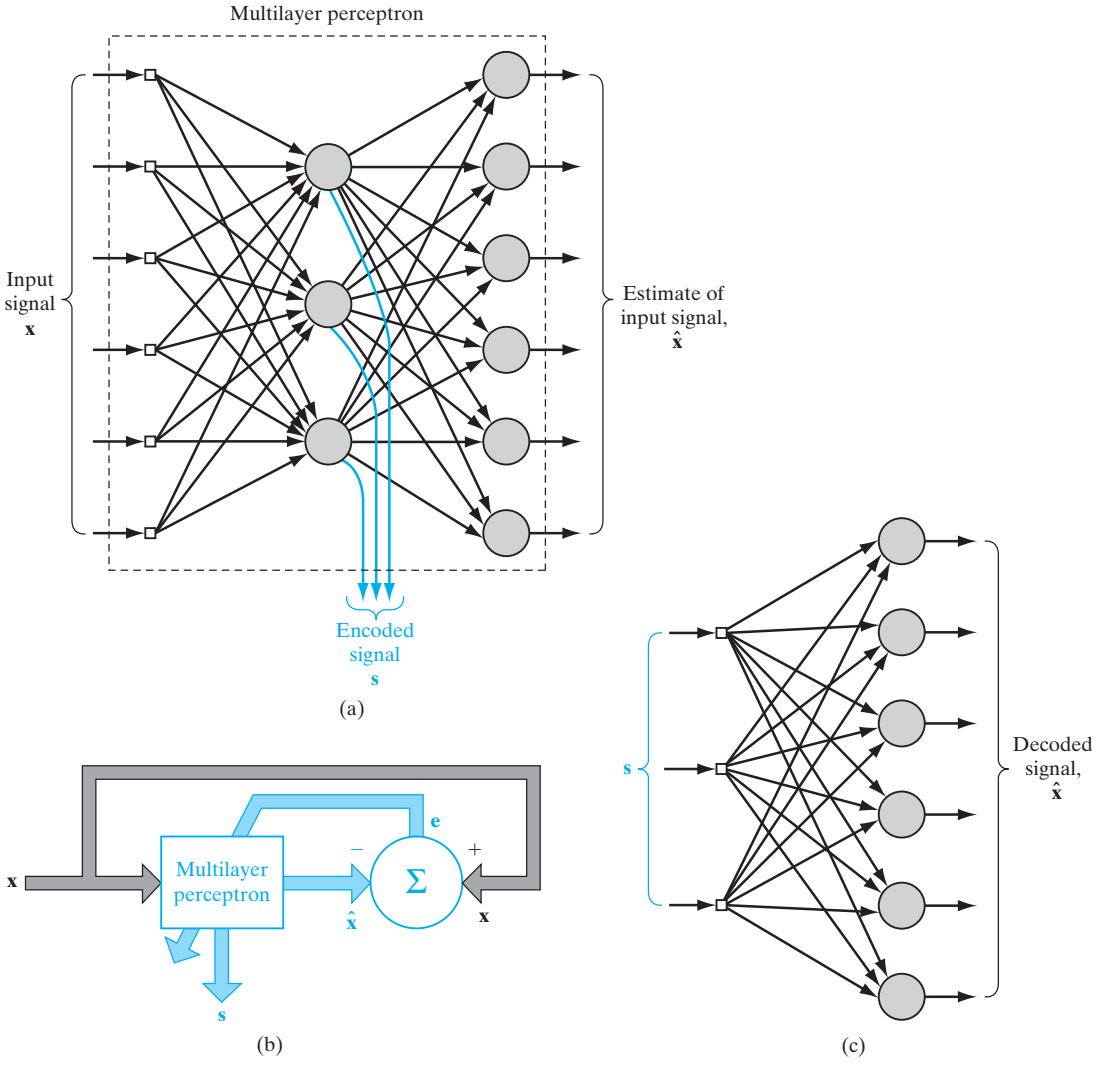


FIGURE 4.19 (a) Replicator network (identity map) with a single hidden layer used as an encoder. (b) Block diagram for the supervised training of the replicator network. (c) Part of the replicator network used as a decoder.

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.

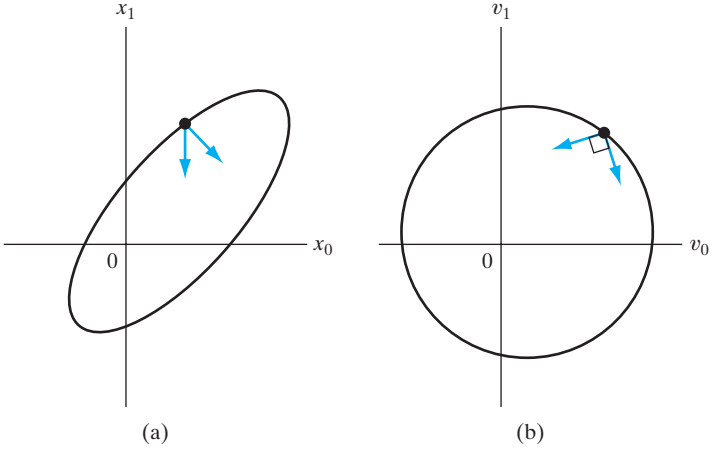


FIGURE 4.20 Interpretation of **A**-conjugate vectors. (a) Elliptic locus in two-dimensional weight space. (b) Transformation of the elliptic locus into a circular locus.

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

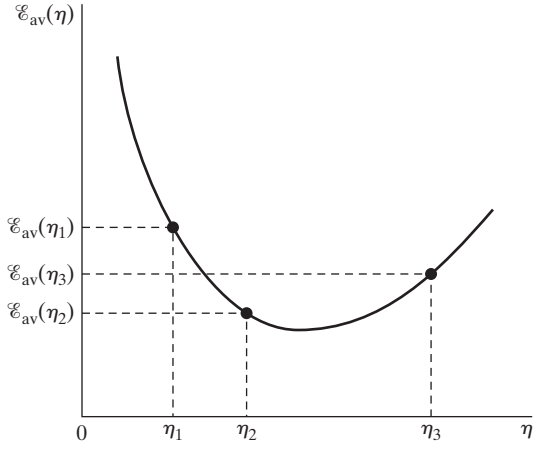


FIGURE 4.21 Illustration of the line search.

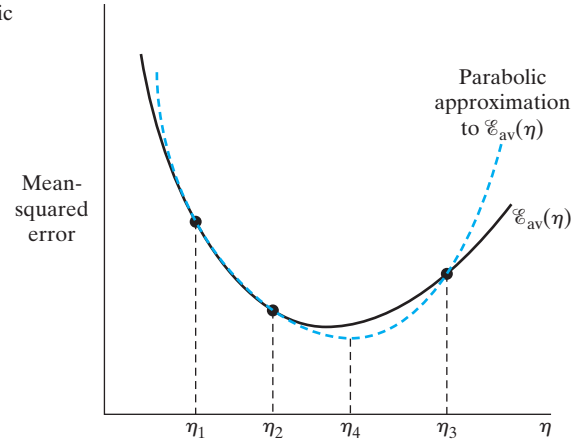


FIGURE 4.22 Inverse parabolic interpolation.

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 $\lambda = 10^{-3}$.

3. Solve Eq. (4.142) for the adjustment $\Delta\mathbf{w}$ at iteration n and evaluate $\mathscr{E}_{av}(\mathbf{w} + \Delta\mathbf{w})$.
4. If $\mathscr{E}_{av}(\mathbf{w} + \Delta\mathbf{w}) \geq \mathscr{E}_{av}(\mathbf{w})$, increase λ by a factor of 10 (or any other substantial factor) and go back to step 3.
5. If, on the other hand, $\mathscr{E}_{av}(\mathbf{w} + \Delta\mathbf{w}) < \mathscr{E}_{av}(w)$, decrease λ by a factor of 10, update the trial solution $\mathbf{w} \rightarrow \mathbf{w} + \Delta\mathbf{w}$, and go back to step 3.

For obvious reasons, a rule for stopping the iterative process is necessary. In Press et al. (1998), it is pointed out that an adjustment in the parameter vector $\mathbf{w}$ that changes $\mathscr{E}_{av}(\mathbf{w})$ by an incrementally small amount is *never* statistically meaningful. We may therefore use this insightful comment as a basis for the stopping rule.

One last comment is in order: To evaluate the partial derivative $\partial F(\mathbf{x}; \mathbf{w})/\partial\mathbf{w}$ at each iteration of the algorithm, we may use back-propagation in the manner described in Section 4.8.

Second-Order Stochastic Gradient Descent for On-line Learning

Up to this point, this section has focused on second-order optimization techniques for batch learning. Hereafter, we turn our attention to second-order stochastic gradient-descent methods for on-line learning. Although these two families of techniques are entirely different, they do share a common purpose:

The second-order information contained in the Hessian (curvature) of the cost function is used to improve the performance of supervised-learning algorithms.

A simple way of expanding on the performance of the optimally annealed on-line learning algorithm considered in Section 4.10 is to replace the learning-rate parameter $\eta(n)$ in Eq. (4.60) with the scaled inverse of the Hessian $\mathbf{H}$, as shown by

$$\underbrace{\hat{\mathbf{w}}(n+1)}_{\text{Updated estimate}} = \underbrace{\hat{\mathbf{w}}(n)}_{\text{Old estimate}} - \underbrace{\frac{1}{n}\mathbf{H}^{-1}}_{\substack{\text{Annealed inverse of the Hessian } \mathbf{H}}}\underbrace{\mathbf{g}(\mathbf{x}(n+1), \mathbf{d}(n+1); \hat{\mathbf{w}}(n))}_{\text{Gradient vector } \mathbf{g}} \tag{4.147}$$

The replacement of $\eta(n)$ with the new term $\frac{1}{n}\mathbf{H}^{-1}$ is intended to accelerate the speed of convergence of the on-line algorithm in an optimally annealed fashion. It is assumed that the Hessian $\mathbf{H}$ is known a priori and its inverse $\mathbf{H}^{-1}$ can therefore be precomputed.

Recognizing the fact that "there is no such thing as a free lunch," the price paid for the accelerated convergence is summarized as follows (Bottou, 2007):

(i) Whereas in the stochastic gradient descent of Eq. (4.60), the computation cost per iteration of the algorithm is $O(W)$, where W is the dimension of the weight vector $\mathbf{w}$ being estimated, the corresponding computation cost per iteration of the second-order stochastic gradient-descent algorithm in Eq. (4.147) is $O(W^2)$.

(ii) For each training example $(\mathbf{x}, \mathbf{d})$ processed by the algorithm of Eq. (4.147), the algorithm requires multiplication of the W-by-l gradient vector $\mathbf{g}$ and the W-by-W inverse matrix $\mathbf{H}^{-1}$ and storage of the product.

(iii) In a general context, whenever some form of *sparsity* exists in the training sample, the natural move is to exploit the sparsity for the purpose of improved algorithmic performance. Unfortunately, the Hessian $\mathbf{H}$ is typically a full matrix and therefore not sparse, which rules out the possibility of exploiting training-sample sparsity.

To overcome these limitations, we may resort to one of the following *approximation procedures*:

(i) *Diagonal approximation:* (Becker and LeCun, 1989). In this procedure, only the diagonal elements of the Hessian are retained, which means that the inverse matrix $\mathbf{H}^{-1}$ will likewise be a diagonal matrix. Matrix theory teaches us that the matrix product $\mathbf{H}^{-1}\mathbf{g}$ will consist of a sum of terms of the form $h_{ii}^{-1}g_i$, where h_{ii} is the ith diagonal element of the Hessian $\mathbf{H}$ and g_i is the corresponding element of the gradient $\mathbf{g}$ for $i = 1, 2, ..., W$. With the gradient vector $\mathbf{g}$ being linear in the weights, it follows that the computational complexity of the approximated second-order on-line learning algorithm is $O(W)$.

(ii) *Low-rank approximation:* (LeCun et al., 1998). By definition, the rank of a matrix equals the number of algebraically independent columns of the matrix. Given a Hessian $\mathbf{H}$, the use of *singular value decomposition* (SVD) provides an important procedure for the low-rank approximation of the Hessian $\mathbf{H}$. Let the rank of $\mathbf{H}$ be denoted by p and a rank r approximation of $\mathbf{H}$ be denoted by $\mathbf{H}_r$, where $r < p$. The squared error between the Hessian and its approximation is defined by the *Frobenius norm*

$$e^2 = \text{tr}[(\mathbf{H} - \mathbf{H}_r)^T(\mathbf{H} - \mathbf{H}_r)] \tag{4.148}$$

where tr[·] denotes the *trace* (i.e., sum of the diagonal components) of the square matrix enclosed inside the square brackets. Applying the SVD to the matrices $\mathbf{H}$ and $\mathbf{H}_r$, we write

$$\mathbf{H} = \mathbf{V}\Sigma\mathbf{U}^T \tag{4.149}$$

and

$$\mathbf{H}_r = \mathbf{V}\Sigma_r\mathbf{U}^T \tag{4.150}$$

where the orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$ define the common *right* and *left singular vectors*, respectively, and the rectangular matrix

$$\Sigma_r = \text{diag}[\lambda_1, \lambda_2, ..., \lambda_r, 0 ..., 0] \tag{4.151}$$

defines the *singular values* of the low-rank approximation $\mathbf{H}_r$. The new square matrix

$$\mathbf{H}_r = \mathbf{U}\Sigma_r\mathbf{V}^T \tag{4.152}$$

provides the *least-squares*, *rank r approximation* to the Hessian $\mathbf{H}$ (Scharf, 1991). Correspondingly, the use of the new matrix $\mathbf{H}_r$ in place of the Hessian $\mathbf{H}$ in the on-line learning algorithm of Eq. (4.147) reduces the computational complexity of the algorithm to somewhere between $O(W)$ and $O(W^2)$.

(iii) *BFGS approximation:* (Schraudolph et al., 2007). As pointed out previously in this section, the BFGS algorithm is considered to be the best form of a quasi-Newton

method. In the 2007 paper by Schraudolph et al., the BFGS algorithm is modified in both its full and limited versions of memory such that it becomes usable for the stochastic aproximation of gradients. The modified algorithm appears to provide a fast, scalable, stochastic quasi-Newton procedure for on-line convex optimization. In Yu et al. (2008), the BFGS quasi-Newton method and its limited-memory variant are extended to deal with non-smooth convex objective functions.

4.17 CONVOLUTIONAL NETWORKS

Up to this point, we have been concerned with the algorithmic design of multilayer perceptrons and related issues. In this section, we focus on the structural layout of the multilayer perceptron itself. In particular, we describe a special class of multilayer perceptrons known collectively as *convolutional networks*, which are well suited for pattern classification. The idea behind the development of these networks is neurobiologically motivated, going back to the pioneering work of Hubel and Wiesel (1962, 1977) on locally sensitive and orientation-selective neurons of the visual cortex of a cat.

A *convolutional network* is a multilayer perceptron designed specifically to recognize two-dimensional shapes with a high degree of invariance to translation, scaling, skewing, and other forms of distortion. This difficult task is learned in a supervised manner by means of a network whose structure includes the following forms of *constraints* (LeCun and Bengio, 2003):

1. *Feature extraction.* Each neuron takes its synaptic inputs from a local *receptive field* in the previous layer, thereby forcing it to extract local features. Once a feature has been extracted, its exact location becomes less important, so long as its position relative to other features is approximately preserved.

2. *Feature mapping.* Each computational layer of the network is composed of multiple *feature maps*, with each feature map being in the form of a plane within which the individual neurons are *constrained* to share the same set of synaptic weights. This second form of structural constraint has the following beneficial effects:

- *shift invariance*, forced into the operation of a feature map through the use of *convolution* with a kernel of small size, followed by a sigmoid function;
- *reduction in the number of free parameters*, accomplished through the use of *weight sharing*.

3. *Subsampling.* Each convolutional layer is followed by a computational layer that performs *local averaging* and *subsampling*, whereby the resolution of the feature map is reduced. This operation has the effect of reducing the sensitivity of the feature map's output to shifts and other forms of distortion.

We emphasize that all weights in all layers of a convolutional network are learned through training. Moreover, the network learns to extract its own features automatically.

Figure 4.23 shows the architectural layout of a convolutional network made up of an input layer, four hidden layers, and an output layer. This network is designed to perform *image processing* (e.g., recognition of handwritten characters). The input layer, made up of 28×28 sensory nodes, receives the images of different characters that have

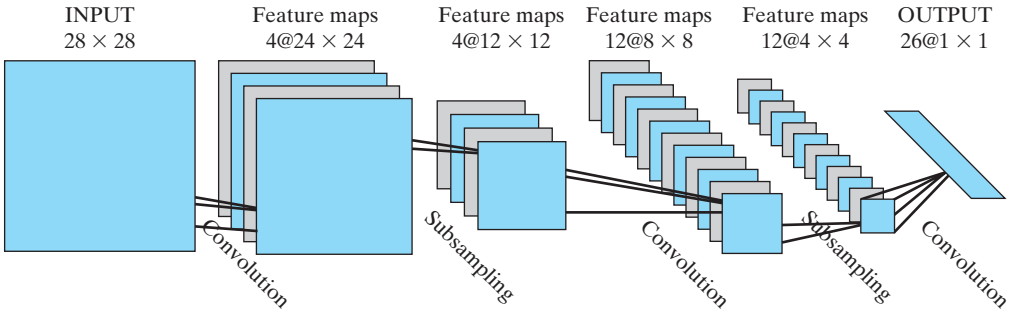


FIGURE 4.23 Convolutional network for image processing such as handwriting recognition. (Reproduced with permission of MIT Press.)

been approximately centered and normalized in size. Thereafter, the computational layouts alternate between convolution and subsampling:

1. The first hidden layer performs convolution. It consists of four feature maps, with each feature map consisting of 24×24 neurons. Each neuron is assigned a receptive field of size 5×5.
2. The second hidden layer performs subsampling and local averaging. It also consists of four feature maps, but each feature map is now made up of 12×12 neurons. Each neuron has a receptive field of size 2×2, a trainable coefficient, a trainable bias, and a sigmoid activation function. The trainable coefficient and bias control the operating point of the neuron; for example, if the coefficient is small, the neuron operates in a quasilinear mode.
3. The third hidden layer performs a second convolution. It consists of 12 feature maps, with each feature map consisting of 8×8 neurons. Each neuron in this hidden layer may have synaptic connections from several feature maps in the previous hidden layer. Otherwise, it operates in a manner similar to the first convolutional layer.
4. The fourth hidden layer performs a second subsampling and local averaging. It consists of 12 feature maps, but with each feature map consisting of 4×4 neurons. Otherwise, it operates in a manner similar to the first subsampling layer.
5. The output layer performs one final stage of convolution. It consists of 26 neurons, with each neuron assigned to one of 26 possible characters. As before, each neuron is assigned a receptive field of size 4×4.

With the successive computational layers alternating between convolution and subsampling, we get a "bipyramidal" effect. That is, at each convolutional or subsampling layer, the number of feature maps is increased while the spatial resolution is reduced, compared with the corresponding previous layer. The idea of convolution followed by subsampling is inspired by the notion of "simple" cells followed by "complex" cells[14] that was first described in Hubel and Wiesel (1962).

The multilayer perceptron described in Fig. 4.23 contains approximately 100,000 synaptic connections, but only about 2,600 free parameters. This dramatic reduction in

the number of free parameters is achieved through the use of weight sharing. The capacity of the learning machine is thereby reduced, which in turn improves the machine's generalization ability. What is even more remarkable is the fact that the adjustments to the free parameters of the network are made by using the stochastic mode of back-propagation learning.

Another noteworthy point is that the use of weight sharing makes it possible to implement the convolutional network in parallel form. This is another advantage of the convolutional network over a fully connected multilayer perceptron.

The lesson to be learned from the convolutional network of Fig. 4.23 is twofold. First, a multilayer perceptron of manageable size is able to learn a complex, high-dimensional, nonlinear mapping by *constraining* its design through the incorporation of prior knowledge about the task at hand. Second, the synaptic weights and bias levels can be learned by cycling the simple back-propagation algorithm through the training sample.

4.18 NONLINEAR FILTERING

The prototypical use of a static neural network, exemplified by the multilayer perceptron, is in *structural pattern recognition*; insofar as applications are concerned, much of the material presented in this chapter has focused on structural pattern recognition. In contrast, in *temporal pattern recognition*, or *nonlinear filtering*, the requirement is to process patterns that evolve over time, with the response at a particular instant of time depending not only on the present value of the input signal, but also on past values. Simply put, *time* is an ordered quantity that constitutes an important ingredient of the learning process in temporal-pattern-recognition tasks.

For a neural network to be dynamic, it must be given *short-term memory* in one form or another. A simple way of accomplishing this modification is through the use of *time delays*, which can be implemented at the synaptic level inside the network or externally at the input layer of the network. Indeed, the use of time delays in neural networks is neurobiologically motivated, since it is well known that signal delays are omnipresent in the brain and play an important role in neurobiological information processing (Braitenberg, 1967, 1977, 1986; Miller, 1987). Time may therefore be built into the operation of a neural network in two basic ways:

- *Implicit representation.* Time is represented by the effect it has on signal processing in an implicit manner. For example, in a digital implementation of the neural network, the input signal is *uniformly sampled*, and the sequence of synaptic weights of each neuron connected to the input layer of the network is convolved with a different sequence of input samples. In so doing, the temporal structure of the input signal is embedded in the spatial structure of the network.
- *Explicit representation.* Time is given its own particular representation inside the network structure. For example, the echolocation system of a bat operates by emitting a short frequency-modulated (FM) signal, so that the same intensity level is maintained for each frequency channel restricted to a very short period within the

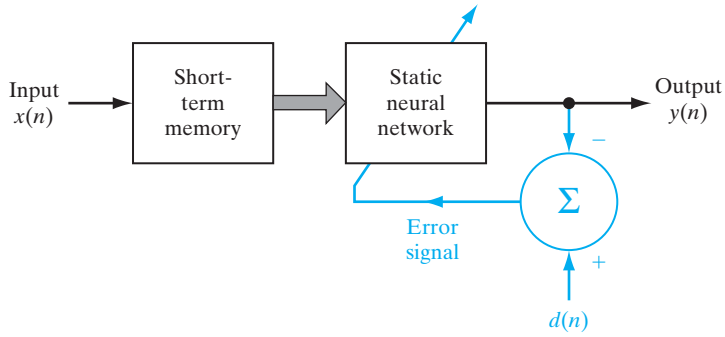


FIGURE 4.24 Nonlinear filter built on a static neural network.

FM sweep. Multiple comparisons between several different frequencies encoded by an array of auditory receptors are made for the purpose of extracting accurate distance (range) information about a target (Suga and Kanwal, 1995). When an echo is received from the target with an unknown delay, a neuron (in the auditory system) with a matching delay line responds, thereby providing an estimate of the range to the target.

In this section, we are concerned with the implicit representation of time, whereby a static neural network (e.g., multilayer perceptron) is provided with *dynamic* properties by external means.

Figure 4.24 shows the block diagram of a *nonlinear filter* consisting of the cascade connection of two subsystems: short-term memory and a static neural network (e.g., multilayer perceptron). This structure provides for a clear-cut separation of processing roles: The static network accounts for nonlinearity, and the memory accounts for time. To be specific, suppose we are given a multilayer perceptron with an input layer of size m. Then, in a corresponding way, the memory is a *single-input*, *multiple-output* (SIMO) structure providing m differently delayed versions of the input signal for stimulating the neural network.

Short-Term Memory Structures

Figure 4.25 shows the block diagram of a *discrete-time memory structure* consisting of p identical sections connected in cascade. Each section is characterized by an impulse response, denoted by $h(n)$, where n denotes discrete time. The number of sections, p, is

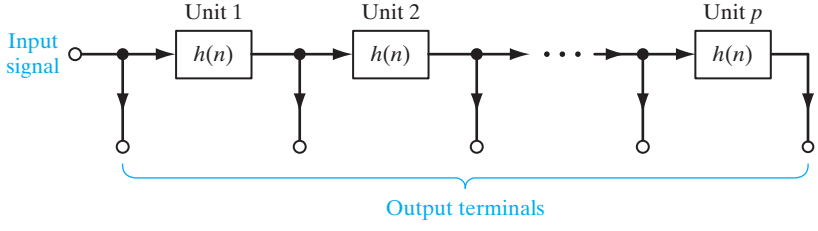


FIGURE 4.25 Generalized tapped-delay-line memory of order p.

called the *order of the memory.* Correspondingly, the number of output terminals (i.e., taps) provided by the memory is $p + 1$, which includes the direct connection from the input to the output. Thus, with m denoting the size of the input layer of the static neural network, we may set

$$m = p + 1$$

The impulse response of each delay section of the memory satisfies two properties:

- *causality*, which means that $h(n)$ is zero for $n < 0$;
- *normalization*, which means that $\sum_{n=0}^{\infty} |h(n)| = 1$.

On this basis, we may refer to $h(n)$ as the *generating kernel* of the discrete-time memory.

The attributes of a memory structure are measured in terms of depth and resolution (deVries and Principe, 1992). Let $h_{\text{overall}}(n)$ denote the overall impulse response of the memory. With p memory sections, it follows that $h_{\text{overall}}(n)$ is defined by p successive convolutions of the impulse response $h(n)$. Accordingly, the *memory depth* D is defined as the *first time moment* of $h_{\text{overall}}(n)$, namely,

$$D = \sum_{n=0}^{\infty} n h_{\text{overall}}(n) \tag{4.153}$$

A memory of low depth D holds information content for a relatively short time interval, whereas a high-depth memory holds it much further into the past. *Memory resolution* R is defined as *the number of taps in the memory structure per unit of time.* A memory of high resolution is therefore able to hold information about the input sequence at a fine level, whereas a low-resolution memory can do so only at a coarser level. For a fixed memory order p, the product of memory depth D and memory resolution R is a constant that turns out to be equal to p.

Naturally, different choices of the generating kernel $h(n)$ result in different values for the depth D and memory resolution R, as illustrated by the following two memory structures:

1. *Tapped-delay-line memory*, for which the generating kernel is simply defined by the unit impulse $\delta(n)$; that is,

$$h(n) = \delta(n) = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases} \tag{4.154}$$

Correspondingly, the overall impulse response is

$$h_{\text{overall}}(n) = \delta(n - p) = \begin{cases} 1, & n = p \\ 0, & n \neq p \end{cases} \tag{4.155}$$

Substituting Eq. (4.155) into Eq. (4.153) yields the memory depth $D = p$, which is intuitively satisfying. Moreover, since there is only one tap per time unit, it follows that the resolution $R = 1$, yielding a depth-resolution product equal to p.

2. *Gamma memory*, for which the generating kernel is defined by

$$h(n) = \mu(1 - \mu)^{n-1}, \qquad n \geq 1 \tag{4.156}$$

where μ is an adjustable parameter (deVries and Principe, 1992). For $h(n)$ to be convergent (i.e., for the short-term memory to be stable), we require that

$$0 < \mu < 2$$

Correspondingly, the overall impulse response of the gamma memory is

$$h_{\text{overall}}(n) = \binom{n-1}{p-1}\mu^p(1 - \mu)^{n-p}, \qquad n \geq p \tag{4.157}$$

where $\binom{\cdot}{\cdot}$ is a binomial coefficient. The impulse response $h_{\text{overall}}(n)$ for varying p represents a discrete version of the integrand of the gamma function (deVries and Principe, 1992)—hence the name "gamma memory." Figure 4.26 plots the impulse response $h_{\text{overall}}(n)$, normalized with respect to μ, for varying memory order $p = 1, 2, 3, 4$ and $\mu = 0.7$. Note also that the time axis has been scaled by the parameter μ, which has the effect of positioning the peak value of $h_{\text{overall}}(n)$ at $n = p - 1$.

It turns out that the depth of gamma memory is (p/μ) and the resolution is μ, again producing a depth-resolution product equal to p. Accordingly, by choosing μ to be less than unity, the gamma memory produces improvement in depth at the expense of resolution. For the special case of $\mu = 1$, the gamma memory reduces to an ordinary tapped-delay-line memory where each section consists simply of a unit-time delay operator.

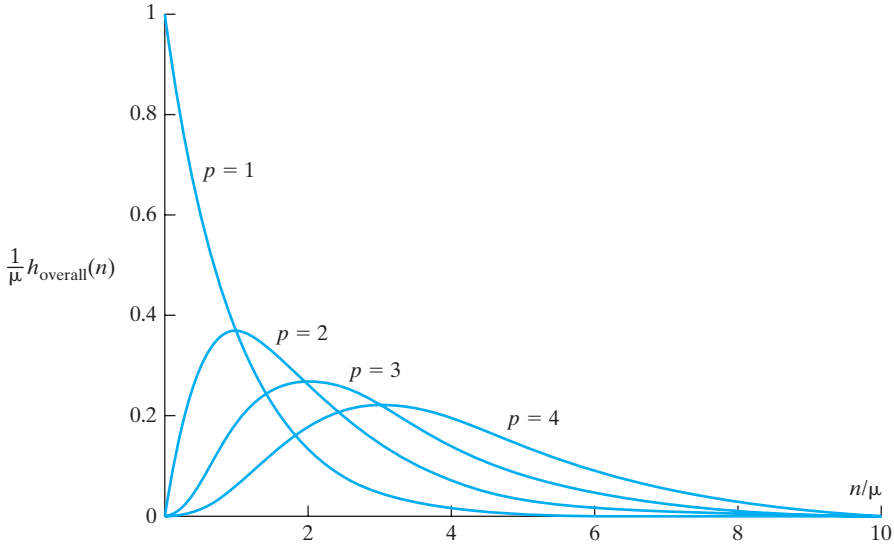


FIGURE 4.26 Family of impulse responses of the gamma memory for order $p = 1, 2, 3, 4$ and $\mu = 0.7$.

Universal Myopic Mapping Theorem

The nonlinear filter of Fig. 4.24 may be generalized to that shown in Fig. 4.27. This generic dynamic structure consists of two functional blocks. The block labeled $\{h_j\}_{j=1}^{L}$ represents *multiple convolutions* in the time domain, that is, a bank of *linear filters* operating in parallel. The h_j are drawn from a large set of real-valued kernels, each one of which represents the impulse response of a linear filter. The block labeled $\mathcal{N}$ represents a static (i.e., memoryless) nonlinear feedforward network such as the multilayer perceptron. The structure of Fig. 4.27 is a *universal dynamic mapper.* In Sandberg and Xu (1997a), it is shown that any shift-invariant *myopic map* can be uniformly approximated arbitrarily well by a structure of the form depicted in Fig. 4.27 under mild conditions. The requirement that a map be myopic is equivalent to "uniformly fading memory"; it is assumed here that the map is *causal*, which means that an output signal is produced by the map at time $n \geq 0$ only when the input signal is applied at time $n = 0$. By "shift invariant," we mean the following: If $y(n)$ is the output of the map generated by an input $x(n)$, then the output of the map generated by the shifted input $x(n - n_0)$ is $y(n - n_0)$, where the time shift n_0 is an integer. In Sandberg and Xu (1997b), it is further shown that for any single-variable, shift-invariant, causal, uniformly fading memory map, there is a gamma memory and static neural network, the combination of which approximates the map uniformly and arbitrarily well.

We may now formally state the *universal myopic mapping theorem*[15] as follows (Sandberg and Xu, 1997a, 1997b):

> *Any shift-invariant myopic dynamic map can be uniformly approximated arbitrarily well by a structure consisting of two functional blocks: a bank of linear filters feeding a static neural network.*

As already mentioned, a multilayer perceptron may serve the role of the static network. It is also noteworthy that this theorem holds when the input and output signals are functions of a finite number of variables, as in image processing, for example.

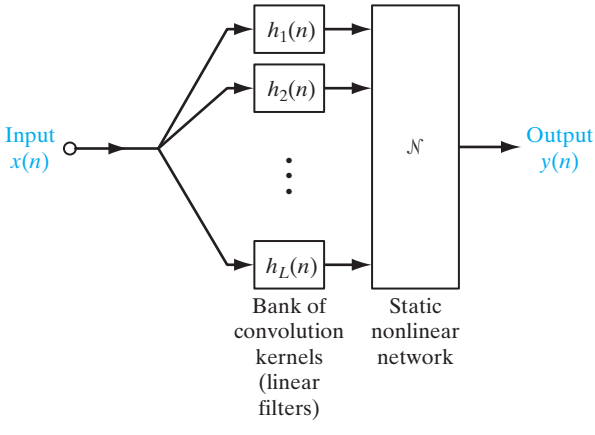


FIGURE 4.27 Generic structure for universal myopic mapping theorem.

Practical Implications of the Theorem

The universal myopic mapping theorem has profound practical implications:

1. The theorem provides justification for *NETtalk*, which was the first demonstration of a massively parallel distributed network that converts English speech to phonemes; a *phoneme* is a basic linguistic unit (Sejnowski and Rosenberg, 1987). Figure 4.28 shows a schematic diagram of the NETtalk system, based on a multilayer perceptron with an input layer of 203 sensory (source) nodes, a hidden layer of 80 neurons, and an output layer of 26 neurons. All the neurons used sigmoid (logistic) activation functions. The synaptic connections in the network were specified by a total of 18,629 weights, including a variable threshold for each neuron; threshold is the negative of bias. The standard back-propagation algorithm was used to train the network. The network had seven groups of nodes in the input layer, with each group encoding one letter of the input text. Strings of seven letters were thus presented to the input layer at any one time. The desired response for the training process was specified as the correct phoneme associated with the center (i.e., fourth) letter in the seven-letter window. The other six letters (three on either side of the center letter) provided a partial *context* for each decision made by the network. The text was stepped through the window on a letter-by-letter basis. At each step in the process, the network computed a phoneme, and after each word the synaptic weights of the network were adjusted according to how closely the computed pronunciation matched the correct one.

The performance of NETtalk exhibited some similarities with observed human performance, as summarized here (Sejnowski and Rosenberg, 1987):

- The training followed a power law.
- The more words the network learned, the better it was at generalizing and correctly pronouncing new words.
- The performance of the network degraded very slowly as synaptic connections in the network were purposely damaged.
- Relearning after damage to the network was much faster than learning during the original training.

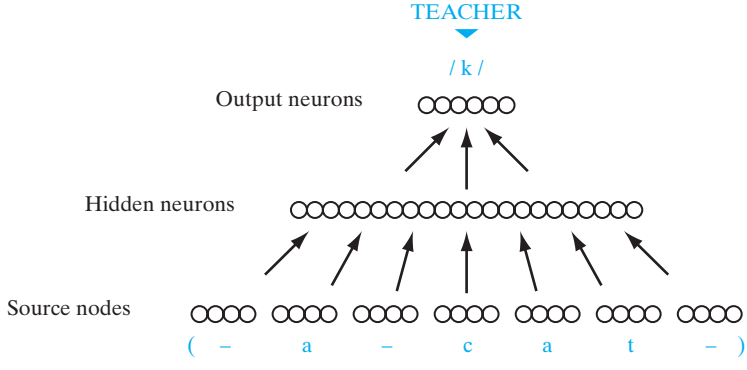


FIGURE 4.28 Schematic diagram of the NETtalk network architecture.

NETtalk was a brilliant illustration in miniature of many aspects of learning, starting out with considerable "innate" knowledge of its input patterns and then gradually acquiring competence at converting English speech to phonemes through practice.

2. The universal myopic theorem lays down the framework for the design of more elaborate models of nonlinear systems. The multiple convolutions at the front end of the structure in Fig. 4.27 may be implemented using linear filters with a finite-duration impulse response (FIR) or infinite-duration impulse response (IIR). Most importantly, the structure of Fig. 4.27 is *inherently stable*, provided that the linear filters are themselves stable. We thus have a clear-cut separation of roles as to how to take care of short-term memory and memoryless nonlinearity in building a stable dynamic system.
3. Given a stationary time series $x(1), x(2), ..., x(n)$, we may use the universal myopic mapping structure of Fig. 4.27 to build a predictive model of the underlying nonlinear physical laws responsible for generating the time series by setting $y(n) = x(n + 1)$, no matter how complex the laws are. In effect, the future sample $x(n + 1)$ plays the role of desired response. When a multilayer perceptron is used as the static network in Fig. 4.27 for such an application, it is advisable to provide a linear neuron for the output unit in the network. This provision will ensure that no amplitude limitation is placed on the dynamic range of the predictive model.

4.19 SMALL-SCALE VERSUS LARGE-SCALE LEARNING PROBLEMS

At various points along the way in this chapter and other parts of the book, we have made references to small-scale and large-scale learning problems. However, we did not elaborate rigorously on the meaning of these two kinds of supervised learning. The purpose of this section is to sharpen the statistical and computational issues that distinguish them from each other.

We begin the discussion with structural risk minimization (SRM), which is entirely statistical in nature; SRM is adequate for dealing with small-scale learning problems. Then we broaden the discussion by considering computational issues that assume a prominent role in dealing with large-scale learning problems.

Structural Risk Minimization

The feasibility of supervised learning depends on the following key question:

Does a training sample consisting of N independent and identically distributed examples

$$(\mathbf{x}_1, d_1), (\mathbf{x}_2, d_2), ..., (\mathbf{x}_N, d_N)$$

contain sufficient information to construct a learning machine capable of good generalization performance?

The answer to this fundamental question lies in the method of *structural risk minimization*, described by Vapnik (1982, 1998).

To describe what we mean by this method, let the natural source or environment responsible for generating the training sample be represented by the *nonlinear* regression model

$$d = f(\mathbf{x}) + \varepsilon \tag{4.158}$$

where, following the terminology introduced in Chapter 2, the vector $\mathbf{x}$ is the regressor, the scalar d is the response, and ε is the explanational (modeling) error. The function f is the unknown, and the objective is to estimate it. To do this estimation, we define the expected risk (i.e., the ensemble-averaged cost function) as

$$J_{\text{actual}}(f) = \mathbb{E}_{\mathbf{x},d}\left[\frac{1}{2}(d - f(\mathbf{x}))^2\right] \tag{4.159}$$

where the expectation is performed jointly with respect to the regressor–response pair $(\mathbf{x}, d)$. In Chapter 5, we will show that the *conditional mean estimator*

$$\hat{f}^* = \mathbb{E}[d|\mathbf{x}] \tag{4.160}$$

is the minimizer of the cost function $J_{\text{actual}}(f)$. Correspondingly, we write $J_{\text{actual}}(\hat{f}^*)$ as the minimum value of the cost function defined in Eq. (4.159); it serves as the *absolute optimum* that is achievable.

Determination of the conditional mean estimator $\hat{f}^*$ requires knowledge of the underlying joint probability distribution of the regressor $\mathbf{x}$ and the response d. Typically, however, we find that this knowledge is not available. To circumvent this difficulty, we look to machine learning for a viable solution. Suppose, for example, we choose a single-layer multilayer perceptron to do the machine learning. Let the function $F(\mathbf{x}; \mathbf{w})$ denote the input–output relationship of the neural network parameterized by the weight vector $\mathbf{w}$. We then make our *first approximation* by setting

$$f(\mathbf{x}) = F(\mathbf{x}; \mathbf{w}) \tag{4.161}$$

Correspondingly, we formulate the model's cost function as

$$J(\mathbf{w}) = \mathbb{E}_{\mathbf{x},d}\left[\frac{1}{2}(d - F(\mathbf{x}; \mathbf{w}))^2\right] \tag{4.162}$$

where, as before, the expectation is performed jointly with respect to the pair $(\mathbf{x}, d)$. This second cost function is naturally formulated differently from the cost function $J_{\text{actual}}(f)$ pertaining to the source—hence the use of different symbols for them. In imposing the equality of Eq. (4.161) on the neural network, we have in effect restricted the choice of the approximating function $F(\mathbf{x}; \mathbf{w})$.

Let

$$\hat{\mathbf{w}}^* = \arg\min_{\mathbf{w}} J(\mathbf{w}) \tag{4.163}$$

be the minimizer of the cost function $J(\mathbf{w})$. The reality, however, is that even if we can find the minimizer $\hat{\mathbf{w}}^*$, it is highly likely that the resulting cost function $J(\hat{\mathbf{w}}^*)$ will be worse than the minimized cost function $J_{\text{actual}}(\hat{f}^*)$. In any event, we cannot do better than $J_{\text{actual}}(\hat{f}^*)$, and thus we write

$$J(\hat{\mathbf{w}}^*) > J_{\text{actual}}(\hat{f}^*) \tag{4.164}$$

Unfortunately, we are still faced with the same practical problem as before in that we may not know the underlying joint probability distribution of the pair $(\mathbf{x}, d)$. To alleviate this difficulty, we make our *second approximation* by using the empirical risk (i.e., the time-averaged energy function)

$$\mathscr{E}_{\text{av}}(N; \mathbf{w}) = \frac{1}{2N} \sum_{n=1}^{N} (d(n) - F(\mathbf{x}(n); \mathbf{w}))^2 \tag{4.165}$$

whose minimizer is defined by

$$\hat{\mathbf{w}}_N = \arg\min_{\mathbf{w}} \mathscr{E}_{\text{av}}(N; \mathbf{w}) \tag{4.166}$$

Clearly, the minimized cost function $J(\hat{\mathbf{w}}_N)$ cannot be smaller than $J(\hat{\mathbf{w}}^*)$. Indeed, it is highly likely to find that

$$J(\hat{\mathbf{w}}_N) > J(\hat{\mathbf{w}}^*) > J_{\text{actual}}(\hat{f}^*) \tag{4.167}$$

With the two approximations that have been made, we may wonder why we should compute the minimum $\hat{\mathbf{w}}_N$ exactly. Before addressing this question, let us examine what happens when the example multilayer perceptron is changed by enlarging the size of the hidden layer.

From Section 4.12, we recall that the multilayer perceptron is a universal approximator of the unknown function $f(\mathbf{x})$. In theory, the parameterized function $F(\mathbf{x}; \mathbf{w})$ approximates the unknown function $f(\mathbf{x})$ with any desired accuracy provided that the size of the hidden layer is large enough. This, in turn, means that $J(\hat{\mathbf{w}}^*)$ becomes closer to the absolute optimum $J_{\text{actual}}(\hat{f}^*)$. However, by enlarging the size of the hidden layer, we may compromise the generalization capability of the multilayer perceptron. In particular, it is possible for the error $(J(\hat{\mathbf{w}}^*) - J_{\text{actual}}(\hat{f}^*))$ to increase as a result of enlarging the hidden layer, unless the size of the training sample, N, is correspondingly increased. The issue just discussed is the essence of Vapnik's *structural risk minimization*, which manifests itself in the "approximation–estimation trade-off."

To elaborate further on this trade-off, let the *excess error* $(J(\hat{\mathbf{w}}_N) - J_{\text{actual}}(\hat{f}^*))$ be decomposed into two terms as follows:

$$\underbrace{J(\hat{\mathbf{w}}_N) - J_{\text{actual}}(\hat{f}^*)}_{\text{Excess error}} = \underbrace{J(\hat{\mathbf{w}}_N) - J(\hat{\mathbf{w}}^*)}_{\text{Estimation error}} + \underbrace{J(\hat{\mathbf{w}}^*) - J_{\text{actual}}(\hat{f}^*)}_{\text{Approximation error}} \tag{4.168}$$

In this classical decomposition of errors, the following points are noteworthy:

(i) The *estimation error* provides a measure of how much performance is lost as a result of using a training sample of some prescribed size N. Moreover, with $\hat{\mathbf{w}}_N$ being dependent on the training sample, the approximation error is therefore relevant in the assessment of network training.

(ii) The *approximation error* provides a measure of how much performance is lost by choosing a model characterized by the approximating function $F(\mathbf{x}, \mathbf{w})$. Moreover, with $\hat{f}^*$ being a conditional estimator of the response d given the regressor $\mathbf{x}$, the estimation error is therefore relevant in the assessment of network testing.

In Vapnik's theoretical framework, the approximation and estimation errors are formulated in terms of the *VC dimension,* commonly denoted by h. This new parameter, short for the *Vapnik– Chervonenkis dimension* (Vapnik and Chervonenkis, 1971), is a measure of the *capacity*, or *expressive power*, of a family of binary classification functions realized by the learning machine.[16] For the example of a single-layer multilayer perceptron, the VC dimension is determined by the size of the hidden layer; the larger this size is, the larger the VC dimension h will be.

To put Vapnik's theory in a practical context, consider a family of *nested* approximating network functions denoted by

$$\mathcal{F}_k = \{F(\mathbf{x}; \mathbf{w})(\mathbf{w} \in \mathcal{W}_k)\}, \qquad k = 1, 2, ..., K \tag{4.169}$$

such that we have

$$\mathcal{F}_1 \subset \mathcal{F}_2 \subset \cdots \subset \mathcal{F}_K$$

where the symbol $\subset$ means "is contained in." Correspondingly, the VC dimensions of the individual subsets of $\mathcal{F}_K$ satisfy the condition

$$h_1 < h_2 < \cdots < h_K$$

In effect, the size of $\mathcal{F}_K$ is a measure of the machine capacity. Hereafter, we use the definition of Eq. (4.169) in place of the VC dimension.

Figure 4.29 plots variations of the approximation and estimation errors versus the size K of the family of approximating network functions $\mathcal{F}_K$. For the example of a single-layer multilayer perceptron, the optimum size of the hidden layer is determined by the point at which the approximation error and the estimation error assume a common value. Before the optimum condition is reached, the learning problem is *overdetermined*, which means that the machine capacity is too small for the amount of detail contained in the training sample. Beyond the minimum point, the learning problem is *underdetermined*, which means that the machine capacity is too large for the training sample.

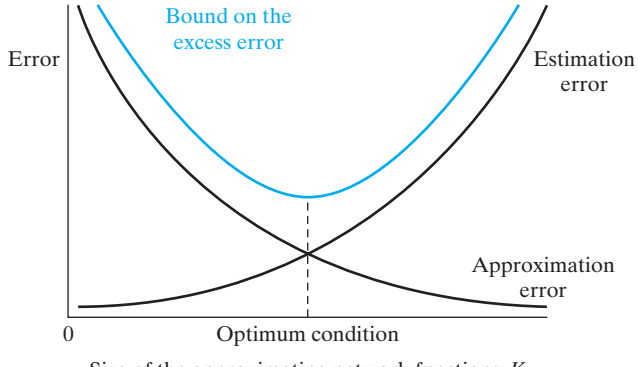


FIGURE 4.29 Variations of the approximation and estimation errors with the size K.

Computational Considerations

The neural network model (e.g., the single-layer multilayer perceptron) must be a *controlled variable*, so that it can be freely adjusted to achieve the best test performance on data not seen before. Another controlled variable is the number of examples to do the training. In order to add practical realism to the supervised-training process, Bottou (2007) has introduced the cost of computation by considering a new controlled variable: *optimization accuracy*.

In practice, it is possible to find that the task of computing the minimizer $\hat{\mathbf{w}}_N$ is rather *costly*. Moreover, in the course of coming up with a satisfactory network design, we usually make many approximations. Suppose, then, we settle on a network model characterized by the weight vector $\tilde{\mathbf{w}}_N$, which is different from $\hat{\mathbf{w}}_N$; in so doing, we will have made our *third*, *and final*, *approximation*. For example, the on-line learning algorithm could be stopped long before convergence has been reached, due to limited computing time. In any event, $\tilde{\mathbf{w}}_N$ is a suboptimal solution that satisfies the condition

$$\mathscr{E}_{\text{av}}(N; \tilde{\mathbf{w}}_N) \leq \mathscr{E}_{\text{av}}(N; \hat{\mathbf{w}}_N) + \rho \tag{4.170}$$

where ρ constitutes a new controlled variable; it provides a measure of *computational accuracy*.

In light of this new practicality, we now have a more complicated problem than that encountered in the method of structural risk minimization. Specifically, we must now adjust three variables:

- the network model (for instance, through the number of hidden neurons in a multilayer perceptron),
- the number of training examples, and
- the optimization accuracy (for instance by prematurely terminating computation of the minimizer $\hat{\mathbf{w}}_N$ and settling on the suboptimal solution $\tilde{\mathbf{w}}_N$).

In order to hit the best test performance, we have to satisfy *budget constraints*, which define the maximum number of training examples that we can use and the maximum computing time that we can afford. In a practical context, we are therefore confronted with a trade-off that is rather complicated. In solving this *constrained-optimization problem*, the trade-off will depend on whether we first hit a limit on the number of examples or impose a limit on the computing time. Which of these two limits is the active budget constraint depends on whether the supervised-learning process is of a small-scale or large-scale kind, as discussed next.

Definitions

According to Bottou (2007), small-scale and large-scale learning problems are respectively defined as follows:

Definition I. Small-scale learning

A supervised-learning problem is said to be of a small-scale kind when the *size of the training sample* (i.e., the number of examples) is the active budget constraint imposed on the learning process.

Definition II. Large-scale learning

A supervised-learning problem is said to be of a large-scale kind when the *computing time* is the active budget constraint imposed on the learning process.

In other words, it is the *active budget constraint* that distinguishes one learning problem from the other.

For an illustrative example of a small-scale learning problem, we may mention the design of an *adaptive equalizer*, the purpose of which is to compensate for the inevitable distortion of information-bearing data transmitted over a communication channel. The LMS algorithm, rooted in stochastic gradient descent and discussed in Chapter 3, is widely used for solving this on-line learning problem (Haykin, 2002).

For an illustrative example of a large-scale learning problem, we may mention the design of a check reader where the training examples consist of joint pairs, each of which describes a particular {*image*, *amount*} pair, where "image" pertains to a check and "amount" pertains to the amount of money inscribed in the check. Such a learning problem has strong structure that is complicated by the following issues (Bottou, 2007):

- field segmentation;
- character segmentation;
- character recognition;
- syntactical interpretation.

The *convolutional network*, embodying differentiable modules as described in Section 4.17 and trained with a stochastic gradient algorithm for a few weeks, is widely used for solving this challenging learning problem (LeCun et al., 1998). Indeed, this novel network has been deployed in industry since 1996, running billions of checks.

Small-Scale Learning Problems

Insofar as small-scale learning problems are concerned, there are three variables available to the designer of a learning machine:

- the number of training examples, N;
- the permissible size K of the family of approximating network functions $\mathcal{F}$;
- the computational error ρ introduced in Eq. (4.170).

With the active budget constraint being the number of examples, the design options in learning problems of the first kind are as follows (Bottou, 2007):

- Reduce the estimation error by making N as large as the budget permits.
- Reduce the optimization error by setting the computational error $\rho = 0$, which means setting $\tilde{\mathbf{w}}_N = \hat{\mathbf{w}}_N$.
- Adjust the size of $\mathcal{F}$ to the extent deemed to be reasonable.

With $\rho = 0$, the method of structural risk minimization, involving the approximation–estimation tradeoff illustrated in Fig. 4.29, is adequate for dealing with small-scale learning problems.

Large-Scale Learning Problems

As pointed out previously, the active budget constraint in large-scale learning problems is the computing time. In tackling learning problems of this second kind, we face *more complicated trade-offs* because we now have to account for the computing time T.

In large-scale learning problems, the excess error is defined by the difference $(J(\tilde{\mathbf{w}}_N) - J_{\text{actual}}(\hat{f}^*))$, which is decomposed into three terms, as shown by the following (Bottou, 2007):

$$\underbrace{J(\tilde{\mathbf{w}}_N) - J_{\text{actual}}(\hat{f}^*)}_{\text{Excess error}} = \underbrace{J(\tilde{\mathbf{w}}_N) - J(\hat{\mathbf{w}}_N)}_{\text{Optimization error}} + \underbrace{J(\hat{\mathbf{w}}_N) - J(\hat{\mathbf{w}}^*)}_{\text{Estimation error}} + \underbrace{J(\hat{\mathbf{w}}^*) - J_{\text{actual}}(\hat{f}^*)}_{\text{Approximation error}} \tag{4.171}$$

The last two terms, constituting the approximation and estimation errors, are common to both small-scale and large-scale learning problems. It is the first term in Eq. (4.171) that distinguishes large-scale learning problems from small-scale ones. This new term, called the *optimization error*, is obviously related to the computational error ρ.

Computation of the *bound* on the approximation error, depicted in Fig. 4.29, is reasonably well understood (in terms of the VC theory) for small-scale learning problems. Unfortunately, the constants involved in the formula for this bound are quite bad when the formula is applied to large-scale learning problems. In these more difficult situations, it is therefore more productive to analyze Eq. (4.171) in terms of convergence rates rather than bounds.

The requirement is to minimize the sum of the three terms in Eq. (4.171) by adjusting the available variables:

- the number of examples, N;
- the permissible size K of approximating network functions, $\mathcal{F}_K$;
- the computational error ρ, which is no longer zero.

Doing this minimization analytically is extremely difficult, due to the fact that the computing time T is actually dependent on all three variables N, $\mathcal{F}$, and ρ. To illustrate the consequences of this dependence, suppose we assign a small value to the error ρ so as to reduce the optimization error. To realize this reduction, unfortunately, we must also increase N, $\mathcal{F}$, or both, any of which would have undesirable effects on the approximation and estimation errors.

Nevertheless, in some cases, it is possible to compute the exponents with respect to which the three errors tend to decrease when ρ decreases and both $\mathcal{F}$ and N increase. Similarly, it is possible to identify the exponents with respect to which the computing time T increases when ρ decreases and both $\mathcal{F}$ and N increase. Putting these pieces together, we have the elements for an approximate solution to trade-offs in tackling large-scale

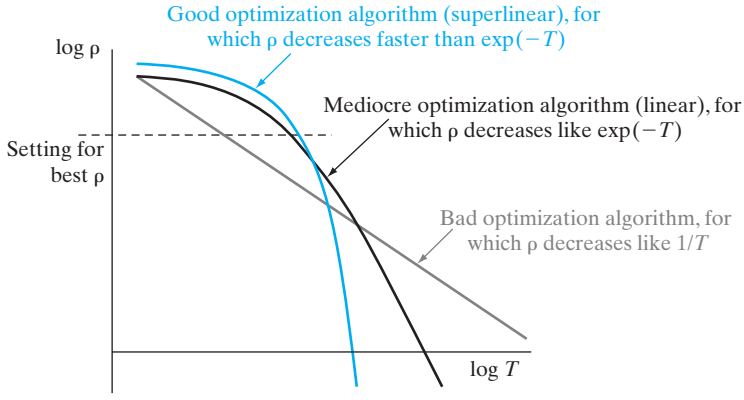


FIGURE 4.30 Variations of the computational error ρ versus the computation time T for three classes of optimization algorithm: bad, mediocre, and good. (This figure is reproduced with the permission of Dr. Leon Bottou.)

learning problems. Most importantly, in the final analysis, the trade-offs depend on the choice of the optimization algorithm.

Figure 4.30 illustrates how a plot of log ρ versus log T is affected by the type of optimization algorithm used to solve a large-scale learning problem. Three categories of optimization algorithms are identified in this figure—namely, *bad*, *mediocre*, and *good*—examples of which respectively include stochastic gradient descent (i.e., on-line learning), gradient descent (i.e., batch learning), and second-order gradient descent (e.g., quasi-Newton optimization algorithm of the BFGS kind or its extension). Table 4.4 summarizes the distinguishing features of these three categories of optimization algorithms.

TABLE 4.4 Summary of Statistical Characteristics of Three Optimization Algorithms*

Algorithm	Cost per iteration	Time to reach ρ
1. Stochastic gradient descent (on-line learning)	$O(m)$	$O\left(\frac{1}{\rho}\right)$
2. Gradient descent (batch learning)	$O(Nm)$	$O\left(\log\frac{1}{\rho}\right)$
3. Second-order gradient descent (on-line learning)	$O(m(m+N))$	$O\left(\log\left(\log\frac{1}{\rho}\right)\right)$

m : dimension of input vector $\mathbf{x}$
N : number of examples used in training
ρ : computational error

*This table is compiled from Bottou (2007).

The message to take from the material presented in this section on supervised learning may now be summed up as follows:

Whereas the study of small-scale learning problems is well-developed, the study of large-scale learning problems is in its early stages of development.

4.20 SUMMARY AND DISCUSSION

The back-propagation algorithm has established itself as a computationally efficient and useful algorithm for the training of multilayer perceptrons. The algorithm derives its name from the fact that the partial derivatives of the cost function (performance measure) with respect to the free parameters (synaptic weights and biases) of the network are determined by back-propagating the error signals (computed by the output neurons) through the network, layer by layer. In so doing, the algorithm solves the credit-assignment problem in a most elegant fashion. The computing power of the algorithm lies in its two main attributes:

- the *local* method, for updating the synaptic weights and biases of the multilayer perceptron;
- the *efficient* method, for computing *all* the partial derivatives of the cost function with respect to these free parameters.

Stochastic and Batch Methods of Training

For a given epoch of training data, the back-propagation algorithm operates in one of two modes: stochastic or batch. In the stochastic mode, the synaptic weights of all neurons in the network are adjusted in a sequential manner, pattern by pattern. Consequently, estimation of the gradient vector of the error surface used in the computation is stochastic in nature—hence the name "stochastic back-propagation learning." On the other hand, in the batch mode, the adjustments to all synaptic weights and biases are made on an epoch-by-epoch basis, with the result that a more accurate estimate of the gradient vector is utilized in the computation. Despite its disadvantages, the stochastic form of back-propagation learning is most frequently used for the training of multilayer perceptrons, particularly for large-scale problems. To achieve the best results, however, careful tuning of the algorithm is required.

Pattern Classification and Nonlinear Filtering

The specific details involved in the design of a multilayer perceptron naturally depend on the application of interest. We may, however, make two distinctions:

1. In pattern classification involving nonlinearly separable patterns, all the neurons in the network are *nonlinear*. The nonlinearity is achieved by employing a sigmoid function, two commonly used forms of which are (a) the logistic function, and (b) the hyperbolic tangent function. Each neuron is responsible for producing a hyperplane of

its own in decision space. Through a supervised learning process, the combination of hyperplanes formed by all the neurons in the network is iteratively adjusted in order to separate patterns drawn from the different classes and not seen before, with the fewest classification errors on average. For pattern classfication, the stochastic back-propagation algorithm is widely used to perform the training, particularly for large-scale problems (e.g., optical character recognition).

2. In nonlinear filtering, the *dynamic range* at the output of the multilayer perceptron should be sufficiently large to accommodate the process values; in this context, the use of linear output neurons is the most sensible choice. As for learning algorithms, we offer the following observations:

- On-line learning is much slower than batch learning.
- Assuming that batch learning is the desired choice, the standard back-propagation algorithm is slower than the conjugate gradient algorithm.

The method of nonlinear filtering, discussed in this chapter, focused on the use of a static network, exemplified by the multilayer perceptron; the input signal is applied to the multilayer perceptron through a short-term memory structure (e.g., tapped delay line or gamma filter) that provides for time, which is an essential dimension of filtering. In Chapter 15, we revisit the design of nonlinear filters for which feedback is applied to a multilayer perceptron, turning it into a recurrent neural network.

Small-scale versus Large-scale Learning Problems

Generally speaking, there are three kinds of error that can arise in the study of machine-learning problems:

1. *Approximation error*, which refers to the error incurred in the training of a neural network or learning machine, given a training sample of some finite size N.

2. *Estimation error*, which refers to the error incurred when the training of the machine is completed and its performance is tested using data not seen before; in effect, estimation error is another way of referring to generalization error.

3. *Optimization error*, the presence of which is attributed to accuracy of the computation involved in training the machine for some prescribed computing time T.

In small-scale learning problems, we find that the *active budget constraint* is the size of the training sample, which implicily means that the optimization error is usually zero in practice. Vapnik's theory of structural risk minimization is therefore adequately equipped to handle small-scale learning problems. On the other hand, in large-scale learning problems, the active budget constraint is the available computing time, T, with the result that the optimization error takes on a critical role of its own. In particular, computational accuracy of the learning process and therefore the optimization error are both strongly affected by the type of optimization algorithm employed to solve the learning problem.

NOTES AND REFERENCES

1. Sigmoid functions are "S" shaped graphs; Mennon et al. (1996) present a detailed study of two classes of sigmoids:
   - *simple sigmoids*, defined to be odd, asymptotically bounded, and completely monotone functions of one variable;
   - *hyperbolic sigmoids*, representing a proper subset of simple sigmoids and a natural generalization of the hyperbolic tangent function.
2. For the special case of the LMS algorithm, it has been shown that use of the momentum constant α reduces the stable range of the learning-rate parameter η and could thus lead to instability if η is not adjusted appropriately. Moreover, the misadjustment increases with increasing α; for details, see Roy and Shynk (1990).
3. A vector $\mathbf{w}^*$ is said to be a *local minimum* of an input—output function F if it is no worse than its *neighbors*—that is, if there exists an ε such that

$$F(\mathbf{w}^*) \leq F(\mathbf{w}) \qquad \text{for all } \mathbf{w} \text{ with } \|\mathbf{w} - \mathbf{w}^*\| < \varepsilon$$

   (Bertsekas, 1995). The vector $\mathbf{w}^*$ is said to be a *global minimum* of the function F if it is no worse than *all* other vectors—that is,

$$F(\mathbf{w}^*) \leq F(\mathbf{w}) \qquad \text{for all } \mathbf{w} \in \mathbb{R}^n$$

   where n is the dimension of $\mathbf{w}$.
4. The first documented description of the use of back propagation for efficient gradient evaluation is attributed to Werbos (1974). The material presented in Section 4.8 follows the treatment given in Saarinen et al. (1992); a more general discussion of the topic is presented by Werbos (1990).
5. Battiti (1992) presents a review of exact and approximate algorithms for computing the Hessian, with particular reference to neural networks.
6. Müller et al. (1998) have studied the application of the annealed on-line learning algorithm of Eq. (4.77) to a nonstationary blind source separation problem, which illustrates the broad algorithmic applicability of adaptive control of the learning rate due to Murata (1998). The issue of blind source separation is discussed in Chapter 10.
7. The formulation of Eq. (4.80) follows a corresponding part of the optimally annealed online learning algorithm due to Sompolinski et al. (1995) that deals with adaptation of the learning-rate parameter. Practical limitations of this algorithm include the need to compute the Hessian at each iteration and the need to know the minimal loss of the learning curve.
8. The universal approximation theorem may be viewed as a natural extension of the *Weierstrass theorem* (Weierstrass, 1885; Kline, 1972). This theorem states

   Any continuous function over a closed interval on the real axis can be expressed in that interval as an absolutely and uniformly convergent series of polynomials.

   Research interest in the virtues of multilayer perceptrons as devices for the representation of arbitrary continuous functions was perhaps first put into focus by Hecht-Nielsen (1987), who invoked an improved version of Kolomogorov's superposition theorem due to Sprecher (1965). Then, Gallant and White (1988) showed that a single-hidden-layer multilayer perceptron with monotone "cosine" squashing at the hidden layer and no squashing at the output behaves like as a special case of a "Fourier network"

that yields a Fourier series approximation to a given function as its output. However, in the context of traditional multilayer perceptrons, it was Cybenko who demonstrated rigorously for the first time that a single hidden layer is sufficient to uniformly approximate any continuous function with support in a unit hypercube; this work was published as a University of Illinois Technical Report in 1988 and republished as a paper one year later (Cybenko, 1988, 1989). In 1989, two other papers were published independently on multilayer perceptrons as universal approximators, one by Funahashi (1989) and the other by Hornik et al. (1990). For subsequent contributions to the approximation problem, see Light (1992b).

9. The history of the development of cross-validation is documented in Stone (1974). The idea of cross-validation had been around at least since the 1930s, but refinement of the technique was accomplished in the 1960s and 1970s. Two important papers from that era are by Stone (1974) and Geisser (1975), who independently and almost simultaneously propounded the idea of cross-validation. The technique was termed the "cross-validating method" by Stone and the "predictive sample reuse method" by Geisser.

10. Hecht-Nielsen (1995) decribes a replicator neural network in the form of a multilayer perceptron with an input layer of source nodes, three hidden layers and an output layer:

- The activation functions of neurons in the first and third hidden layers are defined by the hyperbolic tangent function

$$\varphi^{(1)}(v) = \varphi^{(3)}(v) = \tanh(v)$$

  where v is the induced local field of a neuron in those layers.
- The activation function for each neuron in the second hidden layer is given by

$$\varphi^{(2)}(v) = \frac{1}{2} + \frac{2}{2(N-1)} \sum_{j=1}^{N-1} \tanh\left(a\left(v - \frac{j}{N} \right) \right)$$

  where a is a gain parameter and v is the induced local filed of a neuron in that layer. The function $\varphi^{(2)}(v)$ describes a smooth staircase activation function with N treadles, thereby essentially quantizing the vector of the respective neural outputs into $K = N^n$, where n is the number of neurons in the middle hidden layer.
- The neurons in the output layer are linear, with their activation functions defined by

$$\varphi^{(4)}(v) = v$$

- Based on this neural network structure, Hecht-Nielsen describes a theorem showing that optimal data compression for arbitrary input data vector can be carried out.

11. The classic reference for the conjugate-gradient method is Hestenes and Stiefel (1952). For a discussion of the convergence behavior of the conjugate-gradient algorithm, see Luenberger (1984) and Bertsekas (1995). For a tutorial treatment of the many facets of the conjugate-gradient algorithm, see Shewchuk (1994). For a readable account of the algorithm in the context of neural networks, see Johansson et al. (1990).

12. The conventional form of the conjugate-gradient algorithm requires the use of a line search, which can be time consuming because of its trial-and-error nature. Møller (1993) describes a modified version of the conjugate-gradient algorithm called the scaled conjugate-gradient algorithm, which avoids the use of a line search. Essentially, the line search is replaced by a one-dimensional Levenberg–Marquardt form of algorithm. The motivation for using such methods is to circumvent the difficulty caused by nonpositive-definite Hessian matrices (Fletcher, 1987).

13. The so-called *$\mathcal{R}$-technique*, due to Pearlmutter (1994), provides an efficient procedure for computing a matrix-vector product; as such, this technique can be of practical use in computing the inverse Hessian $\mathbf{H}^{-1}$ in Eq. (4.138). The $\mathcal{R}$-technique is addressed in Problem 4.6.

14. Hubel and Wiesel's notion of "simple" and "complex" cells was first exploited in the neural network literature by Fukushima (1980, 1995) in the design of a learning machine called the *neocognitron*. This learning machine, however, operates in a self-organized manner, whereas the convolutional network described in Fig. 4.23 operates in a supervised manner using labeled examples.

15. For the origins of the universal myopic mapping theorem, see Sandberg (1991).

16. For a detailed account of the VC-dimension and the related bound on empirical risk, see the classic book on statistical learning theory by Vapnik (1998). The VC-dimension is also discussed in the books by Schölkopf and Smola (2002) and Herbrich (2002). A noteworthy comment is in order: the VC-dimension is related to Cover's separating capacity, which will be discussed in the next chapter, Chapter 5.

PROBLEMS

Back-Propagation Learning

4.1 Figure P4.1 shows a neural network involving a single hidden neuron for solving the XOR problem; this network may be viewed as an alternative to that considered in Section 4.5. Show that the network of Fig. P4.1 solves the XOR problem by constucting (a) decision regions, and (b) a truth table for the network.

4.2 Use the back-propagation algorithm for computing a set of synaptic weights and bias levels for a neural network structured as in Fig. 4.8 to solve the XOR problem. Assume the use of a logistic function for the nonlinearity.

4.3 The momentum constant α is normally assigned a positive value in the range $0 < \alpha \leq 1$. Investigate the difference that would be made in the behavior of Eq. (4.43) with respect to time t if α were assigned a negative value in the range $-1 \leq \alpha < 0$.

4.4 Consider the simple example of a network involving a single weight, for which the cost function is

$$\mathcal{E}(w) = k_1(w - w_0)^2 + k_2$$

where w_0, k_1, and k_2 are constants. A back-propagation algorithm with momentum α is used to minimize $\mathcal{E}(w)$.

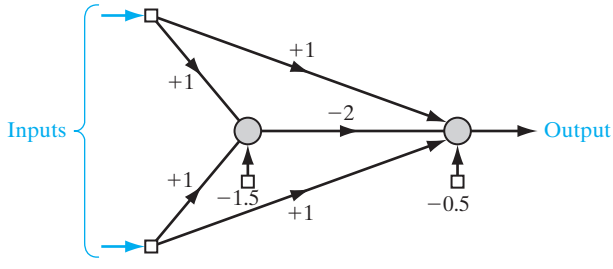


FIGURE P4.1

Explore the way in which the inclusion of the momentum constant α influences the learning process, with particular reference to the number of steps required for convergence versus α.

4.5 Equations (4.51) through (4.53) define the partial derivatives of the approximating function $F(\mathbf{w}, \mathbf{x})$ realized by the multilayer perceptron in Fig. 4.14. Derive these equations from the scenario described by the following conditions:

(a) *Cost function:*

$$\mathscr{E}(n) = \frac{1}{2}[d - F(\mathbf{w}, \mathbf{x})]^2$$

(b) *Output of neuron j:*

$$y_j = \varphi\left(\sum_i w_{ji}\, y_i\right)$$

where w_{ji} is the synaptic weight from neuron i to neuron j, and y_i is the output of neuron i;

(c) *Nonlinearity:*

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

4.6 The $\mathscr{R}$ *technique*, developed by Pearlmutter (1994), provides a computationally fast procedure for evaluating a matrix–vector product. To illustrate this procedure, consider a multilayer perceptron with a single hidden layer; the forward-propagation equations of the network are defined by

$$\begin{aligned} v_j &= \sum_i w_{ji} x_i \\ z_j &= \varphi(v_j) \\ y_k &= \sum_j w_{kj} z_j \end{aligned}$$

$\mathscr{R}[\cdot]$ denotes an *operator* that acts on the quantity enclosed inside the brackets to produce the following results for the example network at hand:

$$\mathscr{R}[v_j] = \sum_i a_{ji} x_i, \qquad \mathscr{R}[w_{ji}] = a_{ji}$$

$$\mathscr{R}[v_j] = \varphi'(v_j)\mathscr{R}[v_j], \qquad \varphi'(v_j) = \frac{\partial}{\partial v_j}\varphi(v_j)$$

$$\mathscr{R}[y_k] = \sum_j w_{kj}\mathscr{R}[z_j] + \sum_i a_{ji} z_j, \qquad \mathscr{R}[w_{kj}] = a_{kj}$$

The $\mathscr{R}$ results are to be viewed as *new* variables. In effect, the operator $\mathscr{R}[\cdot]$ follows the ordinary rules of calculus in addition to the condition

$$\mathscr{R}[\mathbf{w}_j] = \mathbf{a}_j$$

where $\mathbf{w}_j$ is the vector of weights connected to node j and $\mathbf{a}_j$ is the associated vector resulting from application of the $\mathscr{R}$ operator.

(a) Applying the $\mathcal{R}$ technique to the back-propagation algorithm, derive expressions for the elements of the matrix–vector product **Ha**, identifying the new variables for the hidden and output neurons, the matrix **H** is the Hessian. For this application, use the multilayer perceptron described at the beginning of the problem.

(b) Justify the statement that the $\mathcal{R}$ technique is computationally fast.

Supervised Learning Issues

4.7 In this problem, we study the output representation and decision rule performed by a multilayer perceptron. In theory, for an M-class *classification problem* in which the union of the M distinct classes forms the entire input space, we need a total of M outputs to represent all possible classification decisions, as depicted in Fig. P4.7. In this figure, the vector $\mathbf{x}_j$ denotes the jth *prototype* (i.e., unique sample) of an m-dimensional random vector $\mathbf{x}$ to be classified by a multilayer perceptron. The kth of M possible classes to which $\mathbf{x}$ can belong is denoted by $\mathscr{C}_k$. Let y_{kj} be the kth output of the network produced in response to the prototype $\mathbf{x}_j$, as shown by

$$y_{kj} = F_k(\mathbf{x}_j), \qquad k = 1, 2, \ldots, M$$

where the function $F_k(\cdot)$ defines the mapping learned by the network from the input to the k-th output. For convenience of presentation, let

$$\begin{aligned}\mathbf{y}_j &= [y_{1j}, y_{2j}, \ldots, y_{Mj}]^T \\ &= [F_1(\mathbf{x}_j), F_2(\mathbf{x}_j), \ldots, F_M(\mathbf{x}_j)]^T \\ &= \mathbf{F}(\mathbf{x}_j)\end{aligned}$$

where $\mathbf{F}(\cdot)$ is a vector-valued function. The basic question we wish to address in this problem is the following:

> After a multilayer perceptron is trained, what should the optimum decision rule be for classifying the M outputs of the network?

To address this problem, consider the use of a multilayer perceptron embodying a logistic function for its hidden neurons and operating under the following assumptions:

- The size of the training sample is sufficiently large to make a reasonably accurate estimate of the probability of correct classification.
- The back-propagation algorithm used to train the multilayer perceptron does not get stuck in a local minimum.

Specifically, develop mathematical arguments for the property that the M outputs of the multilayer perceptron provide estimates of the a posteriori class probabilities.

4.8 In this problem, we revisit the adaptive control of the learning rate discussed in Section 4.10. The issue of interest is to demonstrate that the asymptotic behavior of the learning-rate parameter $\eta(n)$ in Eq. (4.85) does not converge to zero as the number of iterations increases to infinity.

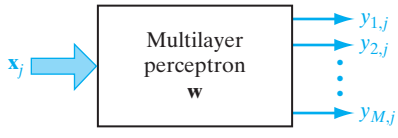


FIGURE P4.7 Block diagram of a pattern classifier for Problem 4.7.

(a) Let $\bar{\mathbf{r}}(n)$ denote the expectation of the auxiliary vector $\mathbf{r}(n)$ with respect to the example $\{\mathbf{x}, \mathbf{d}\}$. Show that if the estimator $\hat{\mathbf{w}}(n)$ is in the close vicinity of the optimal estimator $\mathbf{w}^*$, we may then write

$$\bar{\mathbf{r}}(n+1) \approx (1-\delta)\,\bar{\mathbf{r}}(n) + \delta \mathbf{K}^*(\hat{\mathbf{w}}(n) - \bar{\mathbf{w}}(n))$$

where $\bar{\mathbf{w}}(n)$ is the mean value of the estimator $\hat{\mathbf{w}}(n)$ and δ is a small positive parameter.

(b) In Heskas and Kappen (1991), it is shown that the estimator $\hat{\mathbf{w}}(n)$ is closely approximated by a Gaussian-distributed random vector. Hence, justify the following asymptotic behavior:

$$\lim_{n\to\infty} \hat{\mathbf{w}}(n) \neq \tilde{\mathbf{w}}(n)$$

What does this condition teach us about the asymptotic behavior of the learning-rate parameter $\eta(n)$?

4.9 The composition of the *minimum-description-length (MDL) criterion* is described as follows (see Eq. (2.37)):

$$\text{MDL} = (\text{Error term}) + (\text{Complexity term})$$

Discuss how the weight-decay method used for network pruning fits into the MDL formalism.

4.10 In the *optimal-brain-damage (OBD)* algorithm for network pruning, due to LeCun et al. (1990b), the Hessian $\mathbf{H}$ is approximated by its diagonal version. Using this approximation, derive the OBD procedure as a special case of the optimal-brain-surgeon (OBS) algorithm, studied in Section 4.14.

4.11 In Jacobs (1988), the following heuristics are proposed to accelerate the convergence of on-line back-propagation learning:

(i) Every adjustable network parameter of the cost function should have its own learning-rate parameter.

(ii) Every learning-rate parameter should be allowed to vary from one iteration to the next.

(iii) When the derivative of the cost function with respect to a synaptic weight has the same algebraic sign for several consecutive iterations of the algorithm, the learning-rate parameter for that particular weight should be increased.

(iv) When the algebraic sign of the cost function with respect to a particular synaptic weight alternates for several consecutive iterations of the algorithm, the learning-rate parameter for that weight should be decreased.

These four heuristics satisfy the locality constraint of the back-propagation algorithm.

(a) Use intuitive arguments to justify these four heuristics.

(b) The inclusion of a momentum in the weight update of the back-propagation algorithm may be viewed as a mechanism for satisfying heuristics (iii) and (iv). Demonstrate the validity of this statement.

Second-Order Optimization Methods

4.12 The use of a momentum term in the weight update described in Eq. (4.41) may be considered as an approximation to the conjugate-gradient method (Battiti, 1992). Discuss the validity of this statement.

4.13 Starting with the formula for $\beta(n)$ in Eq. (4.127), derive the *Hesteness-Stiefel formula*,

$$\beta(n) = \frac{\mathbf{r}^T(n)(\mathbf{r}(n) - \mathbf{r}(n-1))}{\mathbf{s}^T(n-1)\mathbf{r}(n-1)}$$

where $\mathbf{s}(n)$ is the direction vector and $\mathbf{r}(n)$ is the residual in the conjugate-gradient method. Use this result to derive the Polak–Ribière formula of Eq. (4.128) and the Fletcher–Reeves formula of Eq. (4.129).

Temporal Processing

4.14 Figure P4.14 illustrates the use of a *Gaussian-shaped time window* as a method for temporal processing, which is motivated by neurobiological considerations (Bodenhausen and Waibel, 1991). The time window associated with synapse i of neuron j is denoted by $\theta(n, \tau_{ji}, \sigma_{ji})$, where τ_{ji} and σ_{ji} are measures of *time delay* and *width* of the windows, respectively, as shown by

$$\theta(n, \tau_{ji}, \sigma_{ji}) = \frac{1}{\sqrt{2\pi}\sigma_{ji}} \exp\left(-\frac{1}{2\sigma_{ji}^2}(n - \tau_{ji})^2\right), \quad i = 1, 2, ..., m_0$$

The output of neuron j is thus defined as

$$y_j(n) = \varphi\left(\sum_{i=1}^{m_0} w_{ji} u_i(n)\right)$$

where $u_i(n)$ is the convolution of the input $x_i(n)$ and the time window $\theta(n, \tau_{ji}, \sigma_{ji})$. The requirement is for the weight w_{ji}, and time delay τ_{ji} of synapse i belonging to neuron j are all to be *learned* in a supervised manner.

This process of learning may be accomplished by using the standard back-propagation algorithm. Demonstrate this learning process by deriving update equations for w_{ji}, τ_{ji}, and σ_{ji}.

Computer Experiments

4.15 Investigate the use of back-propagation learning employing a sigmoidal nonlinearity to achieve one-to-one mappings, as described here:

1. $f(x) = \dfrac{1}{x}, \qquad 1 \le x \le 100$
2. $f(x) = \log_{10} x, \qquad 1 \le x \le 10$
3. $f(x) = \exp(-x), \qquad 1 \le x \le 10$
4. $f(x) = \sin x, \qquad 0 \le x \le \dfrac{\pi}{2}$

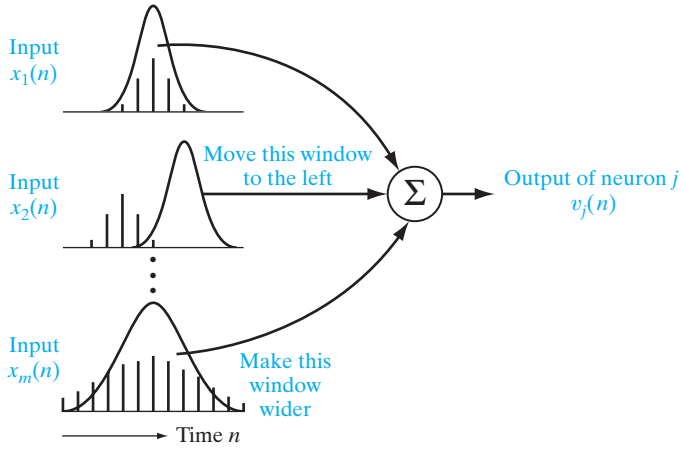


FIGURE P4.14 The figure for Problem 4.14; the instructions appended to the Gaussian windows are aimed at the learning algorithm.

For each mapping, do the following:

(a) Set up two sets of data, one for network training, and the other for testing.
(b) Use the training data set to compute the synaptic weights of the network, assumed to have a single hidden layer.
(c) Evaluate the computation accuracy of the network by using the test data.

Use a single hidden layer, but with a variable number of hidden neurons. Investigate how the network performance is affected by varying the size of the hidden layer.

4.16 Repeat the computer experiment of Section 4.7 for the MLP classifier, where the distance between the two moons is set at $d = 0$. Comment on the findings of your experiment in light of the corresponding experiment performed on the perceptron in Problem 1.6 for the same setting.

4.17 In this computer experiment, we consider a pattern-classification experiment for which the decision boundary is theoretically known. The primary objective of the experiment is to see how the design of the multilayer perceptron could be optimized experimentally in relation to the optimum decision boundary.

Specifically, the requirement is to distinguish between two *equiprobable* classes of "overlapping" two-dimensional Gaussian-distributed patterns, labeled $\mathscr{C}_1$ and $\mathscr{C}_2$. The conditional probability density functions of the two classes are

$$\text{Class } \mathscr{C}_1: \quad p_{\mathbf{X}|\mathscr{C}_1}(\mathbf{x}|\mathscr{C}_1) = \frac{1}{2\pi\sigma_1^2}\exp\left(-\frac{1}{2\sigma_1^2}\|\mathbf{x}-\boldsymbol{\mu}_1\|^2\right)$$

where

$$\boldsymbol{\mu}_1 = \text{mean vector} = [0, 0]^T$$
$$\sigma_1^2 = \text{variance} = 1$$

$$\text{Class } \mathscr{C}_2: \quad p_{\mathbf{X}|\mathscr{C}_2}(\mathbf{x}|\mathscr{C}_2) = \frac{1}{2\pi\sigma_2^2}\exp\left(-\frac{1}{2\sigma_2^2}\|\mathbf{x}-\boldsymbol{\mu}_2\|^2\right)$$

where

$$\boldsymbol{\mu}_2 = [2, 0]^T$$
$$\sigma_2^2 = 4$$

(a) The optimum Bayesian decision boundary is defined by the likelihood ratio test

$$\Lambda(\mathbf{x}) \underset{\mathscr{C}_2}{\overset{\mathscr{C}_1}{\gtrless}} \lambda$$

where

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

and λ is the *threshold* determined by the prior probabilities of the two classes. Show that the optimum decision boundary is a *circle* whose center is located at

$$\mathbf{x}_{\mathscr{C}} = \begin{bmatrix} -2/3 \\ 0 \end{bmatrix}$$

and radius $r = 2.34$.

(b) Assume the use of a single hidden layer. The requirement is to experimentally determine the optimal number of hidden neurons.

- Starting with a multilayer perceptron with two hidden neurons, and using the back-propagation algorithm with learning-rate parameter $\eta = 0.1$ and momentum constant $\alpha = 0$ to train the network, calculate the probability of correct classification for the following scenarios:

Training-sample size	Number of epochs
500	320
2,000	80
8,000	20

- Repeat the experiment, this time using four hidden neurons, with everything else remaining the same as before. Compare the results of this second experiment with those of the previous one and thereby select the network configuration, with two *or* four hidden neurons, that you consider to be the optimal choice.

(c) For the "optimal" network selection made in part (b), we now turn to experimentally find the optimal values of the learning-rate parameter η and momentum constant α. To do this, perform experiments using the following combination of parameters:

$$\eta \in [0.01, 0.1, 0.5]$$
$$\alpha \in [0.0, 0.1, 0.5]$$

Hence, determine the values of η and α that yield the best probability of correct classification.

(d) Having identified the optimum size of hidden layer and the optimum set of η and α, perform one last experiment to find the optimum decision boundary and the corresponding probability of correct classification. Compare the optimum performance so obtained experimentally against the theoretical optimum, and comment on your results.

4.18 In this problem, we use the standard back-propagation algorithm to solve a difficult nonlinear prediction problem and compare its performance with that of the LMS algorithm. The time series to be considered is created using a discrete *Volterra model* that has the form

$$x(n) = \sum_i g_i v(n-i) + \sum_i \sum_j g_{ij} v(n-i)v(n-j) + \cdots$$

where $g_i, g_{ij}, \ldots$, are the Volterra coefficients; the $v(n)$ are samples of a white, independently distributed Gaussian noise sequence; and $x(n)$ is the resulting output of the Volterra model. The first summation term is the familiar moving-average (MA) time-series model, and the remaining summation terms are nonlinear components of ever increasing order. In general, the estimation of the Volterra coefficients is considered to be difficult, primarily because of their nonlinear relationship to the data.

We consider the simple example

$$x(n) = v(n) + \beta v(n-1)v(n-2)$$

The time series has zero mean, is uncorrelated, and therefore has a white spectrum. However, the time-series samples are not independent of each other, and therefore a higher-order predictor can be constructed. The variance of the model output is given by

$$\sigma_x^2 = \sigma_\nu^2 + \beta^2\sigma_\nu^4$$

where σ_ν^2 is the white-noise variance.

(a) Construct a multilayer perceptron with an input layer of six nodes, a hidden layer of 16 neurons, and a single output neuron. A tapped-delay-line memory is used to feed the input layer of the network. The hidden neurons use sigmoid activation functions limited to the interval [0, 1], whereas the output neuron operates as a linear combiner. The network is trained with the standard back-propagation algorithm having the following description:

Learning-rate parameter	$\eta = 0.001$
Momentum constant	$\alpha = 0.6$
Total number of samples processed	100,000
Number of samples per epoch	1,000
Total number of epochs	2,500

The white-noise variance σ_ν^2 is set equal to unity. Hence, with $\beta = 0.5$, we find that the output variance of the predictor is $\sigma_x^2 = 1.25$.

Compute the learning curve of the nonlinear predictor, with the variance of the predictor output $x(n)$ plotted as a function of the number of epochs of training samples up to 2,500 epochs. For the preparation of each epoch used to perform the training, explore the following two modes:

(i) The time ordering of the training sample is maintained from one epoch to the next in exactly the same form as it is generated.

(ii) The ordering of the training sample is randomized from one pattern (state) to another.

Also, use cross-validation (described in Section 4.13) with a validation set of 1,000 samples to monitor the learning behavior of the predictor.

(b) Repeat the experiment, using the LMS algorithm designed to perform a linear prediction on an input of six samples. The learning-rate parameter of the algorithm is set equal to $\eta = 10^{-5}$.

(c) Repeat the entire experiment for $\beta = 1, \sigma_\nu^2 = 2$, and then for $\beta = 2, \sigma_\nu^2 = 5$.

The results of each experiment should reveal that initially the back-propagation algorithm and the LMS algorithm follow essentially a similar path, and then the back-propagation algorithm continues to improve, finally producing a prediction variance approaching the prescribed value of σ_x^2.

4.19 In this experiment, we use a multilayer perceptron trained with the back-propagation algorithm to perform one-step prediction on the *Lorenz attractor*. The dynamics of this attractor are defined by three equations:

$$\frac{dx(t)}{dt} = -\sigma x(t) + \sigma y(t)$$

$$\frac{dy(t)}{dt} = -x(t)z(t) + rx(t) - y(t)$$

$$\frac{dz(t)}{dt} = x(t)y(t) - bz(t)$$

where σ, r, and b are dimensionless parameters. Typical values for these parameters are $\sigma = 10, b = \frac{8}{3}$, and $r = 28$.

The specifications of the multilayer perceptron are as follows:

Number of source nodes: 20
Number of hidden neurons: 200
Number of output neurons: 1

The particulars of the data sets are as follows:

Training sample: 700 data points
Testing sample: 800 data points
Number of epochs used for training: 50

The parameters of the back-propagation algorithm are as follows:

The learning-rate parameter η is annealed linearly from 10^{-1} down to 10^{-5}.
Momentum: $\alpha = 0$

(a) Compute the learning curve of the MLP, plotting the mean-square error versus the number of epochs used to do the training.

(b) Compute the one-step prediction to the Lorenz attractor; specifically, plot the results obtained as a function of time, and compare the prediction against the evolution of the Lorenz attractor.