Artificial Neural Networks

An Introduction

Tutorial Texts Series

- *Artificial Neural Networks An Introduction*, Kevin L. Priddy and Paul E. Keller, Vol. TT68
- *Basics of Code Division Multiple Access (CDMA)*, Raghuveer Rao and Sohail Dianat, Vol. TT67
- *Optical Imaging in Projection Microlithography*, Alfred Kwok-Kit Wong, Vol. TT66
- *Metrics for High-Quality Specular Surfaces*, Lionel R. Baker, Vol. TT65
- *Field Mathematics for Electromagnetics, Photonics, and Materials Science*, Bernard Maxum, Vol. TT64
- *High-Fidelity Medical Imaging Displays*, Aldo Badano, Michael J. Flynn, and Jerzy Kanicki, Vol. TT63
- *Diffractive Optics–Design, Fabrication, and Test*, Donald C. O'Shea, Thomas J. Suleski, Alan D. Kathman, and Dennis W. Prather, Vol. TT62
- *Fourier-Transform Spectroscopy Instrumentation Engineering*, Vidi Saptari, Vol. TT61
- *The Power- and Energy-Handling Capability of Optical Materials, Components, and Systems*, Roger M. Wood, Vol. TT60
- *Hands-on Morphological Image Processing*, Edward R. Dougherty, Roberto A. Lotufo, Vol. TT59
- *Integrated Optomechanical Analysis*, Keith B. Doyle, Victor L. Genberg, Gregory J. Michels, Vol. TT58
- *Thin-Film Design Modulated Thickness and Other Stopband Design Methods*, Bruce Perilloux, Vol. TT57
- *Optische Grundlagen für Infrarotsysteme*, Max J. Riedl, Vol. TT56
- *An Engineering Introduction to Biotechnology*, J. Patrick Fitch, Vol. TT55
- *Image Performance in CRT Displays*, Kenneth Compton, Vol. TT54
- *Introduction to Laser Diode-Pumped Solid State Lasers*, Richard Scheps, Vol. TT53
- *Modulation Transfer Function in Optical and Electro-Optical Systems*, Glenn D. Boreman, Vol. TT52
- *Uncooled Thermal Imaging Arrays, Systems, and Applications*, Paul W. Kruse, Vol. TT51
- *Fundamentals of Antennas*, Christos G. Christodoulou and Parveen Wahid, Vol. TT50
- *Basics of Spectroscopy*, David W. Ball, Vol. TT49
- *Optical Design Fundamentals for Infrared Systems, Second Edition*, Max J. Riedl, Vol. TT48
- *Resolution Enhancement Techniques in Optical Lithography*, Alfred Kwok-Kit Wong, Vol. TT47
- *Copper Interconnect Technology*, Christoph Steinbrüchel and Barry L. Chin, Vol. TT46
- *Optical Design for Visual Systems*, Bruce H. Walker, Vol. TT45
- *Fundamentals of Contamination Control*, Alan C. Tribble, Vol. TT44
- *Evolutionary Computation Principles and Practice for Signal Processing*, David Fogel, Vol. TT43
- *Infrared Optics and Zoom Lenses*, Allen Mann, Vol. TT42
- *Introduction to Adaptive Optics*, Robert K. Tyson, Vol. TT41
- *Fractal and Wavelet Image Compression Techniques*, Stephen Welstead, Vol. TT40
- *Analysis of Sampled Imaging Systems*, R. H. Vollmerhausen and R. G. Driggers, Vol. TT39
- *Tissue Optics Light Scattering Methods and Instruments for Medical Diagnosis*, Valery Tuchin, Vol. TT38
- *Fundamentos de Electro-Óptica para Ingenieros*, Glenn D. Boreman, translated by Javier Alda, Vol. TT37
- *Infrared Design Examples*, William L. Wolfe, Vol. TT36
- *Sensor and Data Fusion Concepts and Applications, Second Edition*, L. A. Klein, Vol. TT35
- *Practical Applications of Infrared Thermal Sensing and Imaging Equipment, Second Edition*, Herbert Kaplan, Vol. TT34
- *Fundamentals of Machine Vision*, Harley R. Myler, Vol. TT33
- *Design and Mounting of Prisms and Small Mirrors in Optical Instruments*, Paul R. Yoder, Jr., Vol. TT32
- *Basic Electro-Optics for Electrical Engineers*, Glenn D. Boreman, Vol. TT31
- *Optical Engineering Fundamentals*, Bruce H. Walker, Vol. TT30
- *Introduction to Radiometry*, William L. Wolfe, Vol. TT29
- *Lithography Process Control*, Harry J. Levinson, Vol. TT28
- *An Introduction to Interpretation of Graphic Images*, Sergey Ablameyko, Vol. TT27
- *Thermal Infrared Characterization of Ground Targets and Backgrounds*, P. Jacobs, Vol. TT26
- *Introduction to Imaging Spectrometers*, William L. Wolfe, Vol. TT25
- *Introduction to Infrared System Design*, William L. Wolfe, Vol. TT24

Artificial Neural Networks

An Introduction

Kevin L. Priddy and Paul E. Keller

Tutorial Texts in Optical Engineering
Volume TT68

SPIE
PRESS

Bellingham, Washington USA

Library of Congress Cataloging-in-Publication Data

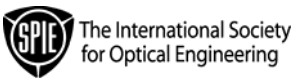
Priddy, Kevin L.
Artificial neural networks : an introduction / Kevin L. Priddy and Paul E. Keller.
p. cm.
Includes bibliographical references and index.
ISBN 0-8194-5987-9
1. Neural networks (Computer science) I. Keller, Paul E. II. Title.

QA76.87.P736 2005
006.3'2--dc22 2005021833

Published by

SPIE—The International Society for Optical Engineering
P.O. Box 10
Bellingham, Washington 98227-0010 USA
Phone: +1 360 676 3290
Fax: +1 360 647 1445
Email: spie@spie.org
Web: http://spie.org

Printed in the United States of America.



Introduction to the Series

Since its conception in 1989, the Tutorial Texts series has grown to more than 60 titles covering many diverse fields of science and engineering. When the series was started, the goal of the series was to provide a way to make the material presented in SPIE short courses available to those who could not attend, and to provide a reference text for those who could. Many of the texts in this series are generated from notes that were presented during these short courses. But as stand-alone documents, short course notes do not generally serve the student or reader well. Short course notes typically are developed on the assumption that supporting material will be presented verbally to complement the notes, which are generally written in summary form to highlight key technical topics and therefore are not intended as stand-alone documents. Additionally, the figures, tables, and other graphically formatted information accompanying the notes require the further explanation given during the instructor's lecture. Thus, by adding the appropriate detail presented during the lecture, the course material can be read and used independently in a tutorial fashion.

What separates the books in this series from other technical monographs and textbooks is the way in which the material is presented. To keep in line with the tutorial nature of the series, many of the topics presented in these texts are followed by detailed examples that further explain the concepts presented. Many pictures and illustrations are included with each text and, where appropriate, tabular reference data are also included.

The topics within the series have grown from the initial areas of geometrical optics, optical detectors, and image processing to include the emerging fields of nanotechnology, biomedical optics, and micromachining. When a proposal for a text is received, each proposal is evaluated to determine the relevance of the proposed topic. This initial reviewing process has been very helpful to authors in identifying, early in the writing process, the need for additional material or other changes in approach that would serve to strengthen the text. Once a manuscript is completed, it is peer reviewed to ensure that chapters communicate accurately the essential ingredients of the processes and technologies under discussion.

It is my goal to maintain the style and quality of books in the series, and to further expand the topic areas to include new emerging fields as they become of interest to our reading audience.

Arthur R. Weeks, Jr.
University of Central Florida

Contents

Preface

This text introduces the reader to the fascinating world of artificial neural networks, a journey that the authors are here to help you with. The authors have written this book for the reader who wants to understand artificial neural networks without necessarily being bogged down in the mathematics. A glossary is included to assist the reader in understanding any unfamiliar terms. For those who desire the math, sufficient detail for most of the common neural network algorithms is included in the appendixes.

The concept of data-driven computing is the overriding principle upon which neural networks have been built. Many problems exist for which data are plentiful, but there is no underlying knowledge of the process that converts the measured inputs into the observed outputs. Artificial neural networks are well suited to this class of problem because they are excellent data mappers in that they map inputs to outputs. This text illustrates how this is done with examples and relevant snippets of theory.

The authors have enjoyed writing the text and welcome readers to dig further and learn how artificial neural networks are changing the world around them.

Acknowledgements

We wish to acknowledge our mentor and friend, Dr. Steven K. Rogers, aka Captain Amerika, for his enthusiasm and encouragement throughout our careers in utilizing artificial neural networks. Many of the concepts and ideas were borrowed from discussions and presentations given by Dr. Rogers.

We could not have done this work without the support of our families and especially our wives, Wendy Priddy and Torie Keller. We wish to acknowledge the assistance of Samuel Priddy in preparing the manuscript. In addition, The International Society for Optical Engineering (SPIE) has been very helpful in encouraging us to write this text and in its final preparation for publication.

Artificial Neural Networks

An Introduction

Chapter 1

Introduction

Artificial neural networks are mathematical inventions inspired by observations made in the study of biological systems, though loosely based on the actual biology. An artificial neural network can be described as mapping an input space to an output space. This concept is analogous to that of a mathematical function. The purpose of a neural network is to map an input into a desired output. While patterned after the interconnections between neurons found in biological systems, artificial neural networks are no more related to real neurons than feathers are related to modern airplanes. Both biological systems, neurons and feathers, serve a useful purpose, but the implementation of the principles involved has resulted in man-made inventions that bear little resemblance to the biological systems that spawned the creative process.

This text starts with the aim of introducing the reader to many of the most popular artificial neural networks while keeping the mathematical gymnastics to a minimum. Many excellent texts, which have detailed mathematical descriptions for many of the artificial neural networks presented in this book, are cited in the bibliography. Additional mathematical background for the neural-network algorithms is provided in the appendixes as well. Artificial neural networks are modeled after early observations in biological systems: myriads of neurons, all connected in a manner that somehow distributes the necessary signals to various parts of the body to allow the biological system to function and survive. No one knows exactly how the brain works or what is happening in the nervous system all of the time, but scientists and medical doctors have been able to uncover the inner workings of the mind and the nervous system to a degree that allows them to completely describe the operation of the basic computational building block of the nervous system and brain: the neuron.

1.1 The Neuron

The human body is made up of a vast array of living cells. Certain cells are interconnected in a way that allows them to communicate pain, or to actuate fibers or tissues. Some cells control the opening and shutting of minuscule valves in the

veins and arteries. Others tell the brain that they are experiencing cold, heat, or any number of sensations. These specialized communication cells are called neurons. Neurons are equipped with long tentacle-like structures that stretch out from the cell body, permitting them to communicate with other neurons. The tentacles that take in signals from other cells and the environment itself are called dendrites, while the tentacles that carry signals from the neuron to other cells are called axons. The interaction of the cell body itself with the outside environment through its dendritic connections and the local conditions in the neuron itself cause the neuron to pump either sodium or potassium in and out, raising and lowering the neuron's electrical potential. When the neuron's electrical potential exceeds a threshold, the neuron fires, creating an action potential that flows down the axons to the synapses and other neurons. The action potential is created when the voltage across the cell membrane of the neuron becomes too large and the cell "fires," creating a spike that travels down the axon to other neurons and cells. If the stimulus causing the buildup in voltage is low, then it takes a long time to cause the neuron to fire. If it is high, the neuron fires much faster.

The firing rate of the neuron can thus be close to zero, as in a case involving no stimulus, to a maximum of approximately 300 pulses per second, as in the case of a stimulus that causes the neuron to fire as fast as possible. Because a neuron is a physical system, it takes time to build up enough charge to cause it to fire. This is where adrenaline comes into play. Adrenaline acts as a bias for the neuron, making it much more likely to fire in the presence of a stimulus. In this book the reader will see how man-made neural networks mimic this potential through weighted interconnections and thresholding terms that allow the artificial neurons to fire more rapidly, just as a biological cell can be biased to fire more rapidly through the introduction of adrenaline. Figure 1.1 is a simplified depiction of a neuron.

The neuron has a central cell body, or soma, with some special attachments, dendrites and axons. The dendrites are special nodes and fibers that receive electrochemical stimulation from other neurons. The axon allows the neuron to communicate with other neighboring neurons. The axon connects to a dendrite fiber through an electrochemical junction known as a synapse. For simplicity, the neuron is depicted with a handful of connections to other cells. In reality, there can be from tens to thousands of interconnections between neurons. The key concept to remember is that neurons receive inputs from other neurons and send outputs to other neurons and cells.

1.2 Modeling Neurons

From the previous section, the reader learned that neurons are connected to other neurons. A simple model of the neuron that shows inputs from other neurons and a corresponding output is depicted in Fig. 1.2. As can be seen in the figure, three neurons feed the single neuron, with one output emanating from the single neuron.

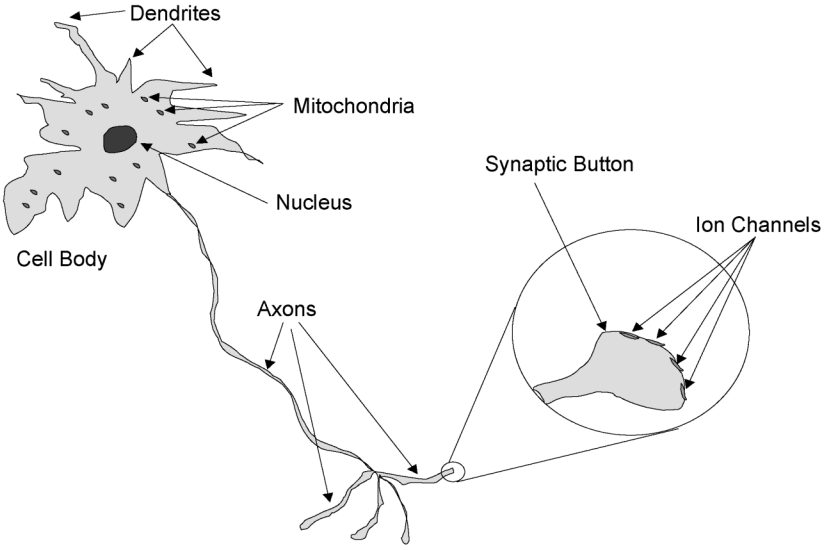


Figure 1.1 Simplified diagram of a neuron.

The reader may recall that neurons send signals to other neurons by sending an action potential down the axon. This is modeled through the use of a transfer function that mimics the firing rate of the neuron action potential, as shown in Fig. 1.3.

Some inputs to the neuron may have more relevance as to whether the neuron should fire, so they should have greater importance. This is modeled by weighting the inputs to the neuron. Thus, the neuron can be thought of as a small computing engine that takes in inputs, processes them, and then transmits an output. The artificial neuron with weighted inputs and a corresponding transfer function is shown in Fig. 1.4. The output for the neuron in Fig. 1.4 is given by

$$z = f\left(\sum_{i=0}^{3} w_i x_i\right). \tag{1.1}$$

The reader has probably already determined that Eq. (1.1) lacks a definition for the transfer function. Many different transfer functions are available to the neural network designer, such as those depicted in Fig. 1.5.

The most commonly used transfer function is the sigmoid or logistic function, because it has nice mathematical properties such as monotonicity, continuity, and differentiability, which are very important when training a neural network with gradient descent. Initially, scientists studied the single neuron with a hard-limiter or step-transfer function. McCulloch and Pitts used an "all or nothing" process to describe neuron activity [McCulloch, 1949]. Rosenblatt [Rosenblatt, 1958] used a hard limiter as the transfer function and termed the hard-limiter neuron a perceptron because it could be taught to solve simple problems. The hard-limiter is an example of a linear equation solver with a simple line forming the decision boundary, as shown in Fig. 1.6.

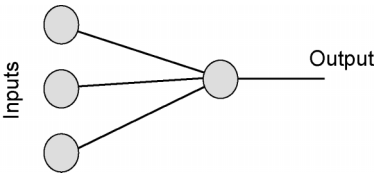


Figure 1.2 Artificial neuron with inputs and a single output.

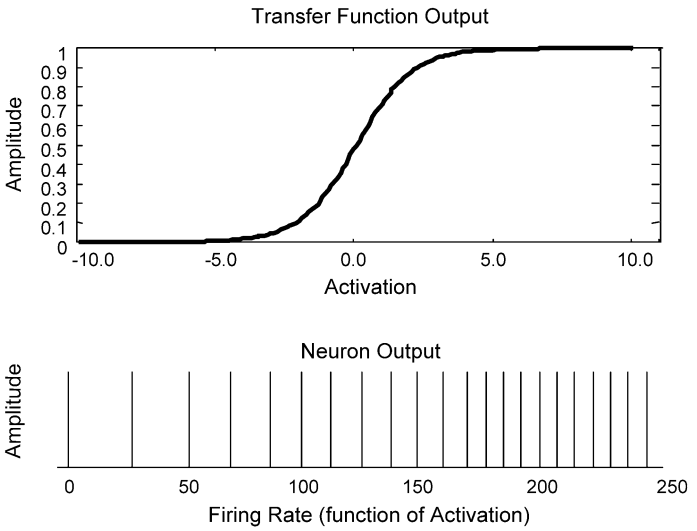


Figure 1.3 Transfer function representation of neuron firing rate.

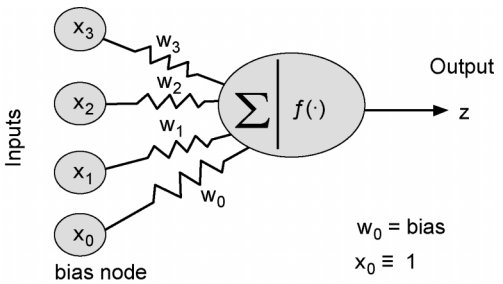


Figure 1.4 Neuron model with weighted inputs and embedded transfer function.

The most common transfer function is the logistic sigmoid function, which is given by the following equation:

$$output = \frac{1}{1 + e^{-(\sum_i w_i x_i + w_0)}}, \tag{1.2}$$

where i is the index on the inputs to the neuron, x_i is the input to the neuron, w_i is the weighting factor attached to that input, and w_0 is the bias to the neuron.

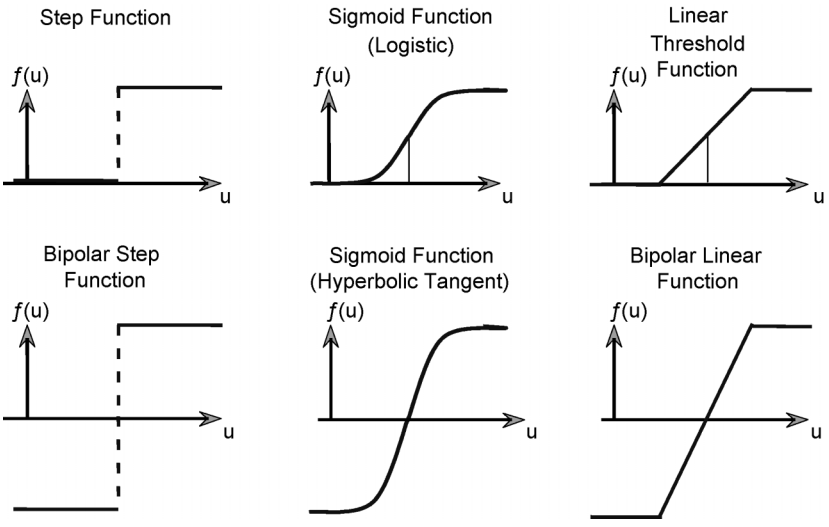


Figure 1.5 Three different types of transfer function—step, sigmoid, and linear in unipolar and bipolar formats.

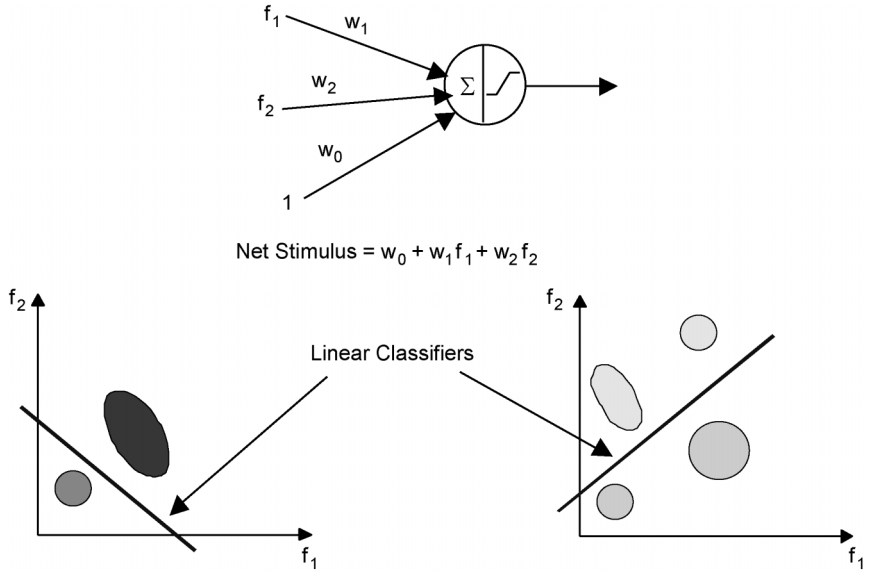


Figure 1.6 Simple neuron model with decision boundaries.

The aggregation and selective use of these decision boundaries is what makes artificial neural networks interesting. Artificial neural networks were created to permit machines to form these decision boundaries with their associated class regions as derived from the data. In fact, a more proper name for artificial neural networks may very well be data-driven computational engines. Neural networks can be used to combine these decision regions together to form higher and higher levels of abstraction, which can result in neural networks with some amazing properties.

To illustrate the how a decision boundary can be formed, consider the neuron in Fig. 1.6 and the associated subfigures. Looking at the subfigure on the lower left, the reader will see a small circle and an ellipse. Each shape represents objects that belong to the same set or class; e.g., the circle could represent all mammals (M) and the ellipse could represent all birds (B). The neuron, combined with a hard-limiter transfer function, can find a line that will separate the two classes {M, B} as depicted by the drawn lines in Fig. 1.6. Each line represents the locus of neuron activation values possible for a given set of weights associated with the neuron. For example, in Fig. 1.6, the reader will see an equation for the neuron net stimulus, or activation, given as

$$\textit{Net Stimulus} = \sum_{i=0}^{2} w_i f_i = w_2 f_2 + w_1 f_1 + w_0, \tag{1.3}$$

which can be rearranged to form

$$f_2 = \frac{w_1 f_1}{w_2} + \frac{(w_0 - \textit{Net Stimulus})}{w_2}, \tag{1.4}$$

which is an equation of a line with a slope of w_1/w_2 and an intercept of $(w_0 - \textit{Net Stimulus})/w_2$ on the f_2 axis. Thus, for a given node with weighted inputs, the activation must lie on the line described by Eq. (1.4).

Decision engines that follow the form of Eq. (1.4) are called linear classifiers. This important observation, that all possible neuron activation values lie on a line, cannot be emphasized enough, because by changing the weights and using a non-linear transfer function a decision engine can be produced that places one set of objects on one side of the line and other objects or classes on the other side of the line. Thus, by changing w_0, w_1, w_2 and the net stimulus value, the position of the line can be moved until one side of the line contains all the examples of a given class and the other side contains all examples of other classes, as shown in Fig. 1.6. When more than two feature dimensions are used, the equation becomes that of a separating hyperplane. In principle, the concept is similar to the line described previously: On one side of the hyperplane is one class, and on the other side, all the other classes. Whenever a decision engine can produce such a line or hyperplane and separate the data, the data is called a linearly separable set.

One of the best-known linear classifiers is the perceptron, first introduced by Rosenblatt [Rosenblatt, 1958]. The perceptron adjusts its weights using the error vector between a data point and the separating decision line. Changes are made to the weights of the network until none of the training-set data samples produces an error. The perceptron algorithm is presented in simplified form in Table 1.1. The reader will find Matlab code in the appendixes that implements the perceptron algorithm.

The perceptron has the ability to form separating lines and hyperplanes to permit linearly separable classification problems to be solved. Many examples of data

exist in the real world in which the classes are not linearly separable, such as that shown in Fig. 1.7. This demonstrates that no single line can separate the green circle from the red kidney-shaped object. It will be shown later that multiple linear classifiers can be combined to produce separable decision regions with a single classifier.

Researchers discovered that combining neurons into layers permits artificial neural networks to solve highly complex classification problems. The next section

Table 1.1 Perceptron Learning Algorithm

1. Assign the desired output of the perceptron $\{-1, 1\}$ to each data point in the training set
2. Augment each set of features for each data point with one to form $\mathbf{x} = \{1, x_1, x_2, \ldots, x_n\}$
3. Choose an update step size (eta $\in (0, 1)$) usually eta $= 0.1$
4. Randomly initialize weights $\mathbf{w} = \{w_0, w_1, w_2, \ldots, w_n\}$
5. Execute perceptron weight adjustment algorithm

```
error_flag = 1
while error_flag = 1
   error = 0
   for ii = 1 to total number of data points in training set
      grab a feature vector (x) and its desired output {1, −1}
      output = signum(wx^T)
      if output not equal to desired
         w = w − eta · x
error = error + output − desired
      end
   end
   if error = 0
      error_flag = 0
   end
end
```

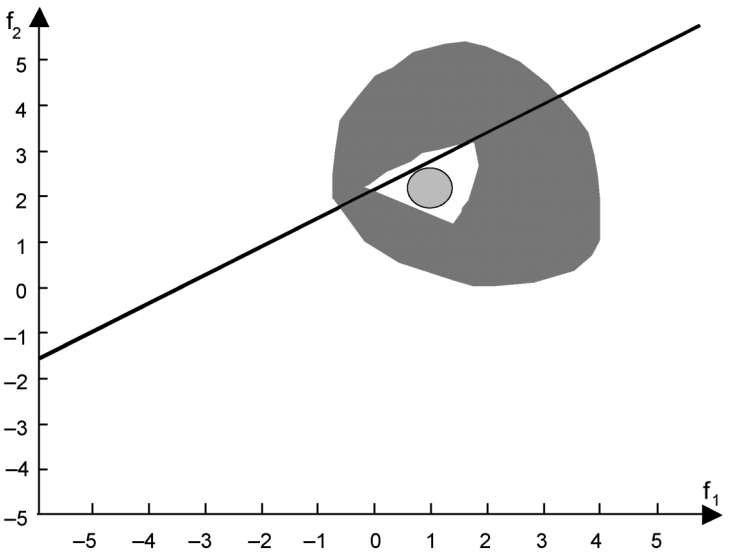


Figure 1.7 Example of a nonlinearly separable classification problem.

introduces the feedforward neural network, which is composed of layers of neurons.

1.3 The Feedforward Neural Network

A depiction of a simple feedforward neural network is given in Fig. 1.8. On the left portion of the figure are inputs to the first layer of neurons, followed by interconnected layers of neurons, and finally with outputs from the final layer of neurons. Note that each layer directly supplies the next layer in the network, feeding the inputs forward through the network, earning this network architecture the feedforward network label. The transfer functions of the neurons do not affect the feedforward behavior of the network. The reader will see many feedforward networks with sigmoid transfer functions, but the neurons in feedforward networks can be any transfer function the designer wishes to use. In addition, the reader should note that a neuron on the first layer could feed a neuron on the third as well as the second layer. Feedforward networks feed outputs from individual neurons forward to one or more neurons or layers in the network. Networks that feed outputs from a neuron back to the inputs of previous layers themselves or other neurons on their own layer are called recurrent networks. Suffice it to say that neural network topologies can vary widely, resulting in differences in architecture, function, and behavior.

Just as the perceptron allowed machines to form decision regions with a single hyperplane, the multilayer feedforward neural network allows machines to form arbitrarily complex decision regions with multiple hyperplanes to solve classification problems as shown in Fig. 1.9.

The process of modifying the weights in a neuron or network to correctly perform a desired input-to-output mapping is termed learning in the neural-network community. As the reader will discover, many methods exist for training neural net-

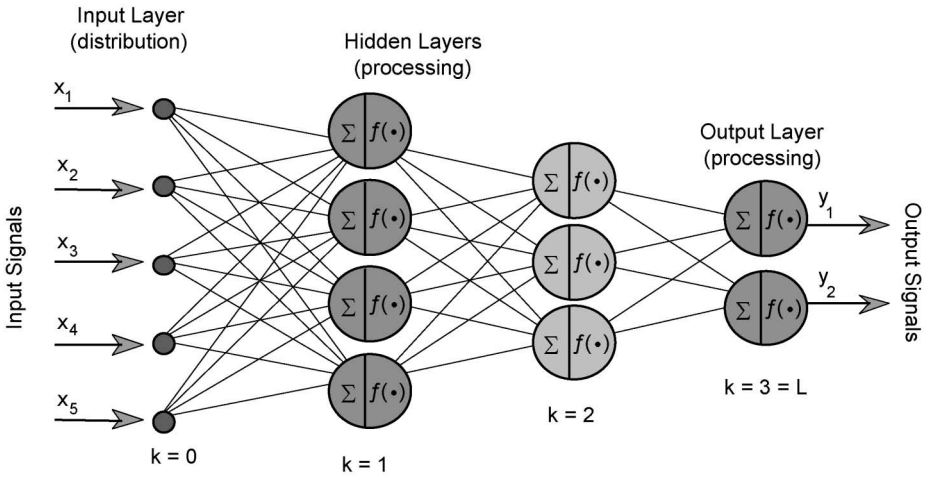
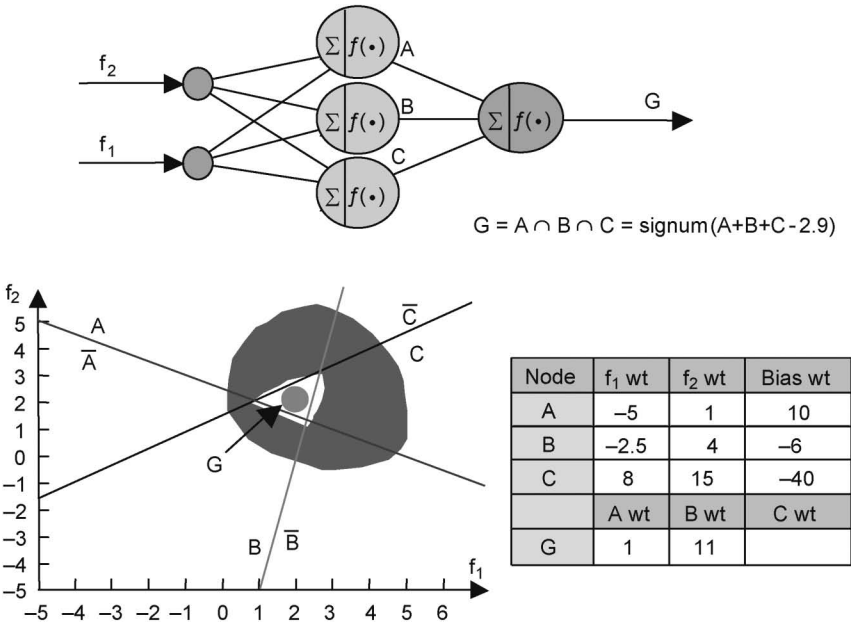


Figure 1.8 Multilayer feedforward neural network.



Node	f_1 wt	f_2 wt	Bias wt	
A	–5	1	10	
B	–2.5	4	–6	
C	8	15	–40	
	A wt	B wt	C wt	Bias wt
G	1	11		–2.9

Figure 1.9 Multilayer feedforward neural network forms complex decision regions to solve nonlinearly separable problem.

works, but all learning is based upon adapting the weights between interconnected neurons. Once a particular network architecture is chosen, the designer must determine which weight to modify to effect the desired network behavior. This is discussed in the next section.

1.3.1 The Credit-Assignment Problem

Once a set of neurons is interconnected, how does the designer train the neurons to do what he/she wants? Which weights should be modified and by how much? These questions are all related to what is often called the credit-assignment problem, which can be defined as the process of determining which weight should be adjusted to effect the change needed to obtain the desired system performance. If things go well, which weight gets the pat on the back with reinforcement of the desired response? Likewise, when things go wrong, which weight gets identified as the guilty party and is penalized? Other questions that need to be answered concern the number of neurons to put in each layer, the number of hidden layers, and the best way to train the network. The answers to all of these questions would fill volumes.

Rosenblatt, with his perceptron, showed some interesting results as long as the classes were linearly separable. Minsky and Papert [Minsky, 1969] showed quite conclusively that there were many classes of problems ill-suited for linear classifiers, so the neural-network community went back to work to find a better solution for the nonlinearly separable class of problems. Werbos showed that by adding

layers of neurons and carefully defining their transfer functions to be sigmoids, the credit-assignment problem could be solved and the issues pointed out by Minsky and Pappert overcome. Thus, by using monotonic continuous-transfer functions, Werbos solved the credit-assignment problem—which weight is the slacker or conversely the worker bee—by taking the derivative of the mean squared error with respect to a given weight (see Appendix A for the derivation). This is often referred to as gradient descent, as depicted in Fig. 1.10. Werbos' result allowed researchers to add any number of hidden layers between the input and output. Researchers discovered that problems occurred with the gradient-descent approach because of the time required to train the network as well as a propensity to find local minima in the overall solution space. Local minima are locations in the solution space that form a minimum, which causes network convergence to a non-optimal solution.

1.3.2 Complexity

Some people argue that a neural network is a black box, because either they cannot understand its inner workings or they believe it is too complex a system for the problem. They would prefer a simpler solution that can be easily dissected. A neural network is a complex structure used to solve complex data-analysis problems. Henry Louis Mencken [Mencken, 1917], an early twentieth-century editor, author, and critic, once wrote, "... there is always an easy solution to every human problem—neat, plausible, and wrong."

While this statement was intended as social commentary, it does indicate mankind's desire for easy solutions. Usually, it is best to find the simplest solution to a problem, but the complexity of the solution must be comparable to the complexity of the problem. This does not mean that a designer should build the most

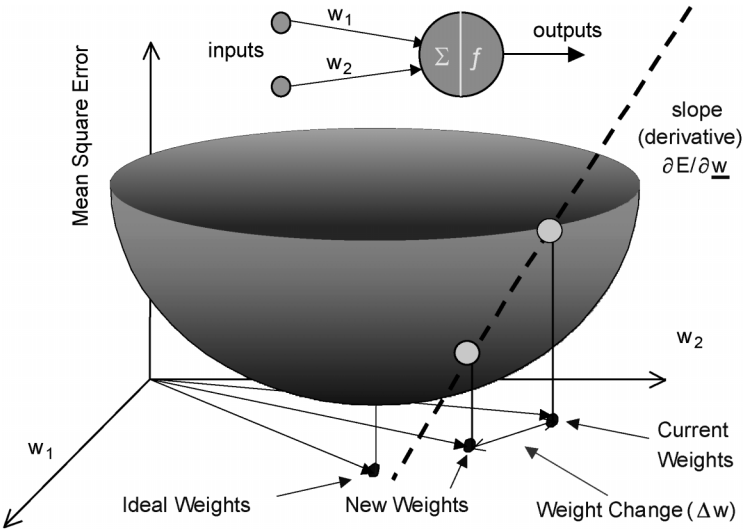


Figure 1.10 Gradient descent for mean-squared-error cost function.

complex system. The designer should develop the simplest solution that solves the problem within the given constraints. Harry Burke [Burke, 1996] presents the idea that a neural network is a powerful tool for capturing complex relationships:

> Artificial neural networks are nonparametric regression models. They can capture any phenomena, to any degree of accuracy (depending on the adequacy of the data and the power of the predictors), without prior knowledge of the phenomena. Further, artificial neural networks can be represented, not only as formulae, but also as graphical models. Graphical models can improve the manipulation, and understanding, of statistical structures. Artificial neural networks are a powerful method for capturing complex phenomena, but their use requires a paradigm shift, from exploratory analysis of the data to exploratory analysis of the model.

1.4 Historical Perspective on Computing with Artificial Neurons

Researchers back to the late nineteenth century, most notably Alexander Bain [Bain, 1873] and William James [James, 1890], realized the potential for man-made systems based on neural models. By the middle of the twentieth century, Warren McCulloch and Walter Pitts [McCulloch, 1943] had shown that groups of neurons were Turing capable, and Donald Hebb [Hebb, 1949] had developed a learning rule that showed how neurons used reinforcement to strengthen connections from important inputs. Inspired by Hebb's work, Belmont Farley and Wesley Clark developed the first digital computer-based artificial neural network in the early 1950s. [Farley, 1954] In their neural network, neurons were randomly connected. This system was followed by a self-assembled neural network developed by Nathaniel Rochester, John Holland, and their colleagues. [Rochester, 1956] Frank Rosenblatt developed the perceptron for pattern classification [Rosenblatt, 1958]. Unfortunately, perceptrons could not perform complex classifications, and the research was abandoned in the late 1960s. Also during this time, the ADALINE (Adaptive Linear Element) was developed by Bernard Widrow and Marcian Hoff [Widrow, 1960]. It implemented an adaptive-learning system that eventually was used for adaptive signal processing to eliminate echoes in telephone systems.

During the 1970s, neural-network research was limited to a few research groups. Based on physiological studies of the nervous system, Stephen Grossberg developed a self-organizing neural-network model known as Adaptive Resonance Theory (ART) [Grossberg, 1976a and c]. Also during this period, Teuvo Kohonen developed both matrix-associative memories [Kohonen, 1972] and a theory in which neurons self-organize into topological and tonotopical mappings of their perceived environment [Kohonen, 1987]. Paul Werbos in 1974 developed a learning rule in which the error in the network's output is propagated backwards through the network, and the network's synaptic weights are adjusted by using a gradient-descent error-minimization approach [Werbos, 1974; Werbos, 1994]. While this work was unknown to most neural network researchers at the time, this technique is the backpropagation of error algorithm. It is often referred to as backpropagation (or backprop for short), and is currently the most widely used artificial neural network model.

John Hopfield, who argued that a neural network's computational abilities were based on the collective action of the network and not the function of the individual neurons, rekindled general interest in neural networks in 1982 [Hopfield, 1982]. He modeled this interaction as an energy-minimization process. In the mid-1980s, David Rumelhart, Geoffrey Hinton, Ronald J. Williams, and others popularized the backpropagation algorithm [Rumelhart, 1986; Parker, 1982; LeCun, 1985].

Clearly, today's work with artificial neural networks is based upon the research of many mathematicians, scientists, and engineers.

Chapter 2

Learning Methods

The past few sections have introduced the reader to the neuron and the feedforward neural network. The neuron is the building block for all the network architectures that will be presented in this book. The concept of a learning system, which follows, will prepare the reader for the rest of the text. The reader may recall that the perceptron learning rule required the addition of a desired output before the network could adapt the weights to find the line that separated one class from the other. This process of using desired outputs for training the neural network is known as supervised training.

2.1 Supervised Training Methods

Supervised training employs a "teacher" to assist in training the network by telling the network what the desired response to a given stimulus should be. This type of training is termed supervised learning and is analogous to a student guided by a teacher. This is exemplified by the diagram in Fig. 2.1. As can be seen in the diagram, the learning system is exposed to the environment, which is represented by a measurement vector of features. The measurement vector is also presented to a teacher who, based on experience, determines the desired response. The desired response is then used to create an error signal that adapts the weights of the learning system. Thus, each input-feature vector has an associated desired-output vector, which is used to train the neural network. The important principle is that supervised learning requires an input and a corresponding desired output. In many situations, it is neither practical nor possible to train the learning system by using supervised-learning methods. The next section introduces the concept of an unsupervised-learning method to handle situations in which supervised training is impractical.

2.2 Unsupervised Training Methods

This section outlines the basic principles of unsupervised learning. The unsupervised-training model (see Fig. 2.2) is similar to the supervised model but differs in

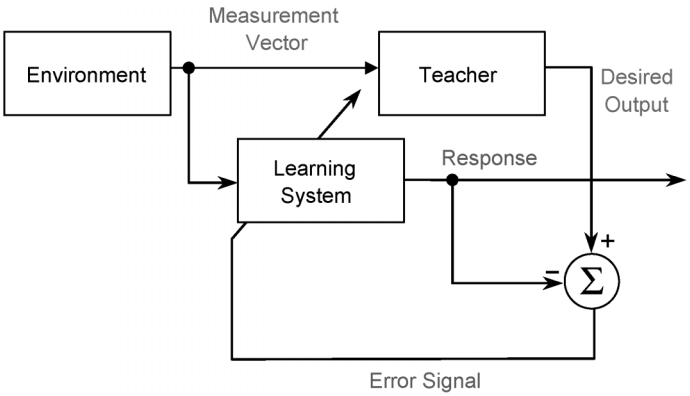


Figure 2.1 Block diagram of supervised-learning model.

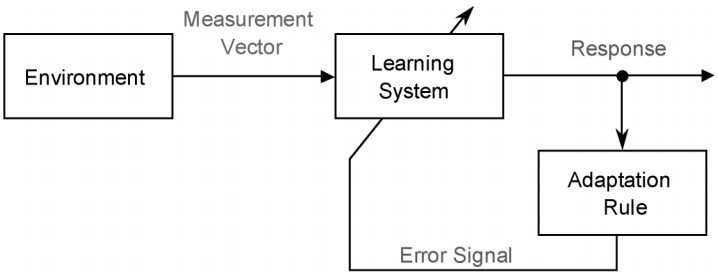


Figure 2.2 Unsupervised-training model.

that no teacher is employed in the training process. It is analogous to students learning the lesson on their own. The learning process is a somewhat open loop, with a set of adaptation rules that govern general behavior. The unsupervised-training model consists of the environment, represented by a measurement vector. The measurement vector is fed to the learning system and the system response is obtained. Based upon the system response and the adaptation rule employed, the weights of the learning system are adjusted to obtain the desired performance. Note that unlike the supervised-training method, the unsupervised method does not need a desired output for each input-feature vector. The adaptation rule in the unsupervised-training algorithm performs the error-signal generation role the teacher performs in the supervised-learning system. Thus, the behavior of the unsupervised learning system depends in large measure on the adaptation rule used to control how the weights are adjusted.

Two of the most popular unsupervised-learning techniques, which will be reviewed in detail later, are the Self-Organizing Map (SOM), developed by Teuvo Kohonen, and the Adaptive Resonance Theory (ART) network, developed by Stephen Grossberg and Gail Carpenter. Many other unsupervised training methods are available to choose from, which all follow the basic model provided in Fig. 2.2.

Chapter 3

Data Normalization

One of the most common tools used by designers of automated recognition systems to obtain better results is to utilize data normalization. There are many types of data normalization. Ideally a system designer wants the same range of values for each input feature in order to minimize bias within the neural network for one feature over another. Data normalization can also speed up training time by starting the training process for each feature within the same scale. Data normalization is especially useful for modeling applications where the inputs are generally on widely different scales. The authors present some of the more common data normalization techniques in this chapter, beginning with statistical or Z-score normalization.

3.1 Statistical or Z-Score Normalization

The statistical or Z-score normalization technique uses the mean (μ) and standard deviation (σ) for each feature across a set of training data to normalize each input feature vector. The mean and standard deviation are computed for each feature, and then the transformation given in Eq. (3.1) is made to each input feature vector as it is presented. This produces data where each feature has a zero mean and a unit variance. Sometimes the normalization technique is applied to all of the feature vectors in the data set first, creating a new training set, and then training is commenced. Once the means and standard deviations are computed for each feature (x_i) over a set of training data, they must be retained and used as weights in the final system design. Equation (3.1) is applied to any input feature vector presented to the network as a preprocessing layer within the neural network structure. Otherwise, the performance of the neural network will vary significantly because it was trained on a different data representation than the unnormalized data. One of the statistical norm's advantages is that it reduces the effects of outliers in the data. The statistical norm is given by

$$x_i' = \left[\frac{(x_i - \mu_i)}{\sigma_i} \right]. \tag{3.1}$$

3.2 Min-Max Normalization

There are times when the neural network designer may wish to constrain the range of each input feature or each output of a neural network. This is often done by rescaling the features or outputs from one range of values to a new range of values. Most often the features are rescaled to lie within a range of 0 to 1 or from -1 to 1. The rescaling is often accomplished by using a linear interpolation formula such as that given in Eq. (3.2):

$$x_i' = (\max_{target} - \min_{target}) \times \left[\frac{(x_i - \min_{value})}{(\max_{value} - \min_{value})} \right] + \min_{target}, \tag{3.2}$$

where $(\max_{value} - \min_{value}) \neq 0$. When $(\max_{value} - \min_{value}) = 0$ for a feature, it indicates a constant value for that feature in the data. When a feature value is found in the data with a constant value, it should be removed because it does not provide any information to the neural network. As can be seen in Eq. (3.2), the maximum and minimum values $(\min_{value}, \max_{value})$ for each feature in the data are calculated and the data are linearly transformed to lie within the desired range of values $(\min_{target}, \max_{target})$. When the min-max normalization is applied, each feature will lie within the new range of values; but the underlying distributions of the corresponding features within the new range of values will remain the same. This allows the designer more flexibility in designing neural networks and in determining which features are of more worth than others when making a decision. Min-max normalization has the advantage of preserving exactly all relationships in the data, and it does not introduce any bias.

3.3 Sigmoidal or Softmax Normalization

Sigmoidal or Softmax normalization is a way of reducing the influence of extreme values or outliers in the data without removing them from the data set. It is useful when you have outlier data that you wish to include in the data set while still preserving the significance of data within a standard deviation of the mean. Data are nonlinearly transformed by using a sigmoidal function: either the logistic sigmoid function or the hyperbolic tangent function. These functions were illustrated in Fig. 1.5. The mean and standard deviation are computed for each feature and used in the transformation given in Eq. (3.3) with a logistic sigmoid. It puts the normalized data in a range of 0 to 1. Equation (3.4) shows the normalization with a hyperbolic tangent; it puts the normalized data in a range of -1 to 1:

$$x_i' \equiv \frac{1}{1 + e^{-\left(\frac{x_i - \mu_i}{\sigma_i}\right)}}, \tag{3.3}$$

$$x_i' \equiv \frac{1 - e^{-\left(\frac{x_i - \mu_i}{\sigma_i}\right)}}{1 + e^{-\left(\frac{x_i - \mu_i}{\sigma_i}\right)}}. \tag{3.4}$$

This transformation is almost linear near the mean value and has a smooth nonlinearity at both extremes to ensure that all values are within a limited range. This maintains the resolution of most values that are within a standard deviation of the mean. Since most values are nearly linearly transformed, it is sometimes called Softmax normalization.

3.4 Energy Normalization

While the statistical, min-max, and sigmoidal norms work on each feature across a set of training vectors, the energy norm works on each input vector independently. The energy normalization transformation using any desired Minkowski norm $(1, 2, \ldots)$ [see Eq. (3.4)], is given in Eq. (3.5). The two most popular energy norms used are the L1 $(n = 1)$ or taxicab norm and the L2 $(n = 2)$ or Euclidean norm. The normalized feature vector is computed by dividing each input feature vector by its length, L1 norm or L2 norm, such that when a dot product of the normalized vector is computed with itself, the length is 1.

$$M_n(\mathbf{x}) = \sqrt[n]{\sum_{i=1}^{N} |x_i|^n}, \tag{3.5}$$

$$x_i' = \frac{x_i}{\sqrt[n]{\sum_{i=1}^{N} |x_i|^n}}. \tag{3.6}$$

As in the case of the previous normalization schemes, once a network has been trained with the energy normalization process, the energy normalization process must be performed on every feature vector presented to the neural network. Failure to perform the normalization will result in poor performance by the neural network because the designer will have asked it to solve one problem, while in practice presenting it with different data than it had been trained with.

3.5 Principal Components Normalization

Another very popular normalization method employed by pattern recognition experts is that of principal components analysis (PCA). PCA is sometimes also referred to as the Hotelling transform [Hotelling, 1933]. Principal components normalization is based on the premise that the salient information in a given set of features lies in those features that have the largest variance. This means that for a given set of data, the features that exhibit the most variance are the most descriptive for determining differences between sets of data. This is accomplished by using eigenvector analysis [Jackson, 1991] on either the covariance matrix or correlation matrix for a set of data. The use of the covariance matrix requires that the features be constrained to similar ranges of values. Hence, a statistical normalization (or norm) or min-max norm may need to be performed on the data prior to performing the PCA transformation. Jackson points out that the principal

components obtained on the covariance matrix differ from those obtained from the correlation matrix. We will demonstrate how the principal components method can assist in solving pattern recognition problems by using the Fisher iris data set. The Fisher iris data set is well known in the pattern classification literature, having been popularized by Sir Ronald Fisher [Fisher, 1936] from data collected by Edgar Anderson [Anderson, 1935]. The Fisher iris data consists of 150 vectors of four features: (1) sepal length, (2) sepal width, (3) petal length, and (4) petal width for three different varieties of iris: (1) setosa, (2) virginica, and (3) versicolor. Figure 3.1 shows a scatter plot for each of the features on the Fisher iris data. As can be observed in the scatter plot, there is significant overlap of the data between most of the features. It will be shown that the use of PCA analysis can assist in finding the best way to dichotomize, i.e., divide up, the classes of data. Figure 3.2 shows the iris data plotted against the first three features. As can be seen in Fig. 3.2 and by observing the scatter plots in Fig. 3.1, the data are overlapped, which makes pattern recognition between the classes difficult. The PCA transformation helps to minimize the overlap between data classes, as shown in Fig. 3.3. By projecting the data onto the eigenvectors, sorted largest to smallest, of the covariance matrix of the original data, a new set of features is obtained that provides the best separation for the data, starting with the maximum variance down to the smallest. As with the other normalization techniques, once the desired eigenvectors are found, all data presented to the neural network will need to be projected into features in the PCA domain before the proper output is obtained from the network.

To illustrate the power of the PCA technique, consider Fig. 3.2, which depicts good separation of the data with the first three features of the Fisher iris data set.

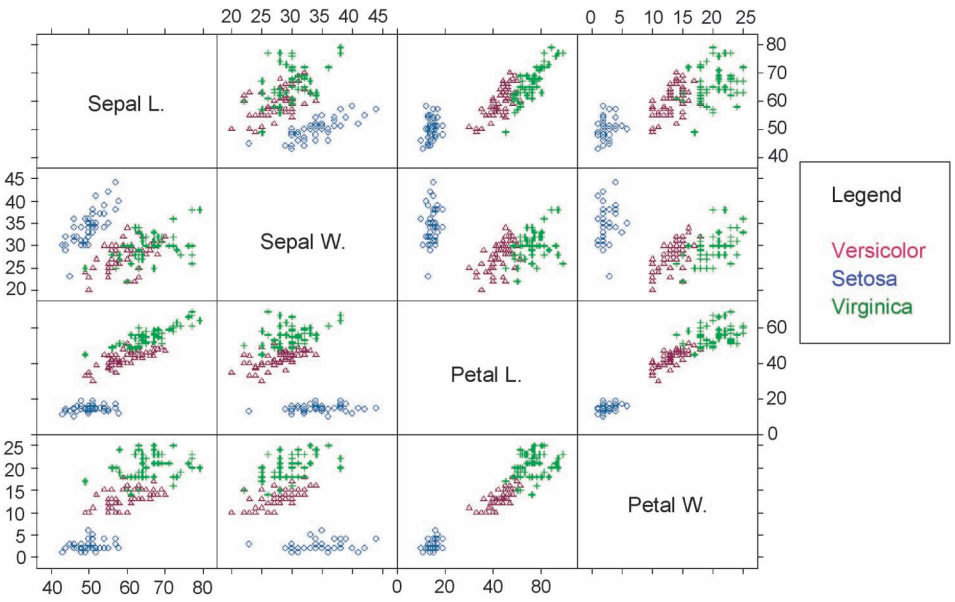


Figure 3.1 Scatter plot of Fisher iris data.

The axes in Fig. 3.2 were rotated by hand until a reasonable amount of separation was obtained. Using PCA, a much better separation of the data are obtained, as shown in Fig. 3.3. Now imagine what would happen if you had 100 features rather than 4 to search for the best separation. You could readily see that the PCA method is far superior to any method that could be obtained by hand. As is shown in Fig. 3.3, the data are now easily separated by any classifier. There are classes of problems where PCA fails and other methods such as independent component analysis [Hyvärinen, 1999; Hyvärinen, 2000] are used to separate data. If you have good features to start with, you will always have a good classifier. This is something that the reader should always remember! Most pattern recognition experts spend

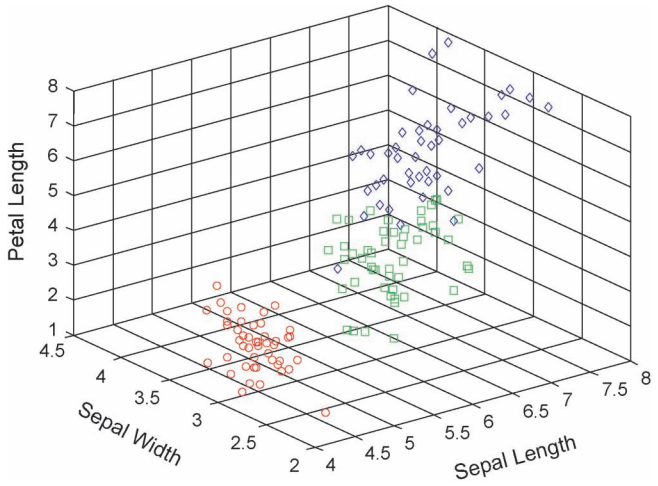


Figure 3.2 Fisher iris data plotted for first three features.

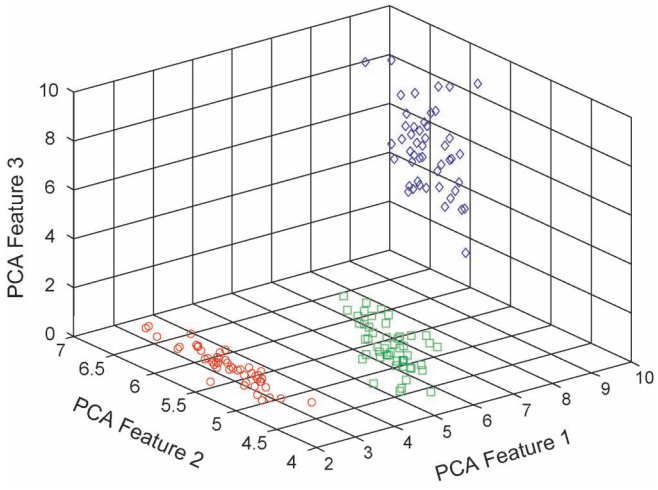


Figure 3.3 Fisher iris data after PCA transformation using projection of the original data onto the first three eigenvectors of the covariance matrix as features.

the majority of their time developing good robust features. Then, the selection of the classifier is a matter of choice based upon system constraints. The eigenvectors obtained from the PCA process for the Fisher iris data are given in Fig. 3.4. Once the eigenvectors of the covariance matrix are found they can be incorporated into the feedforward neural net architecture. A feedforward neural network implementation of the PCA transformation for the Fisher iris data are given in Fig. 3.5. The PCA transformation algorithm is given as Matlab code in Appendix C.

The normalization schemes and PCA transformation presented in this chapter are just a few of many different ways to obtain robust features. Oftentimes, the desired features are problem domain dependent. For example, if you are doing vibration analysis on rotating machinery, it is a natural extension to transform data from the time domain to the frequency domain to improve performance.

e_1	e_2	e_3
03616	0.6565	0.5810
–0.0823	0.7297	–0.5964
0.8566	–0.1758	–0.0725
0.3588	–0.0747	–0.5491

Figure 3.4 First three eigenvectors for Fisher iris data.

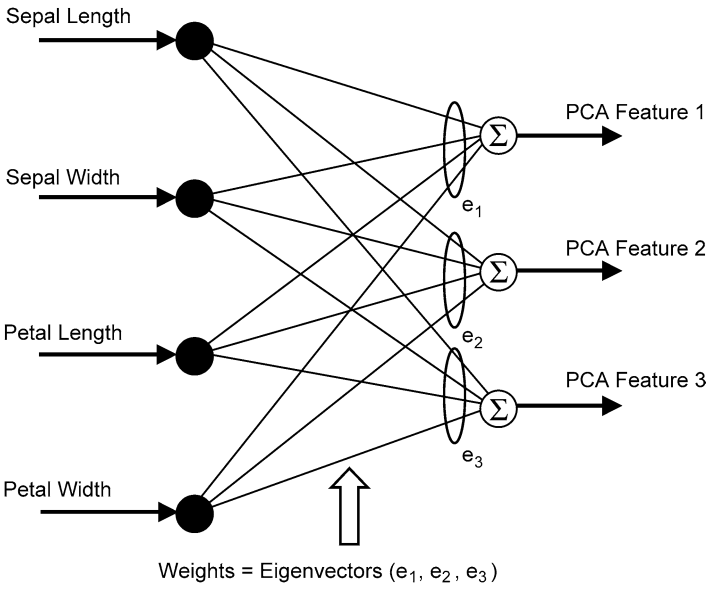


Figure 3.5 Feedforward neural network implementing PCA for the Fisher iris data set.

Chapter 4

Data Collection, Preparation, Labeling, and Input Coding

Since neural networks are data driven, the adage "garbage in, garbage out" is highly relevant to the task of building a neural network. Proper collection, preparation, labeling, and coding of the data can make the difference between a successful and unsuccessful experience with neural networks.

4.1 Data Collection

While the process of collecting data seems simple, the network designer should put some thought into the data-collection process. The designer needs to decide what he wants the neural network to do and what data requirements are needed to train the network. Will it be a classifier, an estimator (modeler), or a self-organizer (clusterer)? The designer needs to determine how and from where to obtain the data and what types of data to collect. He must also determine what the neural network will output in response to the data used as the network input. The steps in a typical data-collection plan are described next.

4.1.1 Data-Collection Plan

1. **Define the problem and how the neural network will be used in its solution.** From this, define the goals and objectives of the data-collection process. What is the neural network expected to do? What data will be needed to accomplish the goals and objectives? By knowing whether a network is expected to identify different input-feature vectors, performing as a classifier; predict the output of a system based upon an input, performing as an estimator; or determine common attributes for a set of input-feature vectors, performing as a clusterer, the designer is in a much better position to determine what data are needed to achieve the desired result.
2. **Identify the quantities of data to be collected.** What measurements are being made and what outputs are the neural network expected to learn? If the designer is developing a classifier, then target classes must be assigned. If

an estimator is desired, then the target estimated outputs must be assigned. If the designer plans a self-organizing system, he still needs to know something about the data to interpret the categories that the neural network will form. Generally, all the data will need to be in numerical form for training and operating the neural network.

3. **Define the methodology for the data collection.** The neural network designer needs to make sure that he will be forming a data set that is representative of the population or space to which the neural network will be exposed during its operation. This means the data used to train the network must be sampled across the population as shown in Fig. 4.1. For example, if the network is trained to predict the outcome of a national election and data from only one region of the country are collected, the neural network will not see a representative sample during training and may be unduly biased during operation. If the network is being trained to utilize time-series data, the designer should make sure the data are sampled often enough to capture any important changes. If the designer is dealing with spatial data (e.g., images), he should be certain to sample at a fine enough resolution to capture important details.
4. **Identify the expected ranges for the data.** The designer should identify the expected range for each feature in the input data to determine whether systematic errors occur during data collection. Using these ranges will allow the designer to flag erroneous data. For example, if a measurement system is collecting ambient-air-temperature data and the data contain readings over 60°C, most likely equipment problems exist, because the hottest ambient air temperature ever recorded on the surface of the Earth was 57.8°C. After the data are collected, the designer should perform basic statistical analysis to determine the range and other statistics of each feature (e.g., minimum, maximum, mean, standard deviation). He should make sure each feature is within the expected ranges. If feature vectors exist with features well beyond the expected values (often termed outliers), the designer should determine

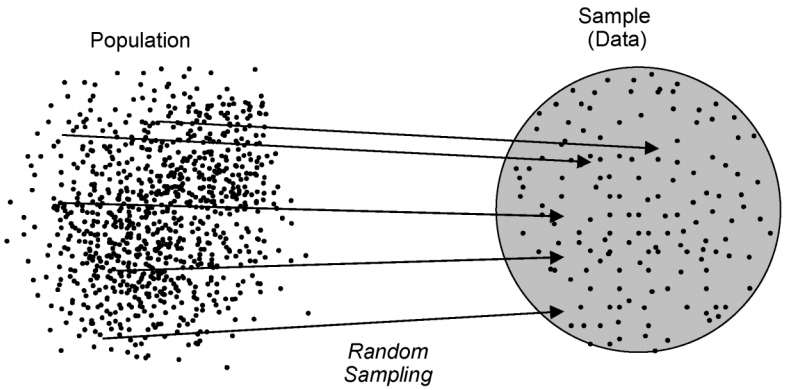


Figure 4.1 The process of sampling a population. Ideally, this sampling should be representative of the whole population of interest.

why these outliers occurred. Outliers should be investigated thoroughly, because they might represent an important condition the designer had not considered. But, if the designer suspects a feature vector is erroneous, it should be removed from the data set. If the data contain a large number of outliers, the designer should suspect the entire data-collection process is flawed, and identify and correct the problem before starting over.

5. **Make sure the data-collection process is repeatable, reproducible, and stable.** The network designer should make sure the data-collection process is repeatable, reproducible, and stable in order to ensure the accuracy of the feature vectors, or samples, is maintained.
6. **Ensure the data are protected from unauthorized access.** The network designer will encounter many applications, where the data or its application could contain information that needs protection from unauthorized access (e.g., personal, strictly private, proprietary, business-sensitive, confidential, official-use-only, or classified data). If necessary, the data-collection plan should include appropriate procedures to protect the data and its intended application from unauthorized access. For example, medical technologies under development often require human subjects' boards to show how patient information was protected from unauthorized access. This often involves scrubbing sensitive information (e.g., name, patient identification, etc.) from the data used during the training and testing process, limiting access to the data, and destroying all data records at the completion of the effort.

4.1.2 Biased Data Set

When the statistics of the training data do not represent the statistics of the data encountered during operation, the network is said to be biased. Biased data sets can result in classifiers with lower classification rates, estimators with lower prediction accuracies, and clustering algorithms (also known as self-organizers) with increased variance in groupings. It must be stressed that a large number of representative training samples, or feature vectors, is necessary to train a neural network so that it will operate properly over the expected range of the input-feature space without memorizing the training data. The ability of a network to perform the desired mapping, from input to output, without memorization, is called generalization. The data must include all the different characteristics necessary to produce the desired output under any anticipated condition. It is also important to select samples or feature vectors that do not have major dominant features that are unrelated to the problem but are common to the specific condition. In many data sets, the unrelated dominant features cannot be determined until a neural network has been trained and checked against a validation data set. (In Appendix B, the authors present a method for determining which features are the most relevant.) While biased data sets are often undesirable, in some situations biased data sets are beneficial. These include situations in which important samples are underrepresented or the outcome

must be biased in one direction or another to reduce error. For example, if a sample in the data set represents a unique situation but has a very low probability of occurrence, it is likely that the neural network will not learn this input-to-output mapping because it is so rare. By training on this case more often (i.e., increasing its *a priori* probability), the neural network is more likely to learn this input-to-output mapping. This biases the neural network but improves the chances that the neural network will learn this rare situation.

One well-known example showing how a biased training set can affect performance is that of a neural network developed for the U.S. Army in the late 1980s. Its purpose was to examine images of a forest scene and determine whether military tanks were in the images [Fraser, 1998]. One hundred images of tanks hiding behind trees and 100 images of trees without tanks were collected. Half of the images were used in training and half were set aside for testing. After training, the neural network was presented with the test data. The network worked remarkably well at discriminating between images with tanks and those without.

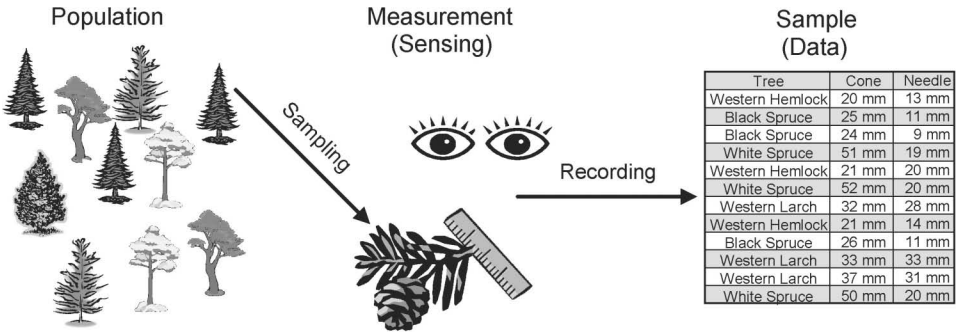
While the U.S. Army was pleased with the results, officials were suspicious of the neural network's success. A second data-collection round was commissioned and the network re-tested. This time, the neural network performed poorly. After a lengthy examination, it was determined that all the original images of tanks had been collected on a cloudy day and all the images without tanks were collected on a sunny day. The neural network had learned to discriminate between a sunny day and a cloudy day, not between images with tanks and those without. It is critically important for the designer to collect training data that is representative of all situations that he anticipates the neural network will process.

4.1.3 Amount of Data

In general, more data are better than fewer data. A larger number of samples (i.e., a bigger data set) will generally give the network a better representation of the desired problem and will increase the likelihood that the neural network will produce the desired outputs. More data also help when noise is present in the data.

4.1.4 Features/Measurements

An individual sample is described by a unique set of measurements. In pattern-recognition vocabulary, these measurements are known as features [Duda, 1973]. Each feature forms a dimension in a space known as feature space. For example, if we wanted to classify different evergreen trees in the forest, we would measure different characteristics of each tree. We could measure the length of the needle and length of the cone as illustrated in Fig. 4.2. This would represent two features. So, for every sample we would measure two quantities and this would form a two-dimensional feature vector to help identify the type of tree. These vectors are then represented by points in feature space. Figure 4.3 shows a feature space for this example with four types of evergreen trees and two features. Most applications will involve many more features and will result in a multidimensional feature space.



Tree	Cone	Needle
Western Hemlock	20 mm	13 mm
Black Spruce	25 mm	11 mm
Black Spruce	24 mm	9 mm
White Spruce	51 mm	19 mm
Western Hemlock	21 mm	20 mm
White Spruce	52 mm	20 mm
Western Larch	32 mm	28 mm
Western Hemlock	21 mm	14 mm
Black Spruce	26 mm	11 mm
Western Larch	33 mm	33 mm
Western Larch	37 mm	31 mm
White Spruce	50 mm	20 mm

Figure 4.2 Process of collecting data through measurement. In this example, a forest represents the population. By measuring two features (needle length and cone length), a data set of samples is generated to represent the population.

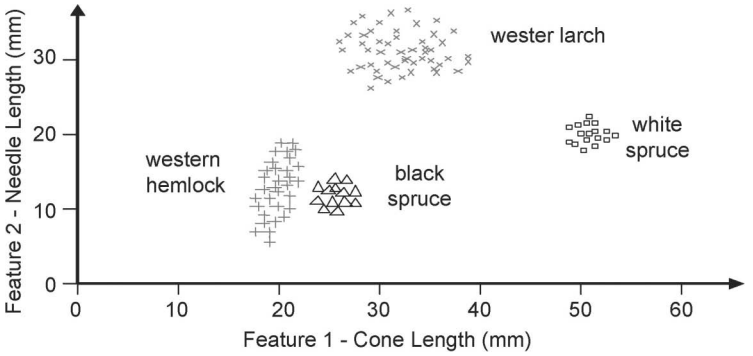


Figure 4.3 Example of a two-dimensional feature space representing vectors from samples taken from four types of evergreen trees.

4.1.5 Data Labeling

For supervised approaches, the data must be labeled or truthed. This requires the neural-network designer, or a model, to assign target values to each sample collected. If the label is not already a number, then it must be converted to a numerical form in order for the neural network to be trained via computer.

While samples do not generally need to be labeled for unsupervised approaches, the neural-network designer does need to know something about the data in order to interpret the results. It is useful to record ancillary information (i.e., metadata) that might not be used as inputs to the unsupervised neural network but that can provide a better understanding of the groupings that it produces.

4.2 Feature Selection and Extraction

Feature selection is key to developing a successful neural network. When the number of features is small and the number of samples is large, the designer can allow

the neural network to choose the importance of each feature in making its decisions. On the other hand, if many features and few samples exist, the designer can have a situation in which the neural network might not produce a unique mapping, resulting in poor performance. We discuss the relationship between the number of samples and the number of features in more detail in Section 4.2.2.

4.2.1 The Curse of Dimensionality

As the number of features or dimensions increases, so does the amount of information. While having a large number of features may seem preferable, it is possible that a neural network would perform worse with more than with fewer features. In addition, as features are added, more samples are needed to prevent the neural network from memorizing the data. This is often termed "the curse of dimensionality," a term coined by Richard Bellman as an observation that the number of data points needed to sample a space grows exponential in its dimensionality. In his book, he states:

> In view of all that we have said in the forgoing sections, the many obstacles we appear to have surmounted, what casts the pall over our victory celebration? It is the curse of dimensionality, a malediction that has plagued the scientist from the earliest days. [Bellman, 1961, p. 97]

As stated previously, a neural network is a mapping of an input space to an output space. Each additional input to the neural network adds another dimension to the space that is being mapped. During training, data representative of every occupied part of input space are needed to train the neural network so that it properly maps input space to output space. There must be sufficient data to populate the space densely enough to represent the mapping relationship, so significantly more samples are required to represent a higher dimension space than a lower dimension space. Covering the input space consumes resources including time, memory, and data. Fortunately, this curse of dimensionality can be ameliorated by proper selection and reduction of features. We present one method for determining which features are the most important or salient in Appendix B. Knowing the salient features allows the neural network designer to reduce the dimensionality of the input data by eliminating poor features.

4.2.2 Feature Reduction/Dimensionality Reduction

For some applications, the number of input features is overwhelming. In these instances, the number of features must be reduced. This is done through feature selection, feature extraction, or a combination of the two. The basic principles of feature reduction follow.

The features in the data set might not be the most efficient in representing the information presented to the neural network because they can include inputs unrelated to the relationship between the input and output space that is to be learned. Feature vectors can contain features that are correlated with one another, which

represents redundant information. When applied properly, feature selection and extraction preserves the information necessary for training while reducing the number of features. However, blindly applying feature reduction can lead to poor performance, and the process of selecting and extracting a set of features to produce a reduced number of inputs can require application-specific domain knowledge. The extracted or retained features should preserve class separation in classifiers, estimation accuracy for estimators, and groupings in self-organizers.

Some common feature-extraction approaches include averaging, principal components analysis (PCA) [Hotelling, 1933], moment invariants [Hu, 1962], Fourier coefficients, wavelet analysis, resampling, linear discriminant analysis (LDA) or Fisher mapping [Fisher, 1936], independent components analysis (ICA) [Hyvärinen, 2001], principal curves [Hastie, 1989], Sammon mapping [Sammon, 1969], and self-organizing maps (SOMs) [Kohonen, 1982]. The description of these techniques would easily fill an entire book. The main thing to remember when dealing with neural networks, and classifiers in general, is that the feature extractor does much of the work.

One example of feature extraction is the Standard and Poor's 500 Index (S&P 500), a weighted average of 500 of the largest publicly traded U.S. companies. Each stock in the index is selected for liquidity, size, and industry and is weighted for market capitalization. Using the S&P 500 feature rather than each of the individual stocks to track the market is a way to reduce the total number of individual stocks that need to be tracked, while retaining important trend information. Thus, the S&P 500 index is an extracted feature that uses domain-specific knowledge. For some applications, an extracted feature can provide all of the information the neural network needs. For other applications, the extracted feature might miss important details related to a specific condition, or stock in the S&P 500 example, that degrades the performance of the neural network.

Another reason to employ feature extraction is to reduce or eliminate feature redundancy. In a weather application, measurements such as temperature, dew point, and humidity are recorded. Humidity is related to temperature and dew point and can be mathematically derived from the two. Therefore, we can reduce the number of features we have from three to two by excluding humidity with no loss of information.

Image analysis is one domain that requires a significant amount of feature extraction prior to neural-network development. A typical image might contain several million colored pixels. It would be nearly impossible to train a usable neural network with several million inputs (and certainly ill advised, because of the curse of dimensionality). Another domain with even more need of feature extraction is video analysis. Many methods have been developed to extract features from images and video. The goal of these feature-extraction methods is to drastically reduce or compress the amount of information into something the neural network can use. Distance metrics, introduced in the next section, offer one method for determining how close two feature vectors are to one another.

4.2.3 Feature Distance Metrics

A distance metric, $d(x, y)$, can be used to measure the similarity of two feature vectors, $\mathbf{x}$ and $\mathbf{y}$, in a feature space of N dimensions. The distance metric must have the following properties:

1. The distance between two identical vectors is zero:

$$d(\mathbf{x}, \mathbf{x}) = 0. \tag{4.1}$$

2. The distance between x and y is the same as the distance between y and x:

$$d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x}). \tag{4.2}$$

4. The sum of two distances must be less than or equal to the sum of the individual distances:

$$d(\mathbf{x}, \mathbf{z}) \leq d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z}). \tag{4.3}$$

Some of the common distance metrics follow.

4.2.3.1 Euclidean distance metric

The Euclidean distance metric for two feature vectors, $\mathbf{x}$ and $\mathbf{y}$, of length N is defined as

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{N} (x_i - y_i)^2}. \tag{4.4}$$

4.2.3.2 Sum of squared difference metric

In many applications the sum of squared difference (SSD) metric is used in place of the Euclidean distance to save processing time. For two vectors, $\mathbf{x}$ and $\mathbf{y}$, of length N it is defined as

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{N} (x_i - y_i)^2. \tag{4.5}$$

4.2.3.3 Taxicab distance metric

The taxicab, city block, or Manhattan distance metric is less computational than the Euclidean distance metric and is easier to implement in specialized hardware. For two vectors, $\mathbf{x}$ and $\mathbf{y}$, of length N it is defined as

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{N} |x_i - y_i|. \tag{4.6}$$

4.2.3.4 Mahalanobis distance metric

The Mahalanobis distance metric is a more advanced version of the Euclidean distance metric. It takes into account the distribution of feature vectors of length N and is useful in comparing feature vectors whose elements are quantities having different ranges and amounts of variation. It is mathematically described by Eq. (4.7), where the feature vectors x and y come from the same distribution and have a covariance matrix $\mathbf{C}$:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(\mathbf{x} - \mathbf{y})\mathbf{C}^{-1}(\mathbf{x} - \mathbf{y})}. \tag{4.7}$$

4.2.3.5 Hamming distance metric

The Hamming distance metric is used to determine the difference between two binary-valued vectors, $\mathbf{x}$ and $\mathbf{y}$, of length N and is defined as

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{N} |x_i - y_i|. \tag{4.8}$$

Notice that the Hamming distance is identical to the taxicab metric with the stipulation that the feature vectors be binary.

The distance metrics presented help to determine how close one feature vector is to another. If a feature vector is within a specific distance of the exemplar (the desired feature vector), then the pattern is classified as a member of the exemplar's group. All the unsupervised methods presented in this book use a distance metric and neighborhood around an exemplar. Figure 10.17 in Section 10.5 illustrates how these distance metrics can be used to group features in the tree recognition example.

Chapter 5

Output Coding

The output neurons of most neural-network architectures produce data in the range of -1 to 1 or 0 to 1, depending on the type of neuron. Therefore, the desired outputs must be coded to fit this scale. Also, non-numeric labels, such as those found with classifiers, must be converted to something numeric. This necessitates coding the target outputs. Once the neural network is trained and in operation, the reverse applies and these unitless values must be converted into useful terms. This is done through post-processing of the neural network outputs.

5.1 Classifier Coding

For classifiers, the outputs are generally coded with a 1 for existence in that class and 0 or -1 for absence from that class. With sigmoidal output neurons, sometimes the target output values are pushed back from the extreme edges of the sigmoid so that 0.9 and 0.1 for a logistic function or 0.9 and -0.9 for a hyperbolic tangent function are used instead. For example, if we were trying to classify trees, there would be one output class for each tree. Table 5.1 shows an output-coding scheme for the four classes, with 0.9 representing existence in that class and 0.1 representing absence from that class.

5.2 Estimator Coding

For estimators, systems that predict an output based upon an input, the target output needs to be scaled within the neuron's output range. The target output is often

Table 5.1 Example output coding scheme for a four-tree classifier.

Tree	Output 1	Output 2	Output 3	Output 4
Black Spruce	0.9	0.1	0.1	0.1
Western Hemlock	0.1	0.9	0.1	0.1
Western Larch	0.1	0.1	0.9	0.1
White Spruce	0.1	0.1	0.1	0.9

scaled back from the extreme edges of the neuron's output range in order to prevent multiple inputs from receiving the same output value, a condition known as data squashing. Sometimes the target output is transformed by the activation function of the output neuron (e.g., sigmoidal function). Another option is to use a linear activation function for the output neuron. Equation (5.1) represents the rescaling of a target output, t, to the neuron's output range and through the neuron's activation function, f.

$$t' = (\max_{target} - \min_{target}) \cdot f\left(\frac{t - \min_{value}}{\max_{value} - \min_{value}}\right) + \min_{target}. \tag{5.1}$$

For example, if one of the outputs for a neural network is an estimate of the ambient air temperature anywhere on the earth's surface, it would be best to use the expected range for ambient air temperature to rescale all target values. The coldest temperature ever recorded was −89.5°C in Vostok, Antarctica. The hottest temperature ever recorded was 57.8°C in El Azizia, Libya. The network designer could use this knowledge to rescale the target temperatures to a range of 0.1 to 0.9 by Eq. (5.2).

$$t' = (0.9 - 0.1) \cdot f\left(\frac{t + 89.5°\text{C}}{57.8°\text{C} + 89.5°\text{C}}\right) + 0.1. \tag{5.2}$$

Output neurons from a neural network can represent either continuous or binary values. The network designer must decide whether to code the output as one continuous value by using a single neuron, or as several binary values by using multiple neurons. For example, an estimation application might output the day of the week. These days could be represented by one continuous output scaled from 1 to 7, or a set of seven binary outputs with one output for each day of the week. Assigning continuous values of 1 through 7 to represent the days of the week implies a predetermined ranking for each day. Sunday (1) and Monday (2) are next to each other on a continuous scale. Sunday (1) and Saturday (7) are at the opposite ends of the scale. If Saturday and Sunday had more in common than Sunday and Monday, the predetermined ranking could confuse the training process. There might be no reason to believe that there is a specific relationship between the days of the week, even though we know there might be. Therefore, by creating seven binary-output neurons, there is no predetermined ranking and each day is treated independently. The disadvantage of having seven outputs instead of one is that additional weights connect to the output layer, which results in more training time and the need for more training data. Also, if the days of the week are related by this order, then the information contained in the order is not used.

Chapter 6

Post-processing

As mentioned in the previous section, the output of a neural network is generally a set of unitless numbers on a scale between 0 to 1 or −1 to 1. Therefore, for applications that require data ranges outside of the neuron output, the data must be rescaled to the desired data range.

Classifiers usually have a separate output for each class. In this case, the outputs need to be thresholded so that a value above the threshold indicates that a given input is classified in that class and a value below the threshold indicates that input is not a member of that class. This thresholding is often accomplished by using a step function like those shown in Fig. 6.1, which results in a binary output as shown on the left side of Fig. 6.1. The value used to threshold the output can be adjusted to produce the optimum ratio of detection to false alarms. Sometimes, it is useful to have an upper and lower threshold for a given classifier design, permitting the classifier to have a "not sure" or indeterminate region. If an output falls above the upper threshold, it is marked as part of the class. If it falls below the lower threshold, it is marked as not part of the class. If it falls between the two thresholds, then the class should be considered indeterminate. This results in two binary outputs: one indicating class membership and one indicating no class membership. If both outputs are low, then class membership is possible but not definite, indicating an indeterminate condition, as shown on the right side of Fig. 6.1.

In many estimation or modeling applications, the output values represent a continuous scale and need to be interpreted as real-world quantities with real-world units. Rescaling the outputs linearly to the range of the real-world quantity will

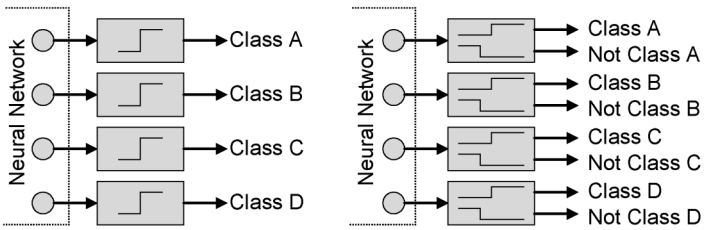


Figure 6.1 Process of thresholding neural-network outputs to determine class membership.

accomplish this, as represented by Eq. (6.1), where y represents the output of the neuron and y' represents the rescaled output. If the target values were not initially scaled by the neuron's activation function, as shown in Eq. (6.1), and the activation function was a sigmoid, the designer can include an output transform by an inverse sigmoid. This is shown in Eq. (6.2).

$$y' = (Max\_value - Min\_value) \times \left(\frac{y - Min\_target}{Max\_target - Min\_target} \right) + Min\_value, \tag{6.1}$$

$$y' = (Max\_value - Min\_value) \times f^{-1}\left(\frac{y - Min\_target}{Max\_target - Min\_target} \right) + Min\_value. \tag{6.2}$$

Returning to the weather estimator example, we rescale the neuron's output with Eq. (6.3). For the maximum (57.8°C) and minimum (−89.5°C) temperature example, scaling the neuron output produces a value that has meaning in terms of recorded temperature:

$$y' = (57.8°\text{C} + 89.5°\text{C}) \cdot \left(\frac{y - 0.1}{0.9 - 0.1} \right) - 89.5°\text{C}. \tag{6.3}$$

Once the network designer has determined the scaling to be performed on the data, he/she can then choose the best learning method to solve a given problem.

Chapter 7

Supervised Training Methods

Most artificial neural networks are trained with supervised learning methods. A simple model for supervised learning is given in Fig. 7.1. The outside world is measured and the measurement vector, x, is given to a knowledge expert who outputs a desired response, $f(x)$. The learning system is exposed to the same measured variable and also computes a result, $\tilde{f}(x)$. The error between the output of the learning system and the desired response from the knowledge expert is measured. The error signal is then used to modify the response of the learning system, adapting weights for neural networks, so that its response more closely matches that of the knowledge expert. The knowledge expert can be a human expert, a function, a set of rules, a set of measured system outputs, and so forth. The learning system can be trained by using any number of adaptation methods such as backpropagation, fuzzy logic, expert-system rules, evolutionary computation, statistical methods, or an *ad hoc* method. The key principle is that a set of input data and the desired system responses are used to adapt the learning system.

The most common neural network trained with supervised training methods is a feedforward neural network containing sigmoid transfer functions. Usually, a form of backpropagation (see Appendix A) is used to train the feedforward neural network, but other training methods could be employed. As mentioned previously,

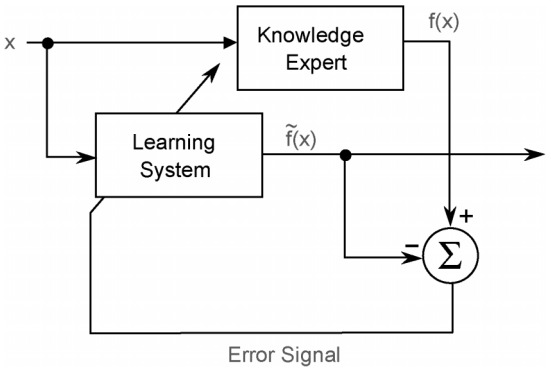


Figure 7.1 Supervised learning model.

the weights of the feedforward network are adjusted such that the error between the desired output and the actual output of the network is minimized by the weight-update method.

7.1 The Effects of Training Data on Neural Network Performance

As machine-learning systems are developed and applied to different problem domains, questions often arise as to the best method for training the system. The machine-learning system can be trained and implemented as a single multiclass system or as a modular system. The system's performance and reusability are traded against one another in the designer's decision process. This section presents four methods of training a feedforward artificial neural network architecture [Priddy, 2004] to illustrate the effect of training on the decision region of the classifier. The four methods involve using a four-class feedforward neural network and a composite network comprised of four individual feedforward neural networks, followed by a decision layer. Both are trained by using the data as found in the problem domain, followed by training both networks on the same training data, augmented with zeros. The feedforward network is an approximation to a Bayes [Ruck, 1990b] optimal discriminant function, a hyperplane that divides classes in an optimal way, which makes it an excellent choice for illustrating the effects of training on decision regions for a machine-learning classifier.

The feedforward multilayer perceptron maps from input space to output space by using a combination of layers and nonlinear neurons. A typical single-hidden-layer feedforward neural network is depicted in Fig. 7.2.

While the multiclass method of utilizing classifiers is valid, it has some drawbacks when the problem domain involves extensive training time for the classifier. This has forced developers to adopt a modular approach to classifier design that can potentially reduce the time required to train or add new classes to the classification system. An example modular system is given in Fig. 7.3.

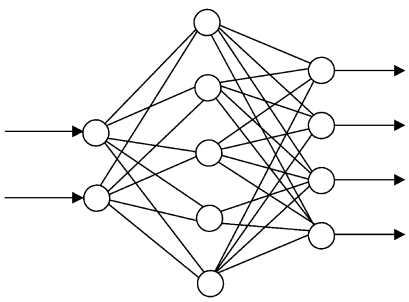


Figure 7.2 Multiclass feedforward neural network.

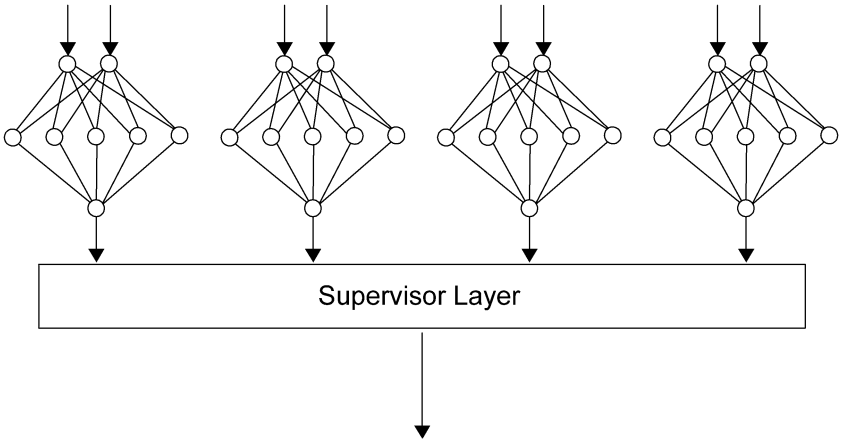


Figure 7.3 Multiple single-class feedforward neural networks, combined to solve a classification problem.

7.1.1 Comparative Analysis

An example problem set is presented that illustrates the effects of training methods and of network architecture. Cybenko [Cybenko, 1989] showed that a single-layer feedforward neural network with logistic sigmoid transfer functions could perform any measurable mapping, given a finite but large enough number of hidden layer nodes. Therefore, one would not expect a large difference in behavior between the two systems in classification performance. To illustrate the differences that do exist, a sample data set is given along with the decision regions formed by the networks. The networks used in this example were created using Netlab [Nabney, 2002], which is available on the Web as a package of modular Matlab routines. As mentioned previously, feedforward networks can contain any desired transfer function, but the feedforward networks presented in this section contained sigmoidal transfer functions. The training set is presented in Fig. 7.4 as a four-class problem with overlap between classes. Given the training data presented in Fig. 7.4, a two-layer feedforward neural network was trained with two input nodes, six hidden-layer nodes, and four output nodes. The neuron transfer functions were sigmoids with the weights between neurons trained with a quasi-Newton training method (see Appendix A), rather than using backpropagation to reduce training time. The resultant decision regions for each class are shown in Fig. 7.5, and the combined result is given in Fig. 7.6.

As can be seen in Fig. 7.5, the feedforward network has done a good job of separating each class. The composite picture in Fig. 7.6 shows how each decision region separates the classes. This training method produces excellent results, but with the constraint that as new data or classes are introduced, it becomes necessary to retrain the entire network. This is simple enough for minor problems such as the one presented here, but has serious ramifications when dealing with systems that have been trained on tens and hundreds of thousands of input feature vectors.

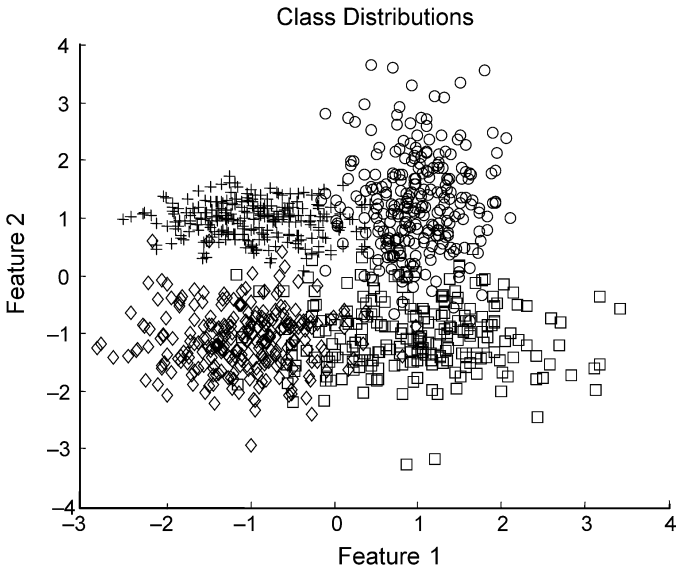


Figure 7.4 Class distributions for four-class classification problem.

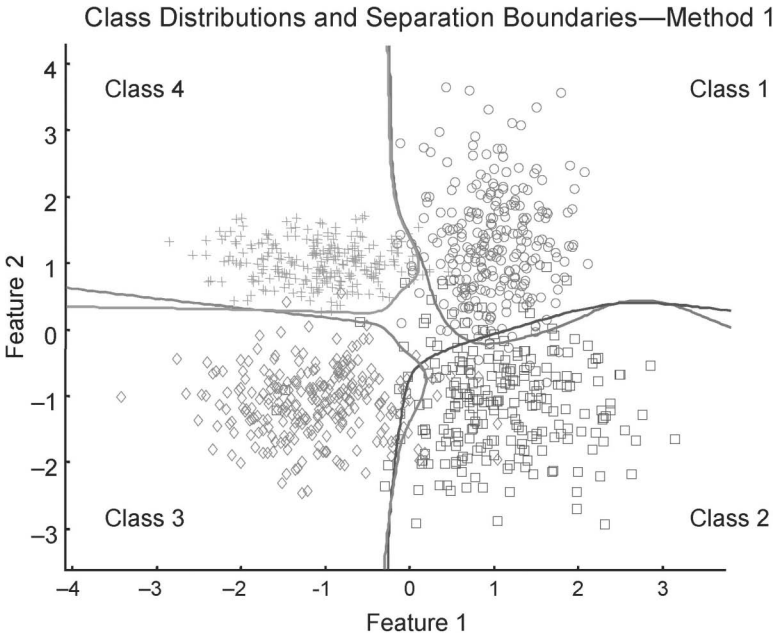


Figure 7.5 Classifier decision boundaries for four-class problem.

Thus, classifier developers often modularize the system as depicted previously in Fig. 7.3. Modularization is accomplished by training a single network for each class and then combining the networks with a supervisory layer such as a max-picker (a component that selects the maximum output). Each individual network is trained

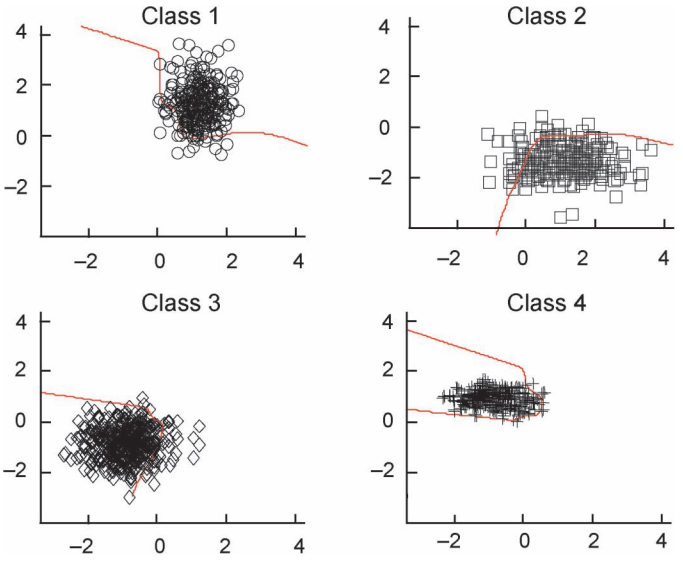


Figure 7.6 Individual decision regions for multiclass classifier trained on the data illustrated in Fig. 7.4 (Method 1).

to output a 1 when feature vectors from its assigned class are presented and a 0 when those from the other classes are presented. In the example presented here, the supervisory layer simply chooses the subnetwork with the maximum output as the class type and reports the type along with the corresponding output of the winning subnetwork. The resultant decision regions for the modularized system (Method 2) are depicted in Fig. 7.7. Thus, it can be seen that while the decision regions for the multiclass (Method 1) and modular (Method 2) approaches are similar, they are definitely different. In both cases (see Figs. 7.5 and 7.7), it can be seen that for some portions of the feature space, decision regions extend much farther than the data sets used to create them.

This can be addressed by a variety of means, such as using Mahalanobis distance [Mahalanobis, 1936] to constrain the network or adding additional training vectors that are outside the data to assist the network in partitioning the feature space. One way to overcome this deficiency is to augment the data with zeros where no samples are found.

The third training method consisted of augmenting each of the class data sets with a set of feature vectors that uniformly spanned the feature space and had a value of zero as the network's target value. Thus, if the data were not contained in the desired class, the network was trained to output a zero. Using this training method on the multiple single-class feedforward neural networks resulted in the decision regions shown in Fig. 7.8. As can be seen in Fig. 7.8, augmenting the known data with zeros results in tight decision regions. Depending upon the desired classifier response, this may or may not be a preferred training method. Generally, networks trained with the third method are best suited to classifier de-

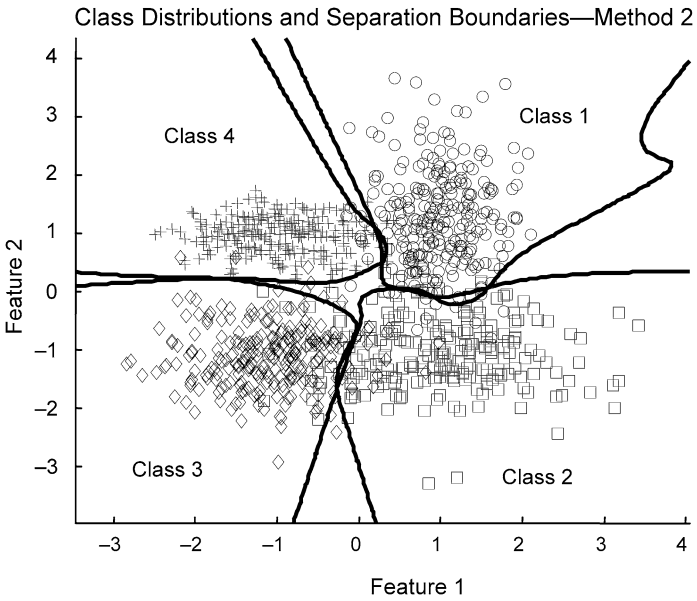


Figure 7.7 Decision boundaries for the modular neural network trained using Method 2.

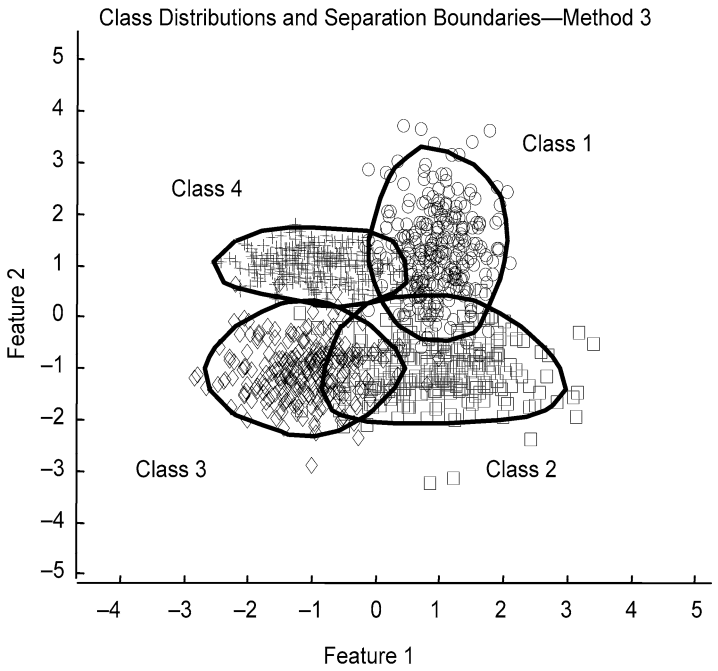


Figure 7.8 Decision regions for modular network trained with augmented data (Method 3).

signs for which the input data ranges are known and the designer is comfortable with the underlying statistics of the input features.

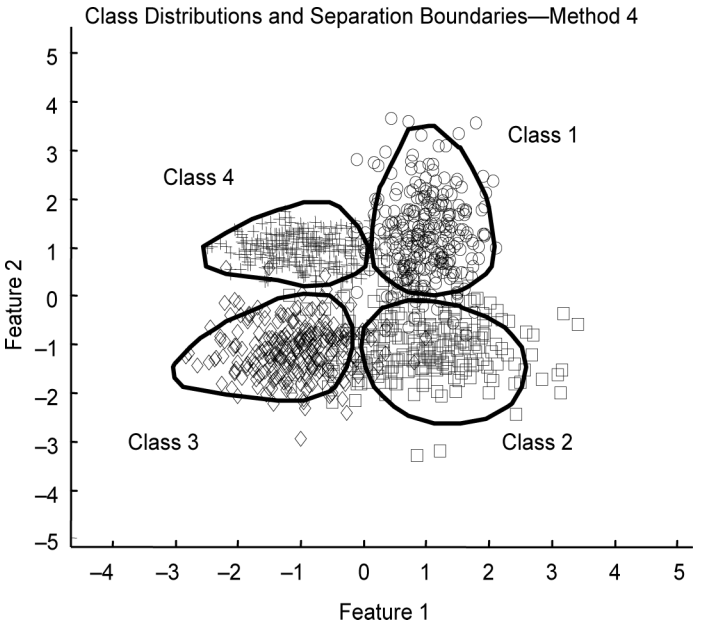


Figure 7.9 Decision regions for multiclass neural network trained on augmented data (Method 4).

Lastly, the multiclass neural network was retrained with the augmented data, with the resultant decision regions depicted in Fig. 7.9. Note that the regions are non-overlapping, which may but is not guaranteed to result in better performance. In the example presented here, the overall performance difference between the networks trained using Method 3 and Method 4 is slight.

The results for each classifier are presented in confusion matrices. A confusion matrix is constructed with the input class or truth for the test feature vectors on the left (row) and the classifier output on the right (column) indices. Thus, an ideal confusion matrix would have values only on the diagonal. Off-axis terms in the rows indicate the confusion of the input with the different possible choices. Off-axis terms in the columns represent labels missed by the classifier.

The confusion matrix results for a test data set are nearly identical for networks trained using Method 1 and Method 4, as shown in Tables 7.1 and 7.3. However, if there were data belonging to a "new" class that is introduced within the measurement space, then the false-alarm rate for Method 1 would increase dramatically, whereas it would not increase nearly as much for the network trained through Method 4. Likewise, if the training data are not representative of the actual distributions seen in the field, but are covered by the decision regions formed by using Method 1, then there would be an increase in the number of correct decisions. Therefore, the designer must know the problem domain well in order to create the optimal classification system.

The point of this exercise, which the authors could not stress more strongly, is that the way the designer sets up the network training really does matter! In the

Table 7.1 Confusion matrix on a test set for a multiclass feedforward network classifier trained using Method 1.

Method 1		Call				Percent
	Threshold = 0.5	Class 1	Class 2	Class 3	Class 4	Identified
Truth	Class 1	237	11	0	5	93.31%
	Class 2	11	216	15	0	89.26%
	Class 3	0	12	233	2	94.33%
	Class 4	6	0	2	249	97.89%
Percent Correctly Classified		93.31%	90.38%	93.20%	97.89%	

Table 7.2 Confusion matrix on a test set for a multiclass feedforward network classifier trained using Method 4.

Method 4		Call				Percent
	Threshold = 0.5	Class 1	Class 2	Class 3	Class 4	Identified
Truth	Class 1	238	10	1	5	93.60%
	Class 2	8	218	16	0	90.08%
	Class 3	0	10	234	3	94.74%
	Class 4	13	0	1	243	94.55%
Percent Correctly Classified		91.89%	91.60%	92.86%	97.81%	

training examples presented in this section, the network topologies were identical: a single multiple-output feedforward neural network and a composite network consisting of multiple single-class feedforward networks. All the authors did was change how the data were presented to each of the networks, which resulted in drastically different decision regions. How the designer sets up the training data, along with the corresponding network topology, will definitely affect the results. This leads to some rules of thumb to use while training feedforward neural networks.

7.2 Rules of Thumb for Training Neural Networks

7.2.1 Foley's Rule

The ratio of the number of samples per class (S) to the number of features (N) should exceed 3 to obtain optimal performance ($S/N > 3$). Foley showed in his seminal paper [Foley, 1972] that when $S/N > 3$, the training-set error would approximate the test-set error and that the resultant error would be close to that of a Bayes optimal classifier.

7.2.2 Cover's Rule

Thomas Cover [Cover, 1965] showed that when the ratio of training samples to the total number of degrees of freedom for a two-class classifier is less than 2,

then the classifier will find a solution even if the classes are drawn from the same distribution. It could be said that if a classifier is given enough rope (that is, degrees of freedom), it will hang itself.

7.2.3 VC Dimension

The Vapnik-Chervonenkis (VC) dimension [Vapnik, 1995] is the size of the largest set S of training samples for which the system can partition all 2^S dichotomies on S. For single hidden-layer feedforward neural networks, the lower bound is the number of weights between the input and the hidden layer, while the upper bound is approximately twice the total number of weights in the network. In practice, neural-net designers often choose the total number of training samples to be 10 times as large as the VC dimension [Rogers, 1997].

7.2.4 The Number of Hidden Layers

Cybenko [Cybenko, 1989] demonstrated that a single hidden layer, given enough neurons, can form any mapping needed. In practice, two hidden layers are often used to speed up convergence. While some feedforward networks have been reported in the literature to contain as many as five or six hidden layers, the additional layers are not necessary. The important thing to remember is that the network learns best when the mapping is simplest, so use good features. A network designer should be able to solve almost any problem with one or two hidden layers.

7.2.5 Number of Hidden Neurons

For a small number of inputs (fewer than 5), approximately twice as many hidden neurons as there are network inputs are used. As the number of inputs increases, the ratio of hidden-layer neurons to inputs decreases. The number of hidden neurons, with their associated weights, should be minimized in order to keep the number of free variables small, decreasing the need for large training sets. Validation-set error (see Section 7.3.3) is often used to determine the optimal number of hidden neurons for a given classification problem.

7.2.6 Transfer Functions

Almost any transfer function can be used in a feedforward neural network. In order to use the backpropagation training method, the transfer function must be differentiable. The most common transfer functions are the logistic sigmoid, the hyperbolic tangent, the Gaussian, and the linear transfer function. In general, linear neurons require very small learning rates in order to train properly. Gaussian transfer functions are employed in radial basis function networks, often used to perform function approximation. For classification problems, nonlinear transfer functions work best.

7.3 Training and Testing

Once the network designer has defined the problem, chosen the neural network architecture, and collected, prepared, and labeled the data, he is ready to train the neural network. The training goal is to find the training parameters that result in the best performance, as judged by the neural network's performance with unfamiliar data. This determines how well the network will generalize. Generalization is a measure of how well the classifier performs on data samples with which it has never been presented, but that are within the acceptable limits of the input feature space. Generalization is used to determine whether the classifier is memorizing the input data. When a network has been overtrained, it will usually memorize the data in classification tasks or will overfit the data when used for estimation tasks. It is critical to use testing data that were not used to train the neural network, since the goal is to find the configuration with the best performance on independent data (e.g., data newly collected or unseen during training).

7.3.1 Split-Sample Testing

To find the optimum neural network configuration, an ideal approach is to randomly sample the population three times to produce three independent data sets: a training set, a validation set, and a test set as shown in Fig. 7.10. This is known as independent-sample testing. However, usually the network designer is given a single data set from which the population has already been sampled. When this situation occurs, a common approach to divide the available data into three disjoint sets, through the use of random selection, is shown in Fig. 7.11. This is known as split-sample testing and is outlined in Table 7.3. Unfortunately, the use of the terms "test set" and "validation set" are reversed between the statistics and machine-learning communities. This discussion will use the definitions common to the statistics community. In statistics terminology [Bishop, 1995], the training set is used to fit models, the validation set is used to estimate prediction error for

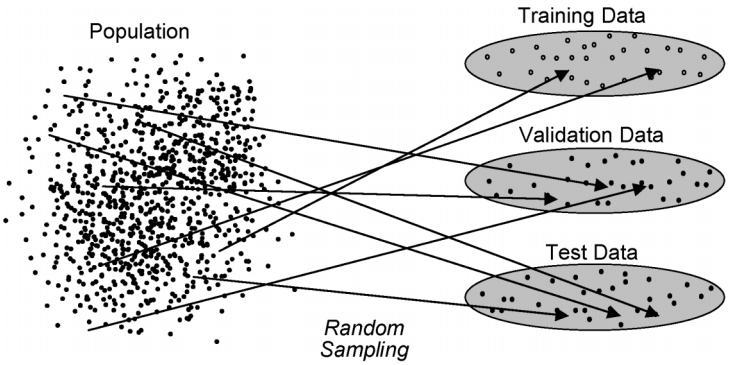


Figure 7.10 Randomly sampling population to form three independent data sets for independent-sample testing.

model selection, and the test set is used to determine the generalization error of the final chosen model.

The training set is a set of samples used to adjust or train the weights in the neural network to produce the desired outcome. The validation set (sometimes called the test set in machine-learning vocabulary) is a set of samples used to find the best neural-network configuration and training parameters. For example, it can be employed to monitor the network error during training to determine the optimal number of training iterations or epochs. It can also be used to determine the optimal number of hidden neurons. The validation set is used to choose between multiple trained networks. When the validation set is used to stop training, the neural network is optimistically biased, having been exposed to the data.

The test set (sometimes called the validation set in machine-learning vocabulary) is a set of samples used only to evaluate the fully trained neural network. Often, it is collected separately from the training and validation sets to help ensure

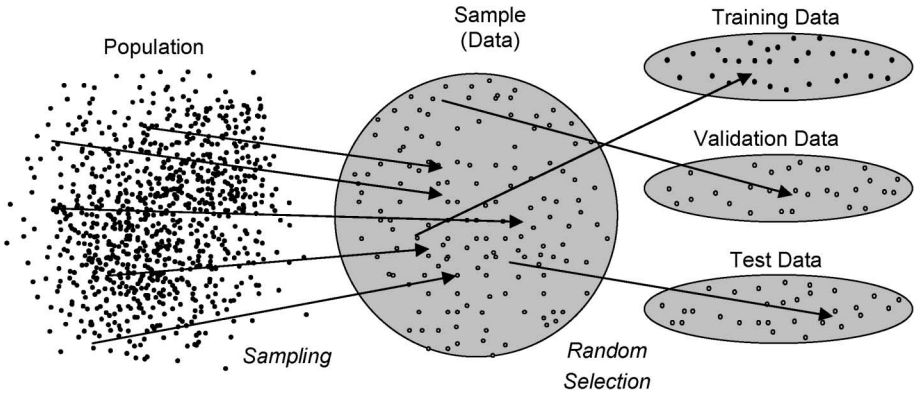


Figure 7.11 Random selection of data from a previously sampled population to form three independent data sets for split-sample testing.

Table 7.3 Procedure for Training and Testing Supervised Neural Networks

Step 1a.	Randomly sample the population in three sessions to form three independent data sets: training set, validation set, and test set as shown in Fig. 7.10 (i.e., independent-sample testing).
Step 1b.	If you only have access to the sampled data, then divide the available data into training, validation, and test sets through random selection as shown in Fig. 7.11 (i.e., split-sample testing).
Step 2.	Choose a neural network, configure its architecture, and set its training parameters.
Step 3.	Train with the training-set data and monitor with the validation set.
Step 4.	Evaluate the neural network by using the validation-set data.
Step 5.	Repeat Steps 2 through 4 with different architectures and training parameters.
Step 7.	Select the best network by identifying the smallest error found with the validation set.
Step 7.	Train the chosen best network with data from the training set, while monitoring with the validation set.
Step 8.	Assess this best neural network by using the test-set data and report only its performance.

independence. The neural network is biased toward both the training and validation sets, so the independent test set must be used to determine generalization error. The test set should never be used to choose between neural networks, so that it remains an unbiased estimate of the network's generalization error [Ripley, 1996].

7.3.2 Use of Validation Error to Stop Training

During supervised learning, the network's output error is monitored. It should decrease monotonically if a gradient-descent or Newton method is used. In addition, the validation set error is also often monitored to determine the optimum point to stop training. Normally, the error on the validation set will also decrease during the initial training phase. However, when the network begins to overfit the data, the output error produced by the validation set will typically begin to rise, as shown in Fig. 7.12. When the validation error increases for an appreciable number of iterations to indicate that the trend is rising, the training is halted, and the weights that were generated at the minimum validation error are used in the network for operation. This approach should give the best generalization. The weights generated beyond this point are more likely to fit the idiosyncrasies of the training data, and will not interpolate or generalize well.

7.3.3 Use of Validation Error to Select Number of Hidden Neurons

In Section 7.2.4, we discussed rules of thumb for estimating the number of hidden neurons necessary to solve the problem. An empirical approach is to retrain the network with varying numbers of hidden neurons and observe the output error as a function of the number of hidden neurons. Figure 7.13 shows the result of one network repetitively trained with different numbers of hidden neurons.

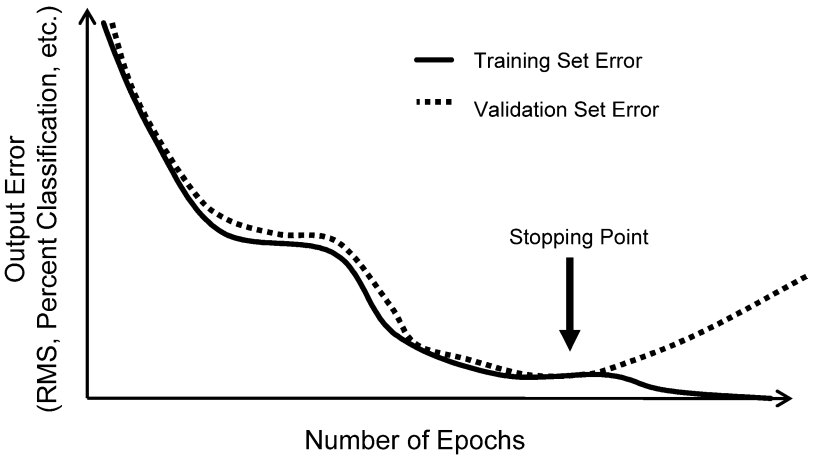


Figure 7.12 Illustration of training set and validation set error as a function of epoch. As training progresses, both errors will drop until a point is reached that validation set error begins to rise. This is the point at which training should cease.

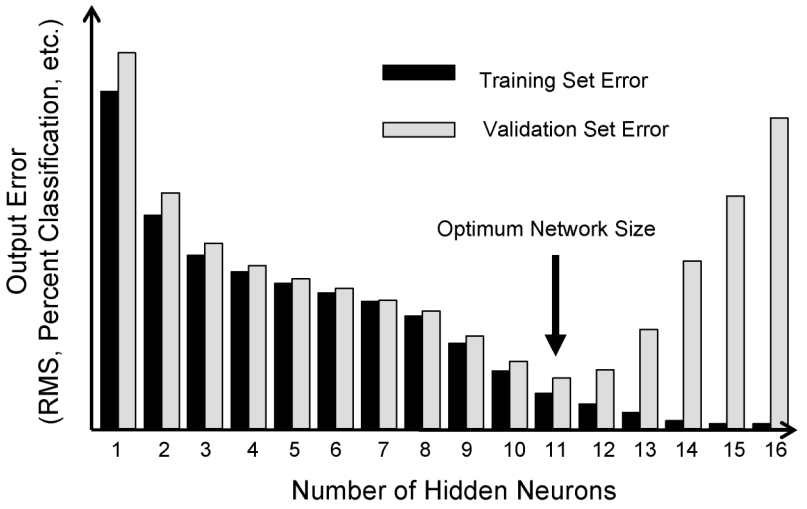


Figure 7.13 Illustration of training set and validation set error as a function of the number of hidden neurons. The optimum number of hidden neurons is determined by finding the lowest validation error as a function of the number of hidden neurons.

Chapter 8

Unsupervised Training Methods

The unsupervised training model (see Fig. 8.1) is similar to the supervised model, but differs in that no teacher is employed in the training process. It is analogous to students learning the lesson on their own. Two of the most popular unsupervised learning techniques used in the neural-network community are the self-organizing map (SOM), developed by Teuvo Kohonen, and the adaptive resonance theory (ART) network, developed by Stephen Grossberg and Gail Carpenter. The unsupervised training model consists of the environment, represented by a measurement vector. The measurement vector is fed to the learning system and the system response is obtained. Based upon the system response and the adaptation rule employed, the learning-system weights are adjusted to obtain the desired performance. The learning process is an open loop with a set of adaptation rules that govern general behavior such as a neighborhood and learning rate in the case of the Kohonen SOM [Kohonen, 1982] and the vigilance parameter in the case of the ART network [Carpenter, 1987].

8.1 Self-Organizing Maps (SOMs)

The function of a self-organizing map (SOM) neural network is to map the relationship among the input patterns to a reduced relationship of the output patterns,

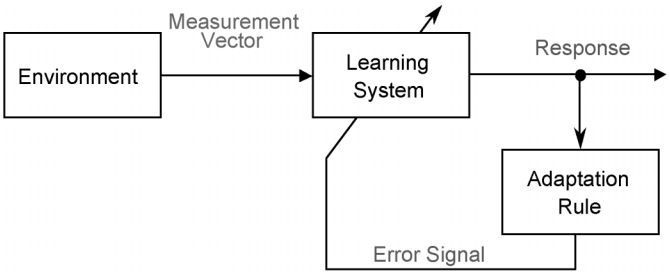


Figure 8.1 Unsupervised training model.

while preserving the general topological relationships. In simple terms, the SOM places things that are similar in the input data into new groupings of reduced dimension that retain the spatial relationship between similar items. An example is depicted in Fig. 8.2 below. Unlike the feedforward neural networks the reader has been introduced to, in the SOM, the weights leading to each neuron are trained to place items with similar characteristics, such as cars or planes, together, yet provide separation for items that are not alike. SOMs are unique in that they construct topology-preserving mappings of the training data where the location of a neuron encodes semantic information. One of the main applications for SOMs is clustering data for display as a two-dimensional image so that the data are easy to visualize. However, the SOM is not limited to a two-dimensional output space, nor only to visualization tasks.

The principles used to construct SOM maps are not limited to two dimensions and can be expanded to any desired dimensionality. A simple Kohonen network is presented in Fig. 8.3. As can be seen, every input neuron is connected to every node in the map. Also, the map is an ordered array of neurons. In the example, the map is a rectangle. However, the map can be a line, a circle, a hexagonal structure, or a multi-dimensional structure of any desired shape. Once a map is constructed, it must be trained to group similar items together, a process called clustering, as shown in Fig. 8.2.

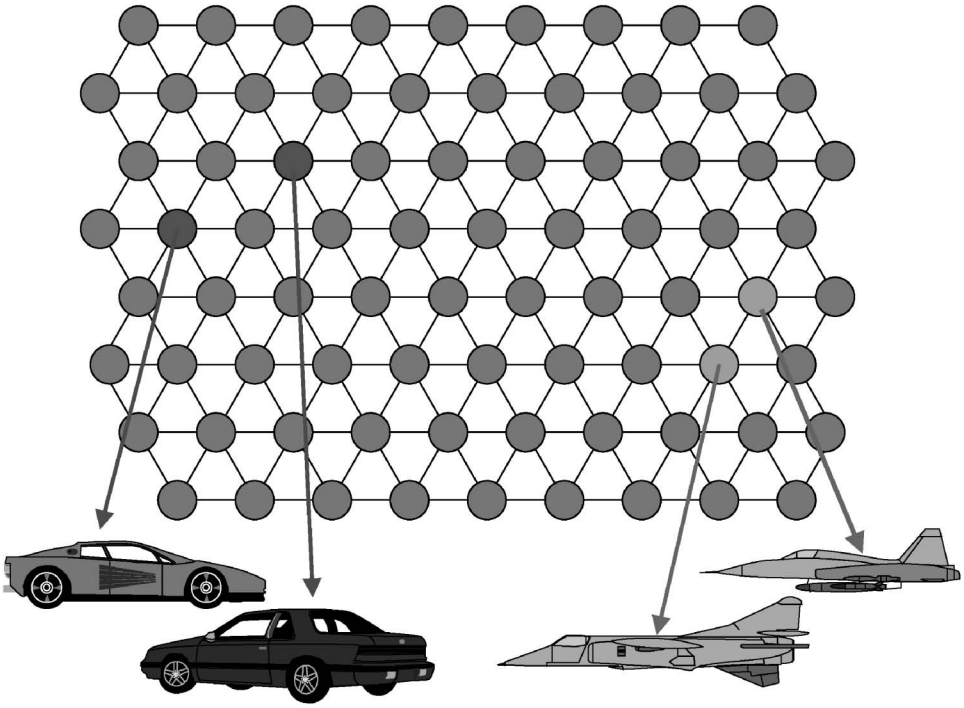


Figure 8.2 Self-organizing map with clusters of similar objects.

8.1.1 SOM Training

The SOM is trained using a combination of neighborhood size, neighborhood update parameters, and a weight-change parameter. The SOM is formed initially with random weights between the input layer neurons and each of the neurons in the SOM. Each neuron in the input layer is connected to every neuron in the SOM. A neighborhood is the region around a given neuron that will be eligible to have the weights adapted, as shown in Fig. 8.4. Neurons outside the region defined by the neighborhood do not undergo any weight adjustment. As the training is performed, the neighborhood size is adjusted downward until it surrounds a single neuron. The use of a neighborhood that reduces in size over time allows the SOM to group similar items together. The neighborhood decay parameter, α, controls the shrinkage of the neighborhood; and the weight-update parameter, η, controls how far each weight is adjusted toward the desired value.

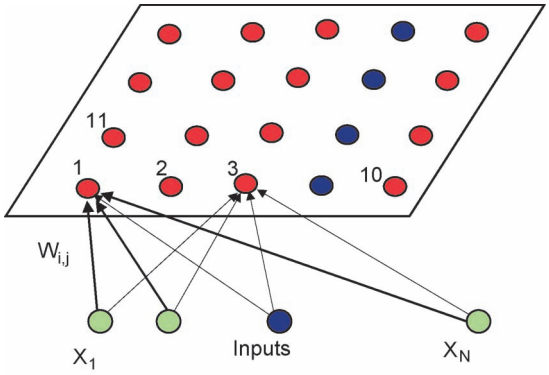


Figure 8.3 Simple Kohonen self-organizing map.

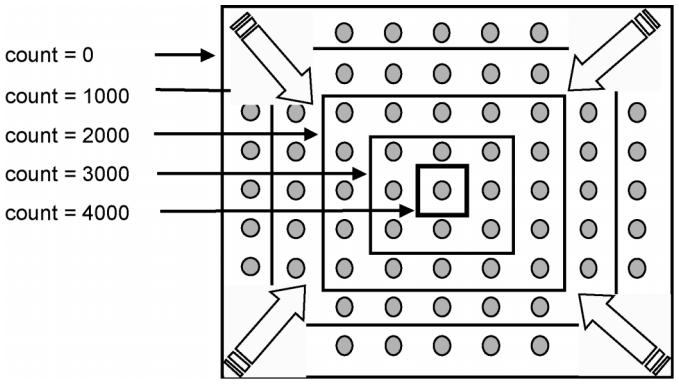


Figure 8.4 SOM neighborhood as a function of count. At count = 0, the entire network of SOM neurons is adjusted. As training continues, the neighborhood is reduced until at the end it is operating on a single neuron.

Once the initial network weights and training parameters are set, the training begins with the introduction of an input feature vector. The distance, usually Euclidean distance, between the input vector and the corresponding weight vector for each neuron in the SOM is measured. The neuron closest in distance to the input is declared the winner (see Fig. 8.5). Once the winner is found, all neurons in the neighborhood are updated to be similar to the input feature vector, according to the update formula as given in the SOM training process shown in Table 8.1.

The weight update for each weight vector in the neighborhood of the winning neuron is depicted in Fig. 8.6. As can be seen in the figure, the weight vector emanating from the input layer to the neuron is moved in the direction of the input vector. Thus, in a simple description, the neuron is being trained to be a "grand-

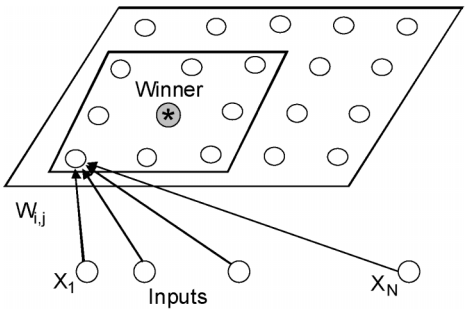


Figure 8.5 SOM with winning neuron and corresponding neighborhood.

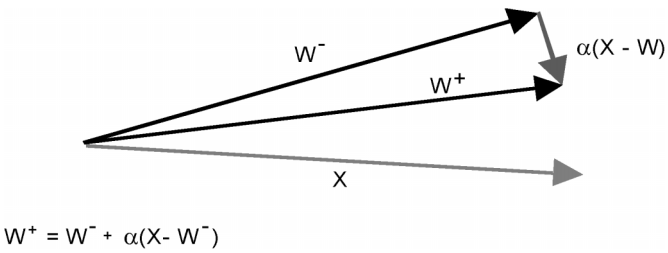


Figure 8.6 Weight update for SOM network in direction of input vector.

Table 8.1 Kohonen SOM Training Process

Step 1:	Choose number of neurons, total iterations (loops), η, α, etc.
Step 2:	Initialize weights
Step 3:	Loop until total iterations meet
Step 4:	Get data example and find winner. Winner is node closest to input vector by a predetermined distance metric (L1, L2, L3, etc.)
Step 5:	Update winner and all neurons in the neighborhood $$w^+ = w^- + \alpha(x - w^-)$$
Step 6:	Update neighborhood size and lower η, α, as needed
Step 7:	End loop

mother cell." That is, the neuron is being trained to respond to a specific input feature vector, such as a picture of your grandmother. Thus, when the neuron is presented with a picture of your grandmother, the neuron fires.

8.1.2 An Example Problem Solution Using the SOM

The SOM can be used to perform a variety of mappings. In the example given in Fig. 8.7, a three-dimensional cube is compressed into a two-dimensional object. By observing Fig. 8.7, you will soon notice that the portions of the cube that lie close together in three-dimensional space are also close together in two-dimensional space, preserving much of the topology in the original cube. The maps given in Fig. 8.7 are produced by polling each of the nodes in the SOM and coloring the result to match the closest class in the original three-dimensional cube. Thus, eight distinct portions exist in the cube and eight corresponding areas in the SOM. The SOM's ability to preserve the topological spacing of items makes it a valuable tool for many pattern-recognition problems.

The reader may be wondering at this point why the SOM would be so valuable. In real-world problems that pattern-recognition experts encounter, a great deal of data are often available, but the majority are not "truthed." In other words, the data are available but the knowledge of what classes they belong to is missing. If a developer has enough time and money, he or she can hire one or more experts to

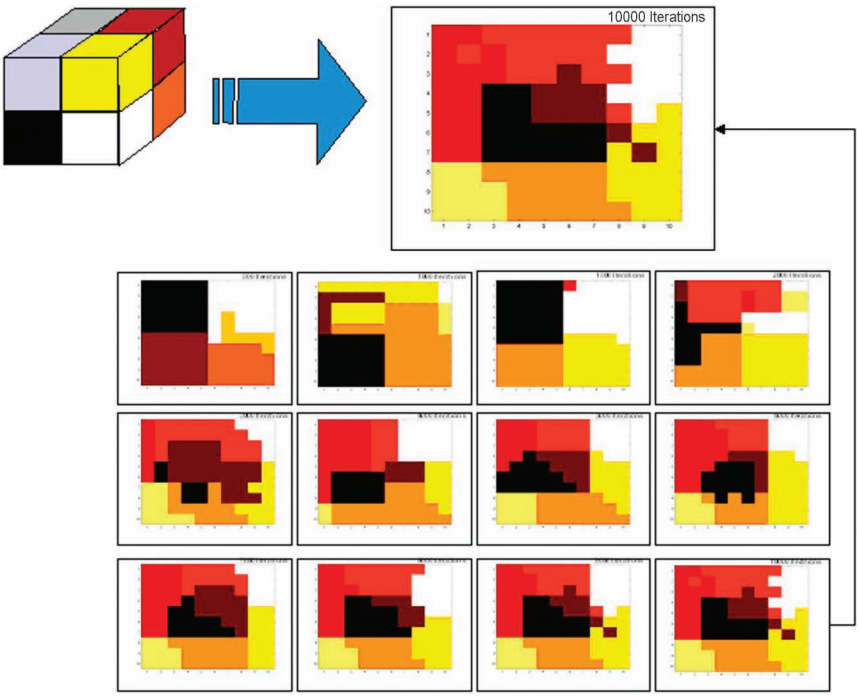


Figure 8.7 Compressing a cube: SOM weights for various network-training iterations.

grade each feature vector and tell him/her what the desired system response should be. This is optimal in one sense, but impractical because experts are expensive, and usually too much data exists to truth by hand. Another method, often incorporated to overcome this difficulty, is to have an expert "truth" a portion of the feature vectors for known conditions, and then to hold these data as a reference set. The designer then uses an unsupervised method, such as the SOM, to cluster the data. Because the SOM preserves topology, we know that points in the map that are close together represent feature vectors that are close together in the feature space of the measurements. Thus, a SOM is often formed using the "untruthed" data. The designer then uses the set of known inputs and corresponding classes and the SOM's topology-preserving feature to identify which neurons lie in which class. Then a supervised neural network maps the SOM nodes to the desired network output. We will illustrate this process using the Fisher iris data set.

As you may recall from the principal components discussion, the Fisher iris data [Fisher, 1936] consist of 150 feature vectors (50 per class) that utilize four features (sepal length, sepal width, petal length, petal width) to discriminate among three classes of iris (setosa, virginica, versicolor). The data points for the three classes are shown in Fig. 8.8. As can be seen, the *setosa* class is separable, but the *virginica* and *versicolor* varieties are not separable using the four features measured by Fisher. A scatter plot for each of the iris classes in Fisher's iris data set is given in Fig. 8.9. The scatter plot also shows that the classes are not separable. What if the reader merely had the set of 150 feature vectors and was asked to discover what he could about the data? Could he determine how many classes existed? How close would the results be to the "known" classes? The authors will now show how it can be done using the SOM.

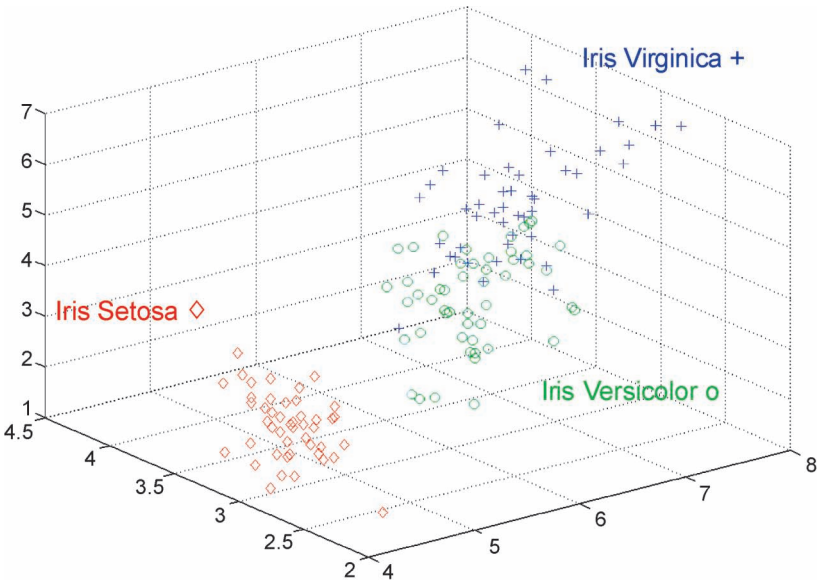


Figure 8.8 Depiction of Fisher's iris data plotted against the first three features.

The authors start by training a SOM for 1000 iterations, about seven passes through the entire training set. The resultant SOM map is given in Fig. 8.10, which shows that the map contains three distinct regions. Examining the map, perhaps the reader can define one class as those pixels that are shades of blue but not cyan, another class as those pixels that are mostly cyan to almost yellowish-green, and

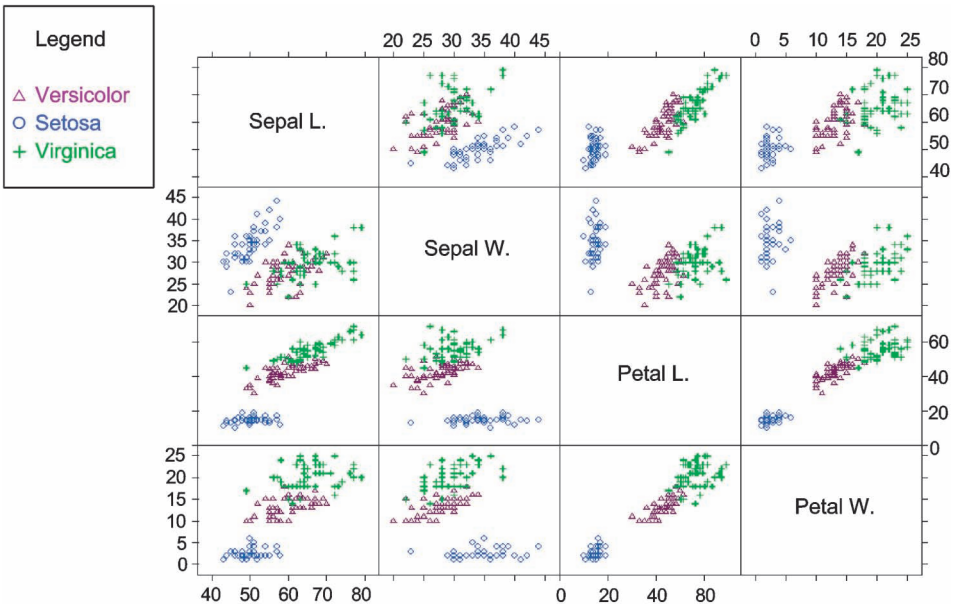


Figure 8.9 Scatter plot for Fisher's iris data.

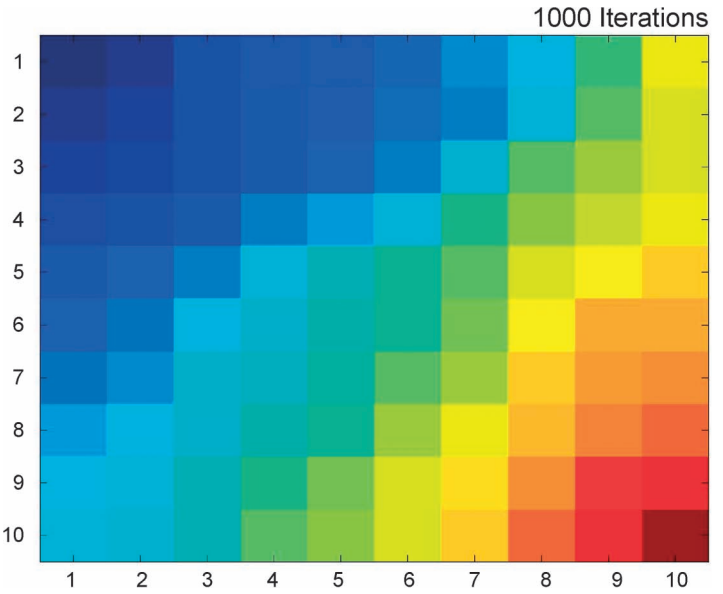


Figure 8.10 SOM output after training on Fisher's iris data set.

the last class yellowish-green to red, as shown in Fig. 8.11. The actual results found using the Fisher iris data set are given in Fig. 8.12. As can be seen in the figure, the initial boundaries were pretty good, but need to be adjusted slightly based upon the "truthed" data. Thus, with no knowledge about how many classes were in the iris set, the SOM net could be used discern three classes and approximately where their boundaries would lie.

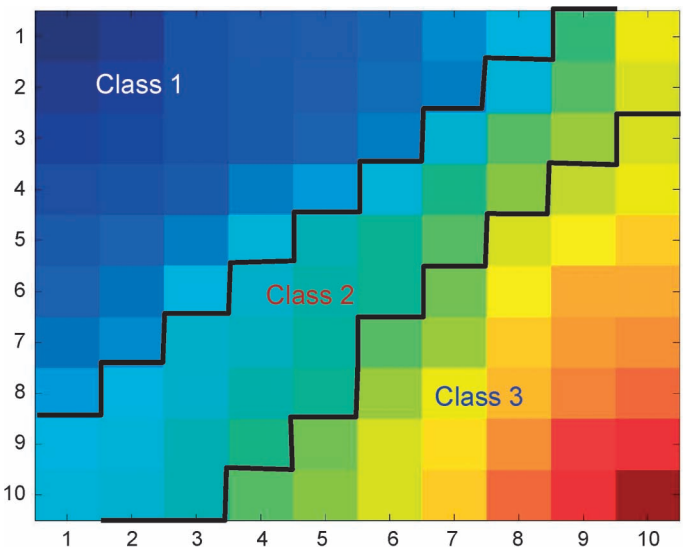


Figure 8.11 Potential class boundaries obtained using SOM output after training on Fisher's iris data set.

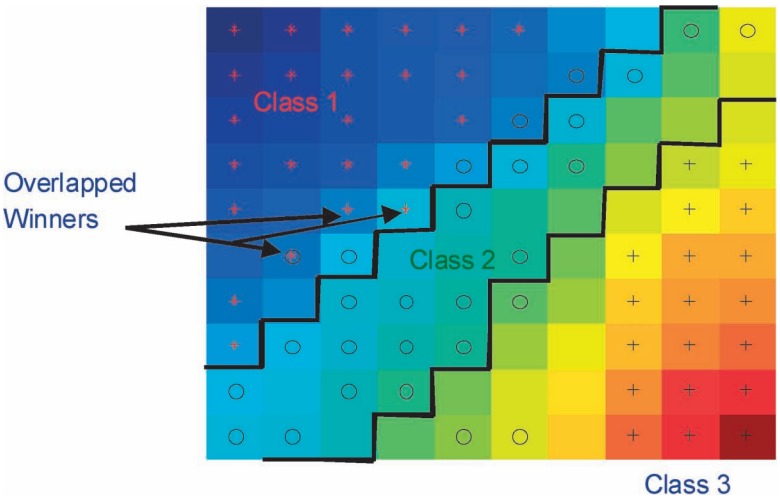


Figure 8.12 Class boundaries obtained using SOM output after training on Fisher's iris data set, along with winning neurons and corresponding iris data.

8.2 Adaptive Resonance Theory Network

The adaptive resonance theory (ART) network was developed by Stephen Grossberg in the 1970s. The first ART network, ART1, performed unsupervised learning for binary input patterns [Grossberg, 1976a, b, c; Grossberg, 1987]. The ART2 network was developed to handle both continuous and binary input patterns [Carpenter, 1987b]. The ART3 network performs parallel searches of distributed recognition codes in a multilevel network hierarchy [Carpenter, 1990]. The FuzzyART network is almost exactly the same as ART1, except that it can also handle continuous input patterns [Carpenter, 1991]. The training rules for FuzzyART are basically the same as those for ART1, except that the intersection operator of ART1 is replaced with a minimum operator working on the fuzzy membership functions. Gaussian ART uses Gaussian classifiers as the discriminates, implemented as radial basis functions [Williamson, 1996].

The authors present a simplified version of the ART network that works equally well on binary and floating-point data. The essence of the ART network is that the network begins with a memory composed of neurons with unassigned weights that, during the course of training, become assigned to cluster centers based upon the input data. Each neuron is assigned a vigilance parameter that controls the size of the hypersphere. This can be considered to be a spherical neighborhood in a high-dimensional feature space, which defines the membership of an input vector to the cluster center represented by the weights of the assigned neuron. As new patterns are presented to the ART network, with separation larger than the vigilance, additional neurons are assigned with their centers at the location in feature space represented by the input. Figure 8.13 contains a simplified block diagram model of an ART network that contains a preprocessing layer that normalizes incoming feature vectors. Generally, the Euclidean energy norm is taken, which stretches or squashes incoming feature vectors onto the unit hypersphere, as depicted in Fig. 8.14.

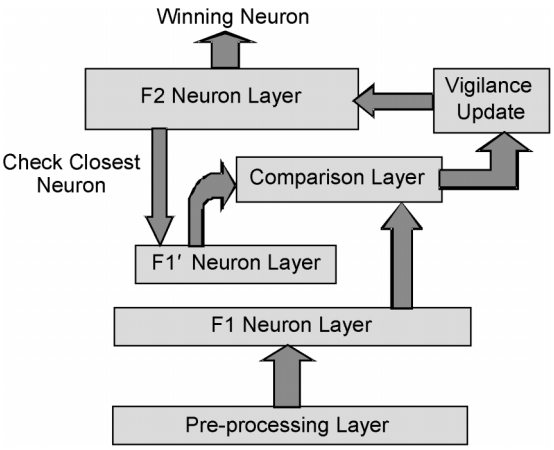


Figure 8.13 Block diagram of adaptive resonance theory (ART) network.

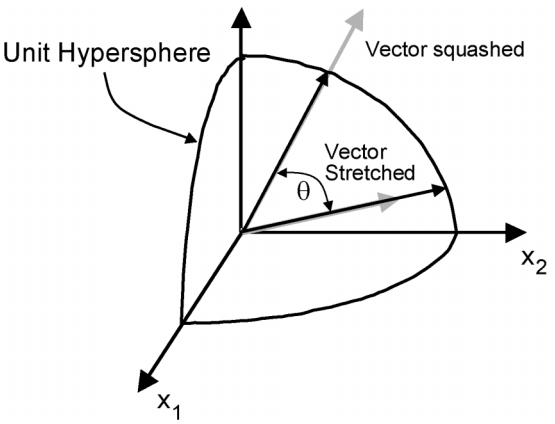


Figure 8.14 Energy normalization of input vectors onto unit hypersphere.

The angle between the vectors, θ, is preserved, allowing the vectors to be discriminated by the ART network. The normalized input is passed onto the F1 neuron layer and the network finds the closest match among the stored neurons on the F2 layer. The comparison layer calculates how similar the vectors are to one another ($-1 \leq S \leq 1$) using the dot product, and compares the magnitude of the result to the vigilance parameter. Thus, inputs that are exactly the same as an F2 neuron would have a similarity value of 1, and those that are antipodal (exactly the opposite) would have a value of -1. The larger the vigilance parameter, the easier it is to add neurons, because a match is harder to achieve.

If an input feature vector meets the vigilance parameter for a given stored neuron, then the neuron weights are adapted to move the center of the neuron to better capture the new input vector. Because of the unsupervised nature of the ART algorithm, the network designer must guard against allowing the neurons to rotate too far. To envision how this could be a problem, consider a series of input vectors spaced 30 degrees apart on the unit hypersphere for a network with a vigilance of 0.8. If, without accounting for prior history, the network blindly adapts the neuron weights, then after the fourth vector is presented, the neuron weights would be orthogonal to the original vector that created the neuron in the first place. Guarding against the ease by which vectors can be rotated is often accomplished by updating the weights by increasingly smaller amounts as new vectors that meet the vigilance condition are used to adapt the weights of the winning neuron. For example, if only one feature vector was used to create the weights of the winning neuron, then the next feature vector that meets the vigilance condition would affect the weights equally with the first. Then, succeeding input vectors that meet the vigilance condition would have one-third the impact, then one-fourth, one-fifth, one-sixth, and so on. In the limit, the stored neurons in the F2 layer would be the means of the cluster centers. This is borne out by comparing the ART-like algorithm's neuron weights shown in Table 8.2 to the means of the normalized input vectors for each of the Fisher iris classes given in Table 8.3. The results in Tables 8.2 and 8.3 are nearly identical.

Table 8.2 ART neuron weights for energy-normalized Fisher iris data.

	Feature 1	Feature 2	Feature 3	Feature 4
Node 1	0.8027	0.5467	0.2352	0.0389
Node 2	0.7494	0.3498	0.5368	0.1669
Node 3	0.7053	0.3192	0.5944	0.2175

Table 8.3 Feature means for each class using energy-normalized Fisher iris data.

	Feature 1	Feature 2	Feature 3	Feature 4
Mean 1	0.8022	0.5477	0.2346	0.0391
Mean 2	0.749	0.3495	0.5375	0.1673
Mean 3	0.7056	0.3185	0.5946	0.217

Chapter 9

Recurrent Neural Networks

Recurrent neural networks are networks that feed the outputs from neurons to other adjacent neurons, to themselves, or to neurons on preceding network layers. Two of the most popular recurrent neural networks are the Hopfield and the Bidirectional Associative Memory (BAM) networks.

9.1 Hopfield Neural Networks

The Hopfield network [Hopfield, 1982] rekindled interest in neural networks in the early 1980s, but it is rarely used today. The Hopfield network is often used as an auto-associative memory or content-associated network with fully recurrent connections between the input and output, as depicted in Fig. 9.1. Its primary purpose is to retrieve stored patterns by presenting a portion of the desired pattern to the network.

In the Hopfield neural network, each neuron is connected to every other neuron through weighted synaptic links. A set of P patterns with bits encoded as $M_{pi} \in \{-1, 1\}$ are encoded in the synaptic weights by using an outer-product learning rule [Hopfield, 1982]. The encoded synaptic weights are determined by

$$w_{ij} = M^T M - pI, \tag{9.1}$$

where p is the total number of training patterns and I is the identity matrix. Table 9.1 summarizes the steps involved in configuring a Hopfield network.

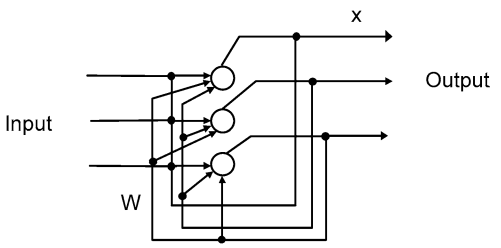


Figure 9.1 Hopfield Neural Network.

Table 9.1 Process of configuring a Hopfield neural network.

Step 1.	Encode each pattern as a binary vector.
Step 2.	Form an N (number of inputs) by P (number of patterns) dimensional matrix to store the pattern vectors. Each vector becomes a row in the matrix.
Step 3.	Encode the weights in the network through the use of an outer product of the pattern matrix and its transpose. Then zero out the diagonal terms. This is described mathematically in Eq. (9.1).

The output of the Hopfield network is given by

$$output = f\left(\sum_{j \neq i} w_{ij} input_i\right). \tag{9.2}$$

In Hopfield's original work [Hopfield, 1982], the step function was used as the activation function for all the neurons in the network.

For the Hopfield network, a noisy or corrupted pattern is presented at the input, and the network iterates until it reaches a steady state or a limit cycle. A pattern is recalled when stability is reached. In general, if the input pattern is some corrupted or noisy version of a pattern stored in the network, then the recalled pattern should be the ideal or uncorrupted version of this input pattern. Figure 9.2 illustrates the process of pattern recall. This specific example was generated on an opto-electronic implementation of the Hopfield network with interconnect weights stored holographically [Keller, 1991].

In the Hopfield representation, the patterns stored in the associative memory are distributed throughout the network. This distribution of patterns provides a large degree of fault tolerance and allows the network to function even with several bad synaptic connections. Unfortunately, this redundancy is costly, and Hopfield-style networks have a low storage capacity and operate in a forced-decision fashion. That is, the Hopfield network will try to converge on the closest memory state whether the input is valid or merely random noise. The maximum number of patterns that can be stored in a Hopfield auto-associative memory has been shown to be

$$P_{\max} = \frac{N}{2\ln(N)} \tag{9.3}$$

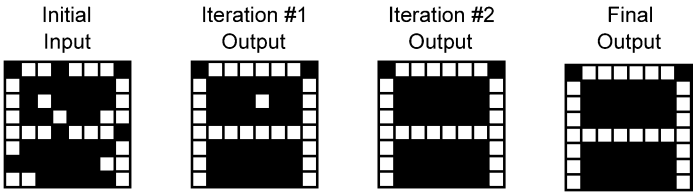


Figure 9.2 Illustration of the recall process of a Hopfield network.

for a network of N neurons [McEliece, 1987]. Several modifications have been made to improve this network. In Hopfield's formulation, the self-connection weights, w_{ii}, are set equal to zero. However, better performance is achieved when the self-connection weights are included [Gindi, 1988]. Also, better memory-recall performance can be achieved by using unipolar quantities and an optimized threshold or offset [Gmitro, 1989].

Hopfield networks have also been devised to solve optimization problems [Hopfield, 1985]. Here, the synaptic weights encode an energy cost function for the problem. An initial or test solution is presented to the network. The network iterates until it reaches stability. At that point, the network's output encodes the solution to the problem. One particular problem that has been encoded in a Hopfield network is the classic *NP*-complete traveling-salesman problem (*NP* = non-polynomial function). The computational complexity of most problems increases as a polynomial function of the number of elements in the problem. Functions that have polynomial complexity are known as *P*-complete problems (*P* = polynomial function) and are relatively easy to compute when compared to *NP*-complete problems. On the other hand, the computational complexity of an *NP*-complete problem is not a polynomial function of the number of elements in the problem and typically increases exponentially with the number of elements [Johnson, 1985].

9.2 The Bidirectional Associative Memory (BAM)

The Hopfield neural network utilizes a symmetric weight matrix that requires the input and output to be the same size. It is an auto-associative neural network, which means the inputs and outputs map to the same state.

What if the network designer wanted a hetero-associative neural network? Could it be done using the outer-product scheme utilized for the Hopfield network? It happens that Kosko [Kosko, 1987] and his associates solved this problem in the 1980s and called the network the bidirectional associative memory, or BAM for short.

But suppose the designer had never heard of Kosko or Hopfield and was faced with this dilemma, what should he or she do? The first step is to ask what is really happening when data passes through the Hopfield neural network. The answer is that the operation of the network could be split into two phases: (1) a forward pass and (2) a backward pass, as depicted in Fig. 9.3, that continues until convergence is reached.

Given this snapshot of the Hopfield neural network, the reader can clearly see that at any given point in time the output and input are computed with a feedforward pass through a weight matrix. In the case of the Hopfield matrix, the weight matrix is symmetric, but there is no reason to assume that this symmetric matrix is the only weight matrix available. What if the output only had three neurons instead of five? What would the matrix W look like? In the pass from the input to the output, it would be a 5×3 matrix, while the pass from the output back to the input would

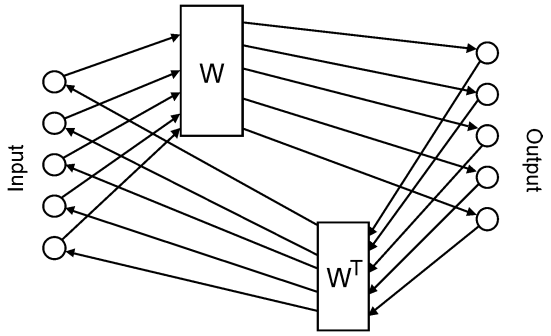


Figure 9.3 Depiction of Hopfield network at a given point in time.

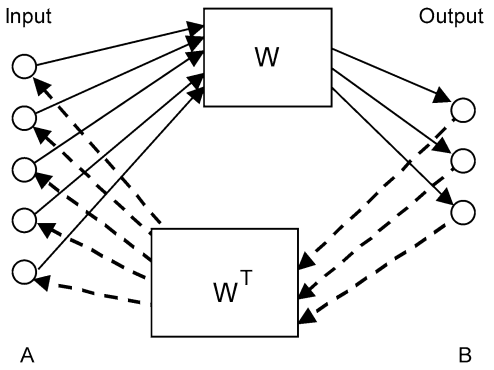


Figure 9.4 Bidirectional associative memory (BAM) neural network.

be a 3×5 matrix. Thinking in terms of a non-symmetric W, the authors finish the problem by showing how the weights needed for W are calculated.

Figure 9.4 illustrates the BAM network. All neurons are connected through feedback to all other neurons in the network. A set of P input patterns with bits encoded as $A_{pi} \in \{-1, 1\}$ and P output patterns with bits encoded as $B_{pi} \in \{-1, 1\}$ are encoded in the synaptic weights by using an outer-product learning rule. The encoded synaptic weights are determined by

$$W = A^T B, \tag{9.4}$$

where A is the set of P input patterns and B is the set of P output patterns.

The output of the BAM network is given by

$$output = f(W \cdot input). \tag{9.5}$$

The input of the BAM network is given by

$$input = f(W^T \cdot output). \tag{9.6}$$

Table 9.2 summarizes the steps involved in configuring the weights of a BAM network.

Thus, the reader can see that the BAM is a generalization of the Hopfield network in that an arbitrary number of input neurons can be associated with an arbitrary number of output neurons. This enables the neural-network designer to associate different sets of items. For example, he may wish to associate objects such as numbers, trucks, and airplanes with their corresponding name, as shown in Fig. 9.5 [Tarr, 1992].

Associating objects with labels is what made hetero-associative networks interesting from an academic point of view. But, just as occurred with the Hopfield neural network, the BAM fell out of use because there was still a problem with the number of items that could be stored before the network would converge to unwanted "spurious" states.

The authors now introduce a use for the Hopfield neural network that works to its strength. Suppose the designer had a set of equations he wanted to solve, but had finite resources to calculate the answer? In other words, he cannot compute the inverse directly because the matrix needed for the inversion is not of sufficient rank or has a bad condition number, or he has very limited computing resources.

Table 9.2 Process of configuring a BAM neural network.

Step 1.	Encode each input pattern as a binary vector.
Step 2.	Form an N (number of inputs) by P (number of patterns) dimensional matrix to store the input pattern vectors. Each vector becomes a row in the matrix, **A**.
Step 3.	Encode each output pattern as a binary vector.
Step 4.	Form an M (number of outputs) by P (number of patterns) dimensional matrix to store the output pattern vectors. Each vector becomes a row in the matrix, **B**.
Step 5.	Encode the weights in the network through the use of an outer product of the transpose of the input pattern matrix, **A**, and the output pattern matrix, **B**. This is described mathematically in Eq. (9.4).

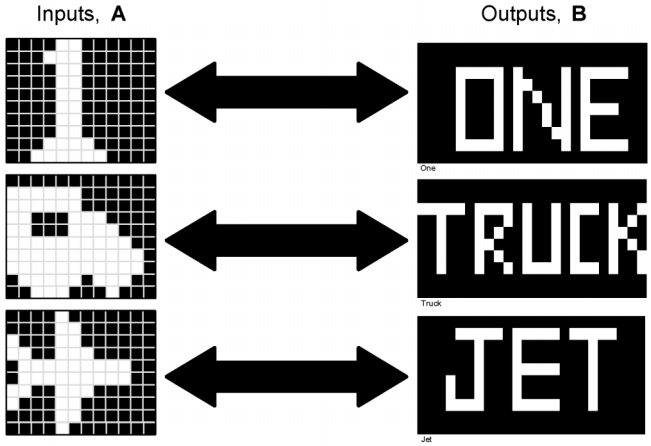


Figure 9.5 Example of the inputs and outputs encoded by a BAM network.

Could he still find a reasonable approximate solution? The next section describes just such a solution using the Hopfield neural network. The new implementation of the Hopfield network is called the generalized linear neural network (GLNN) [Lendaris, 1999].

9.3 The Generalized Linear Neural Network

The generalized linear neural network (GLNN) is an extension of the Hopfield neural network that uses linear activation functions. That is, the summed product is not squashed but used as is. This provides some interesting properties that enable the solution of systems of equations using a Hopfield neural network.

Considering the Hopfield neural network's properties, the reader would see that it multiplies the input by the weight matrix until it converges on a solution or reaches a limit cycle. If care is taken to insure that the network is sufficiently sized for the number of desired memory states, the network can be represented by an equation of the form:

$$x(n+1) = W \cdot x(n) + u, \tag{9.7}$$

where $x(n+1) \equiv$ next state, $W \equiv$ weight matrix, and $u \equiv$ starting input, which in the limit as $n \to \infty$ means that $x(n+1) \cong x(n)$. Taking that thought process further, the reader will find the following:

$$x \cong Wx + u, \tag{9.8}$$

$$x(I - W) \cong u, \tag{9.9}$$

$$x \cong (I - W)^{-1} \cdot u. \tag{9.10}$$

As you can see, the Hopfield network is actually computing an inverse of a matrix. By making the following substitutions:

$$u = \alpha A^T \cdot y, \quad 0 < \alpha < \frac{2}{trace(A^T A)},$$

$$W = I - \alpha \cdot A^T A,$$

and performing some mathematical manipulations, we find

$$x = (A^T A)^{-1} y, \tag{9.11}$$

the Moore-Penrose form of a matrix pseudo-inverse for an overdetermined set of equations. The GLNN for an overdetermined set of equations is shown in Fig. 9.6.

Likewise, when there are fewer equations than unknowns, by substituting:

$$u = y$$

$$W = I - \alpha \cdot AA^T, \quad 0 < \alpha < \frac{2}{trace(A \cdot A^T)},$$

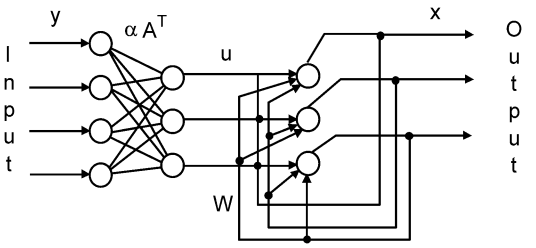


Figure 9.6 Block diagram of the generalized linear neural network for the overdetermined and full-rank cases.

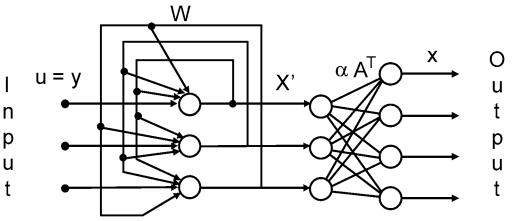


Figure 9.7 Block diagram of the generalized linear neural network for the underdetermined case.

and performing some mathematical manipulations, the reader will find

$$x = (AA^T)^{-1}y, \tag{9.12}$$

which is the Moore-Penrose form of a matrix pseudo-inverse for an underdetermined set of equations. The GLNN for an underdetermined set of equations is shown in Fig. 9.7.

By using the GLNN to solve a system of equations, the designer can avoid the need to take the classical inverse of a set of data. The GLNN allows the designer to compute the inverse on the fly.

9.3.1 GLNN Example

Given the system of inputs and outputs represented by

$$A = \begin{bmatrix} -4.207 & 1.410 & 0.451 & -0.910 \\ -0.344 & -3.473 & 2.380 & 3.267 \\ 3.733 & -1.999 & -3.728 & 2.85 \\ 1.096 & -4.277 & 1.538 & -3.952 \end{bmatrix} \quad y = \begin{bmatrix} 0.902 \\ -27.498 \\ -0.48 \\ 3.734 \end{bmatrix}$$

solve the equation $A \cdot x = y$ for $\mathbf{x}$.

Using the GLNN and iterating the Hopfield network for 100 iterations yields

$$x_{est} = \begin{bmatrix} 1.02 \\ 2.015 \\ -2.977 \\ -3.998 \end{bmatrix}, \quad \text{where} \quad x_{act} = \begin{bmatrix} 1 \\ 2 \\ -3 \\ -4 \end{bmatrix}.$$

Figure 9.8 depicts the error between the GLNN output and the actual value versus iterations for the first element of the **x** vector.

As can be seen in Fig. 9.8, the GLNN solves the set of linear equations in a small number of iterations by utilizing the Hopfield network embedded in the GLNN. The Matlab code needed to calculate a GLNN and its associated Hopfield network is given in Section C.2, with the code used to generate the example given in Section C.4.

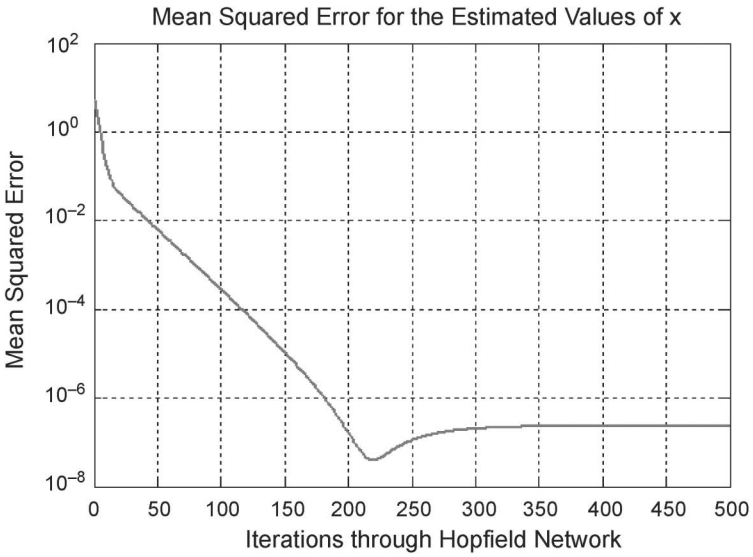


Figure 9.8 GLNN prediction error versus iterations for the example problem.

9.4 Real-Time Recurrent Network

The real-time recurrent neural network of Williams and Zipser [Williams, 1990] as shown in Fig. 9.9, has also found success in a number of applications. Williams and Zipser introduced time into the gradient-descent algorithm, which allows the network to capture time-varying dynamics.

9.5 Elman Recurrent Network

The recurrent neural network developed by Elman [Elman, 1991] uses a single recurrent pass of the hidden layer to store time-varying dynamics, as shown in

Fig. 9.10. Each neuron in the hidden layer is connected to one neuron in the recurrent layer. This copies the information output from the hidden layer and cycles it back through a weighting factor on the next sample presentation. This adds a delay loop to store values from the previous time step to be used in the current time step. This way, a small portion of information from the previous time, $t-1$, is combined with the current time, t. Indirectly, decreasing portions of information from time $t-2, t-3, t-4$, etc., are also captured, thus enabling recurrent operation in this network to model the temporal dynamics of the data. The Elman network has found success in a number of medical applications such as that shown in Section 10.3.

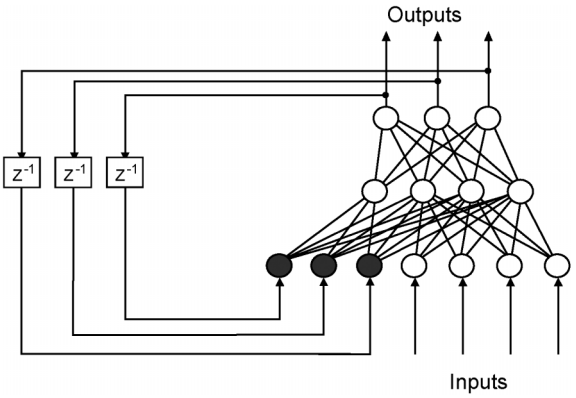


Figure 9.9 Williams and Zipser real-time recurrent neural network.

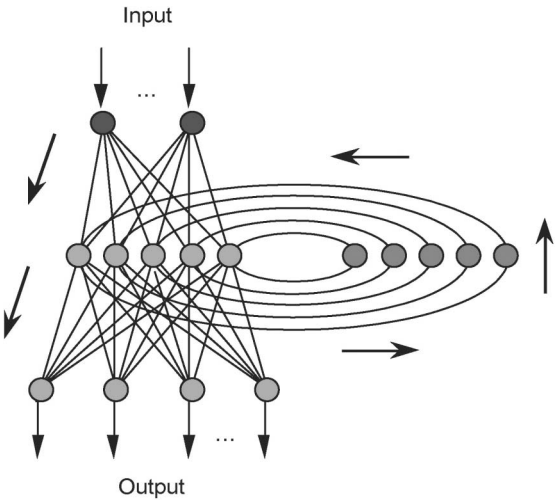


Figure 9.10 Elman neural network with hidden layer holding a single time-delay sample.

Chapter 10

A Plethora of Applications

Artificial neural networks have been used in a staggering number of applications. Not all of them can be listed here, but the authors will highlight areas where neural networks have found success. As has been pointed out, neural networks are very good at mapping inputs to outputs. This ability to map inputs to outputs is very useful in function approximation, pattern recognition, and database mining. The authors present examples showing how neural networks can be applied in these areas without becoming overly detailed. Several come from the authors' work experience, while others come from publicly available data sources.

10.1 Function Approximation

Function approximation can be useful in many applications. For example, the network designer could approximate a function $f(x)$ by using the model in Fig. 10.1, which shows that the desired input is presented to the learning system, with the function generator $f(x)$ acting as the expert. The network output is compared to the desired output ($\widetilde{f}(x)$). The error between the desired output, $f(x)$, and the actual output, $\widetilde{f}(x)$, is used to create an error term that adapts the behavior of the learning system. The authors will illustrate this process using a feedforward neural

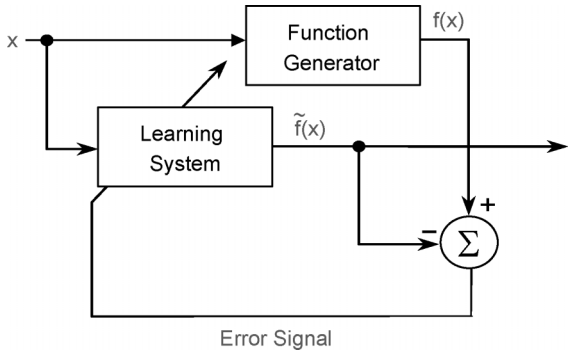


Figure 10.1 Block diagram of function-approximation training model.

network (see Fig. 10.2) to approximate the following function:

$$f(x) = 0.02\left(12 + 3x - 3.5x^2 + 7.2x^3\right)[1 + \cos(4\pi x)][1 + 0.8\sin(3\pi x)] \quad (10.1)$$

on the interval [0, 1]. The transfer functions can be sigmoids, Gaussians, hyperbolic tangents, or any smooth monotonic function. For this example, the authors used the sigmoid function.

The feedforward neural network, as shown in Fig. 10.2, is trained using pairs of data values consisting of the input and the corresponding desired function output. Once training is completed, over thousands of iterations and samples, the resultant neural network approximates the desired function. For the function defined in Eq. (10.1), the approximation by the feedforward network on the interval [0, 1] is very good, as shown in Fig. 10.3. Note that as the neural network is used to approximate the function outside the training interval, the prediction performance is not nearly as good as the region where it was trained.

Figure 10.3 shows that for the input points where the neural network was trained, the outputs are approximated very well. This is a strength of feedforward neural networks: that is, they are good at interpolation and approximation. Outside the regions where the network was trained, the approximation was much worse. This highlights a weakness of simple feedforward neural networks, which is that they do not perform extrapolation tasks well. The reader should also be careful concerning the order of the network used to approximate a function. By using a network with too few neurons, the fit to the function will be of a lower order than the ideal, as in the case of a straight line attempting to fit a series of points of much higher order (see Fig. 10.4). In addition, using too many neurons to fit a function will result in an overfit to the approximation, which results in wild errors for regions outside the points used for training. Thus, an ideal approximating network will be on the same order as the function to be approximated.

x	f(x)
0.0	0.480
…	…
0.2	0.084
…	…
0.4	0.181
…	…
0.6	0.195
…	…
0.8	0.107
…	…
1.0	0.748

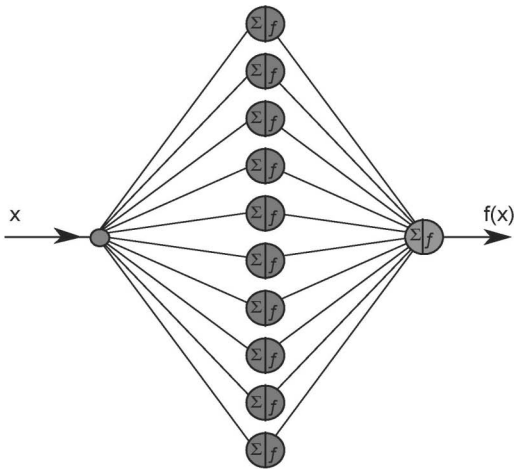


Figure 10.2 Feedforward neural network used to perform function approximation.

The reader may be wondering why one would use a neural network for function approximation when it can simply be computed. The answer is that the process shown in Fig. 10.1 is very general. It can be used to approximate any function [Cybenko, 1989]. Many times in engineering and industrial applications, data are available but a model of the process is desired. This is often termed system identification in control theory. The objective is to model an unknown plant using inputs to the plant along with the associated outputs from the plant to train a neural network that will be used to model the plant, as shown in Fig. 10.5. As the reader can readily see, the form of the method used to model the unknown plant, depicted in Fig. 10.5, is identical to the form of the function-approximation method shown earlier in Fig. 10.1. The supervised-training method is very powerful. The performance of control systems can be improved in other ways, but they are beyond the scope of this book.

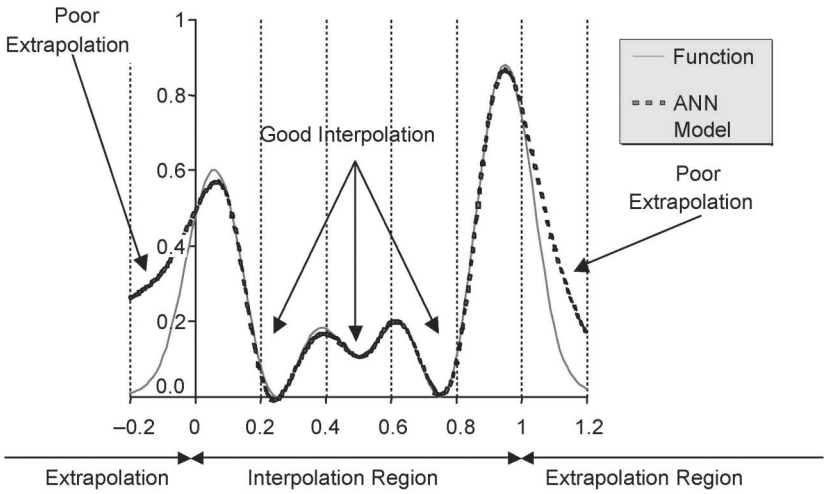


Figure 10.3 Depiction of the function-approximation capability for the feedforward neural network.

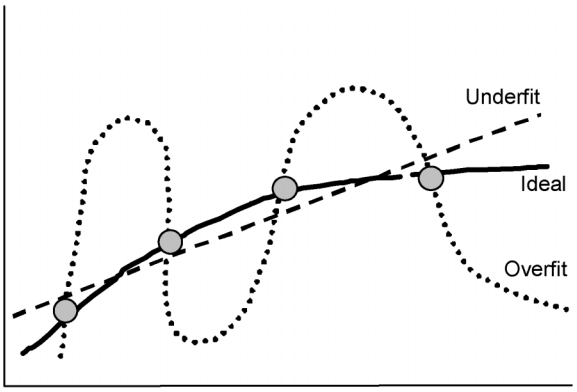


Figure 10.4 Examples of underfit, overfit, and ideal fit for a set of training points.

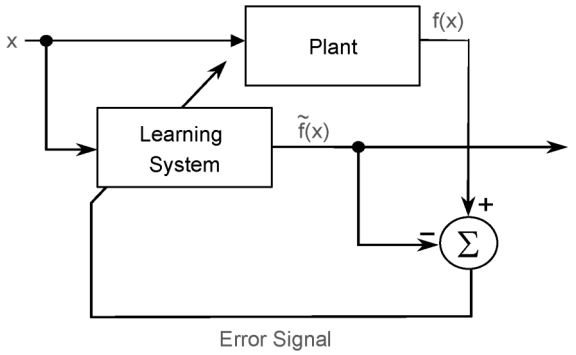


Figure 10.5 Conceptual diagram of a supervised-learning system used to perform system identification.

The reader can obtain additional references related to control applications for neural networks in the references section at the end of this book.

10.2 Function Approximation—Boston Housing Example

As a more extensive example of function approximation, the authors present a neural-network estimator that takes in housing price attributes for a neighborhood, then estimates the median value of owner-occupied homes in that neighborhood. These data come from David Harrison, Jr., and Daniel L. Rubinfeld, [Harrison, 1978] through the University of California-Irvine (UCI) Repository for Machine Learning databases [Blake, 1998]. The data consist of 506 samples of housing price attributes collected in the Boston area in 1978. The attributes include the crime rate (CRIM), the proportion of zoned large residential lots (ZN), the proportion of non-retail business land (INDUS), a binary index indicating whether the tract bounds the Charles River (CHAS), the nitric oxides concentration (NOX), the average number of rooms per dwelling (RM), the proportion of owner-occupied units built prior to 1940 (AGE), the weighted distances to five Boston employment centers (DIS), the index of accessibility to radial highways (RAD), the full-value property tax rate (TAX), the average pupil-to-teacher ratio in local schools (PTRATIO), an ethnic proportion factor (B), an economic-status factor (LSTAT), and the median value of owner-occupied homes at 1978 price levels. The median home value (MHV) is the quantity the neural-network estimator will be trained to predict. The authors wish to point out that any of the attributes could potentially be predicted based upon the other attributes. For example, the neural-network estimator could be used to predict TAX rather than the MHV.

The components of the neural-network price estimator are shown in Fig. 10.6. The 13 housing features are collected for a neighborhood. The 13 features are each scaled by statistical Z-score normalization and fed into a feedforward neural network. The mean and standard deviation are generated from the training set and will be stored as weights to normalize the inputs from the test and validation sets. The neural network's output has a value from 0 to 1, so it must be rescaled to the

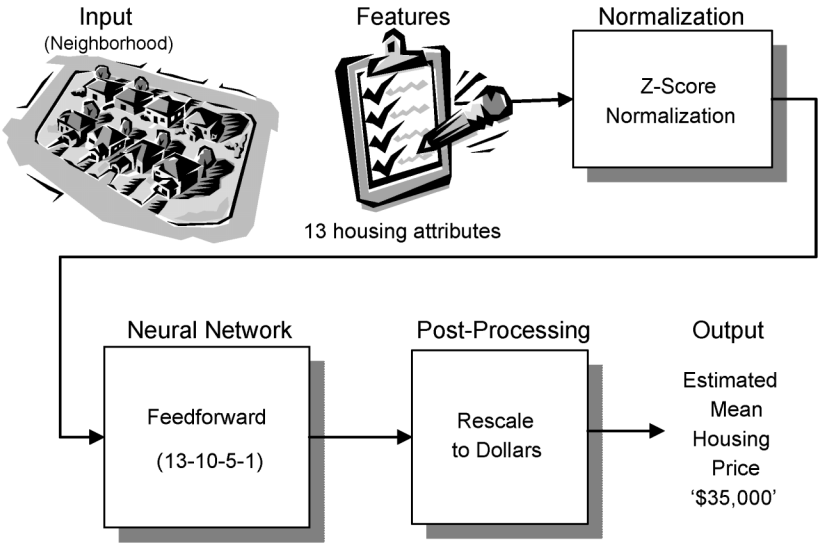


Figure 10.6 Block diagram of the Boston housing-price estimator illustrating the inputs (13 housing attributes), outputs (estimated mean housing price), and major components.

range of housing prices in Boston during 1978, which was $5,000 to $50,000 for the relevant neighborhoods.

The neural network consists of a four-layer feedforward network with 13 inputs, two hidden layers with 10 and 5 neurons respectively, and one output neuron as shown in Fig. 10.7. In general, a single hidden layer can solve most estimation problems. We used two hidden layers here to improve the training time. The number of hidden neurons (10 in the first layer and 5 in the second) was determined through a search process performed by the training software. To develop and test this network, the authors use split-sample testing and break the 506 samples, representing 506 neighborhoods, into a training set of 253 neighborhoods, a validation set of 76 neighborhoods, and a test set of 177 neighborhoods. For this application, the standard backpropagation training algorithm was used to adapt the weights of the feedforward network. The test results for the trained network are given in Table 10.1. An overall summary of this application and the associated network training parameters is given in Table 10.2.

10.3 Function Approximation—Cardiopulmonary Modeling

The next application demonstrates function approximation for a time-varying parameter. It involves a medical system that was developed in 1995 for evaluating graded exercise stress tests for use in cardiopulmonary evaluation [Keller, 1995]. The system is described in U.S. Patent 5,680,866 [Kangas, 1997]. A physician collected a variety of static and dynamic parameters from individuals during graded exercise stress tests, which involved the patient running on a treadmill or pedaling

a stationary bike while several key physiological parameters were recorded to determine how well the patient's heart and lungs respond to physical activity. A photograph of a bicycle graded exercise test setup is shown in Fig. 10.8. The physician felt a bit overwhelmed by the amount of data and was interested in developing a diagnostic aid to help him grade the exercise tests and compare the results to other tests, including previous tests the patient had undergone. The authors decided to build a cardiopulmonary modeler as part of the system to capture the relationship between several measured physiological variables and the workload on the patient.

The role of the cardiopulmonary modeler is to take key measured physiological parameters and predict the metabolic equivalent workload for a patient. This predicted value is then compared to the actual workload to determine whether the

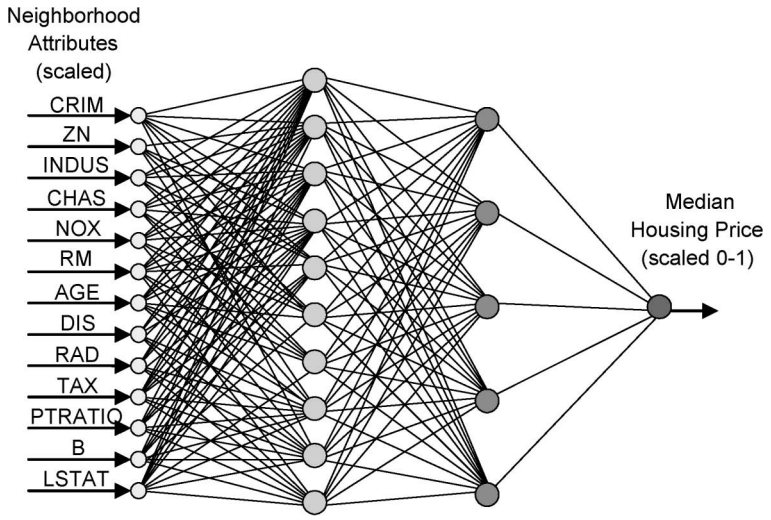


Figure 10.7 Neural network used in Boston housing-price example with 13 inputs from neighborhood attributes, two hidden layers, and one output that represents the median housing price for the neighborhood.

Table 10.1 Results of the Boston housing-price estimator test with a few samples from the test set.

Neighborhood ID	Town	Tract ID	Actual value	Predicted value	Error ($)	Error (%)
71	Burlington	3321	$24,200	$24,000	$200	−0.8%
269	Brookline	4012	$43,500	$44,100	$600	1.4%
3	Swampscott	2022	$34,700	$38,300	$3,600	10.4%
333	Holbrook	4212	$19,400	$20,300	$900	4.6%
4	Marblehead	2031	$33,400	$34,900	$1,500	4.5%
79	Woburn	3335	$21,200	$20,800	$400	−1.9%
210	Waltham	3685	$20,000	$20,500	$500	2.5%
218	Watertown	3702	$28,700	$26,100	$2,600	−9.1%
62	Beverly	2174	$16,000	$15,700	$300	−1.9%
44	Peabody	2103	$24,700	$24,200	$500	−2.0%
343	Hull	5001	$16,500	$15,900	$600	−3.6%

patient is responding as expected. A deviation in response could indicate a problem with the cardiopulmonary system. The overall diagnostic aid included two neural networks for the modeling, one for the static and one for the dynamic variables, and a neural-network model-based reasoning system for diagnostics to compare the modeled to actual outputs. Only the subsystem that performs dynamic modeling will be discussed here. A block diagram of this subsystem is illustrated in Fig. 10.9, with a variety of physiological parameters recorded: oxygen consumption, carbon dioxide production, expired ventilation of oxygen, expired ventilation of carbon dioxide, blood-oxygen saturation, respiratory rate (RR), systolic blood pressure (SBP), diastolic blood pressure (DBP), heart rate (HR), and pulse rate, along with several derived parameters. Workload, ambient temperature, and time were also recorded.

Figure 10.8 A photograph of a graded exercise bicycle test that was used to collect data for this application. This is Dr. Paul A. Allen and his two assistants.

Table 10.2 Summary of the Boston housing-price example.

Application	Predicting mean home price for a neighborhood
Data Model	13 inputs, 1 output, static
Learning	Supervised
Input Features	13 attributes taken from neighborhood assessments
Output	1-Predicted mean home price within neighborhood
Data Samples	506 neighborhoods in suburban Boston
Data Source	David Harrison, Jr. and Daniel L. Rubinfeld
Testing Method	Split-sample testing
Training Set	253 neighborhoods (50%)
Validation Set	76 neighborhoods (15%)
Test Set	177 neighborhoods (35%)
Normalization	Z-score normalization
Neural Network	Feedforward (13-10-5-1)
Training	Backpropagation
Post-processing	Rescale from unitary scale to dollars
Test Result	Root mean squared (RMS) error = \$4,120
	$R^2 = 0.800$

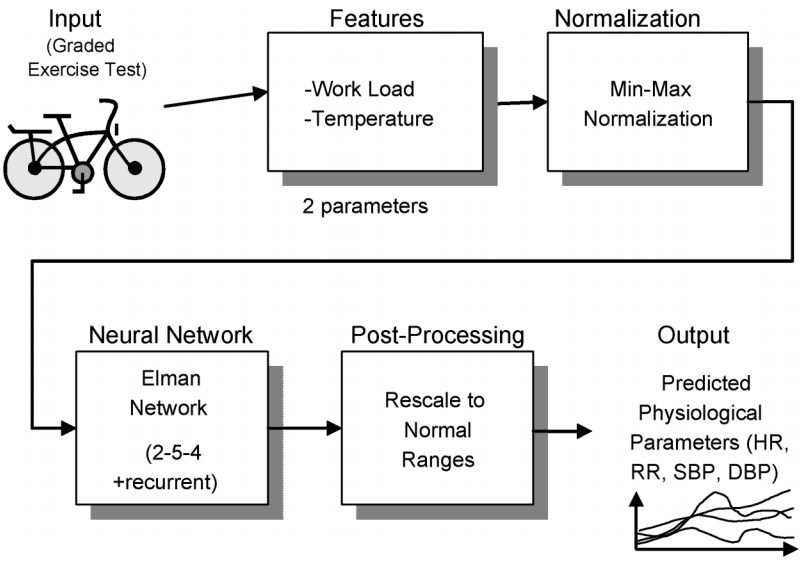


Figure 10.9 Block diagram of the cardiopulmonary-response estimator illustrating the inputs (workload and ambient temperature), outputs (heart rate, respiratory rate, systolic blood pressure, and diastolic blood pressure), and major components.

The first experimental modeler used only four physiological parameters (respiratory rate, heart rate, systolic blood pressure, diastolic blood pressure) along with workload and ambient temperature. That network will be discussed here, although the final modeling system used all of the measured parameters.

Because the workload and physiological parameters varied with time, it was necessary to capture the time element in the model. The use of a recurrent layer added to a feedforward network (see Fig. 10.10), known as an Elman network [Elman, 1990; Elman, 1991], accomplished this. The model has two inputs (workload and ambient temperature) and four predicted outputs (RR, HR, SBP, DBP). Each neuron in the hidden layer is connected to one neuron in the recurrent layer. This copies the information output from the hidden layer and cycles it back through a weighting factor onto the next sample presentation. This has the effect of adding a delay loop to store values from the previous time step to be used in the current time step. This way, a small portion of information from the previous time, $t-1$, is combined with the current time, t. Indirectly, decreasing portions of information from time $t-2$, $t-3$, $t-4$, etc., are also captured, enabling recurrent operation in this network to model the temporal dynamics of the data. A physiological model of the individual can thus be developed over the time span of the test. The recurrent layer also allows the network to adapt to time-varying conditions during operation.

An overall summary of this application is given in Table 10.3. A data set of approximately 80 to 100 patient exercise tests collected by the physician partner, with between 10 and 30 minutes of data collected on each test, were available to develop and test the network. For this application, a modified backpropagation algorithm was used to train the Elman network. The input and target data were normalized

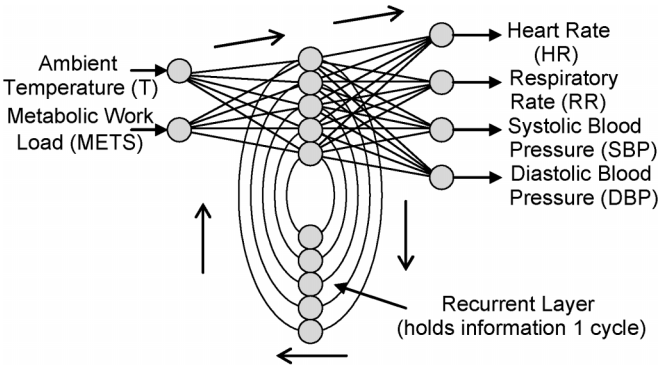


Figure 10.10 Neural network used to model the temporal relationship among four physiological parameters, workload, and ambient temperature. It uses an Elman network with a recurrent layer to capture time-changing information.

Table 10.3 Summary of the cardiopulmonary modeling example.

Application	Predict patient's response to cardiopulmonary stress test
Data Model	2 inputs, 4 outputs, temporally dynamic
Learning	Supervised
Input Features	2—metabolic workload and ambient temperature
Outputs	4—heart rate, breathing rate, systolic blood pressure, diastolic blood pressure
Data Samples	Approximately 80 to 100 patients, some with several tests
Data Source	Dr. Paul A. Allen, M.D., Life Link Inc.
Testing Method	The system was tested on data collected from patients on their second and third tests
Normalization	Min-max normalization
Neural Network	Elman type network
Training	Backpropagation
Post-processing	Rescale data to actual ranges
Test Result	Figure 10.11

to a scale of 0 to 1, based on the calculated minimum and maximum values. The result of one test, showing a typical run with plots of time versus workload, actual and predicted heart rate, and actual and predicted systolic blood pressure, can be seen in Fig. 10.11.

Because the neural network adapts to each individual's stress responses, it becomes a model of the individual's cardiopulmonary system. If the model is developed when the individual is healthy, the predicted physiological parameters can be compared to a later stress test. Any differences between the predicted and actual parameters can be employed to evaluate and diagnose medical conditions that affect the individual's cardiopulmonary system. The stress test can also be compared to general models, based on groupings such as age and gender, to determine whether the individual's cardiopulmonary system is responding as it should for the appropriate age or gender.

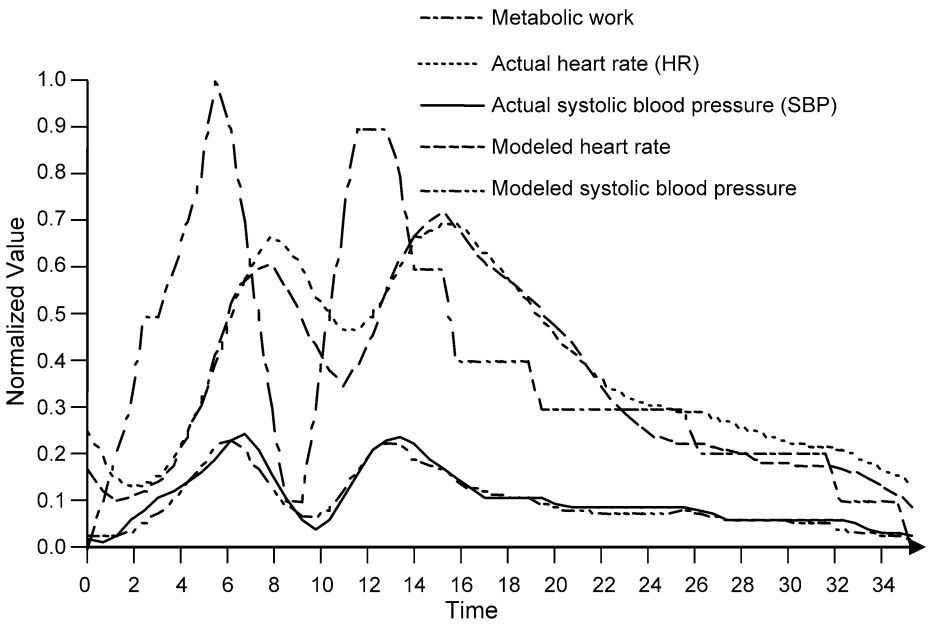


Figure 10.11 A comparison of the modeled and actual heart rate and systolic blood pressure during a graded exercise stress test.

The function approximation and system identification applications all used supervised training. Another application that uses supervised training is that of pattern classification or recognition.

10.4 Pattern Recognition—Tree Classifier Example

Pattern recognition involves taking an input and mapping it to a desired recognition class. We will first illustrate the pattern recognition process with an example that uses cones and leaves to identify four different species of evergreen trees in the Pacific Northwest (see Fig. 10.12). This figure shows the features to be used to identify the species are needle length and cone length. The desired outputs are the four species: black spruce, western hemlock, western larch, and white spruce. We begin by collecting samples of needles and cones for each species and measuring the lengths, resulting in the data shown in Table 10.4.

The raw data have been normalized to assist the neural network during training.

The feature space for the data collected for this application example is depicted in Fig. 10.13. This demonstrates that there is virtually no overlap in the feature space among the four species to be identified. This means that the classifier will not have to work hard to separate the various species from each other and then to use that information to identify a given feature vector with its associated tree species.

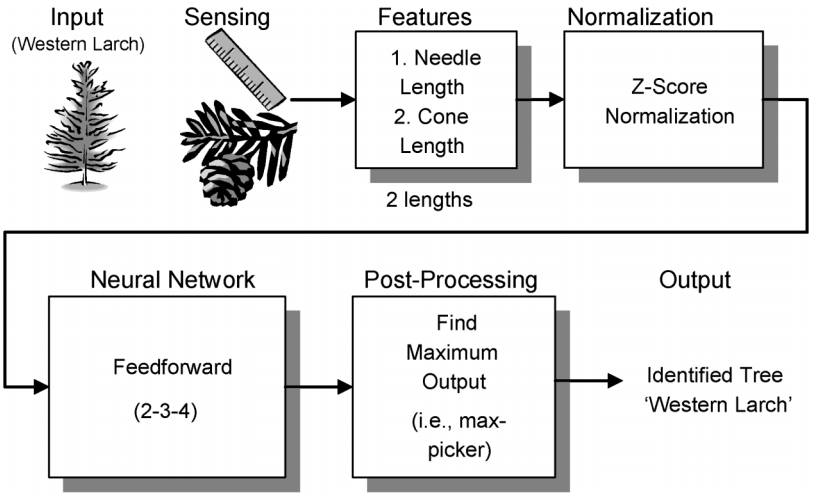


Figure 10.12 Pattern recognition example for distinguishing different evergreen trees in the Pacific Northwest.

Table 10.4 Raw data for distinguishing four evergreen trees in the Pacific Northwest.

Tree	Raw Data		Normalized Data	
	Cone Length	Needle Length	Cone Length	Needle Length
Black Spruce	25 mm	11 mm	–0.32	–0.38
Black Spruce	26 mm	11 mm	–0.29	–0.38
Black Spruce	24 mm	9 mm	–0.35	–0.46
Western Hemlock	20 mm	13 mm	–0.47	–0.30
Western Hemlock	21 mm	14 mm	–0.44	–0.26
Western Hemlock	21 mm	20 mm	–0.44	–0.02
Western Larch	37 mm	31 mm	0.05	0.42
Western Larch	33 mm	33 mm	0.42	0.50
Western Larch	32 mm	28 mm	–0.11	0.30
White Spruce	51 mm	19 mm	0.47	–0.06
White Spruce	50 mm	20 mm	0.44	–0.02
White Spruce	52 mm	20 mm	0.50	–0.02

The use of good features cannot be emphasized enough. The vast majority of the work to be performed in pattern classification is to obtain a good set of features. Once that is accomplished, the classifier system used is not overly critical.

Once the data are collected, normalized, and the associated class identified for each feature vector, the data are presented to a neural network for training. The weights in the neural network are initially set to small random values. We have placed the initial hyperplanes used by the feedforward neural network on the feature space map, as shown in Fig. 10.14.

As the neural network is trained, the hyperplanes shift and begin to carve out decision regions. Feedforward neural networks, with sigmoidal transfer functions,

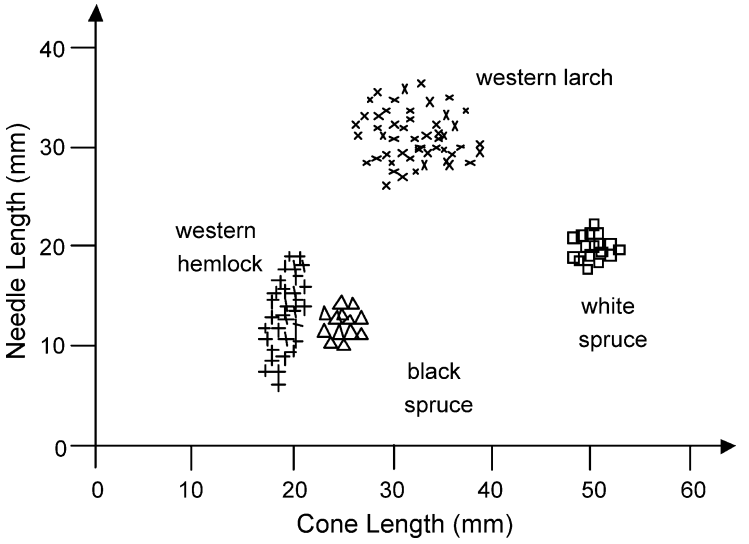


Figure 10.13 Feature space for distinguishing different evergreen trees in the Pacific Northwest.

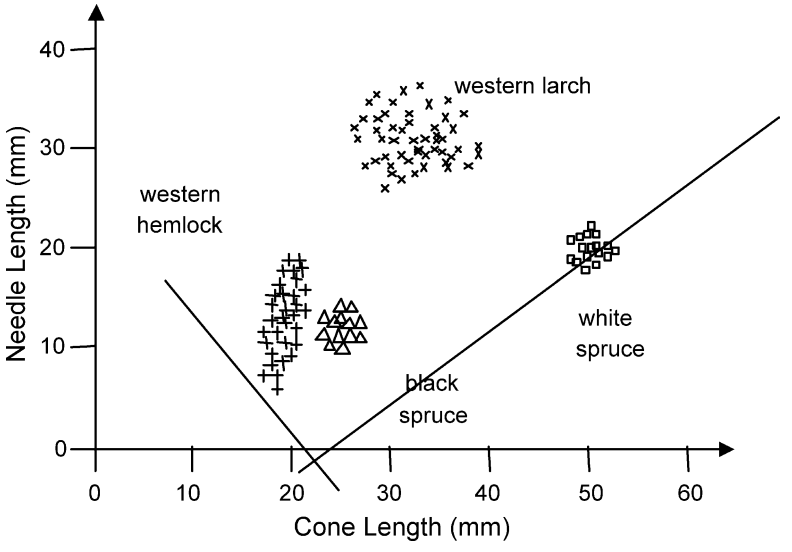


Figure 10.14 Feature space for distinguishing different evergreen trees with initial neural-network hyperplanes.

use the hyperplanes to form what mathematicians term half-spaces. That is, one side of the hyperplane contains one decision region, such as A, and the other side contains the not of that decision region, $\overline{A}$, as depicted in Fig. 10.15.

The four species of evergreen trees can be identified by using three hyperplanes. The logic used by the output layer of the neural network would be something like

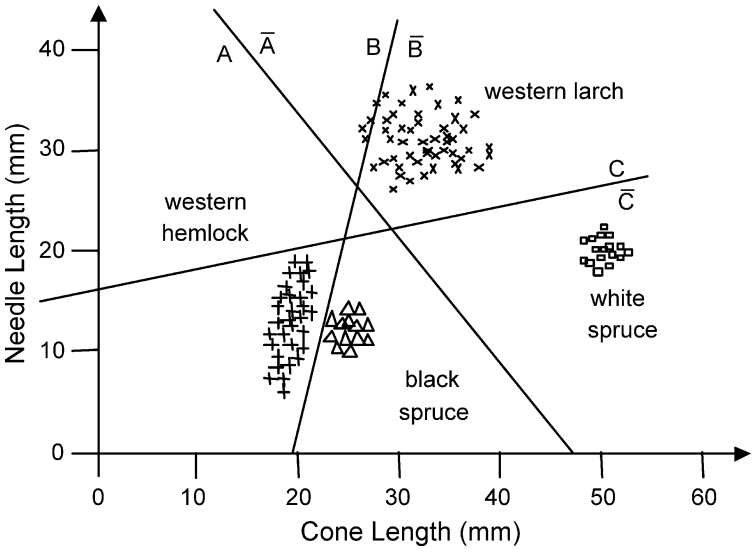


Figure 10.15 Final hyperplanes after 3000 iterations of the network training.

this:

$B \cap A$	*Western Hemlock*
$\overline{A} \cap C$	*Western Larch*
$\overline{A} \cap \overline{C}$	*White Spruce*
$\overline{B} \cap \overline{C}$	*Black Spruce*

The final network and associated decision regions are given in Fig. 10.16. Note that the hyperplanes that define the various tree species do not have limits in the feature space. This means that any new tree species introduced will be recognized as one or more of the learned classes. In addition, notice that the neural network is classifying large areas in the feature space where there has never been a sample. This is something that the reader must always keep in mind when training classifiers that use hyperplanes to make decisions. Making sure that the data in the feature space are used effectively is paramount. These deficiencies can be overcome by partitioning the data in a special way during training, as described in Section 7.3.1.

A summary of the neural network used in this example is given in Table 10.5.

While this example was solved using a feedforward neural network employing hyperplanes, it could just as easily been solved with a clustering algorithm such as the ART or the SOM. Using one of these classifiers with different distance metrics for determining the range of support for each cluster center, or neuron, results in different decision regions that bound each tree species in the feature space, as shown in Fig. 10.17.

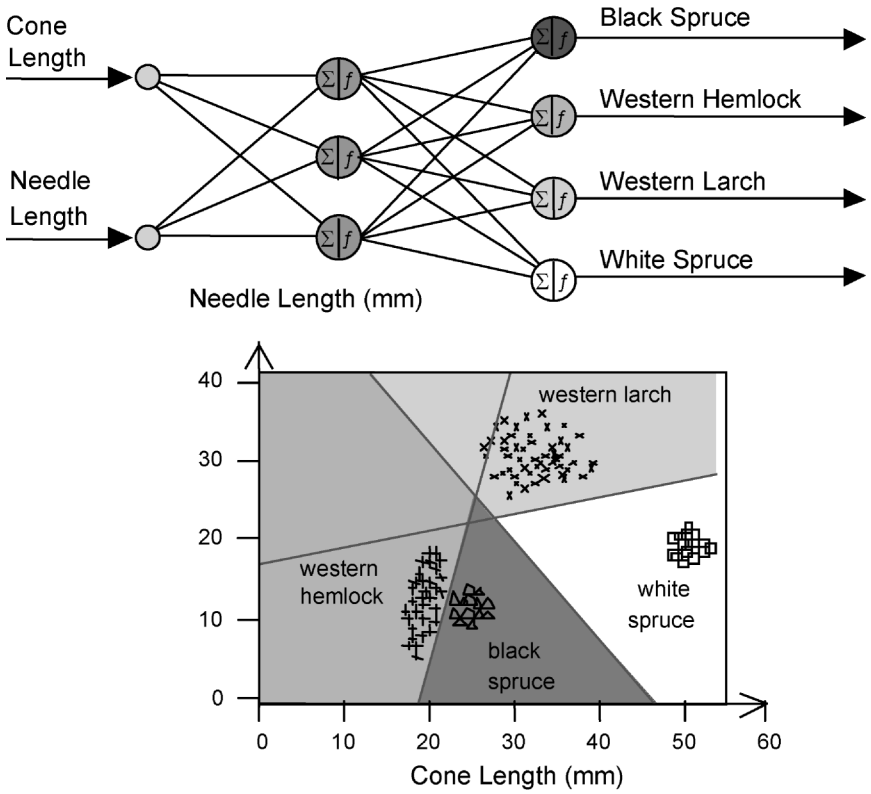


Figure 10.16 Neural network used to classify four species of evergreen with associated decision regions.

Table 10.5 Summary of the tree classifier example.

Application	Recognition of evergreen trees
Data Model	2 inputs, 4 outputs, static
Learning	Supervised
Input Features	2 measurements from trees: needle length and cone length
Outputs	4 classes representing four species of evergreen tree
Data Samples	200 samples from several trees
Data Source	Authors of this book
Testing Method	Split-sample collection for the test set
Training Set	120 samples (60% of total)
Validation Set	30 samples (15% of total)
Test Set	50 samples (25% of total)
Normalization	Z-score normalization
Neural Network	Feedforward network (2-3-4)
Training	Backpropagation
Post-processing	Threshold applied to each output followed by a max-picker to find the most likely class
Test Results	100% classification rate

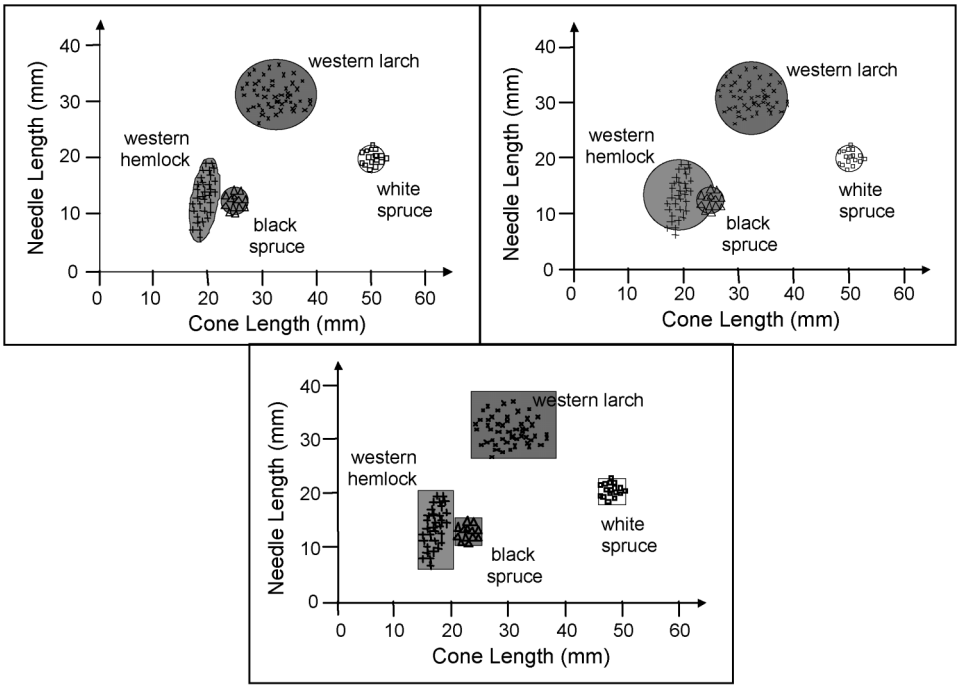


Figure 10.17 Clustering neural-network solutions with associated decision regions using three different distance metrics for each cluster: Mahalanobis (left), Euclidean (right), and taxicab (bottom).

10.5 Pattern Recognition—Handwritten Number Recognition Example

Another pattern recognition success for neural networks has been optical character recognition (OCR) used in applications such as handwriting recognizers for zip codes and personal digital assistants (PDAs), or OCR for scanned pages of text. We present a handwriting recognizer trained with data obtained from Ethem Alpaydin [Alpaydin, 1998] through the UCI Machine Learning Repository [Blake, 1998]. Figure 10.18 summarizes how the data were broken out into training, validation, and test sets.

In this example, the authors present a method of performing handwriting recognition using a simple mouse-driven interface. The reader can easily visualize how this could be extended to laptop finger pads and PDA pen pads. The traditional pattern classification process is shown in Fig. 10.19.

Pattern classification is generally represented by (1) the environment; (2) a sensing system; (3) a set of measurements for the environment obtained for the sensor; (4) extracted features; (5) data normalization; (6) a pattern classifier; and (7) a labeler. The proposed Neural Network Optical Character Recognition (NNOCR) system, shown in Fig. 10.20, uses all of the steps outlined for a pattern recognition system: inputs, feature extraction, data normalization, classifier, and the system

- Problem:
  - Read Handwritten Numbers (e.g., Zip Codes)
- Data Sets:
  - 3823 training samples from 30 people
  - 1797 validation samples from 13 additional people
- Data Source:
  - –E.Alpaydin, C. Kaynak, Department of Computer Engineering, Bogazici University, 80815 Istanbul Turkey alpaydin@boun.edu.tr
  - –http://www.ics.uci.edu/~mlearn/MLSummary.html

Figure 10.18 Data used to train the optical character recognizer.

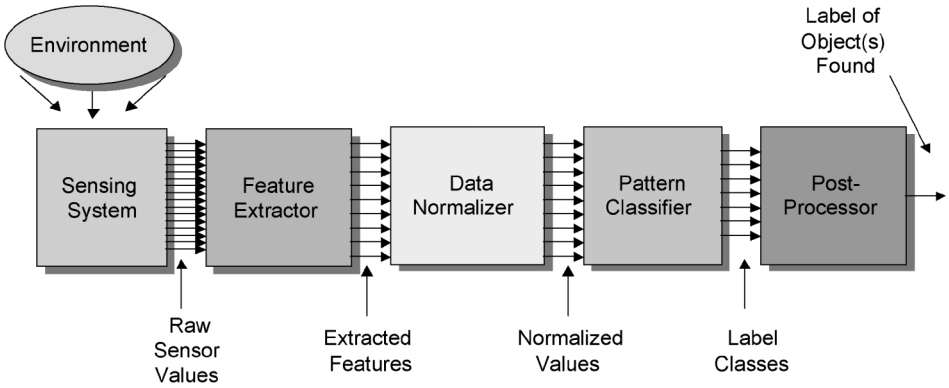


Figure 10.19 Block diagram of pattern classification process.

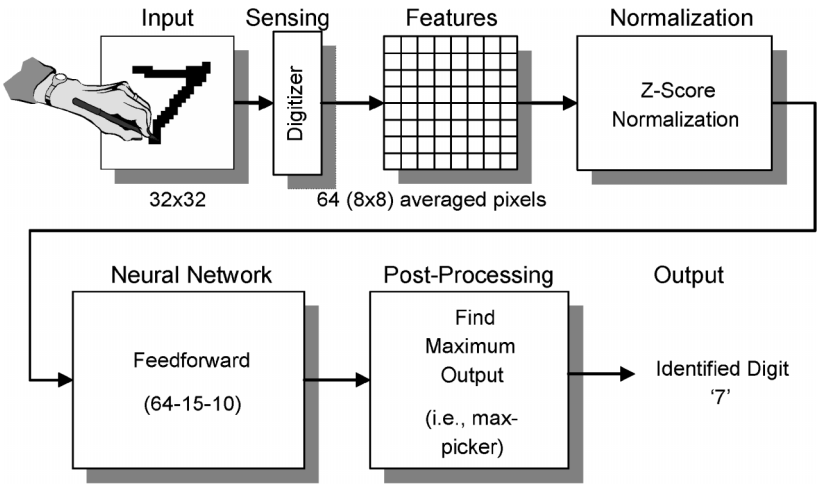


Figure 10.20 Block diagram of neural network OCR system.

output. The training inputs are created by using handwriting samples for each of the characters in the appropriate alphabet, which in this example are the integer numbers $[0, 1, 2, \ldots, 9]$. To create training input, human users enter the desired characters using a mouse, finger on a touchpad, or stylus. The reader will notice that during this process, the inputs are changed from raw pixels to features. The features are averaged pixel values obtained from the input, which converts 1024 input pixel values into 64 features. The recognition system must be able to handle changes in the character set ascribed to translation, scale, and rotation. Examples of various characters that need to be recognized are given in Fig. 10.21. Note how closely the characters 4 and 9 resemble one another.

The system is created using the normalized features as inputs, with the desired numerical value represented by a given neuron, yielding 10 outputs for our example. Each output is trained to be 1 for its associated character and 0 when others are presented. Generally, this is done by dividing available data into a training set and test set. Sometimes, when there are sufficient data, they are divided into training, validation, and test sets. The training set is used to adjust the weights of the neural network. The validation set measures performance without adjusting the weights. The training-set and validation-set error as a function of iterations for the NNOCR are presented in Fig. 10.22. The curves are very similar, showing that the neural network has generalized very well. When a network is overtrained, the

Figure 10.21 Example handwritten characters used to train the NN optical character recognizer.

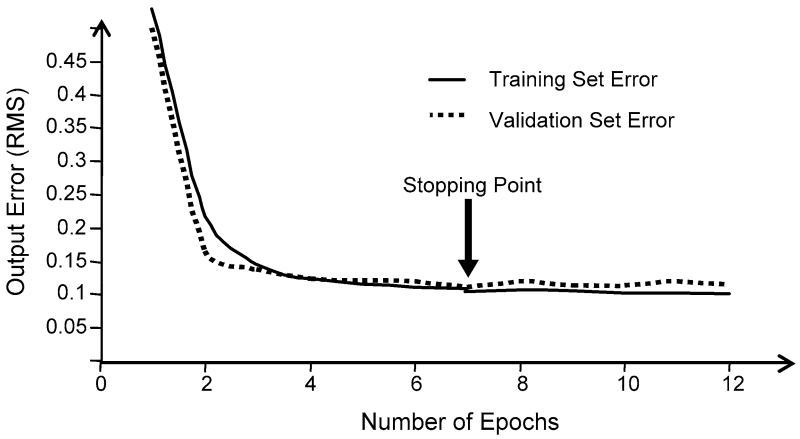


Figure 10.22 Training-set and validation-set error as a function of epochs through the training set.

validation-set error will increase while the training-set error is still decreasing (see Fig. 10.23). A useful insight is that the neural-network training should be halted when the validation-set error is consistently larger than the training-set error. The resultant confusion matrix for the system is shown in Fig. 10.24. The confusion matrix for the NNOCR is nearly ideal, which would contain only diagonal terms. The NNOCR system was implemented using Visual Basic with a mouse interface. The input character is drawn with a mouse and the network produces an output as depicted in Fig. 10.25. The output of each of the 10 nodes is given so the user can see how closely the input matches any of the stored characters. For example, the letters "o" and "a" should most closely match the number 0. The output neuron with the maximum output value, commonly called a max-picker, is chosen as the network output, which is neuron 7. Table 10.6 summarizes the design aspects of the NNOCR.

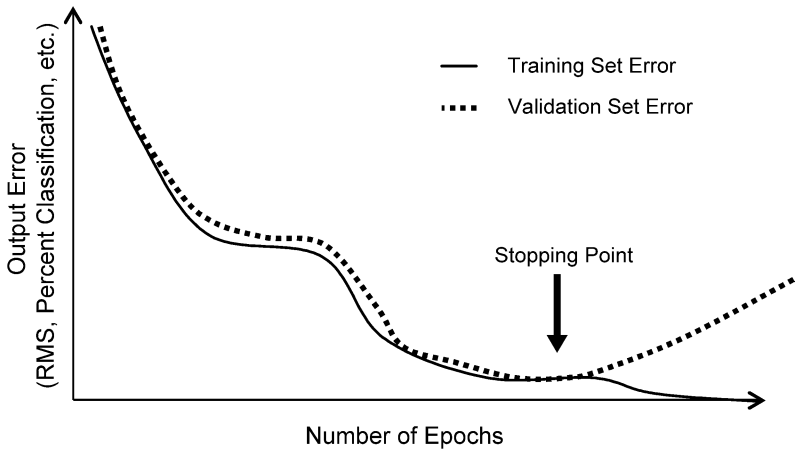


Figure 10.23 Training-set and validation-set error as a function of feature vector presentations.

Desired output

Actual output	0	1	2	3	4	5	6	7	8	9
0	177	0	0	0	0	0	0	0	0	0
1	0	174	6	0	1	0	1	0	14	1
2	0	0	165	4	0	0	0	0	1	2
3	0	0	5	172	0	0	0	0	2	2
4	0	0	0	0	176	0	1	0	1	0
5	0	0	0	1	0	179	0	3	6	2
6	0	0	0	0	0	1	178	0	1	0
7	0	2	0	1	1	0	0	173	0	1
8	1	0	1	3	3	0	1	1	141	5
9	0	6	0	2	0	2	0	2	8	167

Classification: 95% Correct

Figure 10.24 Confusion matrix for the NN optical character recognizer.

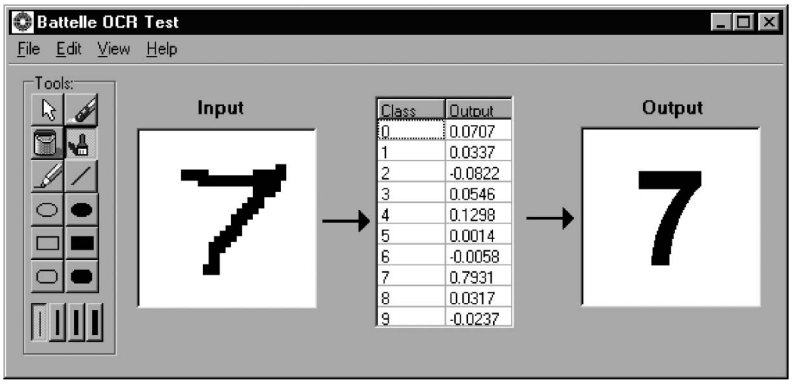


Figure 10.25 Visual Basic GUI for the NN optical character recognizer with the input/output neuron values and final output of the OCR.

Table 10.6 Summary of the NNOCR example.

Application	Recognition of handwritten numbers
Data Model	64 inputs, 10 outputs, static
Learning	Supervised
Input Features	32×32 array of pixels averaged down to 64 features
Outputs	10 classes representing the numbers 0 through 9
Data Samples	5620 handwritten numbers digitized to 1024 pixels
Data Source	Ethem Alpaydin and Cenk Kaynak at Bogazici University in Turkey
Testing Method	Independent sample collection for the test set
Training Set	2676 samples from 30 people (70% of initial data)
Validation Set	1147 samples from the same 30 people (30% of initial)
Test Set	1797 samples from 13 additional people
Normalization	Z-score normalization
Neural Network	Feedforward network (64-15-10)
Training	Backpropagation
Post-processing	Threshold applied to each output followed by a max-picker to find the most likely class
Test Result	95% classification rate (see Fig. 10.24)

10.6 Pattern Recognition—Electronic Nose Example

An electronic nose takes inspiration from biology in both its sensing and pattern recognition traits. Both the olfactory system and the electronic nose consist of an array of chemical-sensing elements and a pattern recognition system. In 1993, one of the authors developed a simple electronic nose prototype to test pattern recognition techniques necessary for building fieldable electronic nose systems [Keller, 1994]. A photo of the system is shown in Fig. 10.26; Fig. 10.27 illustrates the electronic-nose system; and Table 10.7 summarizes the design aspects. While this system is rudimentary by today's standards, it shows how neural networks can be used in fieldable systems. Many of the commercial electronic noses available today employ these basic concepts [Keller, 1999a].

Figure 10.26 Photograph of the prototype electronic nose in operation.

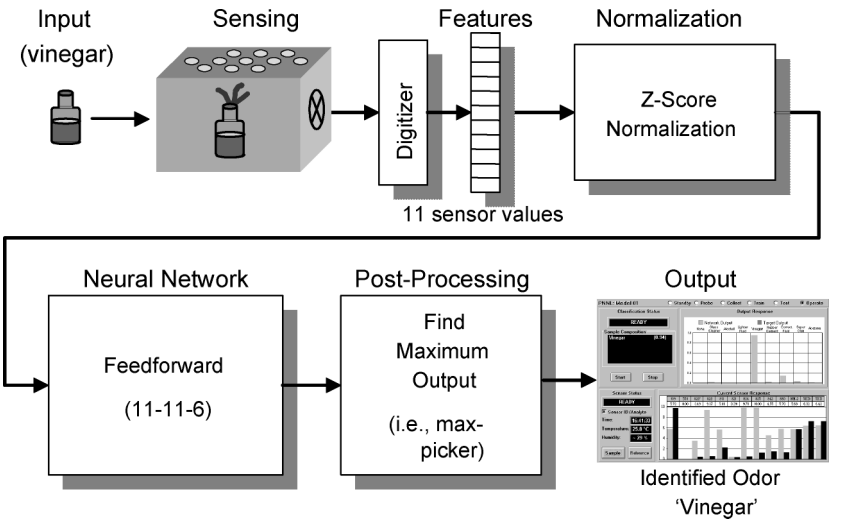


Figure 10.27 Block diagram of the electronic nose illustrating the inputs (nine chemical-sensor values, temperature and humidity), outputs (five chemicals and a category for none), and major components.

The system works by placing a chemical sample in a 5-liter sampling box, which contains a sensor array and mixing fan. The volatile compounds produced are blown over the sensor array by the mixing fan. This transports odorant molecules to the sensors and produces a uniform mixture across the sensor array. The sensors respond physically to the odorant molecules through a chemical reduction process that changes the resistance of the sensor. In this prototype, an array of nine Taguchi-type tin-oxide vapor sensors, one humidity sensor, and one temperature

Table 10.7 Summary of the electronic-nose example.

Application	Odor classification
Data Model	11 inputs, 6 outputs, static
Learning	Supervised
Input Features	11 sensor values (9 chemical, 1 temperature, 1 humidity)
Outputs	6 classes representing 5 chemicals and 1 "none" class
Data Samples	815 odor samples collected from five different household chemicals, along with mixtures of the odors
Data Source	Pacific Northwest National Laboratory
Testing Method	Independent collection of test set Split-sample for validation set
Training Set	433 sensor-response vectors
Validation Set	186 sensor-response vectors
Test Set	196 sensor-response vectors
Normalization	Z-score normalization
Neural Network	Feedforward (A-A-6) ARTmap
Training	Backpropagation Fuzzy ARTmap (vigilance = 0.98 train and 0.80 test)
Post-processing	Threshold applied to each output followed by a max-picker to find the likely odor class or classes
Test Result	Backpropagation: Classification Rate = 92.9% Fuzzy ARTmap: Classification Rate = 93.4%

sensor were used. The electrical signals from the sensors are then sent from the sampling box to an analog-to-digital converter within a desktop computer, so that the digitized sensor values are available to the neural-network software within the computer.

During the data collection process, different chemicals (acetone, ammonia, isopropanol alcohol, lighter fluid, vinegar) along with mixtures were presented to the sampling box and their corresponding sensor values were recorded over several days to collect training and validation data. Additional samples were taken later for testing. This resulted in 619 samples for use in training and validation and 196 samples for testing.

The collected data were normalized by Z-score normalization with the mean and standard deviation computed from the training set. The inputs to the neural networks consisted of the 11 sensor values. The outputs consisted of six classes: one for each chemical plus an additional category for "none." Two types of neural networks were trained for this prototype: a feedforward network trained with backpropagation and a fuzzy ARTmap. The feedforward network had 11 neurons in the hidden layer. The fuzzy ARTmap network is a supervised version of the ART network that was discussed in Section 8.2. We used a training vigilance of 0.98 and a test vigilance of 0.80.

The performance levels of the two networks were essentially equivalent, ranging from 89.7% to 98.2% correct identification on the test set, depending upon the random selection of training patterns. Table 10.8 summarizes the network performances on the test data.

Table 10.8 The performance of the prototype electronic nose when tested on common household chemicals with both the backpropagation and fuzzy ARTmap neural networks.

Number of Training Examples	Number of Test Examples	Input Chemical	Classification Rate on Test Set	
			Backpropagation	Fuzzy ARTmap
67	28	None	96.4%	96.4%
106	25	Lighter Fluid	100.0%	96.0%
75	22	Acetone	100.0%	100.0%
74	27	Ammonia & Lighter Fluid	100.0%	92.6%
64	14	Ammonia	100.0%	100.0%
66	21	Vinegar	81.0%	95.2%
93	28	Isopropanol	92.9%	100.0%
68	26	Ammonia & Vinegar	92.3%	76.9%
5	3	Ammonia & Isopropanol	0.0%	66.7%
1	2	Isopropanol & Vinegar	0.0%	0.0%
619	196	Totals	92.9%	93.4%

10.7 Pattern Recognition—Airport Scanner Texture Recognition Example

Another sensing application that demonstrates the use of neural networks in pattern recognition is airport security [Keller, 2000]. In this application, an individual is imaged with millimeter waves in the frequency range of 12 to 33 GHz (wavelength range of 9 to 25 mm). This wavelength range penetrates clothing to reveal hidden objects carried on the person in or beneath their clothing. A drawing and photograph of the cylindrical scanner is shown in Fig. 10.28, specifically showing hidden weapons and explosives, including nonmetallic items. This raises privacy

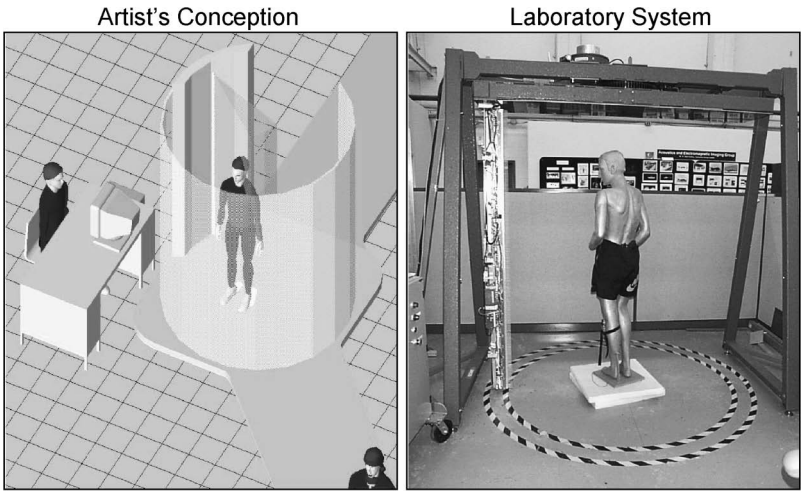


Figure 10.28 Cylindrical millimeter-wave scanner used to screen individuals for hidden weapons and explosives.

issues, because human screeners could see beneath the clothing of the passengers. Therefore, an automated approach to locate potentially threatening objects was needed.

One of the components of this automated system used a feedforward neural network to identify speckle in the images. Plastic, ceramic, and other dielectric items are partially transparent to the millimeter-wave illumination, which often leads to a speckled texture on these items resulting from wave interference to the various coherent reflected and transmitted waves. To an operator, this effect appears as a granular texture. The texture of human skin is smooth and produces very little pixel-to-pixel variation, so the speckle is greatly reduced. Since the texture produced by dielectrics is substantially different from that imaged from the human body, a neural network could be trained to segment dielectric (i.e., plastic) items.

The texture-recognition neural network was designed to examine small regions of the millimeter-wave image. For the system, illustrated in Fig. 10.29, it was found that a 7×7 array of pixels provided enough information for the neural network to identify the speckle effect. Since the images are generally 128×512 pixels in size, the system works by scanning the entire image in a method similar to that of a kernel-based image-processing filter operator. Therefore, the network only sees a small region of the image at any given time. The output of the neural network is thresholded to produce a binary output: speckle or no speckle. This threshold can be varied to change the output sensitivity. This output is then overlaid onto an outline, silhouette, or three-dimensional mannequin model of the scanned individual. The output is further processed by a series of median-window and dilation filters

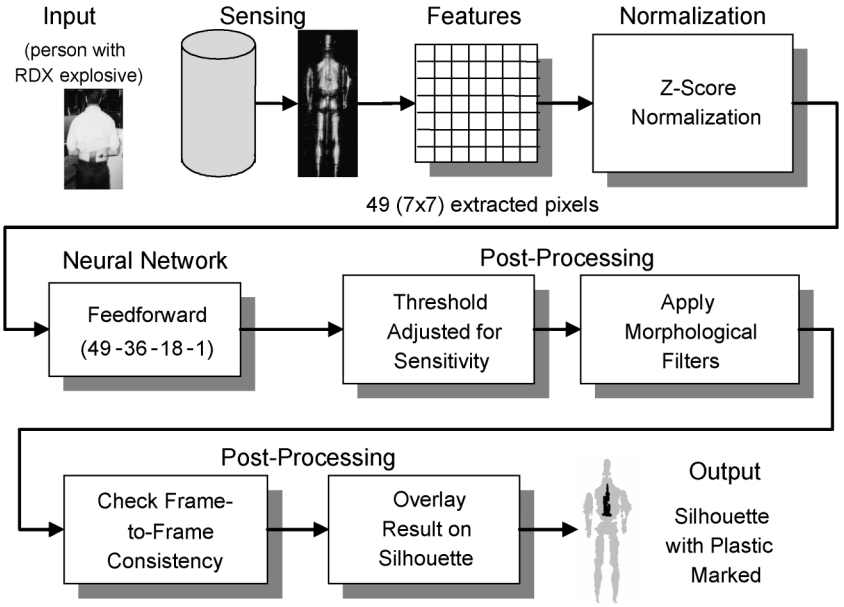


Figure 10.29 Block diagram of the texture detector with the inputs (millimeter-wave image), outputs (processed image), and major components.

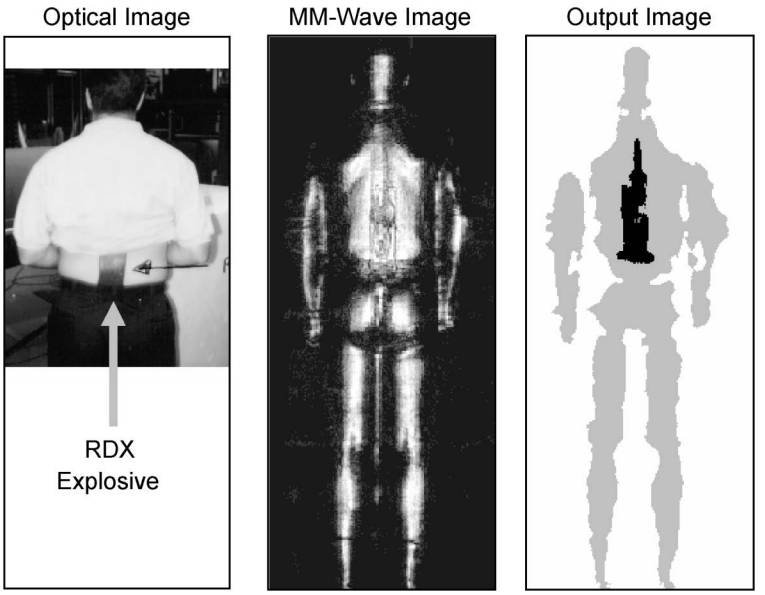


Figure 10.30 A man with RDX explosive strapped to his back is shown on the left. The millimeter-wave image is shown in the middle. Notice the speckle in the center of the back produced by the explosive. On the right is the final output image as the screener sees it.

to remove spurious pixels identified as speckle. Post-processing reduces noise and false alarms in the output images. The entire system recognizes the presence or absence of speckle across various regions in the image. This produces an output image that marks regions of potential interest to the screener. An example is presented in Fig. 10.30. The scanner images a person with RDX explosive and the output of the neural network and associated post-processing filters produces an image that reveals the location of interest while preserving some privacy for the individual.

In this application, almost all the pixels in the data sets do not represent speckle. Approximately 4% of the pixels within the region of the individual represent speckle. So, for training it was necessary to bias the process, increasing the representation of speckle during training by increasing its a priori probability as mentioned in Section 4.1.2.

The results of the test with the human screeners and the speckle detector at two sensitivities are shown in Table 10.9. At low sensitivity, the probability of false alarm (P_{fa}) is substantially lower for the speckle detector than the human screeners, but the detection probability (P_d) is somewhat reduced. For the high-sensitivity case, the probability of detection has increased, but so has the probability of false alarms. Overall, these results show that the performance of the speckle detection algorithm is comparable to that of the human screeners with regard to the detection of plastic threats. A summary for the network training is given in Table 10.10.

Table 10.9 Test results for human screeners and the neural-network-based speckle detector on plastic guns and plastic explosives at low and high sensitivities.

Detection Method	Detection Probability (P_d)	False Alarm Probability (P_{fa})
Human Screener	61.5%	31.0%
Texture Detector Set at Low Sensitivity	62.5%	17.2%
Texture Detector Set at High Sensitivity	75.0%	38.5%

Table 10.10 Summary of the airport scanner texture detection example.

Application	Plastic identification in analysis of speckle in millimeter-wave security images
Data Model	49 inputs, 1 output, static
Learning	Supervised
Input Features	49 extracted millimeter-wave image pixels
Outputs	1 output indicating presence or absence of speckle
Data Samples	50 sequences, each consisting of 36 images per sequence and each containing 128×512 pixels
Data Source	Pacific Northwest National Laboratory
Testing Method	Independent collection of test set
Training Set	17 image sequences
Validation Set	8 image sequences
Test Set	25 image sequences
Normalization	Z-score normalization
Neural Network	Feedforward (49-36-18-1)
Training	Backpropagation
Post-processing	Thresholding, placement of point onto output image, followed by morphological image operators and frame-to-frame consistency check
Test Result	62.5% to 75% classification rate 17.2% to 38.5% false positive rate

10.8 Self-Organization—Serial Killer Data-Mining Example

This application demonstrates that a Self-organizing map can be used to visualize an entire database. Developed by our colleague Lars Kangas [Kangas, 1999] to aid in the investigation of serial murders, it is known as CATCH, for Computer-aided Tracking and Characterization of Homicides, and was developed for the U.S. Department of Justice with assistance from the Washington State Attorney General's Office.

When a serial offender strikes, it usually means that the investigation is unprecedented for the investigating police agency, which can be overwhelmed by incoming leads and pieces of information. These investigations are generally long-term, with the suspect remaining unknown and continuing to commit crimes. CATCH assesses likely characteristics of unknown offenders by relating a specific crime case to other cases and by providing tools for clustering similar cases that may be attributed to the same offenders. One of the clustering tools in CATCH is a self-organizing map that learns to cluster similar cases from approximately 5000

murders and 3000 sexual assaults. These cases reside in a database, with each one detailed by 225 parameters that describe *modus operandi*, the offenders' signature characteristics, and other attributes of the victim, offender, and crime scene [Copson, 1997; Keppel, 1999]. These parameters are derived from a series of questions asked of the local investigator by an investigator from the state crime laboratory.

CATCH maps the 225 parameters, which can be thought of as a 225-dimensional space, to a two-dimensional space organized as 4096 cells in a 64×64 grid, as shown in Fig. 10.31. The proximity of cases within a two-dimensional representation of the clusters allows the analyst to identify similar or serial murders and sexual assaults. This figure highlights a section of the map that contains cases from the Green River murders [Keppel, 1995]. The training phase assigns each crime to exactly one of these cells. The specific cell to which each crime is assigned is based on the clustering algorithm. Similar crimes are placed in closer proximity to each other. Identical or nearly identical crimes may be placed in the same cell. Some cells may not be assigned any crimes during training, but may be assigned new crimes as they are entered into the database. The SOM should be retrained periodically, when a sufficient number of new cases has been added to the database, to take advantage of all available crime data. The SOM portion of CATCH is detailed in Fig. 10.32 and summarized in Table 10.11.

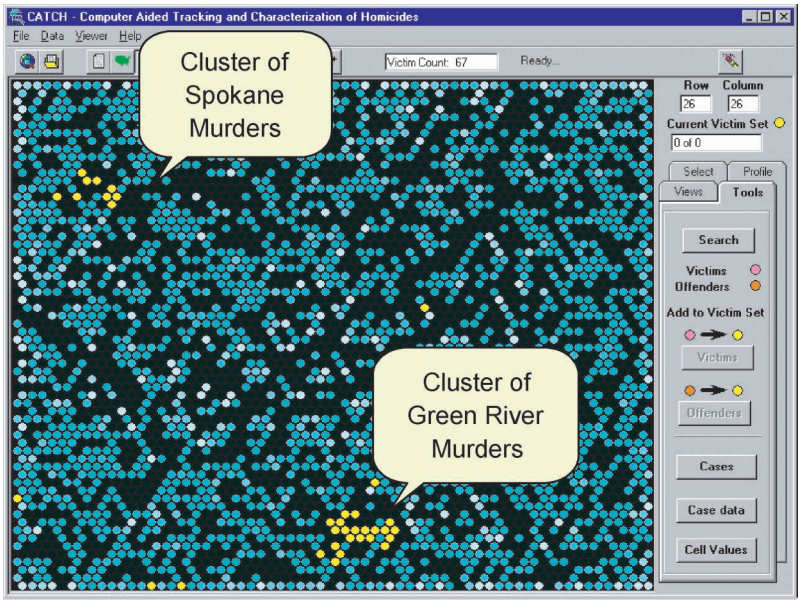


Figure 10.31 A self-organizing map representing about 5000 murders. Each cell in the 64-by-64 map typically contains eight or fewer crimes. A lighter cell color indicates a higher number of crimes in that cell, while a black cell contains no crimes. The white cells within the two highlighted regions indicate cases considered to be committed by two infamous serial killers in the Pacific Northwest.

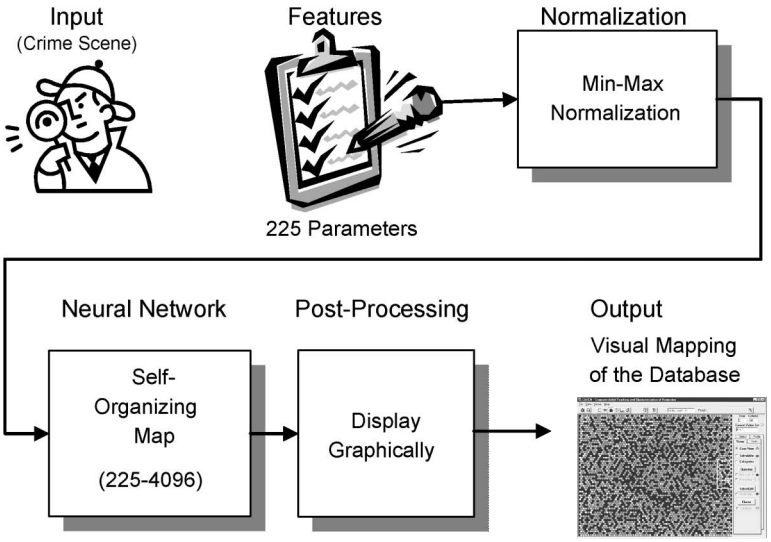


Figure 10.32 Block diagram of CATCH system with the inputs (225 parameters about the offender and victim), outputs (two-dimensional visual mapping of the database), and major components.

Table 10.11 Summary of the CATCH example.

Application	Visualization of murder and sexual-assault cases to determine similarities that might indicate a serial offender
Data Model	225 inputs, 4096 outputs (64×64), static
Learning	Unsupervised
Input Features	225 parameters representing details about the offender, the victim, and the crime scene
Outputs	A two-dimensional map containing 4096 nodes arranged in a 64×64 hexagonal grid
Data Samples	Approximately 5000 murder cases and 3000 serial-rape cases
Data Source	State of Washington Attorney General's Office
Testing Method	All data can be used in training and operation
Neural Network	Self-Organizing Map (225-4096)
Training	Self-Organizing Map
Test Result	See Figure 10.31

10.9 Pulse-Coupled Neural Networks—Image Segmentation Example

In this example, we discuss the application of a significantly different kind of neural network. This pulse-coupled neural network (PCNN) is a model based on the mammalian visual cortex. Eckhorn [Eckhorn, 1990; Eckhorn, 1991] proposed the underlying network to explain the experimentally observed pulse-synchrony process found in the cat visual cortex. Neurons in the visual cortex will fire together to

represent areas of the image field that should be bound together. These could have similar structure or texture.

The PCNN is significantly different from other neural networks in both structure and operation. In the PCNN model, each neuron in the processing layer is tied directly to an image pixel or set of neighboring image pixels as shown in Fig. 10.33 [Linblad, 1998]. Each neuron iteratively processes signals feeding from these nearby image pixels (i.e., feeding inputs) and linking from nearby neurons (i.e., linking inputs) to produce a pulse train. The PCNN does not require training. The properties of the PCNN are adjusted by changing threshold levels and decay-time constants. Figure 10.34 illustrates a PCNC-type neuron. It has inputs (i.e., feeding inputs) from a neighborhood in the image plane and inputs from neighboring neurons in the neuron layer (i.e., linking inputs). Similarities in the input pixels cause the associated neurons to fire in synchrony, indicating similar structure or texture. This synchrony of pulses is then used to segment similar structures

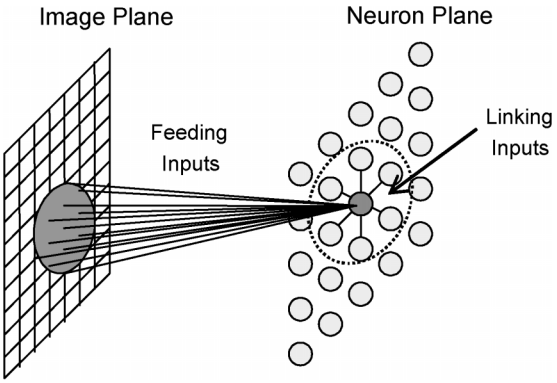


Figure 10.33 A PCNN network with an input image plane and an output neuron plane. A certain region in the image plane is fed into each neuron along with inputs from neighboring neurons.

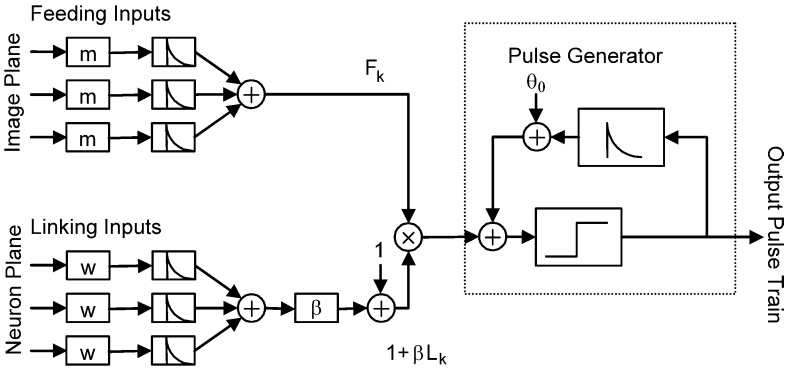


Figure 10.34 A PCNN neuron with links to the image plane and neighboring neurons. This neuron produces an output representative of a pulse train.

or textures in the image. We will discuss a simple application of a PCNN to the segmentation of anatomical structures in a magnetic-resonance image (MRI).

In Fig. 10.35 is a MRI cross-section of the abdomen of a patient with an enlarged kidney, visible on the left side of the image. While the physician can easily see that the kidney is enlarged, image segmentation is used to determine a quanti-

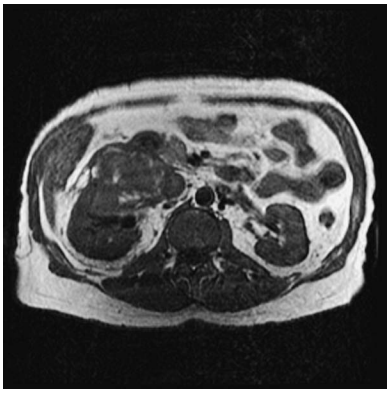


Figure 10.35 Magnetic resonance image (MRI) showing a cross-section of the abdominal region, with an enlarged kidney on the left side of the image.

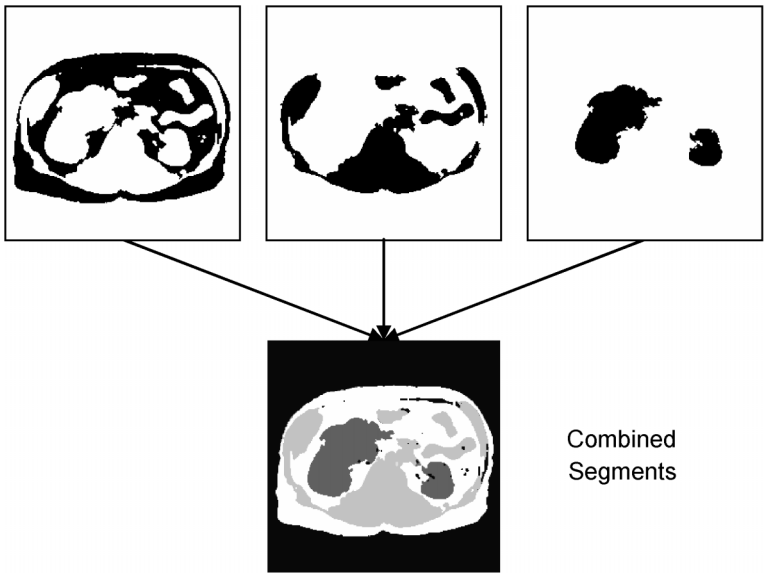


Figure 10.36 Process of combining PCNN image segments to form an image highlighting the important regions. The image in the upper-left segments the fat and muscle tissue within the abdomen and forms the white region in the combined segment image. The abdominal cavity is segmented in the upper-middle image and is represented as light gray in the combined segmented image. The kidneys are segmented in the upper-right image and show as dark gray in the combined segmented image. In this figure, an enlarged kidney is evident in the left when compared to the kidney on the right.

tative figure for the kidney size. Figure 10.36 shows an idealized segmentation of the kidneys produced by a PCNN. This was achieved by running the PCNN with a wide linking radius (8 pixels), followed by an adaptive process of segment combination and spatial filtering with a smoothing filter and a median-window filter [Keller, 1999]. The resulting segmentation process showed that the segmented area of the enlarged kidney was 3.82 times larger than that of the kidney on the right. A full-volume comparison can be computed by processing all the cross-sectional images of the abdomen that contain the kidneys.

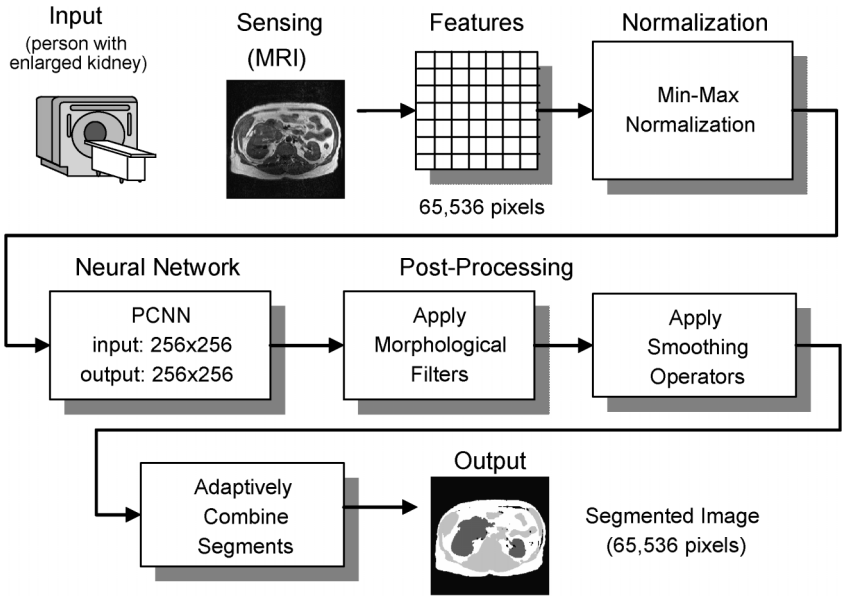


Figure 10.37 Block diagram of the PCNN image-segmenter system with the inputs (MRI image), outputs (segmented image), and major components.

Table 10.12 Summary of the PCNN example.

Application	Segmentation of anatomical structures found within MRI images
Data Model	65,536 inputs, 65,536 outputs, static
Learning	None
Input Features	Entire MRI image
Outputs	Segmented version of entire image
Data Samples	1 image
Data Source	Database of radiological examples for medical students
Testing Method	All data can be used in training and operation
Neural Network	Pulse Coupled Neural Network
Training	None—parameters are manually adjusted
Test Result	See Fig. 10.36

Chapter 11

Dealing with Limited Amounts of Data

The process of collecting and labeling data can be resource intensive, so the amount of data available to the neural-network designer to develop a neural network is often limited. Furthermore, properly training and testing a neural network requires splitting the data into a training, a validation, and a test set, which further reduces the amount of data. Several statistical techniques have been developed for dealing with limited amounts of data. These techniques involve multiple resamplings of the data into a series of sets. The neural-network designer can also use these techniques to judge the performance of neural networks with limited data.

Statisticians originated these techniques for use with statistical estimation and classification, but they are applicable to neural networks. In statistics, they are used to estimate the model's generalization error and to choose its structure [Weiss, 1991; Efron, 1993; Hjorth, 1994; Plutowski, 1994; Shao, 1995]. This is true as well for neural networks. With neural networks, the designer can use these techniques to choose the network architecture, the number of hidden neurons, the salient inputs, training parameters, etc. They can also be used to evaluate the neural network's general performance. A-fold cross-validation, leave-one-out cross-validation, jackknife resampling, and bootstrap resampling are the most common techniques used with neural networks to deal with limited data. When large amounts of data are available and the data are representative of the entire population, then these techniques are generally not needed.

11.1 K-fold Cross-Validation

The k-fold process involves partitioning the data into k separate sets, where k is the number of sets chosen. This division is usually done randomly into k mutually exclusive subsets of approximately equal size. This process is illustrated in Fig. 11.1. A-1 sets are used to train the neural network, and one set is held out and used to test the neural-network architecture and determine its generalization error. This process is repeated k times until all data have been used in both training and testing, but independently. In statistical estimation problems, a value of 10 for k

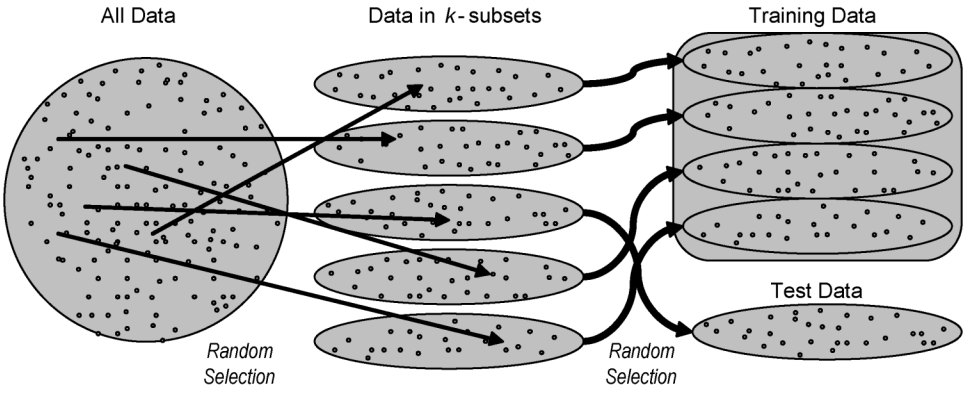


Figure 11.1 Process of randomly breaking data into k-subsets, followed by assignment of subsets to the training and test sets. Training data are used to train the neural network and are generally broken into a training and a validation set.

is popular [Breiman, 1992; Kohavi, 1995]. A specific rule of thumb does not exist for neural networks, but generally a value of 5 to 10 works fine. It is important to remember that k-fold cross-validation is different from the split-sample or holdout validation described earlier.

11.2 Leave-one-out Cross-Validation

The leave-one-out method, illustrated in Fig. 11.2, is similar to k-folding, but only removes one sample at a time for testing. It is the same as k-fold cross-validation where k is equal to the sample size, except that it is more resource intensive than k-fold cross-validation, because it involves leaving out all possible subsets so the entire process is run as many times as there are samples (n).

Leave-one-out cross-validation is also easily confused with jackknifing. Both involve omitting each training case in turn and retraining the network on the remaining subset. However, cross-validation is used to estimate generalization error, while jackknifing is used to estimate the bias of a statistic or neural network.

Leave-one-out cross-validation often works well for continuous-error functions such as the root-mean-square error used in backpropagation. It may perform poorly for discontinuous error functions such as misclassification percentage. If a discontinuous error function is used in the neural-network training, then k-fold cross-validation should be used instead of leave-one-out cross-validation.

11.3 Jackknife Resampling

The jackknife resampling technique was originated by Maurice Quenouille in 1956 [Quenouille, 1956] as a way to study bias in estimators. The name jackknifing comes from John Tukey [Tukey, 1958] after the way a jackknife works. Think of the entire data set as the whole set of knifes. Each subset or each sample is

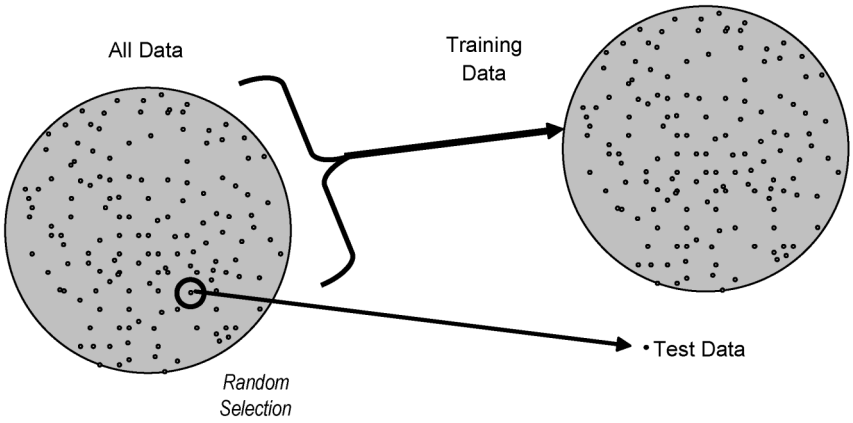


Figure 11.2 Process of randomly selecting a data sample for use in the test set with the remaining data going towards training. Training data are used to train the neural network and are generally broken into a training and a validation set.

a different knife. Each time the developers train a neural network, they pull into the test set the data represented by one of the knives and into the training set the data represented by the remaining knives. In this way, a separate neural network is tested on each subset of data and trained with all the remaining data.

On the surface, jackknifing looks like leave-one-out or k-fold cross-validation. The difference is in what question is being answered by the results for each subset . The purpose of jackknifing is to use the statistics of each subset to determine the neural network's bias, while the purpose of cross-validation (k-fold and leave-one-out) is to estimate the neural network's ability to generalize. If the neural network produces identical or similar statistics for each subset used in the jackknifing process and then for the entire set, then the neural network is considered to be unbiased.

11.4 Bootstrap Resampling

For cross-validation and jackknife resampling to be effective, the error distribution used to select the samples needs to be known. For most cases, this error distribution is not known, so a normal distribution is used. However, a normal distribution might not be representative of the data. The bootstrapping method, invented by Bradley Efron [Efron, 1979], is a more accurate way of dealing with a small sample size. It involves creating a large number of new samples from the original data, resampling the available data to build a sampling distribution. This assumes the sample distribution is representative of the entire population. A Monte Carlo technique, randomly generating new data sets to simulate the process of data generation, accomplishes this resampling. To do this, the Monte Carlo process must draw from an error distribution representative of the problem. The name refers to the phrase "to pull oneself up by one's bootstraps." Figure 11.3 illustrates the process of bootstrapping. The original data set containing n items (10 in this example) is

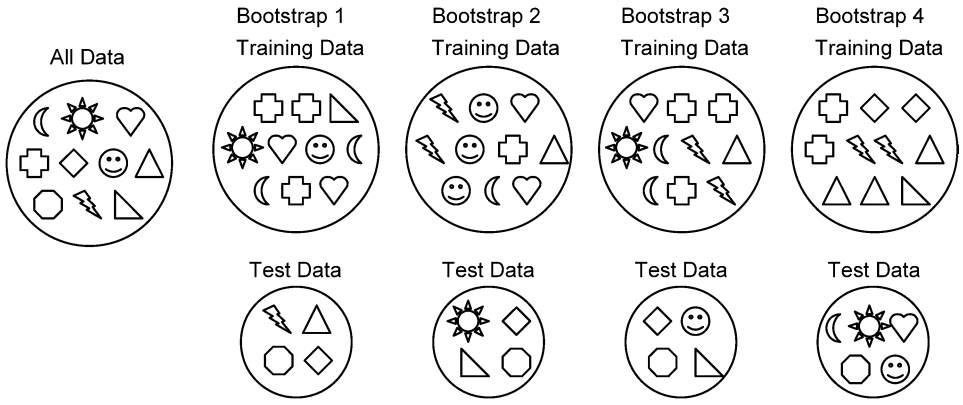


Figure 11.3 Process of forming a series of bootstrap sets from the original data set. Training data are used to train the neural network and are generally broken into a training and a validation set.

randomly sampled with replacement b times to produce a new bootstrap set with exactly n items. The replacement value is randomly selected from the remaining samples and this process can be repeated many times. Four repetitions are shown in this example. A bootstrap set then forms the data used to train a neural network, while the unused data serves to test the neural network. The result is several data sets for use in training and testing. The number of samples (n) in the data set used to train the neural network remains equal to the number of samples in the original data set. The number of test samples can vary. In this illustration, an individual sample can be repeated multiple times in the training data.

Bootstrapping differs from cross-validation in that it involves repeated analysis of subsamples of the data, not subsets of the data. In many cases, bootstrapping seems to work better than the cross-validation methods [Efron, 1983]. More sophisticated versions of bootstrapping exist. One is used to estimate confidence bounds for the network outputs [Efron, 1993]. Another is useful for estimating generalization error in classification problems [Efron, 1997].

While bootstrapping has been used by statisticians for the past 25 years, neural-network designers have only used it for the past decade [Baxt, 1995; Tibshirani, 1996; Masters, 1995]. Bootstrapping is still not a common neural-network technique and knowledge about its use is still somewhat limited; therefore, bootstrapping with neural networks has not been thoroughly researched. It is known that bootstrapping does not work well for some other methodologies [Breiman, 1984; Kohavi, 1995]. It is also known that time-series data are often more complex than typical data used for classification and require complicated methods for bootstrapping [Snijders, 1988; Hjorth, 1994].

In statistics, the bootstrapping technique has been shown to perform remarkably well and to produce surprisingly accurate estimates of statistical sampling distributions. With bootstrapping, the resampled data sets remain the same size as the original. However, some samples are duplicated, while others are discarded. When compared with cross-validation, this resampling increases the variance that

can occur in each run, but bootstrapping provides a more realistic simulation of real life [Efron, 1993]. Bootstrapping preserves the *a priori* probabilities of the classes throughout the random selection process. It has an added benefit in that it can obtain accurate measures of both the bias and variance of the neural-network model. Bootstrapping should be considered when the sample size is small and the error distribution is suspected to be abnormal.

Appendix A. The Feedforward Neural Network

A feedforward network is composed of neurons arranged in layers, as illustrated by Fig. 1.10 in the introduction. Data are introduced into the system through an input layer. This is followed by processing in one or more intermediate (hidden) layers. Output data emerge from the network's final layer. The transfer functions contained in the individual neurons can be almost anything. In this appendix, we describe the mathematics behind the feedforward neural network and the backpropagation algorithm that is commonly used to train feedforward networks with sigmoidal transfer functions. We also mention some of the alternatives to backpropagation for training feedforward networks.

For the mathematical derivations within this appendix, we will use the notations given in Table A.1. This follows the terminology found in many sources that deal with backpropagation. We will also use superscripts to indicate the index of the layer. Subscripts indicate indices for neurons and patterns (data samples).

A.1 Mathematics of the Feedforward Process

The input layer, also known as the zeroth layer, of the network serves to redistribute the input values and does no processing. The outputs of this layer are described mathematically by Eq. (A.1), where x_i represents the input vector and N^0 represents the number of neurons in the input or zeroth layer:

$$o_i^0 = x_i, \quad \text{where } i = 1, \ldots, N^0. \tag{A.1}$$

The input to each neuron in the first hidden layer in the network is a summation of all weighted connections between the input or zeroth layer and the neuron in the first hidden layer. This weighted sum is sometimes called the net stimulus or net input, and is commonly denoted as *net*. We can write the net input to a neuron from the first layer as the product of that input vector, x_i, and the weight factor, w_i, plus a bias term, θ. The total weighted input or net stimulus to the neuron is a summation of these individual input signals and is described by Eq. (A.2), where

Table A.1 A summary of variable names and symbols used in the description of feedforward neural networks and the backpropagation algorithm.

E	total output error when all patterns are presented
E_p	total output error when pattern p is presented
x_{pi}	input to the i^{th} neuron in the input or zeroth layer when pattern p is presented
y_{pi}	output of i^{th} neuron in the final layer when pattern p is presented
t_{pi}	target output for the i^{th} neuron in the final layer when pattern p is presented
$f_j^{\ell}(u)$	activation function for the j^{th} neuron in ℓ^{th} layer
$\dot{f}_j^{\ell}(u)$	derivative of the activation function
o_{pi}^{ℓ}	output of i^{th} neuron in ℓ^{th} layer when pattern p is presented
w_{ki}^{ℓ}	weights linking the i^{th} neuron in the $\ell - 1^{\text{st}}$ layer to the k^{th} neuron in the ℓ^{th} layer
θ	bias term for activation function
w_{j0}^{ℓ}	bias term written as a weight to a unitary input
net	net stimulus to i^{th} neuron in ℓ^{th} layer when pattern p is presented
N^{ℓ}	number of neurons in the ℓ^{th} layer
i, j, k	indices for the neurons and weights $(1, \ldots, N^L)$
L	number of layers for the network excluding the input layer
ℓ	layer index $(0, \ldots, L$; where $0 =$ input and $L =$ output)
P	number of patterns in the training set
p	pattern index $(1, \ldots, P)$
δ_{ik}^{ℓ}	delta term containing the error needed to update the connection linking the i^{th} neuron in the $\ell - 1^{\text{st}}$ layer to the k^{th} neuron in the ℓ^{th} layer
Δw_{ki}^{ℓ}	weight update for the connection linking the i^{th} neuron in the $\ell - 1^{\text{st}}$ layer to the k^{th} neuron in the ℓ^{th} layer
η	learning rate
α	momentum

N represents the number of neurons in the input layer:

$$net\ stimulus = \sum_{i=1}^{N} w_i x_i + \theta. \tag{A.2}$$

The net stimulus to the neuron is transformed by the neuron's activation or transfer function, $f(u)$, to produce a new output value for the neuron. With backpropagation, this transfer function is most commonly either a sigmoid or a linear function. In addition to the net stimulus, a bias term, θ, is generally added to offset the input. Often, the bias is designated as a weight coming from a unitary valued input and denoted as w_0. So, the final output of the neuron is given by the following equation and shown graphically in Fig. A.1.

$$output = f(net) = f\left(\sum_{i=1}^{N} w_i x_i + \theta\right) = f\left(\sum_{i=1}^{N^0} w_i o_i^0 + w_0\right). \tag{A.3}$$

Now we expand to multiple neurons and multiple layers. The outputs of the neurons in one layer, o_i^{ℓ}, are interconnected via weights to neurons in the next layer. So the

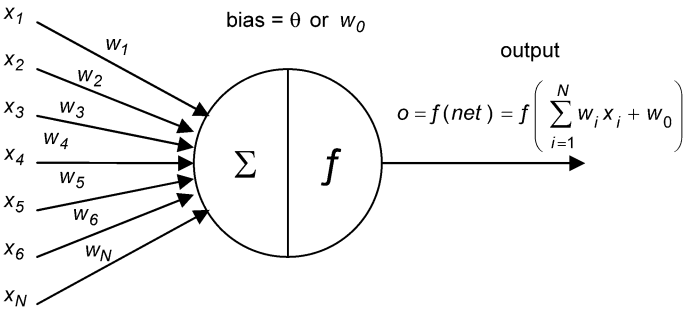


Figure A.1 Process of summing weighted connections into a neuron and then applying an activation function to produce an output.

net stimulus of the j^{th} neuron in the ℓ^{th} layer is given by Eq. (A.4), and the output of the neuron is described by Eq. (A.5).

$$\text{net}_j^\ell = \sum_{i=1}^{N^{\ell-1}} w_{ji}^\ell o_i^{\ell-1} + \theta_j^\ell = \sum_{i=1}^{N^{\ell-1}} w_{ji}^\ell o_i^{\ell-1} + w_{j0}^\ell, \tag{A.4}$$

$$o_j^\ell = f_j^\ell(\text{net}_j^\ell) = f_j^\ell\left(\sum_{i=1}^{N^{\ell-1}} w_{ji}^\ell o_i^{\ell-1} + w_{j0}^\ell\right). \tag{A.5}$$

The outputs that emerge from the network at the final or L^{th} layer are described by

$$y_j = o_j^L, \tag{A.6}$$

where $j = 1, \ldots, N^L$.

Often, the input layer is not included in the network's total layer count since it performs no processing and is only used to redistribute the incoming signals. A common notation for describing the network topology (i.e., distribution of neurons in the network) of a feedforward network is

$$N^0 - N^1 - N^2 - \cdots - N^L. \tag{A.7}$$

For example, a network with three inputs, one hidden layer with 10 neurons, and two outputs, is referred to as a 3-10-2 feedforward network.

A.2 The Backpropagation Algorithm

Earlier we mentioned that backpropagation is the most common technique for training a supervised neural network. In this section, we go through the mathematics behind this algorithm. More complete derivations can be found in Rumelhart's, Pao's, and Haykin's books [Rumelhart, 1986; Pao, 1989, pp. 120–128; Haykin,

1994, pp. 142–156]. The goal of backpropagation, as with most training algorithms, is to iteratively adjust the weights in the network to produce the desired output by minimizing output error. The algorithm's goal is to solve the credit-assignment problem. Backpropagation is a gradient-descent approach in that it uses the minimization of first-order derivatives to find an optimal solution. It works with a training set of input vectors, $\mathbf{x}$, and target output vectors, $\mathbf{t}$. The training algorithm iteratively tries to force the generated outputs represented by vector y to the desired target output vector, $\mathbf{t}$, by adjusting the weights in the network through the use of a generalized delta rule.

A.2.1 Generalized Delta Rule

When an input vector is presented to the feedforward network, the output error can be computed by a squared error. The squared error is calculated as the sum of the squared differences between the target values and output values:

$$E_p = \frac{1}{2}\sum_{j=1}^{N^L}(t_{pj} - y_{pj})^2. \tag{A.8}$$

The output error for all vectors presented to the feedforward network is given by

$$E = \sum_{p=1}^{P} E_p = \frac{1}{2}\sum_{p=1}^{P}\sum_{j=1}^{N^L}(t_{pj} - y_{pj})^2. \tag{A.9}$$

To reduce the error in the network, we minimize error with respect to the weights in the network by taking the partial derivative of error with respect to all of the weights in the network and forcing it to zero. This gradient-descent error minimization is defined by

$$\frac{\partial E}{\partial w_{jk}^{\ell}} \equiv 0. \tag{A.10}$$

The partial derivative of the total error can be broken up into a summation of the errors for each presented pattern:

$$\frac{\partial E}{\partial w_{ji}^{\ell}} = \sum_{p=1}^{P}\frac{\partial E_p}{\partial w_{ji}^{\ell}}. \tag{A.11}$$

Next, we break down the partial derivative into two parts by using the chain rule. The first term contains the change in error with respect to net stimulus and the

second term contains the change in net stimulus with respect to weights:

$$\frac{\partial E_p}{\partial w_{ji}^{\ell}} = \frac{\partial E_p}{\partial \text{net}_{pj}^{\ell}} \frac{\partial \text{net}_{pj}^{\ell}}{\partial w_{ji}^{\ell}}. \tag{A.12}$$

The second term in the chain rule is easier to solve, so we tackle it first. This change in net stimulus as a function of a change in weights can be solved by substituting the net stimulus, which is given in Eq. (A.4). This results in the output of a neuron as shown in Eq. (A.13):

$$\frac{\partial \text{net}_{pj}^{\ell}}{\partial w_{ji}^{\ell}} = \frac{\partial}{\partial w_{ji}^{\ell}} \sum_{k=1}^{N^{\ell}} \left(w_{jk}^{\ell} o_{pk}^{\ell-1} \right) + w_{j0}^{\ell} = o_{pk}^{\ell-1}. \tag{A.13}$$

By substituting Eq. (A.13) back into Eq. (A.12), we get Eq. (A.14). Equation (A.14) looks similar to a learning rule known as the delta rule, which was used in perceptron learning and was a precursor to backpropagation. By defining the change with respect to net stimulus as a delta term, we get Eq. (A.15). Substituting back, we get Eq. (A.16), and call it the generalized delta rule for its similarities to the delta rule developed by Widrow and Hoff in the late 1950s for use in training perceptrons with the least-mean-square method [Widrow, 1960]:

$$\frac{\partial E_p}{\partial w_{ji}^{\ell}} = \frac{\partial E_p}{\partial \text{net}_{pj}^{\ell}} o_{pi}^{\ell-1}, \tag{A.14}$$

$$\delta_{pj}^{\ell} \equiv -\frac{\partial E_p}{\partial \text{net}_{pj}^{\ell}}, \tag{A.15}$$

$$\frac{\partial E_p}{\partial w_{ji}^{\ell}} = -\delta_{pj}^{\ell} o_{pi}^{\ell-1}. \tag{A.16}$$

Now we return to solving the change in error with respect to net stimulus. This must be further broken down, using the chain rule, into a term that measures change in output error with respect to the output of a neuron in any layer and the change in the neuron output with respect to its net stimulus. This gives us

$$\frac{\partial E_p}{\partial \text{net}_{pj}^{\ell}} = \frac{\partial E_p}{\partial o_{pj}^{\ell}} \frac{\partial o_{pj}^{\ell}}{\partial \text{net}_{pj}^{\ell}}. \tag{A.17}$$

The second term in this chain rule is the easier to solve so we tackle it first. By substituting Eq. (A.5), we get the derivative of the neuron's activation function evaluated with the net stimulus of the neuron:

$$\frac{\partial o_{pj}^{\ell}}{\partial \text{net}_{pj}^{\ell}} = \frac{\partial}{\partial \text{net}_{pj}^{\ell}} \dot{f}_j^{\ell}\left(\text{net}_{pj}^{\ell}\right) = \dot{f}_j^{\ell}\left(\text{net}_{pj}^{\ell}\right). \tag{A.18}$$

This leaves the more complicated partial derivative of the error with respect to any neuron's output. When we are computing this for the output layer, we know from Eq. (A.4) that that error is a summed square of the difference between target and output:

$$\frac{\partial E_p}{\partial o_{pj}^{\ell}} = \frac{\partial}{\partial o_{pj}^{\ell}}\left[\frac{1}{2}\sum_{j=1}^{N^L}(t_{pj} - o_{pj})^2\right] = -(t_{pj} - o_{pj}) \quad \text{when } \ell = L. \tag{A.19}$$

The variation in output error as a function of hidden-layer neuron output requires a more complex analysis. Each output in a hidden layer is connected to all neurons in the successive layer, so the variation is distributed to all neurons succeeding (i.e., downstream from) it in the network. So, a change in a weight in one layer affects all outputs downstream. The variation in output error with respect to an internal neuron's output is determined by the variation in its net stimulus. Therefore, we must sum over all variations in downstream layers to get the total variation in output error with respect to the output of a hidden neuron. We write this as Eq. (A.20).

$$\frac{\partial E_p}{\partial o_{pj}^{\ell}} = \sum_{k=1}^{N^{\ell+1}}\left(\frac{\partial E_p}{\partial \text{net}_{pk}^{\ell+1}} \cdot \frac{\partial \text{net}_{pk}^{\ell+1}}{\partial o_{pj}^{\ell}}\right) \quad \text{when } \ell < L \text{ (i.e., hidden layers).} \tag{A.20}$$

The first term in the summation is the delta term that we defined earlier.

$$\frac{\partial E_p}{\partial \text{net}_{pk}^{\ell+1}} = -\delta_{pk}^{\ell+1} \quad \text{when } \ell < L \text{ (i.e., hidden layers).} \tag{A.21}$$

The second term in the summation can be solved by substituting the net stimulus to give us

$$\frac{\partial \text{net}_{pk}^{\ell+1}}{\partial o_{pj}^{\ell}} = \frac{\partial}{\partial o_{pj}^{\ell}}\left(\sum_{i=1}^{N^{\ell}} w_{ki}^{\ell+1} o_i^{\ell} + w_{k0}^{\ell+1}\right) = w_{kj}^{\ell+1} \quad \text{when } \ell < L. \tag{A.22}$$

So Eq. (A.20) becomes

$$\frac{\partial E_p}{\partial o_{pj}^{\ell}} = -\sum_{k=1}^{N^{\ell+1}} \delta_{pk}^{\ell+1} w_{kj}^{\ell+1} \quad \text{when } \ell < L. \tag{A.23}$$

Now we return to the delta term of Eq. (A.15). From this, we see that Eq. (A.20) becomes Eq. (A.24) for the output layer and Eq. (A.25) for the hidden layers:

$$\delta_{pj}^{L} = \dot{f}_j^{L}\left(\text{net}_{pj}^{L}\right) \cdot (t_{pj} - y_{pj}) \quad \text{when } \ell = L \text{ (i.e., output layer),} \tag{A.24}$$

$$\delta_{pj}^{\ell} = \dot{f}_j^{\ell}\left(\text{net}_{pj}^{\ell}\right) \sum_{k=1}^{\ell+1} \delta_{pk}^{\ell+1} w_{kj}^{\ell+1} \quad \text{when } \ell < L \text{ (i.e., hidden layer).} \tag{A.25}$$

We can see from Eq. (A.25) that we need to calculate the delta term for the output layer first, then go backwards from the output layer to the input layer, calculating the deltas for the hidden neurons. The error term shows up directly in the delta term for the output layer. So, we are propagating an error backwards through the network. This gives rise to the name backpropagation of error.

Now, we will return to the original goal—minimizing output error with respect to weights. Figure 1.10 in the introduction shows an idealized gradient descent. The first-order partial derivative of error with respect to weights gives a direction for the weights to change to reduce error. To reduce the output error, we change the weights in the direction indicated by the gradient by subtracting from the weights a portion of this first-order partial derivative scaled by η. This scale factor is called the learning rate and determines the size of the step to take in optimizing the weights. This gives us a weight update equation:

$$\Delta w_{pji}^{\ell} = -\eta \frac{\partial E_p}{\partial w_{pji}^{\ell}}. \tag{A.26}$$

By combining this with Eq. (A.16), we get

$$\Delta w_{pji}^{\ell} = \eta o_{pj}^{\ell} o_{pi}^{\ell-1}. \tag{A.27}$$

Gradient descent assumes an infinitesimal step size. However, if the learning rate is set too low, then the time needed to learn the synaptic weights will be exceedingly long. If the learning rate is set too high, then the algorithm tends to oscillate and the trained network tends to perform poorly because the weight changes are too radical. Therefore, the learning rate is used to control the convergence of the algorithm.

These weight changes can be applied after each pattern is presented, or all computed changes can be summed up until all patterns are presented and then applied to the weights in the network. This summation is shown in Eq. (A.28):

$$\Delta w_{ji}^{\ell} = \sum_{p=1}^{P} \Delta w_{pji}^{\ell}. \tag{A.28}$$

As mentioned earlier, the bias or offset term of the activation function is generally considered to be a weight that is connected to a fixed-unit output node. It is updated the same way as all the other weights in the network.

A.2.2 Backpropagation Process

Table A.2 lists the steps of the backpropagation algorithm. Initially, the weights in the network are set to random values. Each iteration contains two phases: a feedforward phase and an error backpropagation phase.

In the first phase, a training vector, x_{pi}, is presented to the network. It is propagated forward to produce an output vector, y_{pj}. Then, each output is compared to

Table A.2 Backpropagation Training Procedure.

Step 1.	Initialize Weights $w_{ij}^{\ell} =$ Uniformly Random Over $-\varepsilon$ to ε for all i, j, and ℓ.
Step 2.	Pick a labeled pattern (input and target) from the training set and present input pattern to the network. input $= x_{pi}$ target $= t_{pj}$ for $i = 1$ to N^0 (all input nodes) for $j = 1$ to N^L (all output neurons) for $p = 1$ to P (all patterns)
Step 3.	Propagate data forward and generate the output pattern. output $= y_{pj}$ for $j = 1$ to N^L (all output neurons)
Step 4.	Calculate error between target and actual output by using Eq. (A.8).
Step 5.	Propagate error backwards through network and calculate changes to the synaptic weights that will reduce output error by using the weight-update formula, either with or without momentum, as given by Eq. (A.27) or Eq. (A.31).
Step 5a.	If batch mode, do not apply the weight changes until Step 7.
Step 5b.	If incremental mode, apply the weight changes.
Step 6.	If there are more patterns (i.e., $p < P$) in the training set, loop back to Step 2.
Step 7a.	If batch mode, update synaptic-weight values in the network by using the summation of all weight changes from all pattern presentations as given by Eq. (A.28).
Step 8.	If output error is high or the maximum number of iterations has not been met, then loop back to Step 2.

the target output, t_{pj}, and the error between the actual output and target output is calculated with Eq. (A.8).

During the second phase, the synaptic weights are adjusted using the calculated error values that are propagated from the output layer back to the hidden layers. The weights in the network are adjusted using the generalized delta rule as given to minimize the output error. Weights can also be adjusted after each sample is processed, or after all samples have been processed. Adjustment following each sample is known as incremental, instantaneous, continuous, or online learning. Adjustment after all samples are presented is known as batch learning.

This process is repeated with a large number of labeled samples (inputs with targets) until the error converges to a minimum. A small fraction of the observed error is removed by passing the error backwards through the network while performing small adjustments on the synaptic connections that would decrease the error if the same information mapping was tried again in the network. The fraction of error that is removed is called the learning rate, η, and is used to control the algorithm's convergence rate. If the learning rate is set too low, the time needed to learn the synaptic weights will be long. If the learning rate is set too high, the algorithm tends to oscillate and the synaptic weights that are generated will produce poor classifications or estimations.

Many additional details should be considered when using the backpropagation algorithm. The activation function must be differentiable and nondecreasing. A sigmoid function, either logistic or hyperbolic tangent, is generally used as the activa-

tion function. The logistic sigmoid has a derivative in the form given by Eq. (A.29); the hyperbolic tangent, in the form given by Eq. (A.30):

$$\dot{f}(u) = \frac{d}{du}\text{logistic}(u) = \frac{e^{-u}}{(1+e^{-u})^2} = f(u)[1 - f(u)], \tag{A.29}$$

$$\dot{f}(u) = \frac{d}{du}\tanh(u) = 1 - \tanh^2(u) = 1 - f^2(u). \tag{A.30}$$

Both of these derivatives are written in a form that contains the original function, so values calculated during the feedforward phase can be used in the backpropagation phase to decrease computation time.

To reduce the chances of becoming stuck in a local minima or oscillating around the error surface, a momentum term is often added to the weight-update equation [Rumelhart, 1986; Hagiwara, 1992]. This momentum term is called the heavy ball method in numerical analysis [Poljak, 1964; Bertsekas, 1996]. The role of momentum is to filter out rapid changes in error surface, as illustrated in Fig. A.2. In this figure, the error drops through a local minima and rises to a local maxima before falling into the global minimum. The momentum keeps the weight changes going in the same direction even when a local minima is encountered. To include the momentum term in the weight-update equation, we add a fraction of the previous iteration's weight change to the current weight change. This is shown in Eq. (A.31), where α is the momentum, $\Delta w(n-1)$ is the weight update calculated during the previous iteration, and $\Delta w(n)$ is the new weight update:

$$\Delta w_{pij}^{\ell}(n) = \eta \delta_{pj}^{\ell} o_{pij}^{\ell} + \alpha \Delta w_{pij}^{\ell}(n-1). \tag{A.31}$$

The momentum value should be in the range of 0 to 0.9, with 0.5 to 0.9 being common. Momentum affects how quickly learning accelerates or decelerates. Also,

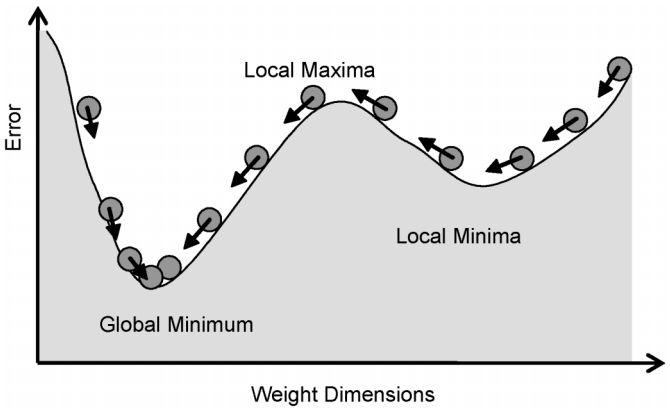


Figure A.2 The use of momentum is illustrated. Think of it as a heavy-ball analogy: A heavy ball rolling downhill will roll up and over a small hill, representing a local maxima.

it allows effective weight steps to be bigger, which often decreases training time by moving weights faster in the direction of prior changes.

Another way to improve performance is to add exponential smoothing [Sejnowski, 1987]. The update formula looks similar to momentum with a smoothing term, σ, used, as shown in Eq. (A.32):

$$\Delta w_{pij}^{\ell}(n) = (1-\sigma)\eta\delta_{pj}^{\ell}o_{pij}^{\ell} + \sigma\Delta w_{pij}^{\ell}(n-1). \tag{A.32}$$

A.2.3 Advantages and Disadvantages of Backpropagation

As we mentioned earlier, backpropagation is the most common algorithm used to train feedforward systems. It has several advantages, listed in Table A.3, and several disadvantages, listed in Table A.4. A modified version of backpropagation has also been developed by Tohru Nitta [Nitta, 1997] for use with complex numbers. In the next section, we will discuss alternatives to backpropagation for training feedforward networks.

A.3 Alternatives to Backpropagation

When backpropagation was becoming popular, computers were significantly limited by today's standards in terms of speed and memory. Several alternatives were

Table A.3 Advantages of the backpropagation algorithm for training feedforward networks.

- It is easy to use, with few parameters to adjust.
- The algorithm easy to implement.
- It is applicable to a wide range of problems.
- It is able to form arbitrarily complex nonlinear mappings (i.e., it is a universal approximator).
- It is popular and widely used for training feedforward networks as well as some recurrent networks.

Table A.4 Disadvantages of the backpropagation algorithm for training feedforward networks.

- There is an inability to know how to precisely generate any arbitrary mapping procedure.
- It is hard to know how many neurons and layers are necessary.
- Learning can be slow.
- New learning will overwrite old learning unless old patterns are repeated in the training process.
- It has no inherent novelty detection so it must be trained on known outcomes.

developed to improve training speed. Some techniques make minor modifications to the gradient-descent approach used in backpropagation. These alterations include changing the way weights are updated, using a different error function, or adding dynamic adjustments to the learning rate and momentum [Chan, 1987; Stornetta, 1987; Jacob, 1988; Vogel, 1988; Huang, 1990; Tollenaere, 1990; Riger, 1991; Sarkar, 1995; Bianchini, 1996; Dai, 1997]. Other techniques try to improve speed by using a second-order derivative [Becker, 1989; Battiti, 1992; Stäger, 1997], while still others use a combination of iterative and direct techniques to compute the weights [Verma, 1997]. The reader may recall that backpropagation uses the derivative of the error with respect to each weight, combined with a fixed step size, to adapt the weights. By using knowledge contained in the second derivative of the error with respect to a weight, the optimum step size could be computed as the network trained. Unfortunately, a second-order derivative of error with respect to weights results in a tensor of rank two that must be solved. A significantly different approach to feedforward training is to use evolutionary computation to train the weights [Fogel, 2000].

A.3.1 Conjugate Gradient Descent

The conjugate gradient-descent optimization technique was developed by Hestenes and Stiefel [Hestenes, 1952] and several enhancements have since been made [Fletcher, 1964]. As an optimization technique, the conjugate gradient descent can be applied to neural-network training by adapting weights as was done in backpropagation. Conjugate gradient descent can work with large numbers of weights, unlike several other alternatives to backpropagation [Stäger, 1997].

Conjugate gradient descent performs a series of line searches across the error surface. It determines the direction of steepest descent and then projects a line in that direction to locate the minimum, then makes an update in weights once per epoch. Another search is then performed along a conjugate direction from this point. This direction is chosen to ensure that all the directions that have been minimized stay minimized. It does this on the assumption that the error surface is quadratic. If the quadratic assumption is wrong and the chosen direction does not slope downward, it will then calculate a line of steepest descent and search that direction. Each epoch involves searching in a specific direction. This results in a search that does not generally follow the steepest descent, but it often produces a faster convergence than a search along the steepest descent direction because it is only searching in one direction at a time. As the algorithm moves closer to a minimum point, the quadratic assumption is more likely to be true and the minimum is then located quickly.

A.3.2 Cascade Correlation

The cascade-correlation neural network was developed by Scott E. Fahlman and Cristian Lebiere at Carnegie Mellon University [Fahlman, 1990] as a way to speed up the training of a feedforward network by training individual neurons one at a

time. Two concepts give rise to the name: cascading architecture and correlation training. The first involves candidate hidden neurons, which are added one at a time in a cascading sequence. The second concept is that the learning algorithm attempts to maximize the magnitude of the correlation between the candidate hidden neuron's output and the error term to be minimized. Variations and enhancements to cascade correlation exist. Prechelt reviewed some enhanced versions to this algorithm and compared them empirically [Prechelt, 1997].

The algorithm begins by constructing a minimal network with no hidden neurons. It minimizes the output error through use of the Widrow-Hoff delta rule [Widrow, 1960] or quick propagation [Fahlman, 1988]. It then creates a candidate hidden neuron. This candidate neuron is trained by maximizing the magnitude of the correlation between the candidate's output and the error term to be minimized. Gradient descent is used to minimize the network's output error, while a gradient ascent is employed to maximize the correlation. If the candidate neuron is successful, its weights are frozen and it is added to the network. That means this neuron does not learn any longer and its weights remain unchanged, so it becomes a permanent feature detector in the network. It is connected to all inputs and all other pre-existing hidden neurons. The algorithm automatically adds new hidden neurons one by one, so that the network is built up neuron by neuron, constructing its own structure, until the desired output error is achieved. This process continues until we are satisfied with the output error or have run out of time. Cascade correlation not only trains the network, but also configures the structure and neurons. Table A.5 lists the steps used in training a cascade-correlation network.

A.3.3 Second-Order Gradient Techniques

The backpropagation algorithm computes the Jacobian of the error with respect to the weights to find the best direction to adjust the weights, and then applies a fixed weight update step size, η, in that direction to adjust the weights. Second-order gradient techniques use the Hessian, $\partial^2 E/\partial w^2$, of the error with respect to the weights to adapt the step size in the direction of the optimal weight update. The second-order gradient techniques essentially start by trying to solve Eq. (A.33)

Table A.5 Cascade-correlation Training Procedure.

Step 1.	Initialize a feedforward network with no hidden neurons.
Step 2.	Train the network until a minimum mean square error is reached.
Step 3.	Add a candidate hidden neuron and initialize its weights.
Step 4.	Train the candidate hidden neuron and stop if the correlation between its output and the network output error is maximized.
Step 5.	Add the hidden candidate neuron to the full neural network by freezing its weights, connecting it to the other hidden neurons (if any), and connecting to the output neurons.
Step 6.	Train the full network that includes the new hidden neuron and stop when the minimum mean square error is reached.
Step 7.	Repeat 3–6 until the desired error is reached.

instead of Eq. (A.26), which relates to backpropagation:

$$\frac{\partial^2 E}{\partial w_{jk}^{l2}} \equiv 0. \tag{A.33}$$

Parker had proposed second-order gradients in his original formulation of backpropagation, but they took too long to compute [Parker, 1982]. To reduce the computational complexity of a second-order gradient, many techniques make some assumption that the error surface, or at least the region near a minimum, can be described by a second-order curve as shown in Fig. A.3. One approach developed in the late 1980s, quick propagation, assumes quadratic error surface, so that the second-order derivative is a constant while another approach tries to approximate a Newton descent (e.g., Quasi-Newton methods, Levenberg–Marquardt).

A.3.3.1 Quick Propagation

Quick propagation is a variation of the standard backpropagation algorithm developed by Scott Fahlman [1989]. It assumes the local is quadratic and employs an approximation to the second-order derivative of the quadratic to make weight changes. It has been shown to be faster than backpropagation for some applications, but is not generally faster than backpropagation. It also can get trapped in local minima or become unstable in a manner similar to backpropagation. For these reasons, it is not considered a general purpose method for training feedforward networks, but a specialized technique that can sometimes produce rapid training.

During the first epoch, the weights are updated through gradient descent, as in backpropagation. From then on, the quadratic error assumption is used. The quadratic calculation is approximated by a difference of gradients between the current epoch (n) and the previous epoch ($n-1$). It assumes that each weight is independent of all others during the weight update calculation. Equation (A.34) shows the basic weight update where $S(n)$ is the gradient, $\partial E/\partial w$, during the current

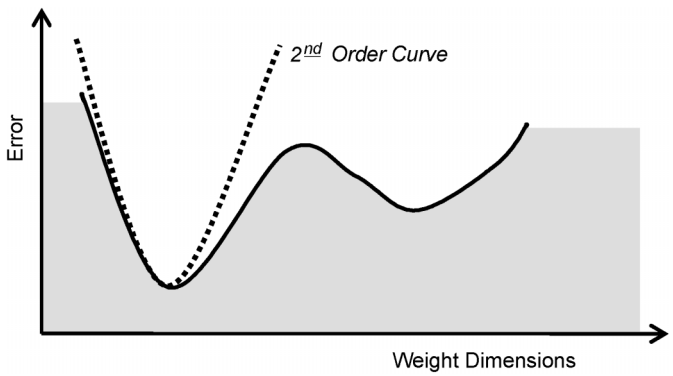


Figure A.3 A minimum can often be approximated by a second-order curve (i.e., parabola, quadratic).

iteration and $S(n-1)$ is the gradient from the previous iteration:

$$\Delta w(n) = \frac{S(n)}{S(n-1) - S(n)} \Delta w(n-1). \tag{A.34}$$

If the slope becomes constant, the gradient will be zero and there will be no updates. Also, if the slope gets steeper (i.e., $S(n) > S(n-1)$), then weight changes are proceeding away from the minimum. Therefore, a second weight update, as given in Eq. (A.35), must be used for all cases that are not trending downward. In this equation, a is an acceleration coefficient:

$$\Delta w(n) = a \Delta w(n-1). \tag{A.35}$$

A.3.3.2 Quasi-Newton

Quasi-Newton methods are popular algorithms for nonlinear optimization. They use second-order derivatives to find the optimal solution, so they generally converge faster than first-order techniques such as the gradient-descent method used in backpropagation. Quasi-Newton methods can be used to train feedforward networks as well, and they can be used in most configurations that work for backpropagation. However, their memory requirements and computation complexity scale as the square of the number of weights, so generally they are not suited for training networks with many weights.

Quasi-Newton assumes that the error surface is quadratic near a minimum, so that if it is close to a minimum, it can solve the minimization in one step through the use of second-order partial derivatives of the error surface with respect to the weights. The second-order derivatives are computed in a Hessian matrix, $\mathbf{H}$. The weight update is a product of the inverse Hessian matrix, $\mathbf{H}$, and the direction of the steepest descent, $\mathbf{g}$. Since it works on the average gradient of the error surface, a batch update of weights is performed at the end of each epoch:

$$\Delta w = -\mathbf{H}^{-1}\mathbf{g}. \tag{A.36}$$

Since determining the weight updates involves the use of a Hessian matrix with all the second-order partial derivatives, the computation is difficult and time consuming. By using approximations to the Hessian matrix, speed can be increased. The most efficient way of computing weight updates is through the Broyden–Fletcher–Goldfarb–Shanno (BFGS) formula [Broyden, 1970a; Broyden, 1970b; Fletcher, 1970; Goldfarb, 1970; Shanno, 1970]. The Davidon–Fletcher–Powell (DFP) algorithm is also efficient [Davidon, 1959; Fletcher, 1963]. In general, quasi-Newton techniques can become stuck in local minima more often than other optimization techniques, and the memory requirements scale as the square of the number of weights in the network.

Since the error surface is not really quadratic in most cases, the technique generates incorrect weight updates. Therefore, it must iteratively build up an approximation to the inverse Hessian. At the start of the process, the Hessian matrix, $\mathbf{H}$, is

initialized to the identity matrix, **I**. The algorithm then starts in the direction of the steepest descent, **g**, which is the same direction found in backpropagation. During each epoch, a backtracking line search is performed in the direction of the weight change. This approximation initially follows the line of steepest descent, then later follows the estimated Hessian more closely.

A.3.3.3 Levenberg–Marquardt

The Levenberg–Marquardt (LM) algorithm [Levenberg, 1944; Marquardt, 1963] is another nonlinear optimization algorithm based on the use of second-order derivatives. It has been adapted for use on training feedforward neural networks by Martin Hagan and Mohammad Menhaj [Hagan, 1994]. As with backpropagation, it computes weight changes. It is, however, more restricted than backpropagation. Like quasi-Newton methods, the memory requirements for the LM algorithm scale as a function of the square of the number of weights, so it is restricted to smaller networks, typically on the order of a few hundred weights. It also works only with summed squared error functions, so it is often used for estimation (i.e., regression) applications [Masters, 1995; Bishop, 1995].

The LM algorithm is a combination of the features of gradient descent found in backpropagation and the Newton method. It assumes that the underlying function being modeled is linear and that the minimum error can be found in one step. It calculates the weight change to make this single step. It tests the network with these new weights to determine whether the new error is lower. A change in weights is only accepted if it improves the error. When the error decreases, the weight change is accepted and the linear assumption is reinforced by decreasing a control parameter, μ. When the error increases, the weight change is rejected and, like backpropagation, it follows a gradient descent by increasing the control parameter to de-emphasize the linear assumption. In this way, the LM algorithm is a compromise between a Newton and gradient-descent process. Near a minimum, the linear assumption is approximately true, so the LM algorithm makes very rapid progress by using this second-order Newton-like feature. Away from the minimum, the linear assumption is often bad, but since it uses gradient descent when error doesn't improve, it still will converge to a minimum error. The process is repeated until the desired error or maximum number of iterations is reached.

The LM algorithm approximates the Hessian matrix used in the quasi-Newton method as the product of a Jacobian matrix of the first-order partial derivatives, with its transpose as shown in Eq. (A.37). Since it uses a Jacobian matrix, **J**, instead of a Hessian matrix, **H**, the calculation is easier:

$$\mathbf{H} \approx \mathbf{J}^T \mathbf{J}. \tag{A.37}$$

The gradient is computed as the product of the Jacobian containing the first-order partial derivatives and a vector, **e**, that contains the errors being minimized:

$$\mathbf{g} = \mathbf{J}^T \mathbf{e}. \tag{A.38}$$

Table A.6 Levenberg–Marquardt Training Procedure.

Step 1.	Initialize Weights $w_{ij}^{\ell} =$ Uniformly Random Over $-\varepsilon$ to ε for all i, j, and ℓ.
Step 2.	Present each pattern to the input of the network.
Step 3.	Propagate data forward and generate the output pattern. Calculate the error between the target output and the actual output.
Step 4.	If there are more patterns (i.e., $p < P$) in the training set, loop back to Step 2.
Step 5.	Now calculate the error vector, **e**, between target and actual output for all patterns presented by using summed squared error as in Eq. (A.8).
Step 6.	Compute the Jacobian matrix, **J**, from the first-order partial derivatives.
Step 7.	Compute the weight update as given in Eq. (A.36).
Step 8.	Recalculate the sum of squared errors. If the new error is lower, reduce μ by a factor β, update the weights by Δw, and go to Step 9. If the new error is higher, increase μ by β and go back to Step 7.
Step 9.	If the norm of the gradient, **g**, is less than the desired amount, stop; otherwise loop back to Step 1.

This gives us a weight-update formulation in Eq. (A.39), where $\mathbf{I}$ is the identity matrix and μ is the control parameter:

$$\Delta w = -(\mathbf{J}^T\mathbf{J} + \mu\mathbf{I})^{-1}\mathbf{J}^T\mathbf{e}. \tag{A.39}$$

From this equation, it is shown that if μ is 0, then this is a Newton method with an approximated Hessian matrix. Larger values of μ make it look more like a gradient-descent method.

A.3.4 Evolutionary Computation

Evolutionary computation mimics the processes of biological evolution, using random variation and natural selection to provide effective solutions for optimization problems. This approach is significantly different from other search and optimization approaches, and has been successfully applied to search and optimization problems for which other approaches have failed. It has proven to be well suited to the optimization of specific nonlinear multivariable systems, and can be useful in training neural networks. Several paradigms in evolutionary computation were developed separately, including genetic algorithms, evolutionary programming, and evolution strategies.

These techniques usually consist of a set of candidate solution states that are generated. This is called the population. A fitness function is used to evaluate each solution in the population. The parameters encoding the solutions may be broken apart and recombined with variation operators such as cross-over, mutation, and forms of recombination to form new solutions (i.e., offspring), which may be more fit than their parents in the previous iteration (i.e., generation). The process is repeated until an acceptable solution is found within specific time constraints.

Evolutionary computation has existed since the late 1940s [Fogel, 1998]. The main branches developed independently during the 1960s. Lawrence Fogel developed evolutionary programming, which used mutations to evolve populations of finite state machines, to solve optimization and prediction problems [Fogel, 1962]. John Holland developed the genetic algorithm (GA) for use in adaptive robust systems [Holland, 1962]. This employed the concept of solution states encoded as binary-valued strings, where a bit is analogous to a gene and the string is analogous to a chromosome, and used cross-over and mutation. Earlier, Alex Fraser [Fraser, 1957] developed the genetic algorithm for modeling genetic systems, and Hans Bremmermann [Bremmermann, 1962] invented genetic algorithms for function optimization. Ingo Rechenberg and Hans-Paul Schwefel developed evolution strategies for parameter optimization [Rechenberg, 1973; Schwefel, 1965], which incorporates many features of the genetic algorithm but uses real-valued in place of binary-valued parameters.

A simple evolutionary-computation approach for training a neural network, based on the discussion in David Fogel's book [Fogel, 2000], is given in Table A.7. Basically, several neural networks are generated with a traditional training algorithm. This forms the initial population, with each network representing a parent in the population. Random variations are applied to the networks to produce offspring networks. These offspring are competitively evaluated and the best performers become the new parents in the next generation. This process continues until a network is found that has the desired outcome.

Table A.7 Evolutionary-computation Training Procedure.

Step 1.	Select a neural-network architecture specifying the number of inputs and outputs.
Step 2.	By using the training data, train several neural networks to produce a population of trained networks with a training algorithm of your choice. Each network can have different numbers of hidden neurons, hidden layers, learning rates, momentums, or other control parameters.
Step 3.	Use the test data to evaluate the networks in the population of trained networks. A common evaluation function is the standard mean squared error between the target output and the actual output that we used with backpropagation.
Step 4.	Create offspring from the parents by randomly varying the number of neurons and layers and/or the neurons' weights and biases. A probability distribution function will determine the likelihood of selecting specific combinations. When new neurons are added, their weights can be initialized by random numbers as shown in Step 1 of backpropagation.
Step 5.	Evaluate the offspring neural networks with test data, as in Step 3.
Step 6.	Conduct a series of competitions between paired networks to determine the relative worth of each proposed network by randomly selecting pairs of offspring networks and evaluating their performance. The better network of the two is marked as the winning network. Repeat this process several times and the networks with the most wins become the parents of the next population.
Step 7.	If the desired result has been obtained; otherwise, return to Step 4.

Appendix B. Feature Saliency

When designing a classifier, the developer is always concerned with which features to use to solve a particular classification or mapping problem. In this appendix, we will present one method in which the weights of a trained neural network can be used to discover which features are important. We begin by asking how, if you were the feedforward network, you could reduce the effect of a "bad" feature on the network. The logical answer is to drive the weights tied to the "bad" feature towards zero, so that it has no contribution to any neuron tied to it in the layer above it. Similarly, you would increase the weights tied to a "good" feature so its effect would be greater. Thus, you may decide to define a saliency metric such as the following:

$$\Lambda_i = \sum_{j=1}^{N^\ell} \left| w_{ji}^\ell \right|, \tag{B.1}$$

where $\ell \equiv$ layer above the neuron of interest, $i \equiv$ neuron index on lower layer, $j \equiv$ neuron index on upper layer, $w_{jl}^\ell \equiv$ weight between neuron i on lower layer $(\ell - 1)$ and neuron j on upper layer (ℓ), $N^\ell \equiv$ number of neurons on layer ℓ.

You could try many other metrics, but this one is simple to compute and yields excellent results. The metric in Eq. (B.1), while simple, is in fact related to the Bayesian nature [Ruck, 1990b; Priddy, 1993] of feedforward neural networks. Ruck showed that the outputs of a feedforward neural network, given a sufficient number of hidden units to model the underlying probability density functions in the training data, approximate the *a posteriori* probabilities for each class in the training set, as depicted in Eq. (B.2):

$$z_j = P(C_j \mid \mathbf{x}), \tag{B.2}$$

where $z_j \equiv$ output of node j on the output layer, $P(C_j \mid \mathbf{x}) \equiv$ the *a posteriori* probability of class C_j given input vector $\mathbf{x}$, $\mathbf{x} \equiv$ input feature vector.

From Bayesian statistics we know that

$$\sum_j z_j = \sum_j P(C_j \mid \mathbf{x}) = 1. \tag{B.3}$$

Thus, the probability of an error for a given output neuron is simply

$$P_{error}(j, \mathbf{x}) = 1 - \sum_j P(C_j \mid \mathbf{x}) = \sum_{k \neq j} z_k. \tag{B.4}$$

By taking the partial derivative of the summed outputs $\sum_{k \neq j} z_k$ with respect to a given input feature, x_i, we can measure how sensitive the output is to that feature. Thus, we can directly measure the saliency, or relevance, of any given feature. Now, let's take the derivative of the probability of error with respect to a given input feature to measure its saliency:

$$\frac{\partial P_{error}(j, \mathbf{x})}{\partial x_i} = \frac{\partial}{\partial x_i} \sum_{k \neq j} z_k. \tag{B.5}$$

The output, z_k, is given by

$$z_k = f_k^{\ell} \left(\sum_{i=1}^{N^{\ell-1}} w_{ki}^{\ell} o_i^{\ell-1} + w_{k0}^{\ell} \right), \tag{B.6}$$

where $\ell \equiv$ the layer above a weight, $\ell - 1 \equiv$ the layer below a weight, $w_{ki}^{\ell} \equiv$ the weight from node i on layer $\ell - 1$ to node k on layer ℓ, $f_k^{\ell}(\,) \equiv$ the transfer function of neuron k on layer ℓ, $o_i^{\ell-1} \equiv$ the output of neuron i on layer $\ell - 1$, which, for a feedforward network with two hidden layers, would be

$$z_k = f_k^3 \left(\sum_{i=1}^{N^2} w_{ki}^3 o_i^2 + w_{k0}^3 \right). \tag{B.7}$$

Note that in Appendix A we defined the transfer function for a sigmoid logistic function to be

$$f(u) = \frac{1}{1 + e^{-u}}. \tag{B.8}$$

The derivative of the logistic function is given as

$$\frac{\partial f(u)}{\partial u} = \frac{\partial}{\partial u} \left(\frac{1}{1 + e^{-u}} \right) = \frac{e^{-u}}{(1 + e^{-u})^2} = f(u)[1 - f(u)]. \tag{B.9}$$

Substituting Eq. (B.7) into Eq. (B.5) yields

$$\frac{\partial P_{error}(j, \mathbf{x})}{\partial x_i} = \frac{\partial}{\partial x_i} \sum_{k \neq j} f_k^3 \left(\sum_{i=1}^{N^2} w_{ki}^3 o_i^2 + w_{k0}^3 \right). \tag{B.10}$$

We now define a new term δ_k^3 to help with subsequent calculations:

$$\delta_k^3 \equiv z_k(1 - z_k). \tag{B.11}$$

Continuing the process yields

$$\frac{\partial P_{error}(j, \mathbf{x})}{\partial x_i} = \sum_{k \neq j} \delta_k^3 \cdot \sum_m w_{km}^3 \cdot \frac{\partial}{\partial x_i}(o_m^2), \tag{B.12}$$

which, when you reach the input layer (layer 0), yields

$$\frac{\partial P_{error}(j, \mathbf{x})}{\partial x_i} = \sum_{k \neq j} \delta_k^3 \cdot \sum_m w_{km}^3 \delta_m^2 \cdot \sum_n w_{mn}^2 \delta_n^1 \cdot w_{ni}^1. \tag{B.13}$$

We now define a saliency metric, Ω_i based on the magnitude of the changes in P_{error} as the feature x_i is varied over its range of values. The saliency can be calculated as follows: Take each training vector ($\mathbf{x}$) in the training set S. For the i^{th} feature, sample at various locations over the expected range of the feature while holding all other features ($\mathbf{x}$) constant. Now compute the L1 norm of the partial derivative of the output z_k for the sample values. Sum the L1 norm over the output nodes (j), the sampled values of $\mathbf{x}(D_i)$, and the training set S, which yields

$$\Omega_i = \sum_j \sum_{\mathbf{x} \in S} \sum_{x_i \in D_i} \left| \frac{\partial P_{error}(j, \mathbf{x})}{\partial x_i} \right|, \tag{B.14}$$

which, by substituting Eq. (B.4) into Eq. (B.14), can be rewritten as

$$\Omega_i = \sum_j \sum_{\mathbf{x} \in S} \sum_{x_i \in D_i} \left| \sum_{k \neq j} \frac{\partial z_k}{\partial x_i} \right|. \tag{B.15}$$

Because z_k is the output of the logistic function, we know each $\partial z_k / \partial x_i$ exists and is finite. Therefore, we can invoke the triangle inequality to obtain

$$\Omega_i = \sum_j \sum_{\mathbf{x} \in S} \sum_{x_i \in D_i} \left| \sum_{k \neq j} \frac{\partial z_k}{\partial x_i} \right| \leq \sum_j \sum_{\mathbf{x} \in S} \sum_{x_i \in D} \sum_{k \neq j} \left| \frac{\partial z_k}{\partial x_i} \right| \leq \text{Constant}. \tag{B.16}$$

Now substituting Eq. (B.13) into Eq. (B.16), we obtain

$$\Omega_i \leqslant \sum_j \sum_{\mathbf{x} \in S} \sum_{x_i \in D} \sum_{k \neq j} \left| \delta_k^3 \cdot \sum_m w_{km}^3 \delta_m^2 \cdot \sum_n w_{mn}^2 \delta_n^1 \cdot w_{ni}^1 \right| \leq C. \tag{B.17}$$

Invoking the triangle inequality again yields

$$\Omega_i \leqslant \sum_j \sum_{\mathbf{x}\in S} \sum_{x_i \in D} \sum_{k\neq j} \left| \delta_k^3 \cdot \sum_m w_{km}^3 \delta_m^2 \cdot \sum_n w_{mn}^2 \delta_n^1 \cdot w_{ni}^1 \right| \tag{B.18}$$

$$\Omega_i \leqslant \sum_j \sum_{\mathbf{x}\in S} \sum_{x_i \in D} \sum_{k\neq j} \delta_k^3 \cdot \sum_m \left| w_{km}^3 \right| \delta_m^2 \cdot \sum_n \left| w_{mn}^2 \right| \delta_n^1 \cdot \left| w_{ni}^1 \right| \leq C. \tag{B.19}$$

While this seems like a great deal of work, what is really important is that we can now clearly see the following:

$$\Omega_i \propto \sum_n \left| w_{ni}^1 \right| = \Lambda_i, \tag{B.20}$$

which is the L1 norm of the weight vector, $\mathbf{w}_i$, emanating from the feature, x_i, in the input vector, $\mathbf{x}$. Thus, the saliency metric we proposed based on intuition, Λ_i, is related to the Bayes probability of error when the network is used as a classifier.

We caution the reader that this derivation involves several implicit assumptions that must be considered whenever this saliency metric is employed: (1) the training set S is representative of all of the conditions expected for the input feature space; (2) the features of vectors in S all have the same range of values, which means a statistical norm or max-min norm must be applied to the data to create S; (3) the network is of sufficient size and capacity to ensure the underlying probability density functions for each class can be adequately modeled by the network; (4) the saliency metric is computed over a fairly large number of networks by using the same topology, but different initial weights and presentation orders for the training vectors in S.

Once the multiple training runs have been completed and the features rank ordered for each run, a histogram is formed for each feature, as shown Table B.1.

Table B.1 Histogram of each feature for a classification problem after training 100 feedforward networks with identical architectures but different starting weights.

	Importance								
	9th	8th	7th	6th	5th	4th	3rd	2nd	1st
Feature 1	0	0	0	0	0	2	4	19	75
Feature 2	15	17	13	8	12	20	9	5	1
Feature 3	4	4	8	4	19	20	24	15	2
Feature 4	24	9	11	23	15	10	7	0	1
Feature 5	16	24	12	19	8	11	8	2	0
Feature 6	24	17	23	15	19	9	0	2	0
Feature 7	14	18	19	10	21	6	10	2	0
Feature 8	2	6	13	16	8	11	18	24	1
Feature 9	1	5	1	5	7	11	19	31	20

Once the histogram is formed, the features can be ranked using a metric, such as

$$\text{Feature\_rank} = \sum_{n=1}^{N} K_n \cdot (N - n - 1), \tag{B.21}$$

where $K_n \equiv$ Count for importance n for the feature of interest, $n \equiv$ the importance value for a given feature (best $= 1$, worst $= N$), $N \equiv$ Total number of features in the input vector ($\mathbf{x}$).

Applying the formula of Eq. (B.21) to the results in Table B.1 yields the following results for each feature tabulated in Table B.2.

Observing Table B.2, the reader will find that the ranks of features 1 and 9 are high and fairly distinct from the other features. The ranks of features 4, 5, 6, and 7 are consistently low, with feature 7 being the worst. The remaining features, 2, 3, and 8, are in between. Thus, more than likely features 4, 5, 6, and 7 can be removed with no loss of performance. For the classification problem that generated the feature ranking shown in Table B.2, the top three features (1, 9, 3) performed as well as all nine features, resulting in a much smaller network to solve the classification problem.

Table B.2 Feature rankings for histograms of Table B.1.

	Rank Score	Overall Rank
Feature 1	767	1
Feature 2	312	5
Feature 3	473	3
Feature 4	260	8
Feature 5	254	6
Feature 6	207	9
Feature 7	274	7
Feature 8	457	4
Feature 9	596	2

Appendix C. Matlab Code for Various Neural Networks

C.1 Matlab Code for Principal-Components Normalization

```
function [C, eigvecs, lambdas] = pca(A,N)
%
%usage:
% [C, eigvecs, lambdas] = pca(A,N)
%
%
%   A – The input data array of feature vectors
%   N – the number of eigenvalues/eigenvectors to return and can be left
%       blank.
%   C – The transformed data array obtained from a projection of A onto the
%       N eigenvectors of the cov(A).
% eigvecs – The N eigenvectors of the cov(A) sorted from largest to smallest in
%           column-major order
% lambdas – The sorted N eigenvalues from largest to smallest.
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% pca.m
%
% Simple Principal Components Analysis
%
% Written By Kevin L. Priddy
% Copyright 2004
% All Rights Reserved
%
% This instantiation returns the transformed data array obtained from a
% projection of A onto the N eigenvectors of the cov(A). If N is missing, N
% is set to min(size(A)).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
B = cov(A) % Compute the covariance of the input array
if exist('N','var')
   if (N > size(B,1)) % Make sure that we don't have too many eigenvalues
      N = size(B,1);
   end
else
   N = size(B,1); % Make sure that we have the right number of eigenvalues
end
[V D] = eig(B);
[rows cols] = size(D);
% Check to see if the result is in descending order and reorder N eigenvectors
% in descending order
if D(1,1) < D(rows,rows) % If true sort the array in descending order
   for ii = 1:N
      eigvecs(:,ii) = V(:,rows-(ii-1));
      lambdas(ii,1) = D(rows-(ii-1),rows-(ii-1));
   end
else
   eigvecs = V(:,1:N);
   for ii = 1:N
      lambdas(ii,1) = D(ii,ii);
   end
end

% Project the original data onto the first N eigenvectors of the covariance matrix.

C = A*eigvecs;
```

C.2 Hopfield Network

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% hop.m
%
% usage: [result, x] = hop(n,u,W)
%
%
% result is the predicted solution
% x is the predicted x values for each iteration through the Hopfield network
% n is the number of allowed iterations
% u is the initial input to the network
% W is the Weight matrix
%
% called by glnn.m
%
% Author: Kevin L. Priddy
% Copyright 2004
% All rights reserved
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function [result, x] = hop(n,u,W)
x(:,1) = W*u;
for ii = 2:n
        x(:,ii) = W*x(:,ii-1) + u;
end
result = x(:,n);
```

C.3 Generalized Neural Network

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% usage: [result, x] = glnn(A,y,n)
%
% glnn.m
%
% A is the data matrix
% y is the set of desired outputs
% x is the solution to the problem
% n is the number of allowed iterations
%
% calls the lin_hop function included in glnn.m
%
% Author: Kevin L. Priddy
% Copyright 2004
% All rights reserved
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [result, x] = glnn(A,y,n)


% full rank and overdetermined cases


[rows cols] = size(A);
if rows >= cols
  alpha = 1/(trace(A'*A)); % 0 < alpha < 2/(trace(A'*A)) to ensure convergence
  W = eye(size(A'*A,2) )-alpha*(A'*A);
  u = alpha*A'*y;
  [result, x] = lin_hop(n,u,W);


else

% underdetermined case

      alpha = 1/(trace(A*A')); % 0 < alpha < 2/(trace(A*A')) to ensure
         convergence
      W = eye(size(A*A',2) )-alpha*(A*A');
```

```
   Wff = alpha*A'*y;
         [result, xp] = lin_hop(n,y,W);
   x = Wff*xp;
end

function [result, x] = lin_hop(n,u,W)
x(:,1) = W*u;
for ii = 2:n
         x(:,ii) = W*x(:,ii-1) + u;
end
result = x(:,n);
```

C.4 Generalized Neural-Network Example

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% glnn_example.m
%
% Demonstrates how the GLNN can solve a set of simultaneous equations
% A is the data matrix
% y is the set of desired outputs
% result is the glnn solution to the problem
% n is the number of allowed iterations
% x is the predicted x values for each iteration through the Hopfield
network
%
% calls the glnn.m function
%
% Author: Kevin L. Priddy
% Copyright 2004
% All rights reserved
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


clear
n = 500; % Set iterations to 500

A = [–4.207 1.410 0.451 –0.910;
        –0.344 –3.473 2.380 3.267;
        3.733 –1.999 –3.728 2.850;
        1.096 –4.277 1.538 –3.952];

y = [0.902; –27.498; –0.480; 3.734];

x_act = [1; 2; –3; –4];

[result, x] = glnn(A,y,n);

% plot the mean squared error for x as a function of n

for ii = 1:n
```

```
   mse(ii) = ((x(:,ii)-x_act)'*(x(:,ii)-x_act))/size(x,1);
end

close all
figure(1)
semilogy(mse, 'r', 'linewidth', 2);
grid on
ylabel('Mean Squared Error');
xlabel('Iterations through Hopfield Network');
title('Mean Squared Error for the Estimated Values of x');

x_act
result
delta = x_act-result
```

C.5 ART-like Network

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% usage neuron = artl(outfile, A, num_nodes, vigilence);
%
% ARTL(outfile, A, num_nodes, vigilence)
% Computes an ART-Like network using floating point numbers
% A is the data matrix
% nodes is the total number of neurons in the F2 Layer
% vigilence is the fitness of a normalized vector to the winning node.
%
% Author: Kevin L. Priddy
% Copyright 2004
% All rights reserved
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function neuron = artl(outfile, A, num_nodes, vigilence);

% First Set up some default variables

[numvecs vecsize] = size(A);

%seed = 12345;
neuron = zeros(nodes,vecsize);
node_wins = zeros(num_nodes,1);
count = 1;
flag = 0;
for ii = 1:numvecs
   temp = A(ii,:);
   temp = temp/norm(temp);

   if(flag == 0) % Set the first neuron to be the first normalized vector in the set
```

```
    neuron(1,:) = temp;
    flag = 1;
    node_wins(1) = 1;
    win_node(ii,1) = 1;
  else
    % Find the winning neuron


    % Set the initial winner to be the first F2 neuron
    winner = 1;
    win_val = temp*neuron(1,:)';

    % Now compare the input to the rest of the F2 neurons
    for jj = 2:count
        dotprod = temp*neuron(jj,:)'
        if(dotprod > win_val)
          winner = jj;
          win_val = dotprod;
        end
    end
    node_wins(winner) = node_wins(winner) + 1;

    % Compare with Vigilence—small number means need a new neuron
    if(win_val < vigilence)
      if(count < num_nodes)
        count = count + 1;
        neuron(count,:) = temp;
        node_wins(count) = 1;
        win_node(ii,1) = count;
      else
        error('You are out of neurons. Decrease vigilence and try again.');
      end
    else
      % adjust centroid of winning neuron. We will weight the shift by the previ-
        ous
      % number of hits for a given winning neuron and then renormalize.
      neuron(winner,:) = temp/(node_wins(winner)) + neuron(winner,:)*
                          ((node_wins(winner)-1)/(node_wins(winner)));
      % renormalize the winner
      neuron(winner,:) = neuron(winner,:)/norm(neuron(winner,:));
      win_node(ii,1) = winner;
    end

  end

end
result = [A win_node];

% Save the result into a file
```

```
save(outfile, 'neuron', 'result', '-ascii');

% Note that the norm(X) function is simply the L2 or Euclidean norm of X.
% In C it would be:
%
% *float norm(*float x, int length)
% {
% int i;
% *float temp, tempval;
% tempval = 0.0;
% for (ii = 0; ii < length-1; ii++)
%    {
%    tempval + = x[ii]*x[ii];
%    }
% tempval = sqrt(tempval);
% for (jj = 0; jj < length-1; jj++)
%    {
%    temp = x[ii]/tempval;
%    }
% return(temp);
% }
```

C.6 Simple Perceptron Algorithm

```
function [W, count] = perceptron(InArray, Desired_Output, eta)
%
%usage:
% [W, count] = perceptron(InArray, Desired_Output, eta)
%
%       InArray – the set of input feature vectors used for training (N vectors *
%             M input features) Desired_Output – The associated output
%             value(-1,1) for a given feature vector (N vectors x 1 output)
%
%       eta – The desired weight update step size (usually set in the
%             range (0.1,1))
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% perceptron.m
%
% Simple Perceptron Program
%
% Written By Kevin L. Priddy
% Copyright 2004
% All Rights Reserved
%
% This instantiation updates the weight vector whenever a feature vector
% presentation produces an error
% Execution stops when there is a hyperplane that produces no errors over
```

```
% the training set
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate the bias term by augmenting each input with a 1
InArray = [InArray ones(size(InArray,1),1)];

% note we've included the bias term in the number of features

[num_vecs num_features] = size(InArray);
bias = 0;
W = zeros(1,num_features);  % Include the bias  term as a weight
error_flag = 1
count = 1;
while error_flag == 1          % Keep adjusting the weights until there are no errors
   error = 0;
   for ii = 1:num_vecs         % Run through the set of input vectors
      invec = [InArray(ii,:)']; % Get an input vector
      out(ii,1) = W*invec;      % Compute the perceptron output and apply the
signum transfer function
      if out(ii,1) >= 0
         out(ii,1) = 1;
      else
         out(ii,1) = -1;
      end

      if out(ii,1) ~= Desired(ii,1) % Update the weights each time an error occurs
         W = W - eta*invec';
         error = error + (out(ii,1) - Desired(ii,1));
      end
   end

   total_error(count) = error   % Update the error for plotting when finished

   if total_error(count) == 0   % Check for NO ERROR case
         error_flag = 0;        % breaks the while loop
   end
      count = count + 1         % update the counter
end

figure(1)
plot(total_error)               % Plot the total error versus count
```

C.7 Kohonen Self-Organizing Feature Map

```
% SOFM(outfile, A,Wrows, Wcols, nh_size, eta, iterations,num_maps)

% Computes a two-dimensional Kohonen Self-organizing Map using square neigh-
borhoods
%
% Written by: Kevin L. Priddy
```

```
% Copyright 2004
% All rights reserved
%
% A is the data matrix (M rows by N features) where M is the total number
% of feature vectors
% Wrows is the number of rows in the SOFM
% Wcols is the number of cols in the SOFM
% nh_size is the starting neighborhood size (nh_size x nh_size)
% eta is the stepsize for the weight update
% iterations is the total number of iterations to be performed
% num_maps is the number of maps to be stored for the total number of iterations

function W = sofm(outfile, A, Wrows, Wcols, nh_size, eta, iterations,num_maps);

% First Set up some default variables

[numvecs vecsize] = size(A);
A_max = max(max(A));
A_min = min(min(A));
map = zeros(10*Wrows*Wcols,vecsize);
% set up initial weight ranges for SOM. Note that we will be
% using the row and column indexing scheme to index each
% row vector that corresponds to the weights from a given node in the
% SOM to the input array

%seed = 12345; % Select a seed if desired

% Spread weights throughout data cloud
W = rand(Wrows*Wcols,vecsize)*(A_max-A_min) + A_min;

%Make sure neighborhood size is odd
if((nh_size > 1)&(mod(nh_size,2) == 0))
   nh_size = nh_size-1;
end
% Calculate until iterations are used up
count = 1;
map_count = 1;
for m = 1:5
   for n = 1:floor((iterations/5) + 0.5)
      %grab an input exemplar from the data set
      temp = A(ceil(rand*(numvecs-1) + 0.5),:);
      %now find closest neuron in SOM to input
      min_D = (W(1,:)-temp)*(W(1,:)-temp)';
      win_row = 1;
      win_col = 1;

      for i = 1:Wrows
         for j = 1:Wcols
         %Compute Euclidean Distance
         D = (W((i-1)*Wcols + j,:) - temp)*(W((i-1)*Wcols + j,:) - temp).';
         if (min_D > D) %Check if D is new winner for W(i,j)
```

```
            win_row = i;
            win_col = j;
            min_D = D;
        end
    end
end
% now we update the winner and everyone in the neighborhood
row_min = win_row-floor(nh_size*0.5);
if row_min < 1
    row_min = 1;
end
col_min = win_col-floor(nh_size*0.5);
if col_min < 1
    col_min = 1;
end
row_max = win_row + floor(nh_size*0.5);
if row_max > Wrows
    row_max = Wrows;
end
col_max = win_col + floor(nh_size*0.5);
if col_max > Wcols
    col_max = Wcols;
end
for i = row_min:row_max
    for j = col_min:col_max
            %win_row
            %win_col
            %temp
            %old_wts = W((i-1)*Wcols + j,:)
            delta = eta * (temp - W((i-1)*Wcols + j,:));
            %delta
            %update the node in the neighborhood
            W((i-1)*Wcols + j,:) = W((i-1)*Wcols + j,:) + delta;
    end
end
if(mod(count,floor(iterations/num_maps)) == 0)
        for ii = 1:Wrows
            for jj = 1:Wcols
                map((map_count-1)*Wrows*Wcols + (ii-1)*Wcols + jj,:) =
                W((ii-1)*Wcols + jj,:);
            end
        end
        map_count = map_count + 1
    end
    count = count + 1;
end
%decrement eta and neighborhood  size
eta = eta*0.8;
```

```
    if nh_size > 1
        nh_size = nh_size - 2;
    end
end
save(outfile, 'map', '-ascii');
```

Appendix D. Glossary of Terms

Activation function: A mathematical function that determines the output signal level of a processing element (neuron) from the input signal levels. It is similar to the firing rate in biological systems.

ANN: artificial neural network.

ART: adaptive resonance theory.

Artificial intelligence (AI): Any computation method based on some understanding of how human intelligence works.

Association neuron: A biological neuron found in the brain or spinal chord that performs computational tasks.

Axon: The long part of a neuron leading away from the cell body. It transmits impulses from one neuron to other neurons.

Backpropagation: Also known as backpropagation of error. This is one of many learning algorithms used to train neural networks and is currently the most widely used algorithm. During training, information is propagated forward through the neural network. The output response is compared to a desired (or target) response. The observed error is propagated backwards through the network and used to make small adjustments in the synaptic connections. This gives rise to its name. The network is trained by repetitively applying this algorithm with a large number of labeled patterns until the output error is minimized.

Backprop: *See* **backpropagation**.

BAM: *See* **bidirectional associative memory**.

Bayes optimal classifier: A classifier that selects the Bayes optimal decision boundary based upon the application of *a priori* information. For example, if the classifier is solving a two-class problem where $P(\text{class1}) + P(\text{class2}) = 1$, then the optimal decision rule would be to choose the output of the classifier to be class 1 if $P(\text{class1}) \geq P(\text{class2})$ and class 2 otherwise. Costs are often associated with making a wrong decision that will influence the decision boundary as well.

Bayes rule: The application of *a priori* probability, e.g., $P(A)$, information to predict *a posteriori* probabilities, e.g., $P(B \mid A)$, of occurrence. Bayes rule is often

used in classification and is defined as

$$P(A \mid B) = \frac{P(A)P(B \mid A)}{P(B)}.$$

Bias: A value used to shift the neuron's firing threshold. In artificial systems, it is an offset to the activation function. Typically, each neuron has its own bias, which is often set by the training algorithm. The term *bias* means something different in statistics terminology. *See* **statistical bias**.

Bidirectional associative memory: Bidirectional associative memory (BAM) is an outer product memory, which can include hetero-associative mappings. For example, the BAM can map an image of a cat to an image of the word cat. Like the Hopfield network when used for pattern association, the BAM will try to map any input to one of its memory patterns. Thus, caution is needed when presenting patterns to the BAM.

CAM: content addressable memory.

Classifier: A system, e.g., neural network, that assigns data to a predefined set of labeled categories.

Clusterer: A system, e.g., neural network, that groups data into groups or clusters.

Confusion matrix: A tabular method of comparing the neural network's output classification to the desired classification. It shows how the classifier confuses the classes. All off-diagonal terms indicate misclassifications. A perfect classifier would have no non-zero off-diagonal terms.

Connection: In an artificial system, a link or pathway between two processing elements (neurons) used to transfer information from one to the other. Generally, a connection has a weight associated with it. The output signal of one neuron is multiplied by this weight as it is fed into the other neuron. This is analogous to a synapse in a biological system.

Connectionist model: Synonym for an artificial neural network.

Confusion matrix: A tabular method of comparing the neural network's output classification to the desired classification.

Credit assignment problem: The process of determining which weight should be adjusted to effect the change needed to obtain the desired system performance.

Cross-validation: A statistical technique used to estimate generalization through the use of subsets of the collected data to maximize its use.

Dendrite: A highly branched, tree-like structure at one end of a neuron that brings impulses (signals) from other neurons into the neuron's cell body.

Epoch: This term refers to a complete pass during which all the training data are presented to the learning algorithm.

Estimator: A system, e.g., neural network, that produces an approximated output based on input data.

Excitatory connection: A connection between neurons that increases a neuron's ability to fire. In artificial systems, this is usually a connection with a positive weight value.

Fault tolerance: The ability of a neural network to function within acceptable limits when some neurons or synapses are malfunctioning.

Feedback network: A neural network in which each neuron can take information from any neuron in the network, including itself.

Feedforward network: A neural network composed of layers. The inputs of the neurons in one layer come from the outputs of neurons in a previous layer.

Function approximator: *See* **estimator**.

Generalization: A neural network's ability to learn the basic structure of the data presented to it without memorizing the data, so that it can produce the same output from similar examples that it has never seen. This is similar to a person's ability to examine known information, draw conclusions, and apply these conclusions to similar but unknown information.

Genetic algorithm (GA): A specific class of evolutionary computation algorithms that performs a stochastic search by using the basic principles of natural selection to optimize a given objective function where parameters are encoded in something analogous to a gene.

GLNN: Generalized linear neural network.

Gradient descent: An optimization approach used in the backpropagation algorithm to change the weights in the network. On an error surface, changes in the output error with respect to the changes in the synaptic weights descend the steepest path toward the point of minimum error.

Hebb's Rule: Connections to a neuron that tend to produce a desired output are strengthened, while connections that tend to produce an undesired output are weakened.

Hidden layer: A layer of processing elements (neurons) that is hidden from direct connections to points outside of the neural network. All connections to and from a hidden layer are internal to the neural network.

Hopfield network: The Hopfield neural network includes a single layer of fully interconnected neurons with the inputs and outputs also connected. The Hopfield network can be used to perform auto-associative mappings, to solve optimization problems, and to solve solutions to sets of equations. When used for pattern association, the Hopfield network will try to map whatever input is provided to one of its memory patterns. Thus, caution is needed when presenting patterns to the network.

Hyperbolic tangent: A trigonometric function used as an activation or transfer function. It has a sigmoidal shape that is commonly used with the networks trained by backpropagation. The function can be mathematically written as

$$f(x) = \tanh\left(\frac{x}{2}\right) = \frac{1 - e^{-x}}{1 + e^{-x}}.$$

Identity function: An activation or transfer function that identically reproduces the input. It is mathematically written as

$$f(x) = x.$$

Inhibitory connection: A connection between neurons that decreases a neuron's ability to fire. In artificial systems, this is usually a connection with a negative weight value.

Input layer: A layer of neurons that feed data into the neural network. In general, the input layer acts as a distribution layer and performs no processing on the data.

Jackknifing or jackknife resampling: A statistical technique used to estimate bias in an estimator or classifier, including neural networks.

Labeled pattern: An input pattern (or stimulus) and its associated output response (target).

Layer: A grouping of neurons (processing elements). *See* also **input layer**, **hidden layer**, and **output layer**.

Learning: In neural networks, both artificial and biological, this is the process of adjusting the synaptic (connection) weights between neurons to produce a desired response as the output of the neural network.

Learning rate: This is a factor used to scale the rate at which the weights in the neural network are adjusted during training. In the backpropagation algorithm, this term is generally denoted by η and is generally set to a value in the range of 0 to 1.

Learning rule (algorithm): Method of configuring and training a neural network. There are many learning rules and the most common is backpropagation.

Levenberg–Marquardt (LM): An alternative to backpropagation for use in training feedforward neural networks based on a nonlinear optimization. It uses an approximation of second-order derivatives to speed up training. It works well for networks with a few hundred weights or less.

Linear function: An activation function that scales the output linearly.

LM: *See* **Levenberg–Marquardt**.

Logistic sigmoid: An activation or transfer function with a sigmoidal shape that is commonly used with the networks trained by backpropagation. The function can be mathematically written as

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

MAM: Matrix associative memory.

Memorization: *See* **overtraining**.

Monte Carlo: Any technique that uses randomness to provide approximate solutions to a variety of problems through the use of computer simulation and statistical sampling.

Momentum: This term denotes the proportion of the last weight change that is added into the new weight change. It often provides a smoothing effect. In the backpropagation algorithm, this term is generally denoted by the symbol α and is often set to a value of 0.9.

Motor neurons: These are biological neurons that relay impulses from the brain or spinal cord and transmit them to the effectors.

MSE (mean squared error): Commonly used as a measure of training and testing accuracy in supervised learning. In the gradient-descent approach taken in backpropagation, this is the factor being minimized. For an output response, y, and a target response, t, the mean squared error is

$$\text{M.S.E.} = \sum_{i=1}^{N} (t_i - y_i)^2.$$

Multilayer perceptron (MLP): A type of feedforward neural network composed of layers of neurons with an input layer, at least one hidden layer, and an output layer. Neurons in one layer are connected to the previous layer through weighted connections. The name is derived from perceptron neural networks, but an MLP can solve problems beyond linearly separable ones.

Nerve: In a biological system, a bundle of neuron fibers used to carry impulses of information.

Neural network: A collection of interconnected neurons working in unison to solve a common task. A highly parallel system of interconnected processing elements. The term can refer to a mathematical model of a collection of neurons in the brain or an information-processing paradigm. When used as an information-processing paradigm, it is also known as an artificial neural network or neurocomputer.

Neurocomputer: Synonym for an artificial neural network.

Neuron: In a biological system, a neuron is a specialized cell in the nervous system that relays and processes information. In an artificial system, it is the fundamental information-processing element involved in transforming and relaying information. Equivalent to processing element, node, and unit.

NN: Neural network.

Node: Synonymous with *processing element* except that it can also include distribution elements that perform no processing.

OLAM: Optimal linear associative memory.

Output layer: A layer of neurons that feed data out of a neural network. This output represents the network's response.

Overtraining: A neural network's tendency to learn specifics about the data presented to it in addition to or instead of learning the data's basic structures. This results in a high error rate when testing the neural network with patterns not used in the training process. This is also known as memorization. In data and curve-fitting methods, this is known as overfitting the data. *See* also **generalization**.

Parallel distributed processing (PDP): Sometimes used as a synonym for artificial neural networks. This refers to the distributed processing and distributed memory properties of a neural network.

Pattern recognition: The ability to recognize and identify patterns based on prior knowledge or training.

PCNN: Pulse-coupled neural network.

Principal-components analysis (PCA): A statistical method to rank the importance of dimension through the use of covariance and to reduce dimensionality by eliminating low-ranking dimensions.

Processing element (PE): In the context of artificial neural networks, a processing element (PE) is synonymous with an artificial neuron. In general, it is the basic component of an information-processing system used for transforming and relaying information.

Pulse-coupled neural network (PCNN): A type of neural network originally developed to explain the experimentally observed pulse-synchrony process found in the cat visual cortex. This model is significantly different from other neural networks in both its structure and operation and is used in image processing and segmentation. No training is involved and its properties are adjusted by changing threshold levels and decay-time constants.

Quick propagation: A modification of backpropagation for training feedforward networks. It assumes a simple quadratic model for the error surface and calculates the weight updates independently along each weight to speed up training.

Radial-basis function (RBF) network: A neural network that uses Gaussian transfer functions to operate on the weighted inputs to produce the neuron output. RBFs are used in many classification and function-approximation applications.

Self-organization: A neural network's ability to create its own organization for the patterns presented during learning. Many neural networks are not self-organizing. Self-organization is related to unsupervised learning.

SOM: Self-organizing map.

Sensory neuron: A biological neuron that relays impulses from receptors to the brain or spinal cord.

Sigmoid: A class of activation functions that has upper and lower saturation limits, is monotonically increasing, and is continuous. It has a shape somewhat similar to a slanted, skewed letter S. The hyperbolic-tangent and logistic-sigmoid functions fall within this class of activation functions.

Statistical bias: The expected value of a statistic differs from that expected from the population. This can occur if certain areas of the population are oversampled or undersampled.

Success criterion: This term denotes the conditions for ending a training process, usually after achieving the desired output-error goal.

Supervised learning: A learning process that requires a set of labeled patterns (i.e., input patterns with known target outputs). This is analogous to a lesson being guided by an instructor. *See also* **unsupervised learning**.

Synapse: In a biological system, a synapse is the electrochemical junction connecting an axon fiber to a dendrite fiber. Information is transmitted from the axon to the dendrite through the synaptic connection. The strength of the connection determines the strength of the signal that is routed from one neuron to another. A synapse is analogous to a connection weight in an artificial system.

Target: The desired output or response of a neural network to a given input pattern.

Test set: A data set used only to evaluate the generalization performance of the fully trained neural network.

Threshold: In biological neurons, the input stimulus level at which the neuron begins to fire. In artificial neurons, a value that shifts the activation function. *See also* **bias**.

Tessellation: Tessellation (tiling) connections allow each node in a destination layer to be connected only to a subset of the nodes in a source layer. These connections are created by a regular tiling of the source layer by the nodes in the destination layer. Tessellation creates nodes with limited receptive fields.

Training schedule: Schedule for adjusting parameters used in the training process.

Training set: This is a set of inputs (and in the case of supervised learning, target outputs) used to train the neural network.

Transfer function: *See* **activation function**.

Unit: *See* **node** and **processing element**.

Unsupervised learning: Learning process that does not require a desired response. The network determines the appropriate response to a given input. This is analogous to a student deriving a lesson totally on his own, and is also a part of the self-organization process. Unsupervised learning is also used in several classical pattern-recognition clustering algorithms.

Validation set: A data set used to tune or adjust the parameters of a neural network during training.

Vigilance: A parameter used in ART networks to determine how closely an input pattern needs to match a candidate category prototype. It is usually denoted by ρ.

Weight: In an artificial neural network, a weight is the strength of a connection between two processing elements (neurons). The output signal of one neuron is multiplied by the weight as it is fed into the other neuron. It is analogous to a synapse in a biological system.

References

D. H. Ackley, G. E. Hinton and T. J. Sejnowski (1985). "A learning algorithm for Boltzmann machines." *Cognitive Science* **9**: 147–169.

E. Alpaydin and C. Kaynak (1998). "Cascading Classifers." *Kybernetika* **34**(4): 369–374.

E. Anderson (1935). "The irises of the Gaspé Peninsula." *Bulletin of the American Iris Society* **59**: 2–5.

A. Bain (1873). *Mind and Body: The Theories of Their Relation*. London, Henry King.

P. V. Balakrishnan, M. C. Cooper, V. S. Jacob and P. A. Lewis. (1994). "A study of the classification capabilities of neural networks using unsupervised learning: A comparison with A-means clustering." *Psychometrika* **59**(4): 509–525.

E. B. Baum and D. Haussler (1989). "What size net gives valid generalization." *Neural Computation* **1**(1): 151–160.

W. G. Baxt and H. White (1995). "Bootstrapping confidence intervals for clinical input variable effects in a network trained to identify the presence of acute myocardial infarction." *Neural Computation* **7**(3): 624–638.

S. Becker and Y. LeCun (1989). Improving the convergence of back-propagation learning with second order methods. *Proceedings of the 1988 Connectionist Summer School*. D. Touretzky, G. E. Hinton and T. Sejnowski. San Mateo, CA, Morgan Kaufmann Publishers: 29–37.

R. E. Bellman (1961). *Adaptive Control Processes: A Guided Tour*. Princeton, NJ, Princeton University Press.

L. M. Belue and K. W. Bauer, Jr. (1995). "Determining input features for multilayer perceptrons." *Neurocomputing* **7**(2): 1A–121.

D. P. Bertsekas and J. Tsitsiklis (1996). *Neuro-Dynamic Programming*. Belmont, MA, Athena Scientific.

M. Bianchini and M. Gori (1996). "Optimal Learning in Artificial Neural Networks: A Review of Theoretical Results." *Neurocomputing* **13**(5): 313–346.

C. M. Bishop (1995). *Neural Networks for Pattern Recognition*. Oxford, Oxford University Press.

C. L. Blake and C. J. Merz (1998). UCI Repository of machine learning databases. Irvine, CA, University of California, Department of Information and Computer Science**:** http://www.ics.uci.edu/~mlearn/MLRepository.html.

L. Breiman (1996). "Heuristics of instability and stabilization in model selection." *Annals of Statistics* **24**(6): 2350–2383.

L. Breiman, J. H. Friedman, R. A. Olshen and C. J. Stone (1984). *Classification and Regression Trees*, Kluwer Academic Publishers.

L. Breiman and P. Spector (1992). "Submodel selection and evaluation in regression: The X-random case." *International Statistical Review* **60**(3): 291–319.

J. S. Bridle (1990). Training Stochastic Model Recognition Algorithms as Networks can lead to Maximum Mutual Information Estimation of Parameters. *Advances in Neural Information Processing Systems*. D. S. Touretzky. San Mateo, CA, Morgan Kaufmann Publishers. **2**: 2A–217.

H. B. Burke (1996). The Importance of Artificial Neural Networks in Biomedicine. *Applications of Neural Networks in Environment, Energy, and Health*. P. E. Keller, S. Hashem, L. J. Kangas and R. T. Kouzes. Singapore, World Scientific Publishing**:** 145–153.

C. Cardaliaguet and E. Guillaume (1992). "Approximation of a function and its derivative with a neural network." *Neural Networks* **5**(2): 207–220.

G. A. Carpenter (1997). "Distributed learning, recognition, and prediction by ART and ARTMAP neural networks." *Neural Networks* **10**(8): 1473–1494.

G. A. Carpenter and S. Grossberg (1987a). "A massively parallel architecture for a self-organizing neural pattern recognition machine." *Computer Vision, Graphics, and Image Processing* **37**(1): 54–115.

G. A. Carpenter and S. Grossberg (1987b). "ART2: Stable self-organization of pattern recognition codes for analog input patterns." *Applied Optics* **26**(23): 4919–4930.

G. A. Carpenter and S. Grossberg (1990). ART3: Self-organization of Distributed Pattern Recognition Codes in Neural Network Hierarchies. *Proceedings of the International Conference on Neural Networks (INNC'90)*. Amsterdam, Kluwer Academic Publishers, North-Holland. **2**: 801–804.

G. A. Carpenter, S. Grossberg, N. Markuzon, J. H. Reynolds and D. B. Rosen (1992). "Fuzzy ARTMAP: A Neural Network Architecture for Incremental Supervised Learning of Analog Multidimensional Maps." *IEEE Transactions on Neural Networks* **3**(5): 698–713.

G. A. Carpenter, S. Grossberg and J. H. Reynolds (1995). "A Fuzzy ARTMAP Nonparametric Probability Estimator for Nonstationary Pattern Recognition Problems." *IEEE Transactions on Neural Networks* **6**(6): 1330–1336.

G. A. Carpenter, S. Grossberg and D. B. Rosen (1991). "Fuzzy ART: Fast Stable Learning and Categorization of Analog Patterns by an Adaptive Resonance Systems." *Neural Networks* **4**(6): 759–771.

G. A. Carpenter and W. D. Ross (1995). "ART-EMAP: A Neural Network Architecture for Object Recognition by Evidence Accumulation." *IEEE Transactions on Neural Networks* **6**(4): 805–818.

T. P. Caudell, S. D. G. Smith, R. Escobedo and M. Anderson (1994). "NIRS: Large Scale ART-1 Neural Architectures for Engineering Design Retrieval." *Neural Networks* **7**(9): 1339–1350.

L.-W. W. Chan and F. Fallside (1987). "An adaptive training algorithm for back-propagation networks." *Computer Speech and Language* **2**: 205–218.

T. Chen and H. Chen (1995). "Universal Approximation to Nonlinear Operators by Neural Networks with Arbitrary Activation Functions and Its Application to Dynamical Systems." *IEEE Transactions on Neural Networks* **6**(4): 9A–917.

G. Copson, R. Badcock, J. Boon and P. Britton (1997). "Articulating a systematic approach to clinical crime profiling." *Criminal Behaviour and Mental Health* **7**: 13–17.

T. M. Cover (1965). "Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition." *IEEE Transactions on Electronic Computers* **E3-14**(3): 326–334.

G. Coward (1992). *Tree Book: Learning to Recognize Trees of British Columbia*. Victoria, BC, Canada, Forestry Canada.

N. Cristianini and J. Shawe-Taylor (2000). *An Introduction to Support Vector Machines*. Cambridge, Cambridge University Press.

G. V. Cybenko (1989). "Approximation by Superpositions of a Sigmoidal Function." *Mathematics of Control, Signals, and Systems* **2**(4): 303–314.

H. Dai and C. Macbeth (1997). "Effects of learning parameters on learning procedure and performance of a BPNN." *Neural Networks* **10**(8): 1505–1521.

J. S. Denker and Y. LeCun (1991). Transforming Neural-Net Output Levels to Probability Distributions. *Advances in Neural Information Processing Systems*. R. Lippmann, J. E. Moody and D. S. Touretzky. San Mateo, CA, Morgan Kaufmann Publishers. **3**: 853–859.

H. Drucker and Y. LeCun (1992). "Improving Generalization Performance Using Double Backpropagation." *IEEE Transactions on Neural Networks* **3**(6): 991–997.

H. Drucker, D. Wu and V. N. Vapnik (1999). "Support Vector Machines for Spam Categorization." *IEEE Transactions on Neural Networks* **10**(5): 1048–1054.

R. O. Duda and P. E. Hart (1973). *Pattern Analysis and Scene Classification*. New York, John Wiley & Sons.

R. O. Duda, P. E. Hart and D. G. Stork (2001). *Pattern Classification, 2nd Edition*. New York, John Wiley & Sons Inc.

R. Eckhorn, H. J. Reitboeck, M. Arndt and P. Dicke (1990). "Feature linking via synchronization among distributed assemblies: Simulations of results from cat visual cortex." *Neural Cooperativity* **2**(3): 293–307.

B. Efron (1979). "Bootstrap methods: Another look at the jackknife." *Annals of Statistics* **7**(1): 1–26.

B. Efron (1982). *The Jackknife, the Bootstrap and Other Resampling Plans*. Philadelphia, Society for Industrial and Applied Mathematics.

B. Efron (1983). "Estimating the error rate of a prediction rule: Improvement on cross-validation." *Journal of the American Statistical Association* **78**: 316–331.

B. Efron and R. Tibshiran (1986). "Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy." *Statistical Science* **1**: 54–77.

B. Efron and R. J. Tibshirani (1993). *An Introduction to the Bootstrap*. London, Chapman & Hall.

B. Efron and R. J. Tibshirani (1997). "Improvements on cross-validation: The .632+ bootstrap method." *Journal of the American Statistical Association* **92**: 548–560.

J. L. Elman (1991). "Distributed representations, simple recurrent networks, and grammatical structure." *Machine Learning* **7**: 195–225.

S. E. Fahlman (1989). Faster Learning Variations on Back Propagation: An Empirical Study. *Proceedings of the 1988 Connectionist Models Summer School*. D. Touretzky, G. E. Hinton and T. Sejnowski. San Mateo, CA, Morgan Kaufmann Publishers: 38–51.

S. E. Fahlman and C. Liebiere (1990). The Cascade—Correlation Learning Architecture. *Advances in Neural Information Processing Systems*. D. S. Touretzky. San Mateo, CA, Morgan Kaufmann Publishers. **2**: 524–532.

B. G. Farley (1960). Self-Organizing Models for Learned Perception. *Self-Organizing Systems*. M. C. Yovits and S. Cameron. Oxford, UK, Pergamon Press.

B. G. Farley and W. A. Clark (1954). "Simulation of Self-Organizing Systems by Digital Computer." *IRE Transactions on Information Theory* **4**(4): 76–84.

B. G. Farley and W. A. Clark (1955). Generalization of pattern recognition in a self-organizing system. *Proceedings of the 1955 Western Joint Computer Conference*: 86–91.

R. A. Fisher (1936). "The use of multiple measurements in taxonomic problems." *Annual Eugenics* **7**(Part II): 179–188.

D. Fogel (1990). "An Information Criterion for Optimal Neural Network Selection." *IEEE Transactions on Neural Networks* **2**(5): 490–497.

D. H. Foley (1972). "Considerations of Sample and Feature Size." *IEEE Transactions on Information Theory* **IT-18**(5): 618–626.

N. Fraser (1998). Neural Network Follies. *http://neil.fraser.name/writing/tank/*.

B. R. Frieden (1983). *Probability, Statistical Optics, and Data Testing*. Berlin Heidelberg New York, Springer-Verlag.

K. Fukunaga (1990). *Introduction to Statistical Pattern Recognition*. San Diego, CA, Academic Press, Inc.

K. Funahashi (1989). "On the approximate realization of continuous mappings by neural networks." *Neural Networks* **2**(3): 183–192.

V. J. Geberth (1996). *Practical Homicide Investigation: Tactics, Procedures, and Forensic Techniques*. Boca Raton, Florida, CRC Publishing.

K. A. Gernoth and J. W. Clark (1995). "Neural Networks That Learn to Predict Probabilities: Global Models of Nuclear Stability and Decay." *Neural Networks* **8**(2): 291–311.

G. R. Gindi, A. F. Gmitro and K. Parthasarathy (1988). "Hopfield Model Associative Memory with Nonzero-Diagonal Terms in Memory Matrix." *Applied Optics* **27**(1): 129–134.

A. F. Gmitro, P. E. Keller and G. R. Gindi (1989). "Statistical performance of outer-product associative memory models." *Applied Optics* **28**(10): 1940–1948.

C. Goutte (1997). "Note on free lunches and cross-validation." *Neural Computation* **9**(6): 12A–1215.

S. Grossberg (1976a). "Adaptive pattern classification and universal recording: I. Parallel development and coding of neural detectors." *Biological Cybernetics* **23**: 121–134.

S. Grossberg (1976b). "On the Development of Feature Detectors in the Visual Cortex with Applications to Learning and Reaction-Diffusion Systems." *Biological Cybernetics* **21**(3): 145–159.

S. Grossberg (1976c). "Adaptive pattern classification and universal recording: II. Feedback, expectation, olfaction, illusions." *Biological Cybernetics* **23**: 187–202.

S. Grossberg (1987). *The Adaptive Brain I: Cognition, Learning, Reinforcement, and Rhythm*. Amsterdam, Elsevier/North-Holland.

M. Hagiwara (1992). Theoretical derivation of momentum term in back-propagation. *Proceedings of the International Joint Conference on Neural Networks (IJCNN'92)*. Piscataway, NJ, IEEE. **1**: 682–686.

C. L. Harris and J. L. Elman (1989). Representing variable information with simple recurrent networks. *Proceedings of the Tenth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ, Lawrence Erlbaum: 635–642.

D. Harrison, Jr. and D. L. Rubinfeld (1978). "Hedonic housing prices and the demand for clean air." *Journal of Environmental Economics and Management* **5**(1): 81–102.

S. Haykin (1994). *Neural Networks: A Comprehensive Foundation*. New York, Macmillan College Publishing Company.

M. A. Hearst, B. Scholkopf, S. Dumais, E. Osuna and J. Platt (1998). "Trends and Controversies: Support Vector Machines." *IEEE Intelligent Systems* **13**(4): 18–28.

D. O. Hebb (1949). *The Organization of Behavior*. New York, John Wiley & Sons.

R. Hecht-Nielsen (1987). "Counterpropation networks." *Applied Optics* **26**(23): 4979–4983.

J. S. U. Hjorth (1994). *Computer Intensive Statistical Methods Validation, Model Selection, and Bootstrap*. London, Chapman & Hall.

A. L. Hodgkin and A. F. Huxley (1952). "A Quantitative Description of Membrane Current and its Application to Conduction and Excitation in Nerve." *Journal of Physiology* **117**: 500–544.

J. J. Hopfield (1982). "Neural networks and physical systems with emergent collective computational abilities." *Proceedings of the National Academy of Sciences USA* **79**: 2554–2558.

J. J. Hopfield (1984). "Neurons with graded response have collective computational properties like those of two-state neurons." *Proceedings of the National Academy of Sciences USA* **81**: 3088–3092.

J. J. Hopfield and D. W. Tank (1985). "'Neural' computation of decisions in optimization problems." *Biological Cybernetics* **52**: 141–152.

J. J. Hopfield and D. W. Tank (1986). "Computing with neural circuits: A model." *Science* **233**: 625–633.

K. Hornik, M. Stinchcombe and H. White (1989). "Multilayer feedforward networks are universal approximators." *Neural Networks* **2**: 359–366.

H. Hotelling (1933). "Analysis of a complex of statistical variables into principal components." *Journal of Educational Psychology* **24**: 417–441 and 498–520.

J. Huang, M. Georgiopoulos and G. L. Heileman (1995). "Fuzzy ART Properties." *Neural Networks* **8**(2): 203–213.

S.-C. Huang and Y.-F. Huang (1990). "Learning Algorithms for Perceptrons Using Back Propagation with Selective Updates." *IEEE Control Systems Magazine* **10**(3): 56–61.

C.-A. Hung and S.-F. Lin (1995). "Adaptive Hamming Net: A Fast-Learning ART 1 Model Without Searching." *Neural Networks* **8**(4): 605–618.

B. Hunt, M. S. Nadar, P. Keller, E. VonColln and A. Goyal (1993). "Synthesis of a Nonrecurrent Associative Memory Model Based on a Nonlinear Transformation in the Spectral Domain." *IEEE Transactions on Neural Networks* **4**(5): 873–878.

C. M. Hurvich and C.-L. Tsai (1989). "Regression and time series model selection in small samples." *Biometrika* **76**: 297–307.

D. R. Hush and B. G. Horne (1993). "Progress in Supervised Neural Networks: What's New Since Lippmann?" *IEEE Signal Processing Magazine* **10**(1): 8–39.

A. Hyvärinen and E. Oja (1999). Independent Component Analysis: A Tutorial. http://www.cis.hut.fi/∼ aapo/ps/NN00.pdf. Espoo, Finland, Helsinki University of Technology.

A. Hyvärinen and E. Oja (2000). "Independent Component Analysis: Algorithms and Applications." *Neural Networks* **13**(4-5): 4A–430.

J. E. Jackson (1991). *A User's Guide to Principal Components*. New York, John Wiley & Sons Inc.: 63–69.

R. A. Jacobs (1988). "Increased Rates of Convergence Through Learning Rate Adaptation." *Neural Networks* **1**(4): 295–307.

W. James (1890). *Principles of Psychology*. New York, Henry Holt.

D. S. Johnson and C. H. Papadimitriou (1985). Computational Complexity. *The Traveling Salesman Problem*. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shmoys. New York, John-Wiley & Sons: 37–85.

B. L. Kalman and S. C. Kwasny (1992). Why Tanh: Choosing a Sigmoidal Function. *Proceedings of the International Joint Conference on Neural Networks (IJCNN'92)*. Piscataway, NJ, IEEE. **4**: 578–581.

L. J. Kangas, K. M. Terrones, R. D. Keppel and R. D. La Moria (1998). Computer Aided Tracking and Characterization of Homicides & Sexual Assaults (CATCH). *Applications and Science of Computational Intelligence II—Proceedings of the SPIE*. K. L. Priddy, P. E. Keller, D. B. Fogel and J. C. Bezdek. Bellingham, WA, SPIE. **3722**: 250–260.

P. P. Kanjilal and D. N. Banerjee (1995). "On the Application of Orthogonal Transformation for the Design and Analysis of Feedforward Networks." *IEEE Transactions on Neural Networks* **6**(5): 1061–1070.

M. Kearns (1997). "A bound on the error of cross validation using the approximation and estimation rates, with consequences for the training-test split." *Neural Computation* **9**(5): 1143–1161.

P. E. Keller and A. D. McKinnon (1999). Pulse-Coupled Neural Networks for Medical Image Analysis. *Applications and Science of Computational Intelligence II—Proceedings of the SPIE*. K. L. Priddy, P. E. Keller, D. B. Fogel and J. C. Bezdek. Bellingham, WA, SPIE. **3722**: 444–451.

P. E. Keller, D. L. McMakin, D. M. Sheen, A. D. McKinnon and J. W. Summet (2000). Privacy Algorithm for Airport Passenger Screening Portal. *Applications and Science of Computation Intelligence III – Proceedings of the SPIE*. K. L. Priddy, P. E. Keller and D. B. Fogel. Bellingham, WA, SPIE. **4055**: 476–483.

R. D. Keppel and R. Walter (1999). "Profiling Killers: A Revised Classification Model for Understanding Sexual Murder." *International Journal of Offender Therapy and Comparative Criminology* **43**(4): 417–437.

R. Kohavi (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'95)*. San Mateo, CA, Morgan Kaufmann Publishers: 1137–1143.

T. Kohonen (1972). "Correlation Matrix Memories." *IEEE Transactions on Computers* **3-21**: 353–359.

T. Kohonen (1982). "Self-organized formation of topologically correct feature maps." *Biological Cybernetics* **43**: 59–69.

T. Kohonen (1987). "Adaptive, associative, and self-organizing functions in neural computing." *Applied Optics* **26**(23): 4910–4918.

T. Kohonen (1988). "The 'Neural' Phonetic Typewriter." *IEEE Computer Magazine* **21**(3): A-22.

T. Kohonen (1989). *Self-Organization and Associative Memory*. Berlin Heidelberg London, Springer-Verlag.

B. Kosko (1987). "Constructing an associative memory." *Byte* **12**(10): 137–144.

B. Kosko (1987). "Adaptive bidirectional memories." *Applied Optics* **26**(23): 4947–4960.

B. Kosko (1988). "Bidirectional associative memory." *IEEE Transactions on Systems, Man and Cybernetics* **SM3-18**(1): 49–60.

B. Kosko (1992). *Neural Networks for Signal Processing*. Upper Saddle River, NJ, Prentice Hall.

A. Kowalczyk (1997). "Estimates of storage capacity of multilayer perceptron with threshold logic hidden units." *Neural Networks* **10**(8): 1417–1433.

M. A. Kraaijveld, J. Mao and A. K. Jain (1995). "A Nonlinear Projection Method Based on Kohonen's Topology Preserving Maps." *IEEE Transactions on Neural Networks* **6**(3): 548–559.

A. H. Kramer and A. L. Sangiovanni-Vincentelli (1989). Efficient parallel learning algorithms for neural networks. *Advances in Neural Information Processing Systems*. D. S. Touretzky. San Mateo, CA, Morgan Kaufmann Publishers. **1**: 40–48.

Y. LeCun (1985). Une procedure d'apprentissage pour reseau a seuil assymetrique. *Cognitiva '85: A la frontière de l 'intelligence Artificielle des Sciences de la Connaissance des Neuronsciences*: 599–604.

G. G. Lendaris, K. Mathia and R. E. Saeks (1999). "Linear Hopfield Networks and Constrained Optimization." *IEEE Transactions of Systems, Man & Cybernetics* **29**(1): 114–118.

K. Levenberg (1944). "A method for the solution of certain problems in least squares." *Quart. Applied Mathematics* **2**: 164–168.

T. Linblad and J. M. Kinser (1998). *Image Processing using Pulse-Coupled Neural Networks*. London, Springer.

R. Lippmann (1987). "An Introduction to Computing with Neural Networks." *IEEE ASSP Magazine* **4**(2): 4–22.

P. C. Mahalanobis (1936). "On the generalized distance in statistics." *Proceedings of the National Institute of Science of India* **2**: 49–53.

O. L. Mangasarian and D. R. Musicant (1999). "Successive Overrelaxation for Support Vector Machines." *IEEE Transactions on Neural Networks* **10**(5): 1032–1037.

J. Mao and A. K. Jain (1996). "A Self-Organizing Network for Hyperellipsoidal Clustering (HEC)." *IEEE Transactions on Neural Networks* **7**(1): 16–29.

D. W. Marquardt (1963). "An algorithm for least-squares estimation of nonlinear parameters." *Journal of the Society of Industrial Applied Mathematics* **11**: 431–441.

S. Marriott and R. F. Harrison (1995). "A Modified Fuzzy ARTMAP Architecture for the Approximation of Noisy Mappings." *Neural Networks* **8**(4): 619–641.

T. Masters (1995). *Advanced Algorithms for Neural Networks: A C++ Sourcebook*. New York, John Wiley & Sons.

W. S. McCulloch and W. H. Pitts (1943). "A Logical Calculus of the Ideas Immanent in Nervous Activity." *Bulletin of Mathematical Biophysics* **5**: 115–133.

R. J. McEliece, E. C. Posner, E. R. Rodemich and S. S. Venkates (1987). "The capacity of the Hopfield associative memory." *IEEE Transactions on Information Theory* **33**(4): 461–482.

R. G. Miller (1974). "The jackknife, a review." *Biometrika* **61**: 1–15.

M. L. Minsky and S. A. Papert (1969). *Perceptrons: An introduction to Computational Geometry*. Cambridge, MA, MIT Press.

C. Z. Mooney and R. D. Duval (1993). *Bootstrapping: A Nonparametric Approach to Statistical Inference*, Sage Publications.

B. Moore (1988). ART 1 and Pattern Clustering. *Proceedings of the 1988 Connectionist Summer School*. D. Touretzky, G. E. Hinton and T. Sejnowski. San Mateo, CA, Morgan Kaufmann Publishers: 174–185.

F. M. Mulier and V. S. Cherkassky (1995). "Statistical Analysis of Self-organization." *Neural Networks* **8**(5): 717–727.

I. Nabney (2002). *Netlab: Algorithms for Pattern Recognition*. Berlin Heidelberg London, Springer-Verlag.

T. Nitta (1997). "An extension of the back-propagation algorithm to complex numbers." *Neural Networks* **10**(8): 1391–1415.

E. Oja (1991). Data compression, feature extraction, and auto-association in feedforward neural networks. *Proceedings of the 1991 International Conference on Artificial Neural Networks (ICANN'91)*. T. Kohonen, K. Makisara, O. Simula and J. Kangas. Amsterdam, Elsevier Science Publishers B. V. **1**: 737–746.

E. Oja (1991). "A simplified neuron model as a principal component analyzer." *Journal of Mathematical Biology* **15**: 267–273.

Y.-H. Pao (1989). *Adaptive Pattern Recognition and Neural Networks*. Reading, MA, Addison-Wesley Publishing Company, Inc.

D. Parker (1982). Learning-logic. *Invention Report S81-64, File 1*. Palo Alto, CA, Stanford University, Office of Technology Licensing.

M. Plutowski, S. Sakata and H. White (1994). Cross-validation estimates IMSE. *Advances in Neural Information Processing Systems*. J. D. Cowan, G. Tesauro and J. Alspector. San Mateo, CA, Morgan Kaufmann Publishers. **6**: 391–398.

T. Poggio (1975). "On optimal nonlinear associative recall." *Biological Cybernetics* **19**: 201–209.

B. T. Poljak (1964a). О некоторых способах ускорения сходимости итерационных методов." *Журн. выч. мат. и мат. физ. — Zhurnal Vychislitel'noi Matematiki I Matematicheskoi Fiziki* **4**(5): 791–803.

B. T. Poljak (1964b). "Some methods of speeding up the convergence of iteration methods." *USSR Computational Mathematics and Mathematical Physics* **4**(5): 1–17.

K. L. Priddy, S. K. Rogers, D. W. Ruck, G. L. Tarr and M. Kabrisky (1993). "Bayesian selection of important features for feedforward neural networks." *Neurocomputing* **5**(2): 91–103.

K.L. Priddy (2004). "A comparative analysis of machine classifiers." *Intelligent Computing: Theory and Applications II, Proceedings of SPIE*. K. L. Priddy. Bellingham, WA, SPIE. **5421**: 142–148.

M. H. Quenouille (1949). "Approximate tests of correlation in time series." *Journal of the Royal Statistical Society B* **11**: 18–84.

M. H. Quenouille (1956). "Notes on bias reduction." *Biometrika* **43**: 353–360.

S. Raudys (2000). "How Good are Support Vector Machines?" *Neural Networks* **13**(1): 17–19.

S. Raudys (2001). *Statistical and Neural Classifiers: An Integrated Approach to Design*. Berlin Heidelberg London, Springer-Verlag.

B. D. Ripley (1996). *Pattern Recognition and Neural Networks*. Cambridge, Cambridge University Press.

N. Rochester, J. H. Holland, H. L. H. and W. L. Duda (1956). "Tests on a Cell Assembly Theory of the Action of the Brain Using a Large Digital Computer." *IRE Transactions on Information Theory* **IT-2**(3): 80–93.

S. K. Rogers (1997). Tools for Pattern Recognition. *EENG 617 Class Handout*. Wright-Patterson AFB, OH, Air Force Institute of Technology.

F. Rosenblatt (1958). "The Perceptron: a Probabilistic Model for Information Storage and Organization in the Brain." *Psychological Review* **65**: 386–408.

F. Rosenblatt (1959). *Principles of Neurodynamics*. New York, Spartan Books.

W. A. Rosenblith and H. B. Barlow (1990). "Sensory communications." *Scientific American*.

D. W. Ruck, S. K. Rogers and M. Kabrisky (1990a). "Feature selection using a multilayer perceptron." *Journal of Neural Network Computing* **2**(2): 40–48.

D. W. Ruck, S. K. Rogers, M. Kabrisky, M. E. Oxley and B. S. Suter (1990b). "The Multilayer Perceptron as an Approximation to a Bayes Optimal Discriminant Function." *IEEE Transactions on Neural Networks* **1**(4): 296–298.

D. E. Rumelhart, G. E. Hinton and R. J. Williams (1986). Learning internal representations by error propagation. *Parallel Distributed Processing: Explorations in the Microstructures of Cognition. 1: Foundations*. D. E. Rumelhart and J. L. McClelland. Cambridge, MA, MIT Press. **1**: 318–362.

T. D. Sanger (1989a). An optimality principle for unsupervised learning. *Advances in Neural Information Processing Systems*. D. S. Touretzky. San Mateo, CA, Morgan Kaufmann Publishers. **1**: A–19.

T. D. Sanger (1989b). "Optimal unsupervised learning in a single-layer linear feedforward neural network." *Neural Networks* **2**(6-7): 459–473.

T. J. Sejnowski and C. R. Rosenberg (1987). "Parallel Networks that Learn to Pronounce English Text." *Complex Systems* **1**: 145–168.

J. Shao (1993). "Linear model selection by cross-validation." *Journal of the American Statistical Association* **88**: 486–494.

J. Shao (1997). "An asymptotic theory for linear model selection." *Statistica Sinica* **7**: 221–264.

J. Shao and D. Tu (1995). *The Jackknife and Bootstrap*. Berlin Heidelberg London, Springer-Verlag.

T. A. B. Snijders (1988). On cross-validation for predictor evaluation in time series. *On Model Uncertainty and Its Statistical Implications*. T. K. Dijkstra. Berlin, Springer-Verlag: 56–69.

F. Stäger and M. Agarwal (1997). "Three methods to speed up the training of feedforward and feedback perceptrons." *Neural Networks* **10**(8): 1435–1443.

M. Stone (1977). "Asymptotics for and against cross-validation." *Biometrika* **64**: 29–35.

M. Stone (1979). "Comments on model selection criteria of Akaike and Schwarz." *Journal of the Royal Statistical Society, Series B* **41**: 276–278.

W. S. Stornetta and B. A. Huberman (1987). An Improved Three-Layer, Back Propagation Algorithm. *Proceedings of the IEEE Conference on Neural Networks (ICNN'87)*. Piscataway, NJ, IEEE. **2**: 637–644.

Y. Suzuki (1995). "Self-Organizing QRS-Wave Recognition in ECG Using Neural Networks." *IEEE Transactions on Neural Networks* **6**(6): 1469–1477.

D. W. Tank and J. J. Hopfield (1986). "Simple Neural Optimization Networks: An A/D Converter, Signal Decision Circuit, and a Linear Programming Circuit." *IEEE Transactions on Circuits and Systems* **33**(5): 533–541.

G. Tarr, K. Priddy and S. Rogers (1992). "NeuralGraphics: A general purpose environment for neural network simulation," *Applications of Artificial Neural Networks III, Proceedings of SPIE.* S. K. Rogers Bellingham, WA, SPIE. **1709**: 1047–1056.

R. Tibshirani (1996). "A comparison of some error estimates for neural network models." *Neural Computation* **8**: 152–163.

J. W. Tukey (1958). "Bias and Confidence in Not-Quite Large Samples." *Annals of Mathematical Statistics* **29**: 614.

V. N. Vapnik (1995). *The Nature of Statistical Learning Theory*. Berlin Heidelberg London, Springer-Verlag.

B. Verma (1997). "Fast Training of Multilayer Perceptrons." *IEEE Transactions on Neural Networks* **8**(6): 1314–1320.

R. Vitthal, P. Sunthar and C. D. Rao (1995). "The Generalized Proportional-Integral-Derivative (PID) Gradient Descent Back Propagation Algorithm." *Neural Networks* **8**(4): 563–569.

S. Watanabe and K. Fukumizu (1995). "Probabilistic Design of Layered Neural Networks Based on Their Unified Framework." *IEEE Transactions on Neural Networks* **6**(3): 691–702.

S. M. Weiss and C. A. Kulikowski (1991). *Computer Systems That Learn*. San Mateo, CA, Morgan Kaufmann Publishers.

P. J. Werbos (1974). "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences." Ph.D. Thesis, Cambridge, MA.

P. J. Werbos (1994). *The Roots of Backpropagation*. New York, John Wiley & Sons.

B. Widrow (1962). Generalization and information Storage in Networks of Adaline Neurons. *Self-Organizing Systems*. M. C. Yovits, G. T. Jacobi and G. D. Goldstein. Washington, D.C., Spartan Books.

B. Widrow and M. E. Hoff, Jr. (1960). Adaptive Switching Circuits. *1960 IRE WESCON Convention Record, Part 4*. New York, Institute of Radio Engineers: 96–104.

B. Widrow and M. A. Lehr (1990). "30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation." *Proceedings of the IEEE* **78**: 1415–1441.

A. L. Wilkes and N. J. Wade (1997). "Bain on Neural Networks." *Brain and Cognition* **33**: 295–305.

R. J. Williams and D. Zipser (1989). "A learning algorithm for continually running fully recurrent neural networks." *Neural Computation* **1**(2): 270–280.

R. J. Williams and D. Zipser (1990). Gradient-based learning algorithms for recurrent connectionist networks. *Technical Report NU-CCS-90-9*. Boston, Northeastern University, College of Computer Science.

R. J. Williams and D. Zipser (1995). Gradient-Based Learning Algorithms for Recurrent Networks and Their Computational Complexity. *Backpropagation:*

Theory, Architectures, and Applications. Y. Chauvin and D. E. Rumelhart. Hillsdale, NJ, Lawrence Erlbaum Publishers: 433–486.

J. R. Williamson (1996). "Gaussian ARTMAP: A neural network for fast incremental learning of noisy multidimensional maps." *Neural Networks* **9**(5): 881–897.

G. V. Wilson and G. S. Pawley (1988). "On the stability of the traveling salesman problem algorithm of Hopfield and Tank." *Biological Cybernetics* **58**(1): 63–70.

Y. Zheng and J. F. Greenleaf (1996). "The Effect of Concave and Convex Weight Adjustments on Self-Organizing Maps." *IEEE Transactions on Neural Networks* **7**(1): 87–96.

H. Zhu and R. Rohwer (1996). "No free lunch for cross-validation." *Neural Computation* **8**(7): 1421–1426.

Index

About the Authors

Paul E. Keller received a B.S. degree in physics from Boise State University in 1985, an M.S. degree in electro-optics from the University of Dayton in 1987, and a Ph.D. in optical sciences from the University of Arizona in 1991. He has been a scientist with Battelle at the Pacific Northwest National Laboratory since 1992. His research areas include optics, photonics, neural networks, sensor data analysis, image processing, and computational physics. He has worked on applications in homeland security, nuclear nonproliferation, national defense, medicine, environmental technologies, telecommunications, computing, and energy distribution. He has taught several short courses on neural networks and their applications and also served as an adjunct faculty at Washington State University in the area of neural networks. He has authored and co-authored more than 60 conference papers, journal articles, and book chapters on the topics of neural networks, medical technologies, optical computing, and Internet technologies. He holds three U.S. patents, with several pending.

Kevin L. Priddy received a B.S. degree in electrical engineering from Brigham Young University in 1982, an M.S. degree in electro-optics from the Air Force Institute of Technology in 1985, and a Ph.D. in electrical engineering from the Air Force Institute of Technology in 1992. Dr. Priddy is currently a team leader in the ATR & Fusion Algorithms Branch, Sensors Directorate, of the Air Force Research Laboratory. Dr. Priddy has over 40 conference papers and journal articles in the fields of artificial neural networks and electrical engineering. He has been awarded two patents and has others pending in the computational intelligence field. His research interests include computational intelligence, automated recognition systems, intelligent sensors, image processing and signal processing.