Neural Network Design

Hagan
Demuth
Beale
De Jesús

2nd Edition

Neural Network Design
2nd Edition

Martin T. Hagan
Oklahoma State University
Stillwater, Oklahoma

Howard B. Demuth
University of Colorado
Boulder, Colorado

Mark Hudson Beale
MHB Inc.
Hayden, Idaho

Orlando De Jesús
Consultant
Frisco, Texas

MTH

To Janet, Thomas, Daniel, Mom and Dad

HBD

To Hal, Katherine, Kimberly and Mary

MHB

To Leah, Valerie, Asia, Drake, Coral and Morgan

ODJ

To: Marisela, María Victoria, Manuel, Mamá y Papá.

Neural Network Design, 2nd Edition, eBook

OVERHEADS and DEMONSTRATION PROGRAMS can be found at the following website:

hagan.okstate.edu/nnd.html

A somewhat condensed paperback version of this text can be ordered from Amazon.

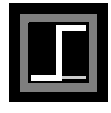
Contents

3

An Illustrative Example

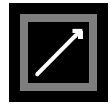
4

Perceptron Learning Rule



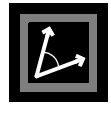
5

Signal and Weight Vector Spaces

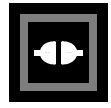


6

Linear Transformations for Neural Networks



7 Supervised Hebbian Learning



8 Performance Surfaces and Optimum Points

9 Performance Optimization

10 Widrow-Hoff Learning

11

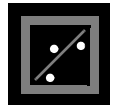
Backpropagation

12

Variations on Backpropagation

13

Generalization



14

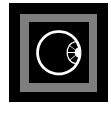
Dynamic Networks

15 Associative Learning

16 Competitive Networks

17 Radial Basis Networks

18 Grossberg Network



19 Adaptive Resonance Theory

20 Stability

21

Hopfield Network

22

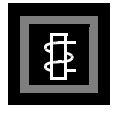
Practical Training Issues

26

Case Study 4: Clustering

27

Case Study 5: Prediction



Appendices

Preface

This book gives an introduction to basic neural network architectures and learning rules. Emphasis is placed on the mathematical analysis of these networks, on methods of training them and on their application to practical engineering problems in such areas as nonlinear regression, pattern recognition, signal processing, data mining and control systems.

Every effort has been made to present material in a clear and consistent manner so that it can be read and applied with ease. We have included many solved problems to illustrate each topic of discussion. We have also included a number of case studies in the final chapters to demonstrate practical issues that arise when using neural networks on real world problems.

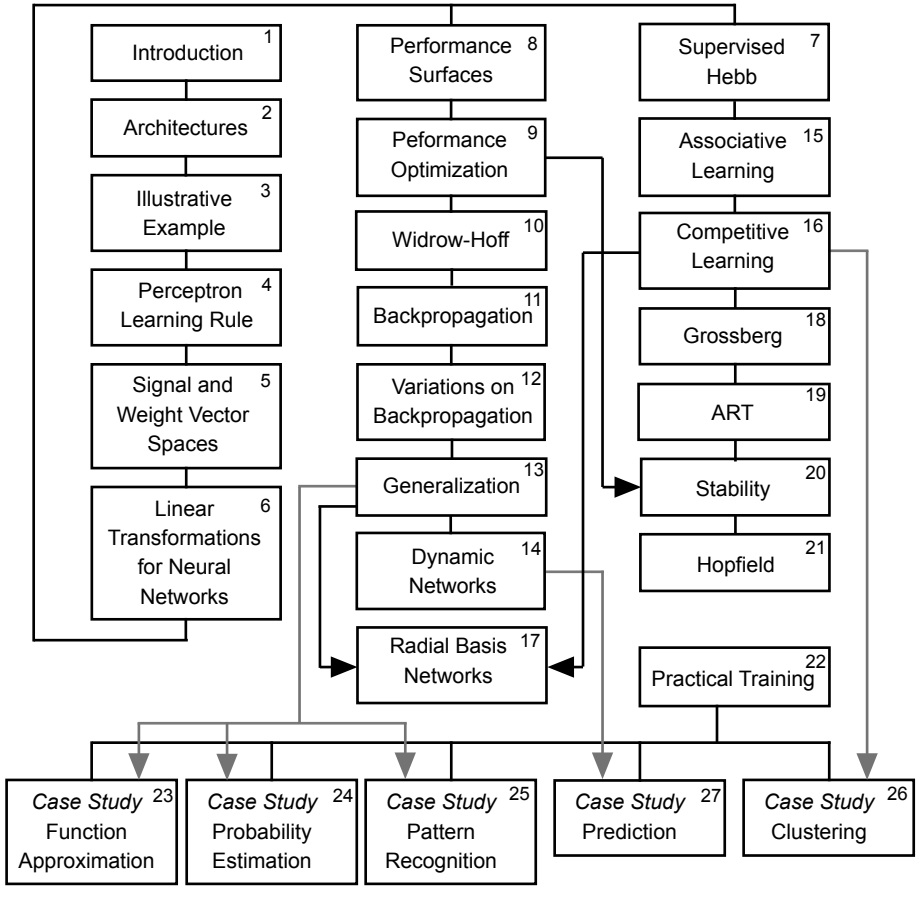
Since this is a book on the design of neural networks, our choice of topics was guided by two principles. First, we wanted to present the most useful and practical neural network architectures, learning rules and training techniques. Second, we wanted the book to be complete in itself and to flow easily from one chapter to the next. For this reason, various introductory materials and chapters on applied mathematics are included just before they are needed for a particular subject. In summary, we have chosen some topics because of their practical importance in the application of neural networks, and other topics because of their importance in explaining how neural networks operate.

We have omitted many topics that might have been included. We have not, for instance, made this book a catalog or compendium of all known neural network architectures and learning rules, but have instead concentrated on the fundamental concepts. Second, we have not discussed neural network implementation technologies, such as VLSI, optical devices and parallel computers. Finally, we do not present the biological and psychological foundations of neural networks in any depth. These are all important topics, but we hope that we have done the reader a service by focusing on those topics that we consider to be most useful in the design of neural networks and by treating those topics in some depth.

This book has been organized for a one-semester introductory course in neural networks at the senior or first-year graduate level. (It is also suitable for short courses, self-study and reference.) The reader is expected to have some background in linear algebra, probability and differential equations.

Each chapter of the book is divided into the following sections: Objectives, Theory and Examples, Summary of Results, Solved Problems, Epilogue, Further Reading and Exercises. The *Theory and Examples* section comprises the main body of each chapter. It includes the development of fundamental ideas as well as worked examples (indicated by the icon shown here in the left margin). The *Summary of Results* section provides a convenient listing of important equations and concepts and facilitates the use of the book as an industrial reference. About a third of each chapter is devoted to the *Solved Problems* section, which provides detailed examples for all key concepts.

The following figure illustrates the dependencies among the chapters.



Chapters 1 through 6 cover basic concepts that are required for all of the remaining chapters. Chapter 1 is an introduction to the text, with a brief historical background and some basic biology. Chapter 2 describes the ba-

sic neural network architectures. The notation that is introduced in this chapter is used throughout the book. In Chapter 3 we present a simple pattern recognition problem and show how it can be solved using three different types of neural networks. These three networks are representative of the types of networks that are presented in the remainder of the text. In addition, the pattern recognition problem presented here provides a common thread of experience throughout the book.

Much of the focus of this book will be on methods for training neural networks to perform various tasks. In Chapter 4 we introduce learning algorithms and present the first practical algorithm: the perceptron learning rule. The perceptron network has fundamental limitations, but it is important for historical reasons and is also a useful tool for introducing key concepts that will be applied to more powerful networks in later chapters.

One of the main objectives of this book is to explain how neural networks operate. For this reason we will weave together neural network topics with important introductory material. For example, linear algebra, which is the core of the mathematics required for understanding neural networks, is reviewed in Chapters 5 and 6. The concepts discussed in these chapters will be used extensively throughout the remainder of the book.

Chapters 7, and 15–19 describe networks and learning rules that are heavily inspired by biology and psychology. They fall into two categories: associative networks and competitive networks. Chapters 7 and 15 introduce basic concepts, while Chapters 16–19 describe more advanced networks.

Chapters 8–14 and 17 develop a class of learning called performance learning, in which a network is trained to optimize its performance. Chapters 8 and 9 introduce the basic concepts of performance learning. Chapters 10–13 apply these concepts to feedforward neural networks of increasing power and complexity, Chapter 14 applies them to dynamic networks and Chapter 17 applies them to radial basis networks, which also use concepts from competitive learning.

Chapters 20 and 21 discuss recurrent associative memory networks. These networks, which have feedback connections, are dynamical systems. Chapter 20 investigates the stability of these systems. Chapter 21 presents the Hopfield network, which has been one of the most influential recurrent networks.

Chapters 22–27 are different than the preceding chapters. Previous chapters focus on the fundamentals of each type of network and their learning rules. The focus is on understanding the key concepts. In Chapters 22–27, we discuss some practical issues in applying neural networks to real world problems. Chapter 22 describes many practical training tips, and Chapters 23–27 present a series of case studies, in which neural networks are applied to practical problems in function approximation, probability estimation, pattern recognition, clustering and prediction.

Software

MATLAB is not essential for using this book. The computer exercises can be performed with any available programming language, and the *Neural Network Design Demonstrations*, while helpful, are not critical to understanding the material covered in this book.

However, we have made use of the MATLAB software package to supplement the textbook. This software is widely available and, because of its matrix/vector notation and graphics, is a convenient environment in which to experiment with neural networks. We use MATLAB in two different ways. First, we have included a number of exercises for the reader to perform in MATLAB. Many of the important features of neural networks become apparent only for large-scale problems, which are computationally intensive and not feasible for hand calculations. With MATLAB, neural network algorithms can be quickly implemented, and large-scale problems can be tested conveniently. These MATLAB exercises are identified by the icon shown here to the left. (If MATLAB is not available, any other programming language can be used to perform the exercises.)

The second way in which we use MATLAB is through the *Neural Network Design Demonstrations*, which can be downloaded from the website hagan.okstate.edu/nnd.html. These interactive demonstrations illustrate important concepts in each chapter. After the software has been loaded into the MATLAB directory on your computer (or placed on the MATLAB path), it can be invoked by typing `nnd` at the MATLAB prompt. All demonstrations are easily accessible from a master menu. The icon shown here to the left identifies references to these demonstrations in the text.

The demonstrations require MATLAB or the student edition of MATLAB, version 2010a or later. See Appendix C for specific information on using the demonstration software.

Overheads

As an aid to instructors who are using this text, we have prepared a companion set of overheads. Transparency masters (in Microsoft Powerpoint format or PDF) for each chapter are available on the web at hagan.okstate.edu/nnd.html.

Acknowledgments

We are deeply indebted to the reviewers who have given freely of their time to read all or parts of the drafts of this book and to test various versions of the software. In particular we are most grateful to Professor John Andreae, University of Canterbury; Dan Foresee, AT&T; Dr. Carl Latino, Oklahoma State University; Jack Hagan, MCI; Dr. Gerry Andeen, SRI; and Joan Miller and Margie Jenks, University of Idaho. We also had constructive inputs from our graduate students in ECEN 5733 at Oklahoma State University, ENEL 621 at the University of Canterbury, INSA 0506 at the Institut National des Sciences Appliquées and ECE 5120 at the University of Colorado, who read many drafts, tested the software and provided helpful suggestions for improving the book over the years. We are also grateful to the anonymous reviewers who provided several useful recommendations.

We wish to thank Dr. Peter Gough for inviting us to join the staff in the Electrical and Electronic Engineering Department at the University of Canterbury, Christchurch, New Zealand, and Dr. Andre Titli for inviting us to join the staff at the Laboratoire d'Analyse et d'Architecture des Systèms, Centre National de la Recherche Scientifique, Toulouse, France. Sabbaticals from Oklahoma State University and a year's leave from the University of Idaho gave us the time to write this book. Thanks to Texas Instruments, Halliburton, Cummins, Amgen and NSF, for their support of our neural network research. Thanks to The Mathworks for permission to use material from the *Neural Network Toolbox*.

1 Introduction

Objectives

As you read these words you are using a complex biological neural network. You have a highly interconnected set of some 10^{11} neurons to facilitate your reading, breathing, motion and thinking. Each of your biological neurons, a rich assembly of tissue and chemistry, has the complexity, if not the speed, of a microprocessor. Some of your neural structure was with you at birth. Other parts have been established by experience.

Scientists have only just begun to understand how biological neural networks operate. It is generally understood that all biological neural functions, including memory, are stored in the neurons and in the connections between them. Learning is viewed as the establishment of new connections between neurons or the modification of existing connections. This leads to the following question: Although we have only a rudimentary understanding of biological neural networks, is it possible to construct a small set of simple artificial "neurons" and perhaps train them to serve a useful function? The answer is "yes." This book, then, is about *artificial* neural networks.

The neurons that we consider here are not biological. They are extremely simple abstractions of biological neurons, realized as elements in a program or perhaps as circuits made of silicon. Networks of these artificial neurons do not have a fraction of the power of the human brain, but they can be trained to perform useful functions. This book is about such neurons, the networks that contain them and their training.

History

The history of artificial neural networks is filled with colorful, creative individuals from a variety of fields, many of whom struggled for decades to develop concepts that we now take for granted. This history has been documented by various authors. One particularly interesting book is *Neurocomputing: Foundations of Research* by John Anderson and Edward Rosenfeld. They have collected and edited a set of some 43 papers of special historical interest. Each paper is preceded by an introduction that puts the paper in historical perspective.

Histories of some of the main neural network contributors are included at the beginning of various chapters throughout this text and will not be repeated here. However, it seems appropriate to give a brief overview, a sample of the major developments.

At least two ingredients are necessary for the advancement of a technology: concept and implementation. First, one must have a concept, a way of thinking about a topic, some view of it that gives a clarity not there before. This may involve a simple idea, or it may be more specific and include a mathematical description. To illustrate this point, consider the history of the heart. It was thought to be, at various times, the center of the soul or a source of heat. In the 17th century medical practitioners finally began to view the heart as a pump, and they designed experiments to study its pumping action. These experiments revolutionized our view of the circulatory system. Without the pump concept, an understanding of the heart was out of grasp.

Concepts and their accompanying mathematics are not sufficient for a technology to mature unless there is some way to implement the system. For instance, the mathematics necessary for the reconstruction of images from computer-aided tomography (CAT) scans was known many years before the availability of high-speed computers and efficient algorithms finally made it practical to implement a useful CAT system.

The history of neural networks has progressed through both conceptual innovations and implementation developments. These advancements, however, seem to have occurred in fits and starts rather than by steady evolution.

Some of the background work for the field of neural networks occurred in the late 19th and early 20th centuries. This consisted primarily of interdisciplinary work in physics, psychology and neurophysiology by such scientists as Hermann von Helmholtz, Ernst Mach and Ivan Pavlov. This early work emphasized general theories of learning, vision, conditioning, etc., and did not include specific mathematical models of neuron operation.

The modern view of neural networks began in the 1940s with the work of Warren McCulloch and Walter Pitts [McPi43], who showed that networks of artificial neurons could, in principle, compute any arithmetic or logical function. Their work is often acknowledged as the origin of the neural network field.

McCulloch and Pitts were followed by Donald Hebb [Hebb49], who proposed that classical conditioning (as discovered by Pavlov) is present because of the properties of individual neurons. He proposed a mechanism for learning in biological neurons (see Chapter 7).

The first practical application of artificial neural networks came in the late 1950s, with the invention of the perceptron network and associated learning rule by Frank Rosenblatt [Rose58]. Rosenblatt and his colleagues built a perceptron network and demonstrated its ability to perform pattern recognition. This early success generated a great deal of interest in neural network research. Unfortunately, it was later shown that the basic perceptron network could solve only a limited class of problems. (See Chapter 4 for more on Rosenblatt and the perceptron learning rule.)

At about the same time, Bernard Widrow and Ted Hoff [WiHo60] introduced a new learning algorithm and used it to train adaptive linear neural networks, which were similar in structure and capability to Rosenblatt's perceptron. The Widrow-Hoff learning rule is still in use today. (See Chapter 10 for more on Widrow-Hoff learning.)

Unfortunately, both Rosenblatt's and Widrow's networks suffered from the same inherent limitations, which were widely publicized in a book by Marvin Minsky and Seymour Papert [MiPa69]. Rosenblatt and Widrow were aware of these limitations and proposed new networks that would overcome them. However, they were not able to successfully modify their learning algorithms to train the more complex networks.

Many people, influenced by Minsky and Papert, believed that further research on neural networks was a dead end. This, combined with the fact that there were no powerful digital computers on which to experiment, caused many researchers to leave the field. For a decade neural network research was largely suspended.

Some important work, however, did continue during the 1970s. In 1972 Teuvo Kohonen [Koho72] and James Anderson [Ande72] independently and separately developed new neural networks that could act as memories. (See Chapter 15 and Chapter 16 for more on Kohonen networks.) Stephen Grossberg [Gros76] was also very active during this period in the investigation of self-organizing networks. (See Chapter 18 and Chapter 19.)

Interest in neural networks had faltered during the late 1960s because of the lack of new ideas and powerful computers with which to experiment. During the 1980s both of these impediments were overcome, and research in neural networks increased dramatically. New personal computers and

workstations, which rapidly grew in capability, became widely available. In addition, important new concepts were introduced.

Two new concepts were most responsible for the rebirth of neural networks. The first was the use of statistical mechanics to explain the operation of a certain class of recurrent network, which could be used as an associative memory. This was described in a seminal paper by physicist John Hopfield [Hopf82]. (Chapter 20 and Chapter 21 discuss these Hopfield networks.)

The second key development of the 1980s was the backpropagation algorithm for training multilayer perceptron networks, which was discovered independently by several different researchers. The most influential publication of the backpropagation algorithm was by David Rumelhart and James McClelland [RuMc86]. This algorithm was the answer to the criticisms Minsky and Papert had made in the 1960s. (See Chapter 11 for a development of the backpropagation algorithm.)

These new developments reinvigorated the field of neural networks. Since the 1980s, thousands of papers have been written, neural networks have found countless applications, and the field has been buzzing with new theoretical and practical work.

The brief historical account given above is not intended to identify all of the major contributors, but is simply to give the reader some feel for how knowledge in the neural network field has progressed. As one might note, the progress has not always been “slow but sure.” There have been periods of dramatic progress and periods when relatively little has been accomplished.

Many of the advances in neural networks have had to do with new concepts, such as innovative architectures and training rules. Just as important has been the availability of powerful new computers on which to test these new concepts.

Well, so much for the history of neural networks to this date. The real question is, “What will happen in the future?” Neural networks have clearly taken a permanent place as important mathematical/engineering tools. They don’t provide solutions to every problem, but they are essential tools to be used in appropriate situations. In addition, remember that we still know very little about how the brain works. The most important advances in neural networks almost certainly lie in the future.

The large number and wide variety of applications of this technology are very encouraging. The next section describes some of these applications.

Applications

A newspaper article described the use of neural networks in literature research by Aston University. It stated that “the network can be taught to recognize individual writing styles, and the researchers used it to compare works attributed to Shakespeare and his contemporaries.” A popular science television program documented the use of neural networks by an Italian research institute to test the purity of olive oil. Google uses neural networks for image tagging (automatically identifying an image and assigning keywords), and Microsoft has developed neural networks that can help convert spoken English speech into spoken Chinese speech. Researchers at Lund University and Skåne University Hospital in Sweden have used neural networks to improve long-term survival rates for heart transplant recipients by identifying optimal recipient and donor matches. These examples are indicative of the broad range of applications that can be found for neural networks. The applications are expanding because neural networks are good at solving problems, not just in engineering, science and mathematics, but in medicine, business, finance and literature as well. Their application to a wide variety of problems in many fields makes them very attractive. Also, faster computers and faster algorithms have made it possible to use neural networks to solve complex industrial problems that formerly required too much computation.

The following note and Table of Neural Network Applications are reproduced here from the *Neural Network Toolbox* for MATLAB with the permission of the MathWorks, Inc.

A 1988 DARPA Neural Network Study [DARP88] lists various neural network applications, beginning with the adaptive channel equalizer in about 1984. This device, which is an outstanding commercial success, is a single-neuron network used in long distance telephone systems to stabilize voice signals. The DARPA report goes on to list other commercial applications, including a small word recognizer, a process monitor, a sonar classifier and a risk analysis system.

Thousands of neural networks have been applied in hundreds of fields in the many years since the DARPA report was written. A list of some of those applications follows.

Aerospace

High performance aircraft autopilots, flight path simulations, aircraft control systems, autopilot enhancements, aircraft component simulations, aircraft component fault detectors

Automotive

Automobile automatic guidance systems, fuel injector control, automatic braking systems, misfire detection, virtual emission sensors, warranty activity analyzers

Banking

Check and other document readers, credit application evaluators, cash forecasting, firm classification, exchange rate forecasting, predicting loan recovery rates, measuring credit risk

Defense

Weapon steering, target tracking, object discrimination, facial recognition, new kinds of sensors, sonar, radar and image signal processing including data compression, feature extraction and noise suppression, signal/image identification

Electronics

Code sequence prediction, integrated circuit chip layout, process control, chip failure analysis, machine vision, voice synthesis, nonlinear modeling

Entertainment

Animation, special effects, market forecasting

Financial

Real estate appraisal, loan advisor, mortgage screening, corporate bond rating, credit line use analysis, portfolio trading program, corporate financial analysis, currency price prediction

Insurance

Policy application evaluation, product optimization

Manufacturing

Manufacturing process control, product design and analysis, process and machine diagnosis, real-time particle identification, visual quality inspection systems, beer testing, welding quality analysis, paper quality prediction, computer chip quality analysis, analysis of grinding operations, chemical product design analysis, machine maintenance analysis, project bidding, planning and management, dynamic modeling of chemical process systems

Medical

Breast cancer cell analysis, EEG and ECG analysis, prosthesis design, optimization of transplant times, hospital expense reduction, hospital quality improvement, emergency room test advisement

Oil and Gas

Exploration, smart sensors, reservoir modeling, well treatment decisions, seismic interpretation

Robotics

Trajectory control, forklift robot, manipulator controllers, vision systems, autonomous vehicles

Speech

Speech recognition, speech compression, vowel classification, text to speech synthesis

Securities

Market analysis, automatic bond rating, stock trading advisory systems

Telecommunications

Image and data compression, automated information services, real-time translation of spoken language, customer payment processing systems

Transportation

Truck brake diagnosis systems, vehicle scheduling, routing systems

Conclusion

The number of neural network applications, the money that has been invested in neural network software and hardware, and the depth and breadth of interest in these devices is enormous.

Biological Inspiration

The artificial neural networks discussed in this text are only remotely related to their biological counterparts. In this section we will briefly describe those characteristics of brain function that have inspired the development of artificial neural networks.

The brain consists of a large number (approximately 10^{11}) of highly connected elements (approximately 10^4 connections per element) called neurons. For our purposes these neurons have three principal components: the dendrites, the cell body and the axon. The dendrites are tree-like receptive networks of nerve fibers that carry electrical signals into the cell body. The cell body effectively sums and thresholds these incoming signals. The axon is a single long fiber that carries the signal from the cell body out to other neurons. The point of contact between an axon of one cell and a dendrite of another cell is called a synapse. It is the arrangement of neurons and the strengths of the individual synapses, determined by a complex chemical process, that establishes the function of the neural network. Figure 1.1 is a simplified schematic diagram of two biological neurons.

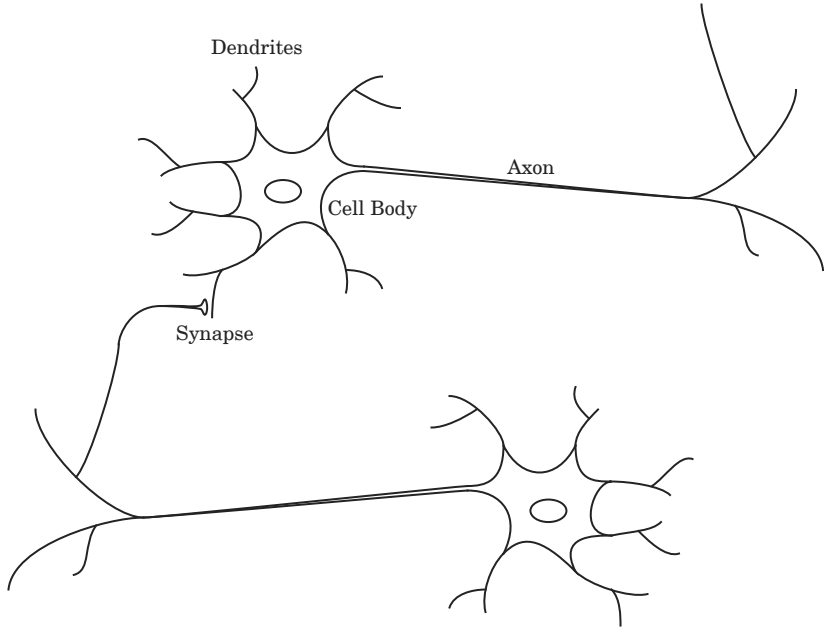


Figure 1.1 Schematic Drawing of Biological Neurons

Some of the neural structure is defined at birth. Other parts are developed through learning, as new connections are made and others waste away. This development is most noticeable in the early stages of life. For example,

it has been shown that if a young cat is denied use of one eye during a critical window of time, it will never develop normal vision in that eye. Linguists have discovered that infants over six months of age can no longer discriminate certain speech sounds, unless they were exposed to them earlier in life [WeTe84].

Neural structures continue to change throughout life. These later changes tend to consist mainly of strengthening or weakening of synaptic junctions. For instance, it is believed that new memories are formed by modification of these synaptic strengths. Thus, the process of learning a new friend's face consists of altering various synapses. Neuroscientists have discovered [MaGa2000], for example, that the hippocampi of London taxi drivers are significantly larger than average. This is because they must memorize a large amount of navigational information—a process that takes more than two years.

Artificial neural networks do not approach the complexity of the brain. There are, however, two key similarities between biological and artificial neural networks. First, the building blocks of both networks are simple computational devices (although artificial neurons are much simpler than biological neurons) that are highly interconnected. Second, the connections between neurons determine the function of the network. The primary objective of this book will be to determine the appropriate connections to solve particular problems.

It is worth noting that even though biological neurons are very slow when compared to electrical circuits (10^{-3} s compared to 10^{-10} s), the brain is able to perform many tasks much faster than any conventional computer. This is in part because of the massively parallel structure of biological neural networks; all of the neurons are operating at the same time. Artificial neural networks share this parallel structure. Even though most artificial neural networks are currently implemented on conventional digital computers, their parallel structure makes them ideally suited to implementation using VLSI, optical devices and parallel processors.

In the following chapter we will introduce our basic artificial neuron and will explain how we can combine such neurons to form networks. This will provide a background for Chapter 3, where we take our first look at neural networks in action.

Further Reading

[Ande72] J. A. Anderson, "A simple neural network generating an interactive memory," *Mathematical Biosciences*, Vol. 14, pp. 197–220, 1972.

Anderson proposed a "linear associator" model for associative memory. The model was trained, using a generalization of the Hebb postulate, to learn an association between input and output vectors. The physiological plausibility of the network was emphasized. Kohonen published a closely related paper at the same time [Koho72], although the two researchers were working independently.

[AnRo88] J. A. Anderson and E. Rosenfeld, *Neurocomputing: Foundations of Research*, Cambridge, MA: MIT Press, 1989.

Neurocomputing is a fundamental reference book. It contains over forty of the most important neurocomputing writings. Each paper is accompanied by an introduction that summarizes its results and gives a perspective on the position of the paper in the history of the field.

[DARP88] *DARPA Neural Network Study*, Lexington, MA: MIT Lincoln Laboratory, 1988.

This study is a compendium of knowledge of neural networks as they were known to 1988. It presents the theoretical foundations of neural networks and discusses their current applications. It contains sections on associative memories, recurrent networks, vision, speech recognition, and robotics. Finally, it discusses simulation tools and implementation technology.

[Gros76] S. Grossberg, "Adaptive pattern classification and universal recoding: I. Parallel development and coding of neural feature detectors," *Biological Cybernetics*, Vol. 23, pp. 121–134, 1976.

Grossberg describes a self-organizing neural network based on the visual system. The network, which consists of short-term and long-term memory mechanisms, is a continuous-time competitive network. It forms a basis for the adaptive resonance theory (ART) networks.

[Gros80] S. Grossberg, "How does the brain build a cognitive code?" *Psychological Review*, Vol. 88, pp. 375–407, 1980.

Grossberg's 1980 paper proposes neural structures and mechanisms that can explain many physiological behaviors including spatial frequency adaptation, binocular rivalry, etc. His systems perform error correction by themselves, without outside help.

[Hebb 49] D. O. Hebb, *The Organization of Behavior*. New York: Wiley, 1949.

The main premise of this seminal book is that behavior can be explained by the action of neurons. In it, Hebb proposed one of the first learning laws, which postulated a mechanism for learning at the cellular level.

Hebb proposes that classical conditioning in biology is present because of the properties of individual neurons.

[Hopf82] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences*, Vol. 79, pp. 2554–2558, 1982.

Hopfield describes a content-addressable neural network. He also presents a clear picture of how his neural network operates, and of what it can do.

[Koho72] T. Kohonen, "Correlation matrix memories," *IEEE Transactions on Computers*, vol. 21, pp. 353–359, 1972.

Kohonen proposed a correlation matrix model for associative memory. The model was trained, using the outer product rule (also known as the Hebb rule), to learn an association between input and output vectors. The mathematical structure of the network was emphasized. Anderson published a closely related paper at the same time [Ande72], although the two researchers were working independently.

[MaGa00] E. A. Maguire, D. G. Gadian, I. S. Johnsrude, C. D. Good, J. Ashburner, R. S. J. Frackowiak, and C. D. Frith, "Navigation-related structural change in the hippocampi of taxi drivers," Proceedings of the National Academy of Sciences, Vol. 97, No. 8, pp. 4398-4403, 2000.

Taxi drivers in London must undergo extensive training, learning how to navigate between thousands of places in the city. This training is colloquially known as "being on The Knowledge" and takes about 2 years to acquire on av-

erage. This study demonstrated that the posterior hippocampi of London taxi drivers were significantly larger relative to those of control subjects.

[McPi43] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics.*, Vol. 5, pp. 115–133, 1943.

This article introduces the first mathematical model of a neuron, in which a weighted sum of input signals is compared to a threshold to determine whether or not the neuron fires. This was the first attempt to describe what the brain does, based on computing elements known at the time. It shows that simple neural networks can compute any arithmetic or logical function.

[MiPa69] M. Minsky and S. Papert, *Perceptrons*, Cambridge, MA: MIT Press, 1969.

A landmark book that contains the first rigorous study devoted to determining what a perceptron network is capable of learning. A formal treatment of the perceptron was needed both to explain the perceptron's limitations and to indicate directions for overcoming them. Unfortunately, the book pessimistically predicted that the limitations of perceptrons indicated that the field of neural networks was a dead end. Although this was not true it temporarily cooled research and funding for research for several years.

[Rose58] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, Vol. 65, pp. 386–408, 1958.

Rosenblatt presents the first practical artificial neural network — the perceptron.

[RuMc86] D. E. Rumelhart and J. L. McClelland, eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1, Cambridge, MA: MIT Press, 1986.

One of the two key influences in the resurgence of interest in the neural network field during the 1980s. Among other topics, it presents the backpropagation algorithm for training multilayer networks.

[WeTe84] J. F. Werker and R. C. Tees, "Cross-language speech perception: Evidence for perceptual reorganization during the first year of life," Infant Behavior and Development, Vol. 7, pp. 49-63, 1984.

This work describes an experiment in which infants from the Interior Salish ethnic group in British Columbia, and other infants outside that group, were tested on their ability to discriminate two different sounds from the Thompson language, which is spoken by the Interior Salish. The researchers discovered that infants less than 6 or 8 months of age were generally able to distinguish the sounds, whether or not they were Interior Salish. By 10 to 12 months of age, only the Interior Salish children were able to distinguish the two sounds.

[WiHo60] B. Widrow and M. E. Hoff, "Adaptive switching circuits,"*1960 IRE WESCON Convention Record*, New York: IRE Part 4, pp. 96–104, 1960.

This seminal paper describes an adaptive perceptron-like network that can learn quickly and accurately. The authors assume that the system has inputs and a desired output classification for each input, and that the system can calculate the error between the actual and desired output. The weights are adjusted, using a gradient descent method, so as to minimize the mean square error. (Least Mean Square error or LMS algorithm.)

This paper is reprinted in [AnRo88].

2 Neuron Model and Network Architectures

Objectives

In Chapter 1 we presented a simplified description of biological neurons and neural networks. Now we will introduce our simplified mathematical model of the neuron and will explain how these artificial neurons can be interconnected to form a variety of network architectures. We will also illustrate the basic operation of these networks through some simple examples. The concepts and notation introduced in this chapter will be used throughout this book.

This chapter does not cover all of the architectures that will be used in this book, but it does present the basic building blocks. More complex architectures will be introduced and discussed as they are needed in later chapters. Even so, a lot of detail is presented here. Please note that it is not necessary for the reader to memorize all of the material in this chapter on a first reading. Instead, treat it as a sample to get you started and a resource to which you can return.

Theory and Examples

Notation

Unfortunately, there is no single neural network notation that is universally accepted. Papers and books on neural networks have come from many diverse fields, including engineering, physics, psychology and mathematics, and many authors tend to use vocabulary peculiar to their specialty. As a result, many books and papers in this field are difficult to read, and concepts are made to seem more complex than they actually are. This is a shame, as it has prevented the spread of important new ideas. It has also led to more than one "reinvention of the wheel."

In this book we have tried to use standard notation where possible, to be clear and to keep matters simple without sacrificing rigor. In particular, we have tried to define practical conventions and use them consistently.

Figures, mathematical equations and text discussing both figures and mathematical equations will use the following notation:

Scalars — small *italic* letters: a,b,c

Vectors — small **bold** nonitalic letters: **a,b,c**

Matrices — capital **BOLD** nonitalic letters: **A,B,C**

Additional notation concerning the network architectures will be introduced as you read this chapter. A complete list of the notation that we use throughout the book is given in Appendix B, so you can look there if you have a question.

Neuron Model

Single-Input Neuron

Weight
Bias
Net Input
Transfer Function

A single-input neuron is shown in Figure 2.1. The scalar input p is multiplied by the scalar *weight* w to form wp, one of the terms that is sent to the summer. The other input, 1, is multiplied by a *bias* b and then passed to the summer. The summer output n, often referred to as the *net input*, goes into a *transfer function* f, which produces the scalar neuron output a. (Some authors use the term "activation function" rather than *transfer function* and "offset" rather than *bias*.)

If we relate this simple model back to the biological neuron that we discussed in Chapter 1, the weight w corresponds to the strength of a synapse, the cell body is represented by the summation and the transfer function, and the neuron output a represents the signal on the axon.

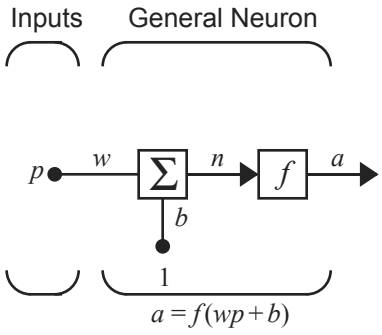


Figure 2.1 Single-Input Neuron

The neuron output is calculated as

$$a = f(wp + b).$$

If, for instance, $w = 3$, $p = 2$ and $b = -1.5$, then

$$a = f(3(2) - 1.5) = f(4.5)$$

The actual output depends on the particular transfer function that is chosen. We will discuss transfer functions in the next section.

The bias is much like a weight, except that it has a constant input of 1. However, if you do not want to have a bias in a particular neuron, it can be omitted. We will see examples of this in Chapters 3, 7 and 16.

Note that w and b are both *adjustable* scalar parameters of the neuron. Typically the transfer function is chosen by the designer and then the parameters w and b will be adjusted by some learning rule so that the neuron input/output relationship meets some specific goal (see Chapter 4 for an introduction to learning rules). As described in the following section, we have different transfer functions for different purposes.

Transfer Functions

The transfer function in Figure 2.1 may be a linear or a nonlinear function of n. A particular transfer function is chosen to satisfy some specification of the problem that the neuron is attempting to solve.

A variety of transfer functions have been included in this book. Three of the most commonly used functions are discussed below.

Hard Limit Transfer Function

The *hard limit transfer function*, shown on the left side of Figure 2.2, sets the output of the neuron to 0 if the function argument is less than 0, or 1 if its argument is greater than or equal to 0. We will use this function to create neurons that classify inputs into two distinct categories. It will be used extensively in Chapter 4.

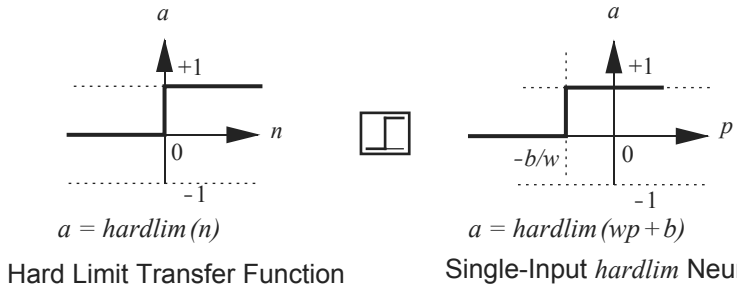


Figure 2.2 Hard Limit Transfer Function

The graph on the right side of Figure 2.2 illustrates the input/output characteristic of a single-input neuron that uses a hard limit transfer function. Here we can see the effect of the weight and the bias. Note that an icon for the hard limit transfer function is shown between the two figures. Such icons will replace the general f in network diagrams to show the particular transfer function that is being used.

Linear Transfer Function

The output of a *linear transfer function* is equal to its input:

$$a = n, \tag{2.1}$$

as illustrated in Figure 2.3.

Neurons with this transfer function are used in the ADALINE networks, which are discussed in Chapter 10.

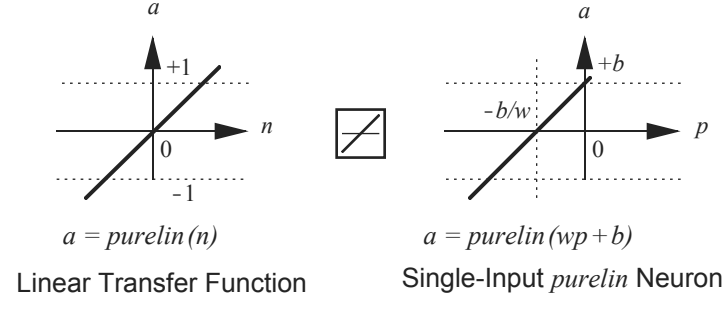


Figure 2.3 Linear Transfer Function

The output (a) versus input (p) characteristic of a single-input linear neuron with a bias is shown on the right of Figure 2.3.

Log-Sigmoid Transfer Function

The *log-sigmoid transfer function* is shown in Figure 2.4.

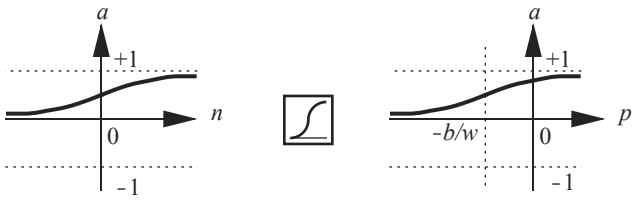


Figure 2.4 Log-Sigmoid Transfer Function

This transfer function takes the input (which may have any value between plus and minus infinity) and squashes the output into the range 0 to 1, according to the expression:

$$a = \frac{1}{1 + e^{-n}}. \tag{2.2}$$

The log-sigmoid transfer function is commonly used in multilayer networks that are trained using the backpropagation algorithm, in part because this function is differentiable (see Chapter 11).

Most of the transfer functions used in this book are summarized in Table 2.1. Of course, you can define other transfer functions in addition to those shown in Table 2.1 if you wish.

To experiment with a single-input neuron, use the Neural Network Design Demonstration One-Input Neuron **nnd2n1.**

Name	Input/Output Relation	Icon	MATLAB Function
Hard Limit	$a = 0 \quad n < 0$ $a = 1 \quad n \geq 0$		hardlim
Symmetrical Hard Limit	$a = -1 \quad n < 0$ $a = +1 \quad n \geq 0$		hardlims
Linear	$a = n$		purelin
Saturating Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n \leq 1$ $a = 1 \quad n > 1$		satlin
Symmetric Saturating Linear	$a = -1 \quad n < -1$ $a = n \quad -1 \leq n \leq 1$ $a = 1 \quad n > 1$		satlins
Log-Sigmoid	$a = \frac{1}{1 + e^{-n}}$		logsig
Hyperbolic Tangent Sigmoid	$a = \frac{e^{n} - e^{-n}}{e^{n} + e^{-n}}$		tansig
Positive Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n$		poslin
Competitive	$a = 1$ neuron with max n $a = 0$ all other neurons		compet

Table 2.1 Transfer Functions

Multiple-Input Neuron

Typically, a neuron has more than one input. A neuron with R inputs is shown in Figure 2.5. The individual inputs $p_1, p_2, \ldots, p_R$ are each weighted by corresponding elements $w_{1,1}, w_{1,2}, \ldots, w_{1,R}$ of the *weight matrix* $\mathbf{W}$.

Weight Matrix

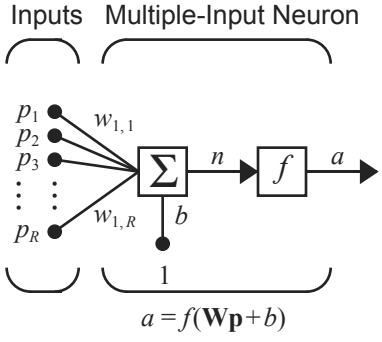


Figure 2.5 Multiple-Input Neuron

The neuron has a bias b, which is summed with the weighted inputs to form the net input n:

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \cdots + w_{1,R}p_R + b . \tag{2.3}$$

This expression can be written in matrix form:

$$n = \mathbf{W}\mathbf{p} + b , \tag{2.4}$$

where the matrix $\mathbf{W}$ for the single neuron case has only one row.

Now the neuron output can be written as

$$a = f(\mathbf{W}\mathbf{p} + b) . \tag{2.5}$$

Fortunately, neural networks can often be described with matrices. This kind of matrix expression will be used throughout the book. Don't be concerned if you are rusty with matrix and vector operations. We will review these topics in Chapters 5 and 6, and we will provide many examples and solved problems that will spell out the procedures.

Weight Indices

We have adopted a particular convention in assigning the indices of the elements of the weight matrix. The first index indicates the particular neuron destination for that weight. The second index indicates the source of the signal fed to the neuron. Thus, the indices in $w_{1,2}$ say that this weight represents the connection *to* the first (and only) neuron *from* the second source. Of course, this convention is more useful if there is more than one neuron, as will be the case later in this chapter.

Abbreviated Notation

We would like to draw networks with several neurons, each having several inputs. Further, we would like to have more than one layer of neurons. You can imagine how complex such a network might appear if all the lines were drawn. It would take a lot of ink, could hardly be read, and the mass of detail might obscure the main features. Thus, we will use an *abbreviated notation*. A multiple-input neuron using this notation is shown in Figure 2.6.

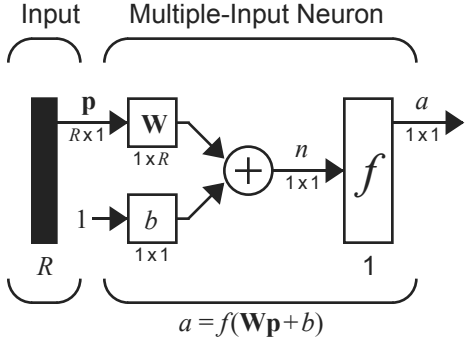


Figure 2.6 Neuron with R Inputs, Abbreviated Notation

As shown in Figure 2.6, the input vector $\mathbf{p}$ is represented by the solid vertical bar at the left. The dimensions of $\mathbf{p}$ are displayed below the variable as $R \times 1$, indicating that the input is a single vector of R elements. These inputs go to the weight matrix $\mathbf{W}$, which has R columns but only one row in this single neuron case. A constant 1 enters the neuron as an input and is multiplied by a scalar bias b. The net input to the transfer function f is n, which is the sum of the bias b and the product $\mathbf{Wp}$. The neuron's output a is a scalar in this case. If we had more than one neuron, the network output would be a vector.

The dimensions of the variables in these abbreviated notation figures will always be included, so that you can tell immediately if we are talking about a scalar, a vector or a matrix. You will not have to guess the kind of variable or its dimensions.

Note that the number of inputs to a network is set by the external specifications of the problem. If, for instance, you want to design a neural network that is to predict kite-flying conditions and the inputs are air temperature, wind velocity and humidity, then there would be three inputs to the network.

To experiment with a two-input neuron, use the Neural Network Design Demonstration Two-Input Neuron (**nnd2n2**)**.**

Network Architectures

Commonly one neuron, even with many inputs, may not be sufficient. We might need five or ten, operating in parallel, in what we will call a "layer." This concept of a layer is discussed below.

A Layer of Neurons

Layer

A single-*layer* network of S neurons is shown in Figure 2.7. Note that each of the R inputs is connected to each of the neurons and that the weight matrix now has S rows.

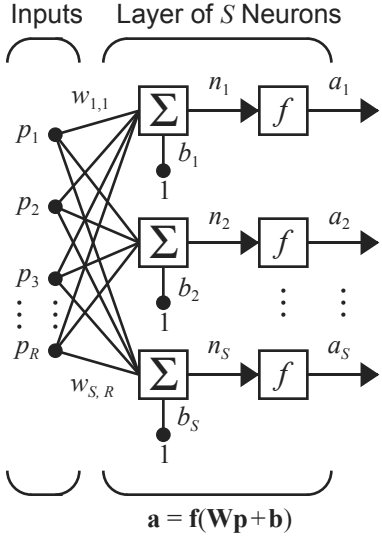


Figure 2.7 Layer of S Neurons

The layer includes the weight matrix, the summers, the bias vector $\mathbf{b}$, the transfer function boxes and the output vector $\mathbf{a}$. Some authors refer to the inputs as another layer, but we will not do that here.

Each element of the input vector $\mathbf{p}$ is connected to each neuron through the weight matrix $\mathbf{W}$. Each neuron has a bias b_i, a summer, a transfer function f and an output a_i. Taken together, the outputs form the output vector $\mathbf{a}$.

It is common for the number of inputs to a layer to be different from the number of neurons (i.e., $R \neq S$).

You might ask if all the neurons in a layer must have the same transfer function. The answer is no; you can define a single (composite) layer of neurons having different transfer functions by combining two of the networks

shown above in parallel. Both networks would have the same inputs, and each network would create some of the outputs.

The input vector elements enter the network through the weight matrix **W**:

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}. \tag{2.6}$$

As noted previously, the row indices of the elements of matrix **W** indicate the destination neuron associated with that weight, while the column indices indicate the source of the input for that weight. Thus, the indices in $w_{3,2}$ say that this weight represents the connection *to* the third neuron *from* the second source.

Fortunately, the *S*-neuron, *R*-input, one-layer network also can be drawn in abbreviated notation, as shown in Figure 2.8.

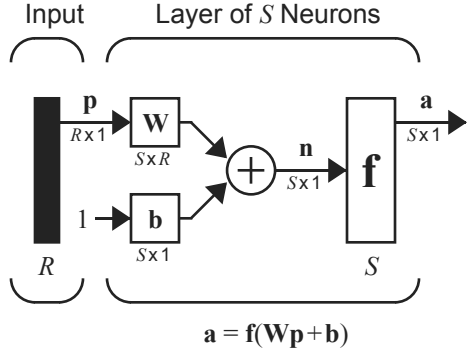


Figure 2.8 Layer of S Neurons, Abbreviated Notation

Here again, the symbols below the variables tell you that for this layer, **p** is a vector of length R, **W** is an $S \times R$ matrix, and **a** and **b** are vectors of length S. As defined previously, the layer includes the weight matrix, the summation and multiplication operations, the bias vector **b**, the transfer function boxes and the output vector.

Multiple Layers of Neurons

Now consider a network with several layers. Each layer has its own weight matrix **W**, its own bias vector **b**, a net input vector **n** and an output vector **a**. We need to introduce some additional notation to distinguish between

Layer Superscript

these layers. We will use superscripts to identify the layers. Specifically, we append the number of the layer as a *superscript* to the names for each of these variables. Thus, the weight matrix for the first layer is written as $\mathbf{W}^1$, and the weight matrix for the second layer is written as $\mathbf{W}^2$. This notation is used in the three-layer network shown in Figure 2.9.

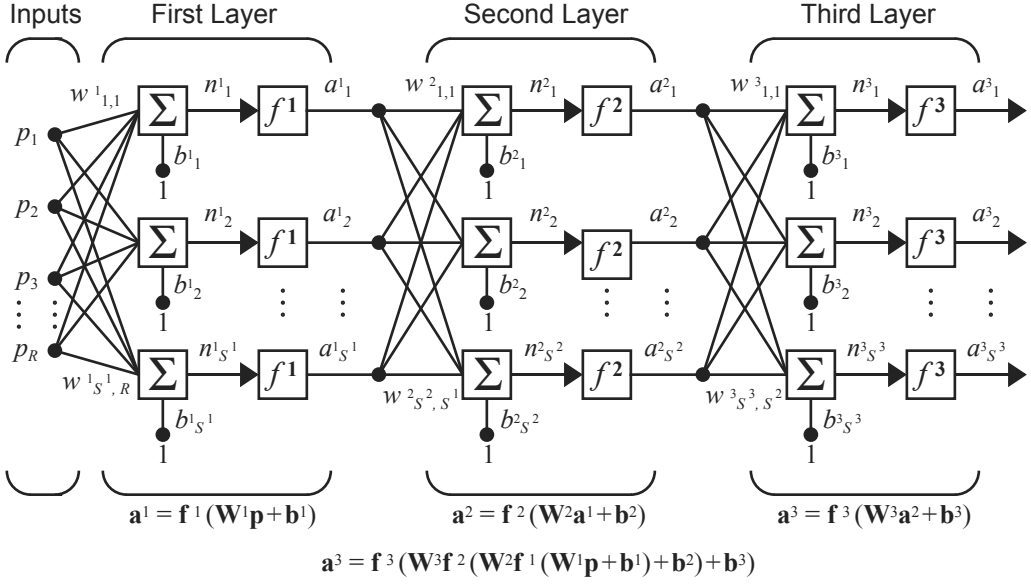


Figure 2.9 Three-Layer Network

As shown, there are R inputs, S^1 neurons in the first layer, S^2 neurons in the second layer, etc. As noted, different layers can have different numbers of neurons.

The outputs of layers one and two are the inputs for layers two and three. Thus layer 2 can be viewed as a one-layer network with $R = S^1$ inputs, $S = S^2$ neurons, and an $S^2 \times S^1$ weight matrix $\mathbf{W}^2$. The input to layer 2 is $\mathbf{a}^1$, and the output is $\mathbf{a}^2$.

Output Layer
Hidden Layers

A layer whose output is the network output is called an *output layer*. The other layers are called *hidden layers*. The network shown above has an output layer (layer 3) and two hidden layers (layers 1 and 2).

The same three-layer network discussed previously also can be drawn using our abbreviated notation, as shown in Figure 2.10.

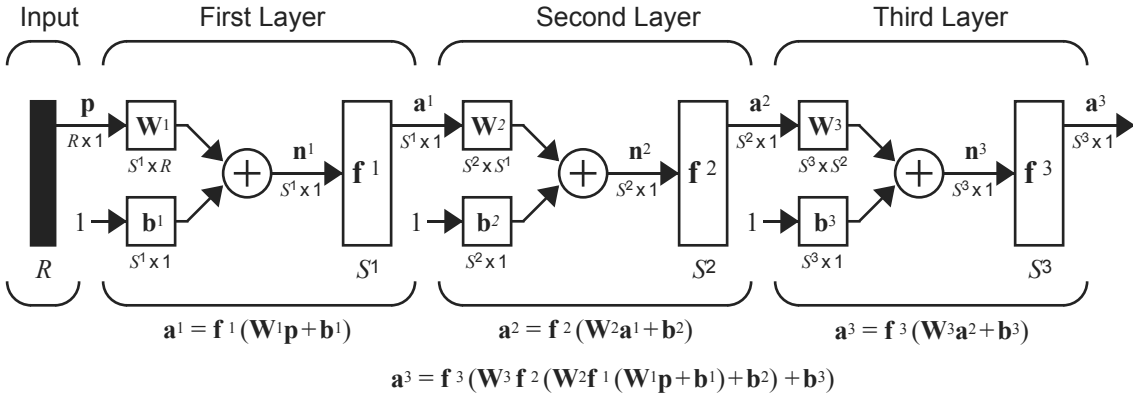


Figure 2.10 Three-Layer Network, Abbreviated Notation

Multilayer networks are more powerful than single-layer networks. For instance, a two-layer network having a sigmoid first layer and a linear second layer can be trained to approximate most functions arbitrarily well. Single-layer networks cannot do this.

At this point the number of choices to be made in specifying a network may look overwhelming, so let us consider this topic. The problem is not as bad as it looks. First, recall that the number of inputs to the network and the number of outputs from the network are defined by external problem specifications. So if there are four external variables to be used as inputs, there are four inputs to the network. Similarly, if there are to be seven outputs from the network, there must be seven neurons in the output layer. Finally, the desired characteristics of the output signal also help to select the transfer function for the output layer. If an output is to be either -1 or 1, then a symmetrical hard limit transfer function should be used. Thus, the architecture of a single-layer network is almost completely determined by problem specifications, including the specific number of inputs and outputs and the particular output signal characteristic.

Now, what if we have more than two layers? Here the external problem does not tell you directly the number of neurons required in the hidden layers. In fact, there are few problems for which one can predict the optimal number of neurons needed in a hidden layer. This problem is an active area of research. We will develop some feeling on this matter as we proceed to Chapter 11, Backpropagation.

As for the number of layers, most practical neural networks have just two or three layers. Four or more layers are used rarely.

We should say something about the use of biases. One can choose neurons with or without biases. The bias gives the network an extra variable, and so you might expect that networks with biases would be more powerful

than those without, and that is true. Note, for instance, that a neuron without a bias will always have a net input n of zero when the network inputs $\mathbf{p}$ are zero. This may not be desirable and can be avoided by the use of a bias. The effect of the bias is discussed more fully in Chapters 3, 4 and 5.

In later chapters we will omit a bias in some examples or demonstrations. In some cases this is done simply to reduce the number of network parameters. With just two variables, we can plot system convergence in a two-dimensional plane. Three or more variables are difficult to display.

Recurrent Networks

Delay

Before we discuss recurrent networks, we need to introduce some simple building blocks. The first is the *delay* block, which is illustrated in Figure 2.11.

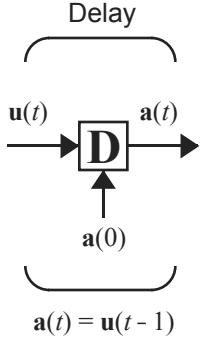


Figure 2.11 Delay Block

The delay output $\mathbf{a}(t)$ is computed from its input $\mathbf{u}(t)$ according to

$$\mathbf{a}(t) = \mathbf{u}(t-1). \tag{2.7}$$

Thus the output is the input delayed by one time step. (This assumes that time is updated in discrete steps and takes on only integer values.) Eq. (2.7) requires that the output be initialized at time $t = 0$. This initial condition is indicated in Figure 2.11 by the arrow coming into the bottom of the delay block.

Integrator

Another related building block, which we will use for the continuous-time recurrent networks in Chapters 18–21, is the *integrator*, which is shown in Figure 2.12.

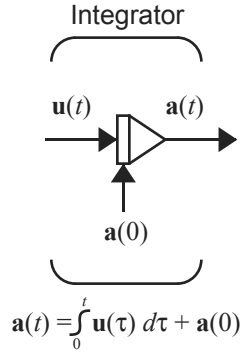


Figure 2.12 Integrator Block

The integrator output $\mathbf{a}(t)$ is computed from its input $\mathbf{u}(t)$ according to

$$\mathbf{a}(t) = \int_0^t \mathbf{u}(\tau)d\tau + \mathbf{a}(0)\,. \tag{2.8}$$

The initial condition $\mathbf{a}(0)$ is indicated by the arrow coming into the bottom of the integrator block.

Recurrent Network

We are now ready to introduce recurrent networks. A *recurrent network* is a network with feedback; some of its outputs are connected to its inputs. This is quite different from the networks that we have studied thus far, which were strictly feedforward with no backward connections. One type of discrete-time recurrent network is shown in Figure 2.13.

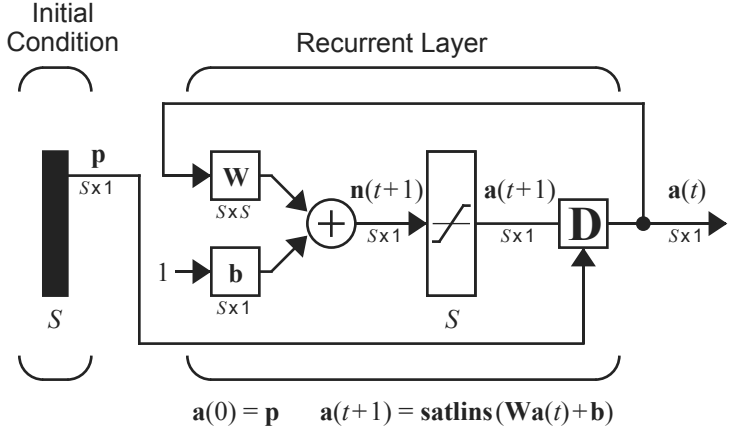


Figure 2.13 Recurrent Network

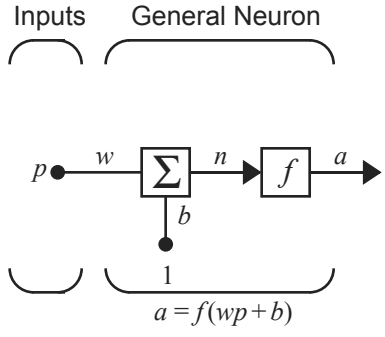
In this particular network the vector $\mathbf{p}$ supplies the initial conditions (i.e., $\mathbf{a}(0) = \mathbf{p}$). Then future outputs of the network are computed from previous outputs:

$$\mathbf{a}(1) = \mathbf{satlins}(\mathbf{W}\mathbf{a}(0) + \mathbf{b}), \ \mathbf{a}(2) = \mathbf{satlins}(\mathbf{W}\mathbf{a}(1) + \mathbf{b}), \ldots$$

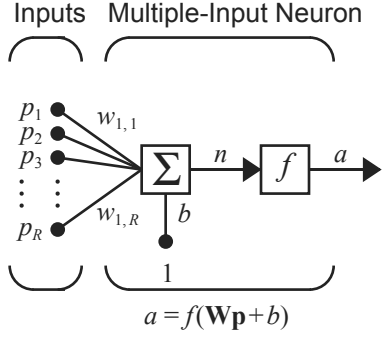
Recurrent networks are potentially more powerful than feedforward networks and can exhibit temporal behavior. These types of networks are discussed in Chapters 3, 14 and 18–21.

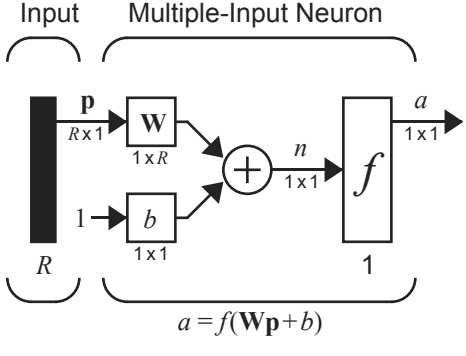
Summary of Results

Single-Input Neuron



Multiple-Input Neuron

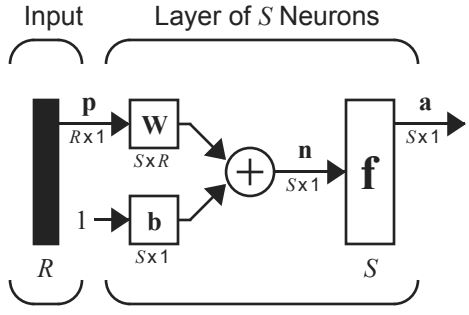




Transfer Functions

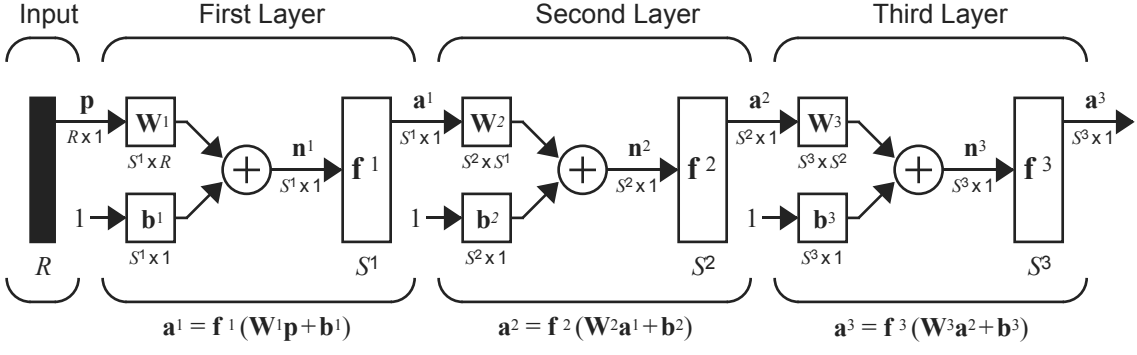
Name	Input/Output Relation	Icon	MATLAB Function
Hard Limit	$a = 0 \quad n < 0$ $a = 1 \quad n \geq 0$		hardlim
Symmetrical Hard Limit	$a = -1 \quad n < 0$ $a = +1 \quad n \geq 0$		hardlims
Linear	$a = n$		purelin
Saturating Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n \leq 1$ $a = 1 \quad n > 1$		satlin
Symmetric Saturating Linear	$a = -1 \quad n < -1$ $a = n \quad -1 \leq n \leq 1$ $a = 1 \quad n > 1$		satlins
Log-Sigmoid	$a = \frac{1}{1 + e^{-n}}$		logsig
Hyperbolic Tangent Sigmoid	$a = \frac{e^{n} - e^{-n}}{e^{n} + e^{-n}}$		tansig
Positive Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n$		poslin
Competitive	$a = 1$ neuron with max n $a = 0$ all other neurons		compet

Layer of Neurons



$$\mathbf{a} = \mathbf{f}(\mathbf{W}\mathbf{p}+\mathbf{b})$$

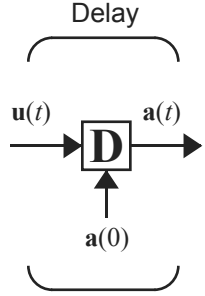
Three Layers of Neurons



$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1) \qquad \mathbf{a}^2 = \mathbf{f}^2(\mathbf{W}^2\mathbf{a}^1+\mathbf{b}^2) \qquad \mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{a}^2+\mathbf{b}^3)$$

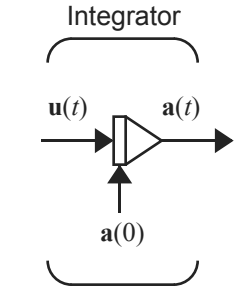
$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$$

Delay

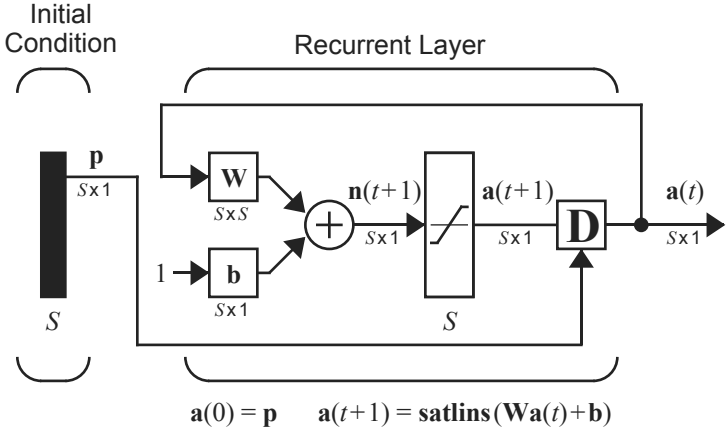


$$\mathbf{a}(t) = \mathbf{u}(t-1)$$

Integrator



Recurrent Network



How to Pick an Architecture

Problem specifications help define the network in the following ways:

1. Number of network inputs = number of problem inputs
2. Number of neurons in output layer = number of problem outputs
3. Output layer transfer function choice at least partly determined by problem specification of the outputs

Solved Problems

P2.1 The input to a single-input neuron is 2.0, its weight is 2.3 and its bias is -3.

i. What is the net input to the transfer function?

ii. What is the neuron output?

i. The net input is given by:

$$n = wp + b = (2.3)(2) + (-3) = 1.6$$

ii. The output cannot be determined because the transfer function is not specified.

P2.2 What is the output of the neuron of P2.1 if it has the following transfer functions?

i. Hard limit

ii. Linear

iii. Log-sigmoid

i. For the hard limit transfer function:

$$a = hardlim(1.6) = 1.0$$

ii. For the linear transfer function:

$$a = purelin(1.6) = 1.6$$

iii. For the log-sigmoid transfer function:

$$a = logsig(1.6) = \frac{1}{1 + e^{-1.6}} = 0.8320$$

Verify this result using MATLAB and the function **`logsig`**, which is in the MININNET directory (see Appendix B).

P2.3 Given a two-input neuron with the following parameters: $b = 1.2$**,** $\mathbf{W} = \begin{bmatrix} 3 & 2 \end{bmatrix}$ **and** $\mathbf{p} = \begin{bmatrix} -5 & 6 \end{bmatrix}^T$**, calculate the neuron output for the following transfer functions:**

i. A symmetrical hard limit transfer function

ii. A saturating linear transfer function

iii. A hyperbolic tangent sigmoid (tansig) transfer function

First calculate the net input n:

$$n = \mathbf{W}\mathbf{p} + b = \begin{bmatrix} 3 & 2 \end{bmatrix} \begin{bmatrix} -5 \\ 6 \end{bmatrix} + (1.2) = -1.8 .$$

Now find the outputs for each of the transfer functions.

i. $a = hardlims(-1.8) = -1$

ii. $a = satlin(-1.8) = 0$

iii. $a = tansig(-1.8) = -0.9468$

P2.4 A single-layer neural network is to have six inputs and two outputs. The outputs are to be limited to and continuous over the range 0 to 1. What can you tell about the network architecture? Specifically:

i. How many neurons are required?

ii. What are the dimensions of the weight matrix?

iii. What kind of transfer functions could be used?

iv. Is a bias required?

The problem specifications allow you to say the following about the network.

i. Two neurons, one for each output, are required.

ii. The weight matrix has two rows corresponding to the two neurons and six columns corresponding to the six inputs. (The product $\mathbf{W}\mathbf{p}$ is a two-element vector.)

iii. Of the transfer functions we have discussed, the *logsig* transfer function would be most appropriate.

iv. Not enough information is given to determine if a bias is required.

Epilogue

This chapter has introduced a simple artificial neuron and has illustrated how different neural networks can be created by connecting groups of neurons in various ways. One of the main objectives of this chapter has been to introduce our basic notation. As the networks are discussed in more detail in later chapters, you may wish to return to Chapter 2 to refresh your memory of the appropriate notation.

This chapter was not meant to be a complete presentation of the networks we have discussed here. That will be done in the chapters that follow. We will begin in Chapter 3, which will present a simple example that uses some of the networks described in this chapter, and will give you an opportunity to see these networks in action. The networks demonstrated in Chapter 3 are representative of the types of networks that are covered in the remainder of this text.

Exercises

E2.1 A single input neuron has a weight of 1.3 and a bias of 3.0. What possible kinds of transfer functions, from Table 2.1, could this neuron have, if its output is given below. In each case, give the value of the input that would produce these outputs.

i. 1.6

ii. 1.0

iii. 0.9963

iv. -1.0

E2.2 Consider a single-input neuron with a bias. We would like the output to be -1 for inputs less than 3 and +1 for inputs greater than or equal to 3.

i. What kind of a transfer function is required?

ii. What bias would you suggest? Is your bias in any way related to the input weight? If yes, how?

iii. Summarize your network by naming the transfer function and stating the bias and the weight. Draw a diagram of the network. Verify the network performance using MATLAB.

E2.3 Given a two-input neuron with the following weight matrix and input vector: $\mathbf{W} = \begin{bmatrix} 3 & 2 \end{bmatrix}$ and $\mathbf{p} = \begin{bmatrix} -5 & 7 \end{bmatrix}^T$, we would like to have an output of 0.5. Do you suppose that there is a combination of bias and transfer function that might allow this?

i. Is there a transfer function from Table 2.1 that will do the job if the bias is zero?

ii. Is there a bias that will do the job if the linear transfer function is used? If yes, what is it?

iii. Is there a bias that will do the job if a log-sigmoid transfer function is used? Again, if yes, what is it?

iv. Is there a bias that will do the job if a symmetrical hard limit transfer function is used? Again, if yes, what is it?

E2.4 A two-layer neural network is to have four inputs and six outputs. The range of the outputs is to be continuous between 0 and 1. What can you tell about the network architecture? Specifically:

i. How many neurons are required in each layer?

ii. What are the dimensions of the first-layer and second-layer weight matrices?

iii. What kinds of transfer functions can be used in each layer?

iv. Are biases required in either layer?

E2.5 Consider the following neuron.

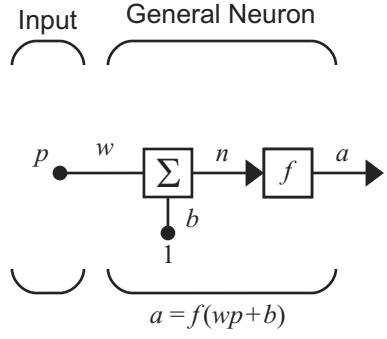


Figure P15.1 General Neuron

Sketch the neuron response (plot a versus p for $-2<p<2$) for the following cases.

i. $w = 1, b = 1, f = hardlims$.

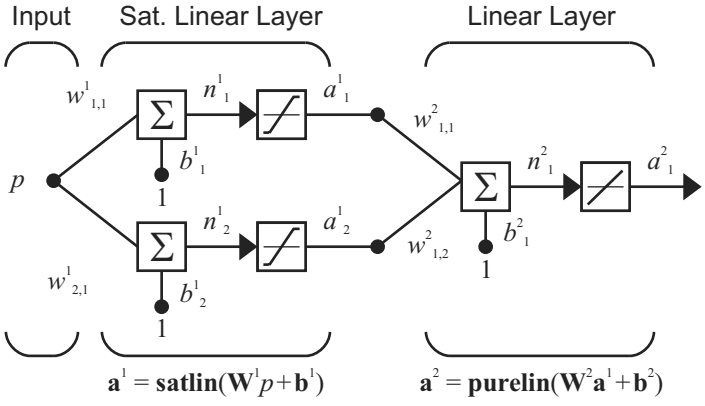
ii. $w = -1, b = 1, f = hardlims$.

iii. $w = 2, b = 3, f = purelin$.

iv. $w = 2, b = 3, f = satlins$.

v. $w = -2, b = -1, f = poslin$.

E2.6 Consider the following neural network.



$$w^1_{1,1} = 2,\ w^1_{2,1} = 1,\ b^1_1 = 2,\ b^1_2 = -1,\ w^2_{1,1} = 1,\ w^2_{1,2} = -1,\ b^2_1 = 0$$

Sketch the following responses (plot the indicated variable versus p for $-3 < p < 3$).

i. n^1_1.

ii. a^1_1.

iii. n^1_2

iv. a^1_2.

v. n^2_1.

vi. a^2_1.

3 An Illustrative Example

Objectives

Think of this chapter as a preview of coming attractions. We will take a simple pattern recognition problem and show how it can be solved using three different neural network architectures. It will be an opportunity to see how the architectures described in the previous chapter can be used to solve a practical (although extremely oversimplified) problem. Do not expect to completely understand these three networks after reading this chapter. We present them simply to give you a taste of what can be done with neural networks, and to demonstrate that there are many different types of networks that can be used to solve a given problem.

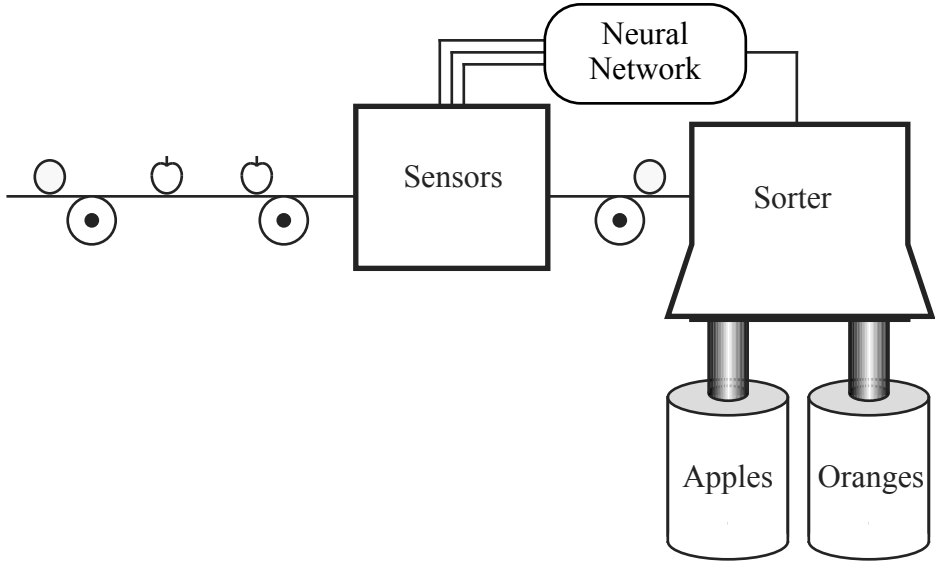
The three networks presented in this chapter are representative of the types of networks discussed in the remaining chapters: feedforward networks (represented here by the perceptron), competitive networks (represented here by the Hamming network) and recurrent associative memory networks (represented here by the Hopfield network).

Theory and Examples

Problem Statement

A produce dealer has a warehouse that stores a variety of fruits and vegetables. When fruit is brought to the warehouse, various types of fruit may be mixed together. The dealer wants a machine that will sort the fruit according to type. There is a conveyer belt on which the fruit is loaded. This conveyer passes through a set of sensors, which measure three properties of the fruit: *shape*, *texture* and *weight*. These sensors are somewhat primitive. The shape sensor will output a 1 if the fruit is approximately round and a -1 if it is more elliptical. The texture sensor will output a 1 if the surface of the fruit is smooth and a -1 if it is rough. The weight sensor will output a 1 if the fruit is more than one pound and a -1 if it is less than one pound.

The three sensor outputs will then be input to a neural network. The purpose of the network is to decide which kind of fruit is on the conveyor, so that the fruit can be directed to the correct storage bin. To make the problem even simpler, let's assume that there are only two kinds of fruit on the conveyor: apples and oranges.



As each fruit passes through the sensors it can be represented by a three-dimensional vector. The first element of the vector will represent shape, the second element will represent texture and the third element will represent weight:

$$\mathbf{p} = \begin{bmatrix} shape \\ texture \\ weight \end{bmatrix}. \tag{3.1}$$

Therefore, a prototype orange would be represented by

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, \tag{3.2}$$

and a prototype apple would be represented by

$$\mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}. \tag{3.3}$$

The neural network will receive one three-dimensional input vector for each fruit on the conveyer and must make a decision as to whether the fruit is an *orange* ($\mathbf{p}_1$) or an *apple* ($\mathbf{p}_2$).

Now that we have defined this simple (trivial?) pattern recognition problem, let's look briefly at three different neural networks that could be used to solve it. The simplicity of our problem will facilitate our understanding of the operation of the networks.

Perceptron

The first network we will discuss is the perceptron. Figure 3.1 illustrates a single-layer perceptron with a symmetric hard limit transfer function *hardlims*.

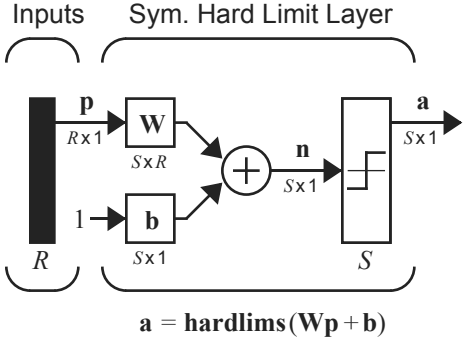


Figure 3.1 Single-Layer Perceptron

Two-Input Case

Before we use the perceptron to solve the orange and apple recognition problem (which will require a three-input perceptron, i.e., $R = 3$), it is useful to investigate the capabilities of a two-input/single-neuron perceptron ($R = 2$), which can be easily analyzed graphically. The two-input perceptron is shown in Figure 3.2.

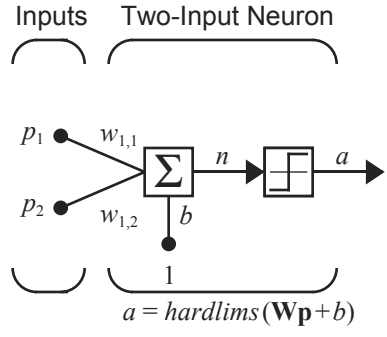


Figure 3.2 Two-Input/Single-Neuron Perceptron

Single-neuron perceptrons can classify input vectors into two categories. For example, for a two-input perceptron, if $w_{1,1} = -1$ and $w_{1,2} = 1$ then

$$a = hardlims(n) = hardlims(\begin{bmatrix}-1 & 1\end{bmatrix}\mathbf{p} + b). \tag{3.4}$$

Therefore, if the inner product of the weight matrix (a single row vector in this case) with the input vector is greater than or equal to $-b$, the output will be 1. If the inner product of the weight vector and the input is less than $-b$, the output will be -1. This divides the input space into two parts. Figure 3.3 illustrates this for the case where $b = -1$. The blue line in the figure represents all points for which the net input n is equal to 0:

$$n = \begin{bmatrix}-1 & 1\end{bmatrix}\mathbf{p} - 1 = 0. \tag{3.5}$$

Notice that this decision boundary will always be orthogonal to the weight matrix, and the position of the boundary can be shifted by changing b. (In the general case, $\mathbf{W}$ is a matrix consisting of a number of row vectors, each of which will be used in an equation like Eq. (3.5). There will be one boundary for each row of $\mathbf{W}$. See Chapter 4 for more on this topic.) The shaded region contains all input vectors for which the output of the network will be 1. The output will be -1 for all other input vectors.

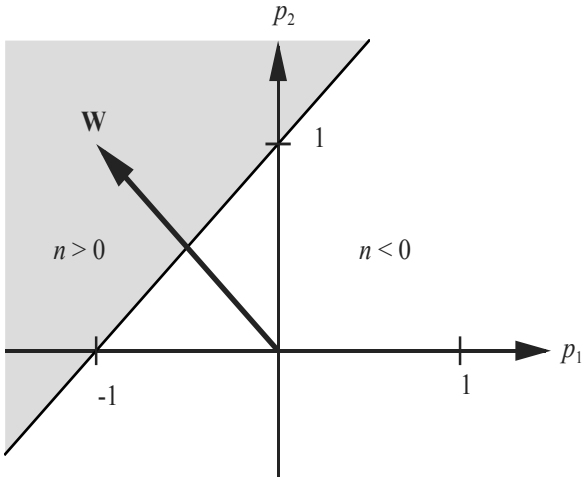


Figure 3.3 Perceptron Decision Boundary

The key property of the single-neuron perceptron, therefore, is that it can separate input vectors into two categories. The decision boundary between the categories is determined by the equation

$$\mathbf{W}\mathbf{p} + b = 0 . \tag{3.6}$$

Because the boundary must be linear, the single-layer perceptron can only be used to recognize patterns that are linearly separable (can be separated by a linear boundary). These concepts will be discussed in more detail in Chapter 4.

Pattern Recognition Example

Now consider the apple and orange pattern recognition problem. Because there are only two categories, we can use a single-neuron perceptron. The vector inputs are three-dimensional ($R = 3$), therefore the perceptron equation will be

$$a = hardlims\left(\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \end{bmatrix}\begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} + b\right) . \tag{3.7}$$

We want to choose the bias b and the elements of the weight matrix so that the perceptron will be able to distinguish between apples and oranges. For example, we may want the output of the perceptron to be 1 when an apple is input and -1 when an orange is input. Using the concept illustrated in Figure 3.3, let's find a linear boundary that can separate oranges and ap-

ples. The two prototype vectors (recall Eq. (3.2) and Eq. (3.3)) are shown in Figure 3.4. From this figure we can see that the linear boundary that divides these two vectors symmetrically is the p_1, p_3 plane.

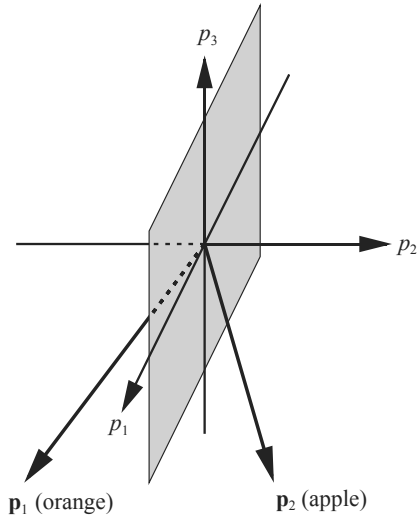


Figure 3.4 Prototype Vectors

The p_1, p_3 plane, which will be our decision boundary, can be described by the equation

$$p_2 = 0, \tag{3.8}$$

or

$$\begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} + 0 = 0. \tag{3.9}$$

Therefore the weight matrix and bias will be

$$\mathbf{W} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}, b = 0. \tag{3.10}$$

The weight matrix is orthogonal to the decision boundary and points toward the region that contains the prototype pattern $\mathbf{p}_2$ (*apple*) for which we want the perceptron to produce an output of 1. The bias is 0 because the decision boundary passes through the origin.

Now let's test the operation of our perceptron pattern classifier. It classifies perfect apples and oranges correctly since

Orange:

$$a = hardlims\left(\begin{bmatrix}0 & 1 & 0\end{bmatrix}\begin{bmatrix}1\\-1\\-1\end{bmatrix} + 0\right) = -1(orange), \tag{3.11}$$

Apple:

$$a = hardlims\left(\begin{bmatrix}0 & 1 & 0\end{bmatrix}\begin{bmatrix}1\\1\\-1\end{bmatrix} + 0\right) = 1(apple). \tag{3.12}$$

But what happens if we put a not-so-perfect orange into the classifier? Let's say that an orange with an elliptical shape is passed through the sensors. The input vector would then be

$$\mathbf{p} = \begin{bmatrix}-1\\-1\\-1\end{bmatrix}. \tag{3.13}$$

The response of the network would be

$$a = hardlims\left(\begin{bmatrix}0 & 1 & 0\end{bmatrix}\begin{bmatrix}-1\\-1\\-1\end{bmatrix} + 0\right) = -1(orange). \tag{3.14}$$

In fact, any input vector that is closer to the orange prototype vector than to the apple prototype vector (in Euclidean distance) will be classified as an orange (and vice versa).

To experiment with the perceptron network and the apple/orange classification problem, use the Neural Network Design Demonstration Perceptron Classification (**nnd3pc**).

This example has demonstrated some of the features of the perceptron network, but by no means have we exhausted our investigation of perceptrons. This network, and variations on it, will be examined in Chapters 4 through 13. Let's consider some of these future topics.

In the apple/orange example we were able to design a network graphically, by choosing a decision boundary that clearly separated the patterns. What about practical problems, with high dimensional input spaces? In Chapters 4, 7, 10 and 11 we will introduce learning algorithms that can be used to

train networks to solve complex problems by using a set of examples of proper network behavior.

The key characteristic of the single-layer perceptron is that it creates linear decision boundaries to separate categories of input vector. What if we have categories that cannot be separated by linear boundaries? This question will be addressed in Chapter 11, where we will introduce the multilayer perceptron. The multilayer networks are able to solve classification problems of arbitrary complexity.

Hamming Network

The next network we will consider is the Hamming network [Lipp87]. It was designed explicitly to solve binary pattern recognition problems (where each element of the input vector has only two possible values — in our example 1 or –1). This is an interesting network, because it uses both feedforward and recurrent (feedback) layers, which were both described in Chapter 2. Figure 3.5 shows the standard Hamming network. Note that the number of neurons in the first layer is the same as the number of neurons in the second layer.

The objective of the Hamming network is to decide which prototype vector is closest to the input vector. This decision is indicated by the output of the recurrent layer. There is one neuron in the recurrent layer for each prototype pattern. When the recurrent layer converges, there will be only one neuron with nonzero output. This neuron indicates the prototype pattern that is closest to the input vector. Now let's investigate the two layers of the Hamming network in detail.

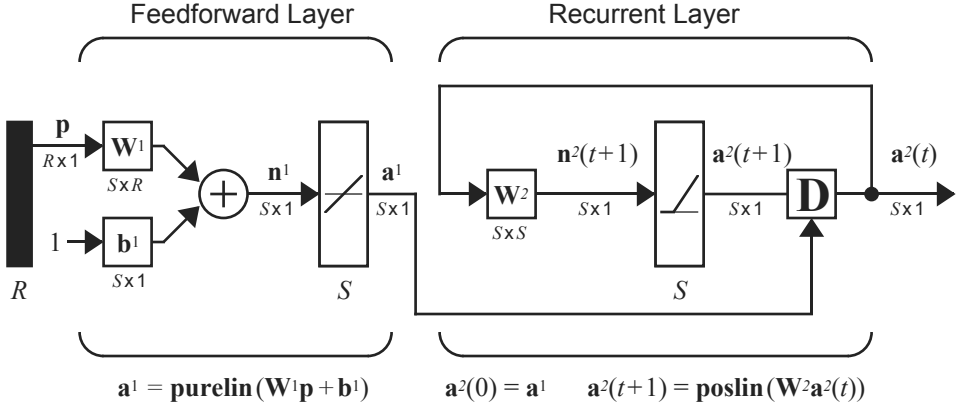


Figure 3.5 Hamming Network

Feedforward Layer

The feedforward layer performs a correlation, or inner product, between each of the prototype patterns and the input pattern (as we will see in Eq. (3.17)). In order for the feedforward layer to perform this correlation, the rows of the weight matrix in the feedforward layer, represented by the connection matrix $\mathbf{W}^1$, are set to the prototype patterns. For our apple and orange example this would mean

$$\mathbf{W}^1 = \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \end{bmatrix} = \begin{bmatrix} 1 & -1 & -1 \\ 1 & 1 & -1 \end{bmatrix}. \tag{3.15}$$

The feedforward layer uses a linear transfer function, and each element of the bias vector is equal to R, where R is the number of elements in the input vector. For our example the bias vector would be

$$\mathbf{b}^1 = \begin{bmatrix} 3 \\ 3 \end{bmatrix}. \tag{3.16}$$

With these choices for the weight matrix and bias vector, the output of the feedforward layer is

$$\mathbf{a}^1 = \mathbf{W}^1\mathbf{p} + \mathbf{b}^1 = \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \end{bmatrix}\mathbf{p} + \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} \mathbf{p}_1^T\mathbf{p} + 3 \\ \mathbf{p}_2^T\mathbf{p} + 3 \end{bmatrix}. \tag{3.17}$$

Note that the outputs of the feedforward layer are equal to the inner products of each prototype pattern with the input, plus R. For two vectors of equal length (norm), their inner product will be largest when the vectors point in the same direction, and will be smallest when they point in opposite directions. (We will discuss this concept in more depth in Chapters 5, 8 and 9.) By adding R to the inner product we guarantee that the outputs of the feedforward layer can never be negative. This is required for proper operation of the recurrent layer.

This network is called the Hamming network because the neuron in the feedforward layer with the largest output will correspond to the prototype pattern that is closest in Hamming distance to the input pattern. (The Hamming distance between two vectors is equal to the number of elements that are different. It is defined only for binary vectors.) We leave it to the reader to show that the outputs of the feedforward layer are equal to $2R$ minus twice the Hamming distances from the prototype patterns to the input pattern.

Recurrent Layer

The recurrent layer of the Hamming network is what is known as a "competitive" layer. The neurons in this layer are initialized with the outputs of the feedforward layer, which indicate the correlation between the prototype patterns and the input vector. Then the neurons compete with each other to determine a winner. After the competition, only one neuron will have a nonzero output. The winning neuron indicates which category of input was presented to the network (for our example the two categories are *apples* and *oranges*). The equations that describe the competition are:

$$\mathbf{a}^2(0) = \mathbf{a}^1 \quad \text{(Initial Condition)}, \tag{3.18}$$

and

$$\mathbf{a}^2(t+1) = \mathbf{poslin}(\mathbf{W}^2\mathbf{a}^2(t)). \tag{3.19}$$

(Don't forget that the superscripts here indicate the layer number, not a power of 2.) The *poslin* transfer function is linear for positive values and zero for negative values. The weight matrix $\mathbf{W}^2$ has the form

$$\mathbf{W}^2 = \begin{bmatrix} 1 & -\varepsilon \\ -\varepsilon & 1 \end{bmatrix}, \tag{3.20}$$

where ε is some number less than $1/(S-1)$, and S is the number of neurons in the recurrent layer. (Can you show why ε must be less than $1/(S-1)$?)

An iteration of the recurrent layer proceeds as follows:

$$\mathbf{a}^2(t+1) = \mathbf{poslin}\left(\begin{bmatrix} 1 & -\varepsilon \\ -\varepsilon & 1 \end{bmatrix}\mathbf{a}^2(t)\right) = \mathbf{poslin}\left(\begin{bmatrix} a_1^2(t) - \varepsilon a_2^2(t) \\ a_2^2(t) - \varepsilon a_1^2(t) \end{bmatrix}\right). \tag{3.21}$$

Each element is reduced by the same fraction of the other. The larger element will be reduced by less, and the smaller element will be reduced by more, therefore the difference between large and small will be increased. The effect of the recurrent layer is to zero out all neuron outputs, except the one with the largest initial value (which corresponds to the prototype pattern that is closest in Hamming distance to the input).

To illustrate the operation of the Hamming network, consider again the oblong orange that we used to test the perceptron:

$$\mathbf{p} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}. \tag{3.22}$$

The output of the feedforward layer will be

$$\mathbf{a}^1 = \begin{bmatrix} 1 & -1 & -1 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} + \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} (1+3) \\ (-1+3) \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}, \tag{3.23}$$

which will then become the initial condition for the recurrent layer.

The weight matrix for the recurrent layer will be given by Eq. (3.20) with $\varepsilon = 1/2$ (any number less than 1 would work). The first iteration of the recurrent layer produces

$$\mathbf{a}^2(1) = \mathbf{poslin}(\mathbf{W}^2\mathbf{a}^2(0)) = \begin{cases} \mathbf{poslin}\left(\begin{bmatrix} 1 & -0.5 \\ -0.5 & 1 \end{bmatrix}\begin{bmatrix} 4 \\ 2 \end{bmatrix}\right) \\ \mathbf{poslin}\left(\begin{bmatrix} 3 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 3 \\ 0 \end{bmatrix} \end{cases}. \tag{3.24}$$

The second iteration produces

$$\mathbf{a}^2(2) = \mathbf{poslin}(\mathbf{W}^2\mathbf{a}^2(1)) = \begin{cases} \mathbf{poslin}\left(\begin{bmatrix} 1 & -0.5 \\ -0.5 & 1 \end{bmatrix}\begin{bmatrix} 3 \\ 0 \end{bmatrix}\right) \\ \mathbf{poslin}\left(\begin{bmatrix} 3 \\ -1.5 \end{bmatrix}\right) = \begin{bmatrix} 3 \\ 0 \end{bmatrix} \end{cases}. \tag{3.25}$$

Since the outputs of successive iterations produce the same result, the network has converged. Prototype pattern number one, the *orange*, is chosen as the correct match, since neuron number one has the only nonzero output. (Recall that the first element of $\mathbf{a}^1$ was $(\mathbf{p}_1^T\mathbf{p} + 3)$.) This is the correct choice, since the Hamming distance from the *orange* prototype to the input pattern is 1, and the Hamming distance from the *apple* prototype to the input pattern is 2.

To experiment with the Hamming network and the apple/orange classification problem, use the Neural Network Design Demonstration Hamming Classification (`nnd3hamc`).

There are a number of networks whose operation is based on the same principles as the Hamming network; that is, where an inner product operation (feedforward layer) is followed by a competitive dynamic layer. These competitive networks will be discussed in Chapters 15, 16, 18 and 19. They are *self-organizing* networks, which can learn to adjust their prototype vectors based on the inputs that have been presented.

Hopfield Network

The final network we will discuss in this brief preview is the Hopfield network. This is a recurrent network that is similar in some respects to the recurrent layer of the Hamming network, but which can effectively perform the operations of both layers of the Hamming network. A diagram of the Hopfield network is shown in Figure 3.6. (This figure is actually a slight variation of the standard Hopfield network. We use this variation because it is somewhat simpler to describe and yet demonstrates the basic concepts.)

The neurons in this network are initialized with the input vector, then the network iterates until the output converges. When the network is operating correctly, the resulting output should be one of the prototype vectors. Therefore, whereas in the Hamming network the nonzero neuron indicates which prototype pattern is chosen, the Hopfield network actually produces the selected prototype pattern at its output.

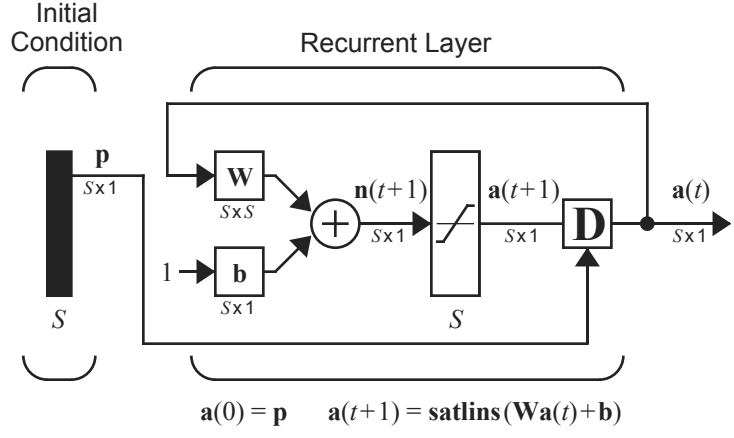


Figure 3.6 Hopfield Network

The equations that describe the network operation are

$$\mathbf{a}(0) = \mathbf{p} \tag{3.26}$$

and

$$\mathbf{a}(t+1) = \mathbf{satlins}(\mathbf{W}\mathbf{a}(t) + \mathbf{b}), \tag{3.27}$$

where *satlins* is the transfer function that is linear in the range [-1, 1] and saturates at 1 for inputs greater than 1 and at -1 for inputs less than -1.

The design of the weight matrix and the bias vector for the Hopfield network is a more complex procedure than it is for the Hamming network, where the weights in the feedforward layer are the prototype patterns. Hopfield design procedures will be discussed in detail in Chapter 21.

To illustrate the operation of the network, we have determined a weight matrix and a bias vector that can solve our orange and apple pattern recognition problem. They are given in Eq. (3.28).

$$\mathbf{W} = \begin{bmatrix} 0.2 & 0 & 0 \\ 0 & 1.2 & 0 \\ 0 & 0 & 0.2 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 0.9 \\ 0 \\ -0.9 \end{bmatrix} \tag{3.28}$$

Although the procedure for computing the weights and biases for the Hopfield network is beyond the scope of this chapter, we can say a few things about why the parameters in Eq. (3.28) work for the apple and orange example.

We want the network output to converge to either the orange pattern, $\mathbf{p}_1$, or the apple pattern, $\mathbf{p}_2$. In both patterns, the first element is 1, and the third element is -1. The difference between the patterns occurs in the second element. Therefore, no matter what pattern is input to the network, we want the first element of the output pattern to converge to 1, the third element to converge to -1, and the second element to go to either 1 or -1, whichever is closer to the second element of the input vector.

The equations of operation of the Hopfield network, using the parameters given in Eq. (3.28), are

$$\begin{aligned} a_1(t+1) &= satlins(0.2a_1(t) + 0.9) \\ a_2(t+1) &= satlins(1.2a_2(t)) \\ a_3(t+1) &= satlins(0.2a_3(t) - 0.9) \end{aligned} \tag{3.29}$$

Regardless of the initial values, $a_i(0)$, the first element will be increased until it saturates at 1, and the third element will be decreased until it saturates at -1. The second element is multiplied by a number larger than 1. Therefore, if it is initially negative, it will eventually saturate at -1; if it is initially positive it will saturate at 1.

(It should be noted that this is not the only (**W**, **b**) pair that could be used. You might want to try some others. See if you can discover what makes these work.)

Let's again take our oblong orange to test the Hopfield network. The outputs of the Hopfield network for the first three iterations would be

$$\mathbf{a}(0) = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}, \mathbf{a}(1) = \begin{bmatrix} 0.7 \\ -1 \\ -1 \end{bmatrix}, \mathbf{a}(2) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, \mathbf{a}(3) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \tag{3.30}$$

The network has converged to the *orange* pattern, as did both the Hamming network and the perceptron, although each network operated in a different way. The perceptron had a single output, which could take on values of -1 (*orange*) or 1 (*apple*). In the Hamming network the single nonzero neuron indicated which prototype pattern had the closest match. If the first neuron was nonzero, that indicated *orange*, and if the second neuron was nonzero, that indicated *apple*. In the Hopfield network the prototype pattern itself appears at the output of the network.

To experiment with the Hopfield network and the apple/orange classification problem, use the Neural Network Design Demonstration Hopfield Classification (**nnd3hopc**).

As with the other networks demonstrated in this chapter, do not expect to feel completely comfortable with the Hopfield network at this point. There are a number of questions that we have not discussed. For example, "How do we know that the network will eventually converge?" It is possible for recurrent networks to oscillate or to have chaotic behavior. In addition, we have not discussed general procedures for designing the weight matrix and the bias vector. These topics will be discussed in detail in Chapters 20 and 21.

Epilogue

The three networks that we have introduced in this chapter demonstrate many of the characteristics that are found in the architectures which are discussed throughout this book.

Feedforward networks, of which the perceptron is one example, are presented in Chapters 4, 7, 11, 12, 13 and 17. In these networks, the output is computed directly from the input in one pass; no feedback is involved. Feedforward networks are used for pattern recognition, as in the apple and orange example, and also for function approximation (see Chapter 11). Function approximation applications are found in such areas as adaptive filtering (see Chapter 10) and automatic control.

Competitive networks, represented here by the Hamming network, are characterized by two properties. First, they compute some measure of distance between stored prototype patterns and the input pattern. Second, they perform a competition to determine which neuron represents the prototype pattern closest to the input. In the competitive networks that are discussed in Chapters 16, 18 and 19, the prototype patterns are adjusted as new inputs are applied to the network. These adaptive networks learn to cluster the inputs into different categories.

Recurrent networks, like the Hopfield network, were originally inspired by statistical mechanics. They have been used as associative memories, in which stored data is recalled by association with input data, rather than by an address. They have also been used to solve a variety of optimization problems. We will discuss these recurrent networks in Chapters 20 and 21.

We hope this chapter has piqued your curiosity about the capabilities of neural networks and has raised some questions. A few of the questions we will answer in later chapters are:

1. How do we determine the weight matrix and bias for perceptron networks with many inputs, where it is impossible to visualize the decision boundary? (Chapters 4 and 10)

2. If the categories to be recognized are not linearly separable, can we extend the standard perceptron to solve the problem? (Chapters 11, 12 and 13)

3. Can we learn the weights and biases of the Hamming network when we don't know the prototype patterns? (Chapters 16, 18 and 19)

4. How do we determine the weight matrix and bias vector for the Hopfield network? (Chapter 21)

5. How do we know that the Hopfield network will eventually converge? (Chapters 20 and 21)

Exercises

E3.1 In this chapter we have designed three different neural networks to distinguish between apples and oranges, based on three sensor measurements (shape, texture and weight). Suppose that we want to distinguish between bananas and pineapples:

$$\mathbf{p}_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} \text{ (Banana)}$$

$$\mathbf{p}_2 = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix} \text{ (Pineapple)}$$

i. Design a perceptron to recognize these patterns.

ii. Design a Hamming network to recognize these patterns.

iii. Design a Hopfield network to recognize these patterns.

iv. Test the operation of your networks by applying several different input patterns. Discuss the advantages and disadvantages of each network.

E3.2 Consider the following prototype patterns.

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 0.5 \end{bmatrix}, \ \mathbf{p}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

i. Find and sketch a decision boundary for a perceptron network that will recognize these two vectors.

ii. Find weights and bias which will produce the decision boundary you found in part i, and sketch the network diagram.

iii. Calculate the network output for the following input. Is the network response (decision) reasonable? Explain.

$$\mathbf{p} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

iv. Design a Hamming network to recognize the two prototype vectors above.

v. Calculate the network output for the Hamming network with the input vector given in part iii, showing all steps. Does the Hamming network produce the same decision as the perceptron? Explain why or why not. Which network is better suited to this problem? Explain.

E3.3 Consider a Hopfield network, with the following weight and bias.

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

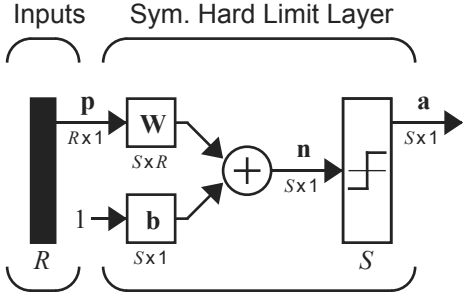
i. The following input (initial condition) is applied to the network. Find the network response (show the network output at each iteration until the network converges).

$$\mathbf{p} = \begin{bmatrix} 0.9 \\ 1 \end{bmatrix}$$

ii. Draw a sketch indicating what region of the input space will converge to the same final output that you found in part i. (In other words, for what other **p** vectors will the network converge to the same final output?) Explain how you obtained your answer.

iii. What other prototypes will this network converge to, and what regions of the input space correspond to each prototype (sketch the regions). Explain how you obtained your answer.

E3.4 Consider the following perceptron network.



$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -2 \\ 0 \end{bmatrix}$$

$$\mathbf{a} = \mathbf{hardlims}(\mathbf{W}\mathbf{p} + \mathbf{b})$$

i. How many different classes can this network classify?

ii. Draw a diagram illustrating the regions corresponding to each class. Label each region with the corresponding network output.

iii. Calculate the network output for the following input.

$$\mathbf{p} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

iv. Plot the input from part iii in your diagram from part ii, and verify that it falls in the correctly labeled region.

E3.5 We want to design a perceptron network to output a 1 when either of these two vectors are input to the network:

$$\left\{ \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right\},$$

and to output a -1 when either of the following vectors are input to the network:

$$\left\{ \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 2 \end{bmatrix} \right\}.$$

i. Find and sketch a decision boundary for a network that will solve this problem.

ii. Find weights and biases that will produce the decision boundary you found in part i. Show all work.

iii. Draw the network diagram using abreviated notation.

iv. For each of the four vectors given above, calculate the net input, **n**, and the network output, **a**, for the network you have designed. Verify that your network solves the problem.

v. Are there other weights and biases that would solve the problem? If so, would you consider your weights best? Explain.

E3.6 We have the folowing two prototype vectors:

$$\left\{ \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$$

i. Find and sketch a decision boundary for a perceptron network that will recognize these two vectors.

ii. Find weights and bias that will produce the decision boundary you found in part i.

iii. Draw the network diagram using abreviated notation.

iv. For the vector given below, calculate the net input, n, and the network output, a, for the network you have designed. Does the network produce a good output? Explain.

$$\begin{bmatrix} 0.5 \\ -0.5 \end{bmatrix}$$

v. Design a Hamming network to recognize the two vectors used in part i.

vi. Calculate the network output for the Hamming network for the input vector given in part iv. Does the network produce a good output? Explain.

vii. Design a Hopfield network to recognize the two vectors used in part i.

viii. Calculate the network output for the Hopfield network for the input vector given in part iv. Does the network produce a good output? Explain.

E3.7 We want to design a Hamming network to recognize the following prototype vectors:

$$\left\{ \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\}.$$

i. Find the weight matrices and bias vectors for the Hamming network.

ii. Draw the network diagram.

iii. Apply the following input vector and calculate the total network response (iterating the second layer to convergence). Explain the meaning of the final network output.

$$\mathbf{p} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

iv. Sketch the decision boundaries for this network. Explain how you determined the boundaries.

4 Perceptron Learning Rule

Objectives

One of the questions we raised in Chapter 3 was: "How do we determine the weight matrix and bias for perceptron networks with many inputs, where it is impossible to visualize the decision boundaries?" In this chapter we will describe an algorithm for *training* perceptron networks, so that they can *learn* to solve classification problems. We will begin by explaining what a learning rule is and will then develop the perceptron learning rule. We will conclude by discussing the advantages and limitations of the single-layer perceptron network. This discussion will lead us into future chapters.

Theory and Examples

In 1943, Warren McCulloch and Walter Pitts introduced one of the first artificial neurons [McPi43]. The main feature of their neuron model is that a weighted sum of input signals is compared to a threshold to determine the neuron output. When the sum is greater than or equal to the threshold, the output is 1. When the sum is less than the threshold, the output is 0. They went on to show that networks of these neurons could, in principle, compute any arithmetic or logical function. Unlike biological networks, the parameters of their networks had to be designed, as no training method was available. However, the perceived connection between biology and digital computers generated a great deal of interest.

In the late 1950s, Frank Rosenblatt and several other researchers developed a class of neural networks called perceptrons. The neurons in these networks were similar to those of McCulloch and Pitts. Rosenblatt's key contribution was the introduction of a learning rule for training perceptron networks to solve pattern recognition problems [Rose58]. He proved that his learning rule will always converge to the correct network weights, if weights exist that solve the problem. Learning was simple and automatic. Examples of proper behavior were presented to the network, which learned from its mistakes. The perceptron could even learn when initialized with random values for its weights and biases.

Unfortunately, the perceptron network is inherently limited. These limitations were widely publicized in the book *Perceptrons* [MiPa69] by Marvin Minsky and Seymour Papert. They demonstrated that the perceptron networks were incapable of implementing certain elementary functions. It was not until the 1980s that these limitations were overcome with improved (multilayer) perceptron networks and associated learning rules. We will discuss these improvements in Chapters 11 and 12.

Today the perceptron is still viewed as an important network. It remains a fast and reliable network for the class of problems that it can solve. In addition, an understanding of the operations of the perceptron provides a good basis for understanding more complex networks. Thus, the perceptron network, and its associated learning rule, are well worth discussing here.

In the remainder of this chapter we will define what we mean by a learning rule, explain the perceptron network and learning rule, and discuss the limitations of the perceptron network.

Learning Rules

Learning Rule

As we begin our presentation of the perceptron learning rule, we want to discuss learning rules in general. By *learning rule* we mean a procedure for modifying the weights and biases of a network. (This procedure may also

be referred to as a training algorithm.) The purpose of the learning rule is to train the network to perform some task. There are many types of neural network learning rules. They fall into three broad categories: supervised learning, unsupervised learning and reinforcement (or graded) learning.

Supervised Learning
Training Set

In *supervised learning*, the learning rule is provided with a set of examples (the *training set*) of proper network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}, \tag{4.1}$$

Target

where $\mathbf{p}_q$ is an input to the network and $\mathbf{t}_q$ is the corresponding correct (*target*) output. As the inputs are applied to the network, the network outputs are compared to the targets. The learning rule is then used to adjust the weights and biases of the network in order to move the network outputs closer to the targets. The perceptron learning rule falls in this supervised learning category. We will also investigate supervised learning algorithms in Chapters 7–14.

Reinforcement Learning

Reinforcement learning is similar to supervised learning, except that, instead of being provided with the correct output for each network input, the algorithm is only given a grade. The grade (or score) is a measure of the network performance over some sequence of inputs. This type of learning is currently much less common than supervised learning. It appears to be most suited to control system applications (see [BaSu83], [WhSo92]).

Unsupervised Learning

In *unsupervised learning*, the weights and biases are modified in response to network inputs only. There are no target outputs available. At first glance this might seem to be impractical. How can you train a network if you don't know what it is supposed to do? Most of these algorithms perform some kind of clustering operation. They learn to categorize the input patterns into a finite number of classes. This is especially useful in such applications as vector quantization. We will see in Chapters 15–19 that there are a number of unsupervised learning algorithms.

Perceptron Architecture

Before we present the perceptron learning rule, let's expand our investigation of the perceptron network, which we began in Chapter 3. The general perceptron network is shown in Figure 4.1.

The output of the network is given by

$$\mathbf{a} = \mathbf{hardlim}(\mathbf{W}\mathbf{p} + \mathbf{b}). \tag{4.2}$$

(Note that in Chapter 3 we used the *hardlims* transfer function, instead of *hardlim*. This does not affect the capabilities of the network. See Exercise E4.10.)

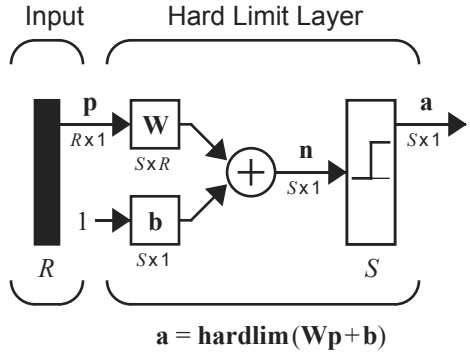


Figure 4.1 Perceptron Network

It will be useful in our development of the perceptron learning rule to be able to conveniently reference individual elements of the network output. Let's see how this can be done. First, consider the network weight matrix:

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}. \tag{4.3}$$

We will define a vector composed of the elements of the ith row of $\mathbf{W}$:

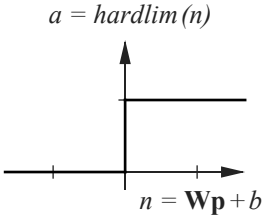
$${}_i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix}. \tag{4.4}$$

Now we can partition the weight matrix:

$$\mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ \vdots \\ {}_S\mathbf{w}^T \end{bmatrix}. \tag{4.5}$$

This allows us to write the ith element of the network output vector as

$$a_i = hardlim(n_i) = hardlim({}_i\mathbf{w}^T\mathbf{p} + b_i). \tag{4.6}$$



Recall that the *hardlim* transfer function (shown at left) is defined as:

$$a = hardlim(n) = \begin{cases} 1 & if \ n \geq 0 \\ 0 & otherwise. \end{cases} \tag{4.7}$$

Therefore, if the inner product of the ith row of the weight matrix with the input vector is greater than or equal to $-b_i$, the output will be 1, otherwise the output will be 0. *Thus each neuron in the network divides the input space into two regions*. It is useful to investigate the boundaries between these regions. We will begin with the simple case of a single-neuron perceptron with two inputs.

Single-Neuron Perceptron

Let's consider a two-input perceptron with one neuron, as shown in Figure 4.2.

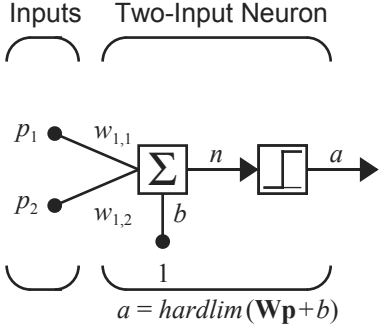


Figure 4.2 Two-Input/Single-Output Perceptron

The output of this network is determined by

$$\begin{aligned} a &= hardlim(n) = hardlim(\mathbf{Wp} + b) \\ &= hardlim({}_1\mathbf{w}^T\mathbf{p} + b) = hardlim(w_{1,1}p_1 + w_{1,2}p_2 + b) \end{aligned} \tag{4.8}$$

Decision Boundary

The *decision boundary* is determined by the input vectors for which the net input n is zero:

$$n = {}_1\mathbf{w}^T\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b = 0. \tag{4.9}$$

To make the example more concrete, let's assign the following values for the weights and bias:

$$w_{1,1} = 1, \ w_{1,2} = 1, \ b = -1. \tag{4.10}$$

The decision boundary is then

$$n = {}_1\mathbf{w}^T\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b = p_1 + p_2 - 1 = 0 . \quad (4.11)$$

This defines a line in the input space. On one side of the line the network output will be 0; on the line and on the other side of the line the output will be 1. To draw the line, we can find the points where it intersects the p_1 and p_2 axes. To find the p_2 intercept set $p_1 = 0$:

$$p_2 = -\frac{b}{w_{1,2}} = -\frac{-1}{1} = 1 \quad \text{if } p_1 = 0 . \quad (4.12)$$

To find the p_1 intercept, set $p_2 = 0$:

$$p_1 = -\frac{b}{w_{1,1}} = -\frac{-1}{1} = 1 \quad \text{if } p_2 = 0 . \quad (4.13)$$

The resulting decision boundary is illustrated in Figure 4.3.

To find out which side of the boundary corresponds to an output of 1, we just need to test one point. For the input $\mathbf{p} = \begin{bmatrix} 2 & 0 \end{bmatrix}^T$, the network output will be

$$a = hardlim({}_1\mathbf{w}^T\mathbf{p} + b) = hardlim\left(\begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \end{bmatrix} - 1 \right) = 1 . \quad (4.14)$$

Therefore, the network output will be 1 for the region above and to the right of the decision boundary. This region is indicated by the shaded area in Figure 4.3.

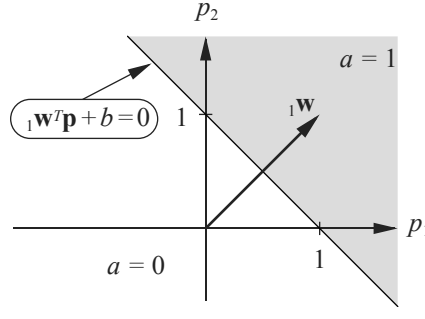
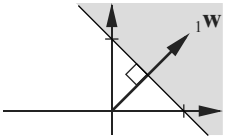
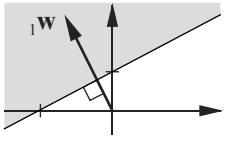


Figure 4.3 Decision Boundary for Two-Input Perceptron



We can also find the decision boundary graphically. The first step is to note that the boundary is always orthogonal to ${}_1\mathbf{w}$, as illustrated in the adjacent figures. The boundary is defined by

$${}_1\mathbf{w}^T\mathbf{p} + b = 0. \tag{4.15}$$

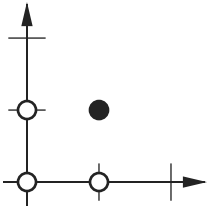


For all points on the boundary, the inner product of the input vector with the weight vector is the same. This implies that these input vectors will all have the same projection onto the weight vector, so they must lie on a line orthogonal to the weight vector. (These concepts will be covered in more detail in Chapter 5.) In addition, any vector in the shaded region of Figure 4.3 will have an inner product greater than $-b$, and vectors in the unshaded region will have inner products less than $-b$. Therefore the weight vector ${}_1\mathbf{w}$ will always point toward the region where the neuron output is 1.

After we have selected a weight vector with the correct angular orientation, the bias value can be computed by selecting a point on the boundary and satisfying Eq. (4.15).

Let's apply some of these concepts to the design of a perceptron network to implement a simple logic function: the AND gate. The input/target pairs for the AND gate are

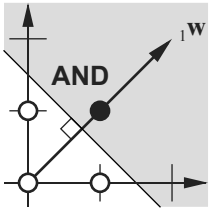
$$\left\{\mathbf{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 0\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 0\right\}\left\{\mathbf{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 1\right\}.$$



The figure to the left illustrates the problem graphically. It displays the input space, with each input vector labeled according to its target. The dark circles ● indicate that the target is 1, and the light circles ○ indicate that the target is 0.

The first step of the design is to select a decision boundary. We want to have a line that separates the dark circles and the light circles. There are an infinite number of solutions to this problem. It seems reasonable to choose the line that falls "halfway" between the two categories of inputs, as shown in the adjacent figure.

Next we want to choose a weight vector that is orthogonal to the decision boundary. The weight vector can be any length, so there are infinite possibilities. One choice is



$${}_1\mathbf{w} = \begin{bmatrix}2\\2\end{bmatrix}, \tag{4.16}$$

as displayed in the figure to the left.

Finally, we need to find the bias, b. We can do this by picking a point on the decision boundary and satisfying Eq. (4.15). If we use $\mathbf{p} = \begin{bmatrix} 1.5 & 0 \end{bmatrix}^T$ we find

$$ {}_1\mathbf{w}^T\mathbf{p} + b = \begin{bmatrix} 2 & 2 \end{bmatrix}\begin{bmatrix} 1.5 \\ 0 \end{bmatrix} + b = 3 + b = 0 \quad \Rightarrow \quad b = -3 . \tag{4.17} $$

We can now test the network on one of the input/target pairs. If we apply $\mathbf{p}_2$ to the network, the output will be

$$ \begin{aligned} a &= hardlim({}_1\mathbf{w}^T\mathbf{p}_2 + b) = hardlim\left(\begin{bmatrix} 2 & 2 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix} - 3 \right) \\ a &= hardlim(-1) = 0, \end{aligned} \tag{4.18} $$

which is equal to the target output t_2. Verify for yourself that all inputs are correctly classified.

*To experiment with decision boundaries, use the Neural Network Design Demonstration Decision Boundaries (*nnd4db*).*

Multiple-Neuron Perceptron

Note that for perceptrons with multiple neurons, as in Figure 4.1, there will be one decision boundary for each neuron. The decision boundary for neuron i will be defined by

$$ {}_i\mathbf{w}^T\mathbf{p} + b_i = 0 . \tag{4.19} $$

A single-neuron perceptron can classify input vectors into two categories, since its output can be either 0 or 1. A multiple-neuron perceptron can classify inputs into many categories. Each category is represented by a different output vector. Since each element of the output vector can be either 0 or 1, there are a total of 2^S possible categories, where S is the number of neurons.

Perceptron Learning Rule

Now that we have examined the performance of perceptron networks, we are in a position to introduce the perceptron learning rule. This learning rule is an example of supervised training, in which the learning rule is provided with a set of examples of proper network behavior:

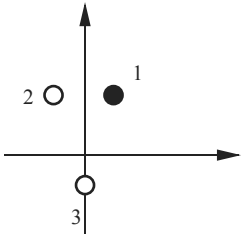
$$ \{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}, \tag{4.20} $$

where $\mathbf{p}_q$ is an input to the network and $\mathbf{t}_q$ is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The learning rule then adjusts the weights and biases of the network in order to move the network output closer to the target.

Test Problem

In our presentation of the perceptron learning rule we will begin with a simple test problem and will experiment with possible rules to develop some intuition about how the rule should work. The input/target pairs for our test problem are

$$\left\{\mathbf{p}_1 = \begin{bmatrix}1\\2\end{bmatrix}, t_1 = 1\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}-1\\2\end{bmatrix}, t_2 = 0\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}0\\-1\end{bmatrix}, t_3 = 0\right\}.$$



The problem is displayed graphically in the adjacent figure, where the two input vectors whose target is 0 are represented with a light circle ○, and the vector whose target is 1 is represented with a dark circle ●. This is a very simple problem, and we could almost obtain a solution by inspection. This simplicity will help us gain some intuitive understanding of the basic concepts of the perceptron learning rule.

The network for this problem should have two-inputs and one output. To simplify our development of the learning rule, we will begin with a network without a bias. The network will then have just two parameters, $w_{1,1}$ and $w_{1,2}$, as shown in Figure 4.4.

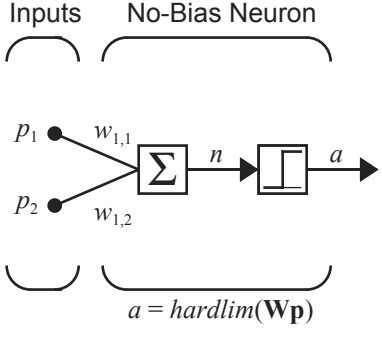
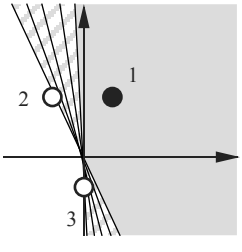
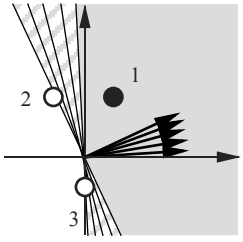


Figure 4.4 Test Problem Network



By removing the bias we are left with a network whose decision boundary must pass through the origin. We need to be sure that this network is still able to solve the test problem. There must be an allowable decision boundary that can separate the vectors $\mathbf{p}_2$ and $\mathbf{p}_3$ from the vector $\mathbf{p}_1$. The figure to the left illustrates that there are indeed an infinite number of such boundaries.



The adjacent figure shows the weight vectors that correspond to the allowable decision boundaries. (Recall that the weight vector is orthogonal to the decision boundary.) We would like a learning rule that will find a weight vector that points in one of these directions. Remember that the length of the weight vector does not matter; only its direction is important.

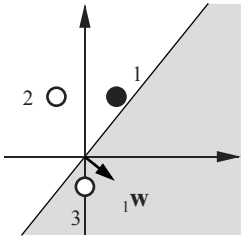
Constructing Learning Rules

Training begins by assigning some initial values for the network parameters. In this case we are training a two-input/single-output network without a bias, so we only have to initialize its two weights. Here we set the elements of the weight vector, ${}_1\mathbf{w}$, to the following randomly generated values:

$${}_1\mathbf{w}^T = \begin{bmatrix} 1.0 & -0.8 \end{bmatrix}. \tag{4.21}$$

We will now begin presenting the input vectors to the network. We begin with $\mathbf{p}_1$:

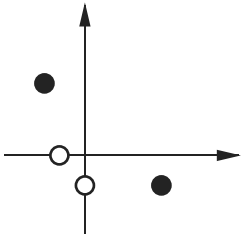
$$\begin{aligned} a &= hardlim({}_1\mathbf{w}^T\mathbf{p}_1) = hardlim\left(\begin{bmatrix} 1.0 & -0.8 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right) \\ a &= hardlim(-0.6) = 0 . \end{aligned} \tag{4.22}$$



The network has not returned the correct value. The network output is 0, while the target response, t_1, is 1.

We can see what happened by looking at the adjacent diagram. The initial weight vector results in a decision boundary that incorrectly classifies the vector $\mathbf{p}_1$. We need to alter the weight vector so that it points more toward $\mathbf{p}_1$, so that in the future it has a better chance of classifying it correctly.

One approach would be to set ${}_1\mathbf{w}$ equal to $\mathbf{p}_1$. This is simple and would ensure that $\mathbf{p}_1$ was classified properly in the future. Unfortunately, it is easy to construct a problem for which this rule cannot find a solution. The diagram to the lower left shows a problem that cannot be solved with the weight vector pointing directly at either of the two class 1 vectors. If we apply the rule ${}_1\mathbf{w} = \mathbf{p}$ every time one of these vectors is misclassified, the network's weights will simply oscillate back and forth and will never find a solution.

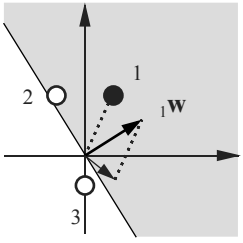


Another possibility would be to add $\mathbf{p}_1$ to ${}_1\mathbf{w}$. Adding $\mathbf{p}_1$ to ${}_1\mathbf{w}$ would make ${}_1\mathbf{w}$ point more in the direction of $\mathbf{p}_1$. Repeated presentations of $\mathbf{p}_1$ would cause the direction of ${}_1\mathbf{w}$ to asymptotically approach the direction of $\mathbf{p}_1$. This rule can be stated:

$$\text{If } t = 1 \text{ and } a = 0, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}. \tag{4.23}$$

Applying this rule to our test problem results in new values for ${}_1\mathbf{w}$:

$$ {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}_1 = \begin{bmatrix} 1.0 \\ -0.8 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 1.2 \end{bmatrix}. \tag{4.24} $$



This operation is illustrated in the adjacent figure.

We now move on to the next input vector and will continue making changes to the weights and cycling through the inputs until they are all classified correctly.

The next input vector is $\mathbf{p}_2$. When it is presented to the network we find:

$$ \begin{aligned} a &= hardlim({}_1\mathbf{w}^T\mathbf{p}_2) = hardlim\left(\begin{bmatrix} 2.0 & 1.2 \end{bmatrix} \begin{bmatrix} -1 \\ 2 \end{bmatrix} \right) \\ &= hardlim(0.4) = 1 . \end{aligned} \tag{4.25} $$

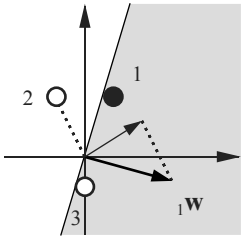
The target t_2 associated with $\mathbf{p}_2$ is 0 and the output a is 1. A class 0 vector was misclassified as a 1.

Since we would now like to move the weight vector ${}_1\mathbf{w}$ away from the input, we can simply change the addition in Eq. (4.23) to subtraction:

$$ \text{If } t = 0 \text{ and } a = 1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}. \tag{4.26} $$

If we apply this to the test problem we find:

$$ {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}_2 = \begin{bmatrix} 2.0 \\ 1.2 \end{bmatrix} - \begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix}, \tag{4.27} $$



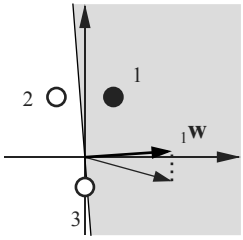
which is illustrated in the adjacent figure.

Now we present the third vector $\mathbf{p}_3$:

$$ \begin{aligned} a &= hardlim({}_1\mathbf{w}^T\mathbf{p}_3) = hardlim\left(\begin{bmatrix} 3.0 & -0.8 \end{bmatrix} \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right) \\ &= hardlim(0.8) = 1 . \end{aligned} \tag{4.28} $$

The current ${}_1\mathbf{w}$ results in a decision boundary that misclassifies $\mathbf{p}_3$. This is a situation for which we already have a rule, so ${}_1\mathbf{w}$ will be updated again, according to Eq. (4.26):

$$ {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}_3 = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix} - \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 3.0 \\ 0.2 \end{bmatrix}. \tag{4.29} $$



The diagram to the left shows that the perceptron has finally learned to classify the three vectors properly. If we present any of the input vectors to the neuron, it will output the correct class for that input vector.

This brings us to our third and final rule: if it works, don't fix it.

$$\text{If } t = a, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}. \tag{4.30}$$

Here are the three rules, which cover all possible combinations of output and target values:

$$\begin{aligned} &\text{If } t = 1 \text{ and } a = 0, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}. \\ &\text{If } t = 0 \text{ and } a = 1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}. \\ &\text{If } t = a, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}. \end{aligned} \tag{4.31}$$

Unified Learning Rule

The three rules in Eq. (4.31) can be rewritten as a single expression. First we will define a new variable, the perceptron error e:

$$e = t - a. \tag{4.32}$$

We can now rewrite the three rules of Eq. (4.31) as:

$$\begin{aligned} &\text{If } e = 1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}. \\ &\text{If } e = -1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}. \\ &\text{If } e = 0, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}. \end{aligned} \tag{4.33}$$

Looking carefully at the first two rules in Eq. (4.33) we can see that the sign of $\mathbf{p}$ is the same as the sign on the error, e. Furthermore, the absence of $\mathbf{p}$ in the third rule corresponds to an e of 0. Thus, we can unify the three rules into a single expression:

$${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + e\mathbf{p} = {}_1\mathbf{w}^{old} + (t - a)\mathbf{p}. \tag{4.34}$$

This rule can be extended to train the bias by noting that a bias is simply a weight whose input is always 1. We can thus replace the input $\mathbf{p}$ in Eq. (4.34) with the input to the bias, which is 1. The result is the perceptron rule for a bias:

$$b^{new} = b^{old} + e. \tag{4.35}$$

Training Multiple-Neuron Perceptrons

The perceptron rule, as given by Eq. (4.34) and Eq. (4.35), updates the weight vector of a single neuron perceptron. We can generalize this rule for the multiple-neuron perceptron of Figure 4.1 as follows. To update the *i*th row of the weight matrix use:

$$ {}_i\mathbf{w}^{new} = {}_i\mathbf{w}^{old} + e_i\mathbf{p}\,. \tag{4.36} $$

To update the *i*th element of the bias vector use:

$$ b_i^{new} = b_i^{old} + e_i\,. \tag{4.37} $$

Perceptron Rule

The *perceptron rule* can be written conveniently in matrix notation:

$$ \mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T, \tag{4.38} $$

and

$$ \mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}\,. \tag{4.39} $$

To test the perceptron learning rule, consider again the apple/orange recognition problem of Chapter 3. The input/output prototype vectors will be

$$ \left\{\mathbf{p}_1 = \begin{bmatrix}1\\-1\\-1\end{bmatrix}, t_1 = \begin{bmatrix}0\end{bmatrix}\right\} \qquad \left\{\mathbf{p}_2 = \begin{bmatrix}1\\1\\-1\end{bmatrix}, t_2 = \begin{bmatrix}1\end{bmatrix}\right\}. \tag{4.40} $$

(Note that we are using 0 as the target output for the orange pattern, $\mathbf{p}_1$, instead of -1, as was used in Chapter 3. This is because we are using the *hardlim* transfer function, instead of *hardlims*.)

Typically the weights and biases are initialized to small random numbers. Suppose that here we start with the initial weight matrix and bias:

$$ \mathbf{W} = \begin{bmatrix}0.5 & -1 & -0.5\end{bmatrix}, b = 0.5\,. \tag{4.41} $$

The first step is to apply the first input vector, $\mathbf{p}_1$, to the network:

$$ \begin{aligned} a &= hardlim(\mathbf{W}\mathbf{p}_1 + b) = hardlim\left(\begin{bmatrix}0.5 & -1 & -0.5\end{bmatrix}\begin{bmatrix}1\\-1\\-1\end{bmatrix} + 0.5\right) \\ &= hardlim(2.5) = 1 \end{aligned} \tag{4.42} $$

Then we calculate the error:

$$e = t_1 - a = 0 - 1 = -1 . \tag{4.43}$$

The weight update is

$$\begin{aligned} \mathbf{W}^{new} &= \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} + (-1)\begin{bmatrix} 1 & -1 & -1 \end{bmatrix} \\ &= \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix} . \end{aligned} \tag{4.44}$$

The bias update is

$$b^{new} = b^{old} + e = 0.5 + (-1) = -0.5 . \tag{4.45}$$

This completes the first iteration.

The second iteration of the perceptron rule is:

$$a = hardlim\,(\mathbf{W}\mathbf{p}_2 + b) = hardlim\,(\begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + (-0.5))$$

$$= hardlim\,(-0.5) = 0 \tag{4.46}$$

$$e = t_2 - a = 1 - 0 = 1 \tag{4.47}$$

$$\mathbf{W}^{new} = \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix} + 1\begin{bmatrix} 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 0.5 & 1 & -0.5 \end{bmatrix} \tag{4.48}$$

$$b^{new} = b^{old} + e = -0.5 + 1 = 0.5 \tag{4.49}$$

The third iteration begins again with the first input vector:

$$a = hardlim\,(\mathbf{W}\mathbf{p}_1 + b) = hardlim\,(\begin{bmatrix} 0.5 & 1 & -0.5 \end{bmatrix}\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 0.5)$$

$$= hardlim\,(0.5) = 1 \tag{4.50}$$

$$e = t_1 - a = 0 - 1 = -1 \tag{4.51}$$

$$\begin{aligned} \mathbf{W}^{new} &= \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} 0.5 & 1 & -0.5 \end{bmatrix} + (-1)\begin{bmatrix} 1 & -1 & -1 \end{bmatrix} \\ &= \begin{bmatrix} -0.5 & 2 & 0.5 \end{bmatrix} \end{aligned} \tag{4.52}$$

$$b^{new} = b^{old} + e = 0.5 + (-1) = -0.5 . \tag{4.53}$$

If you continue with the iterations you will find that both input vectors will now be correctly classified. The algorithm has converged to a solution. Note that the final decision boundary is not the same as the one we developed in Chapter 3, although both boundaries correctly classify the two input vectors.

*To experiment with the perceptron learning rule, use the Neural Network Design Demonstration Perceptron Rule (*nnd4pr*).*

Proof of Convergence

Although the perceptron learning rule is simple, it is quite powerful. In fact, it can be shown that the rule will always converge to weights that accomplish the desired classification (assuming that such weights exist). In this section we will present a proof of convergence for the perceptron learning rule for the single-neuron perceptron shown in Figure 4.5.

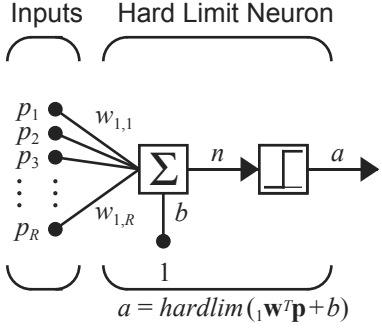


Figure 4.5 Single-Neuron Perceptron

The output of this perceptron is obtained from

$$a = hardlim(_1\mathbf{w}^T\mathbf{p} + b) . \tag{4.54}$$

The network is provided with the following examples of proper network behavior:

$$\{\mathbf{p}_1, t_1\}, \{\mathbf{p}_2, t_2\}, \ldots, \{\mathbf{p}_Q, t_Q\} . \tag{4.55}$$

where each target output, t_q, is either 0 or 1.

Notation

To conveniently present the proof we will first introduce some new notation. We will combine the weight matrix and the bias into a single vector:

$$\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w} \\ b \end{bmatrix}. \tag{4.56}$$

We will also augment the input vectors with a 1, corresponding to the bias input:

$$\mathbf{z}_q = \begin{bmatrix} \mathbf{p}_q \\ 1 \end{bmatrix}. \tag{4.57}$$

Now we can express the net input to the neuron as follows:

$$n = {}_1\mathbf{w}^T\mathbf{p} + b = \mathbf{x}^T\mathbf{z}. \tag{4.58}$$

The perceptron learning rule for a single-neuron perceptron (Eq. (4.34) and Eq. (4.35)) can now be written

$$\mathbf{x}^{new} = \mathbf{x}^{old} + e\mathbf{z}. \tag{4.59}$$

The error e can be either 1, -1 or 0. If $e = 0$, then no change is made to the weights. If $e = 1$, then the input vector is added to the weight vector. If $e = -1$, then the negative of the input vector is added to the weight vector. If we count only those iterations for which the weight vector is changed, the learning rule becomes

$$\mathbf{x}(k) = \mathbf{x}(k-1) + \mathbf{z}'(k-1), \tag{4.60}$$

where $\mathbf{z}'(k-1)$ is the appropriate member of the set

$$\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_Q, -\mathbf{z}_1, -\mathbf{z}_2, \ldots, -\mathbf{z}_Q\}. \tag{4.61}$$

We will assume that a weight vector exists that can correctly categorize all Q input vectors. This solution will be denoted $\mathbf{x}^*$. For this weight vector we will assume that

$$\mathbf{x}^{*T}\mathbf{z}_q > \delta > 0 \text{ if } t_q = 1, \tag{4.62}$$

and

$$\mathbf{x}^{*T}\mathbf{z}_q < -\delta < 0 \text{ if } t_q = 0. \tag{4.63}$$

Proof

We are now ready to begin the proof of the perceptron convergence theorem. The objective of the proof is to find upper and lower bounds on the length of the weight vector at each stage of the algorithm.

Assume that the algorithm is initialized with the zero weight vector: $\mathbf{x}(0) = \mathbf{0}$. (This does not affect the generality of our argument.) Then, after k iterations (changes to the weight vector), we find from Eq. (4.60):

$$\mathbf{x}(k) = \mathbf{z}'(0) + \mathbf{z}'(1) + \cdots + \mathbf{z}'(k-1). \tag{4.64}$$

If we take the inner product of the solution weight vector with the weight vector at iteration k we obtain

$$\mathbf{x}^{*T}\mathbf{x}(k) = \mathbf{x}^{*T}\mathbf{z}'(0) + \mathbf{x}^{*T}\mathbf{z}'(1) + \cdots + \mathbf{x}^{*T}\mathbf{z}'(k-1). \tag{4.65}$$

From Eq. (4.61)–Eq. (4.63) we can show that

$$\mathbf{x}^{*T}\mathbf{z}'(i) > \delta. \tag{4.66}$$

Therefore

$$\mathbf{x}^{*T}\mathbf{x}(k) > k\delta. \tag{4.67}$$

From the Cauchy-Schwartz inequality (see [Brog91])

$$(\mathbf{x}^{*T}\mathbf{x}(k))^2 \le \|\mathbf{x}^*\|^2 \|\mathbf{x}(k)\|^2, \tag{4.68}$$

where

$$\|\mathbf{x}\|^2 = \mathbf{x}^T\mathbf{x}. \tag{4.69}$$

If we combine Eq. (4.67) and Eq. (4.68) we can put a lower bound on the squared length of the weight vector at iteration k:

$$\|\mathbf{x}(k)\|^2 \ge \frac{(\mathbf{x}^{*T}\mathbf{x}(k))^2}{\|\mathbf{x}^*\|^2} > \frac{(k\delta)^2}{\|\mathbf{x}^*\|^2}. \tag{4.70}$$

Next we want to find an upper bound for the length of the weight vector. We begin by finding the change in the length at iteration k:

$$\begin{aligned} \|\mathbf{x}(k)\|^2 &= \mathbf{x}^T(k)\mathbf{x}(k) \\ &= [\mathbf{x}(k-1) + \mathbf{z}'(k-1)]^T[\mathbf{x}(k-1) + \mathbf{z}'(k-1)] \\ &= \mathbf{x}^T(k-1)\mathbf{x}(k-1) + 2\mathbf{x}^T(k-1)\mathbf{z}'(k-1) \\ &\quad + \mathbf{z}'^T(k-1)\mathbf{z}'(k-1) \end{aligned} \tag{4.71}$$

Note that

$$\mathbf{x}^T(k-1)\mathbf{z}'(k-1) \le 0, \tag{4.72}$$

since the weights would not be updated unless the previous input vector had been misclassified. Now Eq. (4.71) can be simplified to

$$\|\mathbf{x}(k)\|^2 \leq \|\mathbf{x}(k-1)\|^2 + \|\mathbf{z}'(k-1)\|^2 . \tag{4.73}$$

We can repeat this process for $\|\mathbf{x}(k-1)\|^2$, $\|\mathbf{x}(k-2)\|^2$, etc., to obtain

$$\|\mathbf{x}(k)\|^2 \leq \|\mathbf{z}'(0)\|^2 + \cdots + \|\mathbf{z}'(k-1)\|^2 . \tag{4.74}$$

If $\Pi = max\{\|\mathbf{z}'(i)\|^2\}$, this upper bound can be simplified to

$$\|\mathbf{x}(k)\|^2 \leq k\Pi . \tag{4.75}$$

We now have an upper bound (Eq. (4.75)) and a lower bound (Eq. (4.70)) on the squared length of the weight vector at iteration k. If we combine the two inequalities we find

$$k\Pi \geq \|\mathbf{x}(k)\|^2 > \frac{(k\delta)^2}{\|\mathbf{x}^*\|^2} \text{ or } k < \frac{\Pi\|\mathbf{x}^*\|^2}{\delta^2} . \tag{4.76}$$

Because k has an upper bound, this means that the weights will only be changed a finite number of times. Therefore, the perceptron learning rule will converge in a finite number of iterations.

The maximum number of iterations (changes to the weight vector) is inversely related to the square of δ. This parameter is a measure of how close the solution decision boundary is to the input patterns. This means that if the input classes are difficult to separate (are close to the decision boundary) it will take many iterations for the algorithm to converge.

Note that there are only three key assumptions required for the proof:

1. A solution to the problem exists, so that Eq. (4.66) is satisfied.
2. The weights are only updated when the input vector is misclassified, therefore Eq. (4.72) is satisfied.
3. An upper bound, Π, exists for the length of the input vectors.

Because of the generality of the proof, there are many variations of the perceptron learning rule that can also be shown to converge. (See Exercise E4.13.)

Limitations

The perceptron learning rule is guaranteed to converge to a solution in a finite number of steps, so long as a solution exists. This brings us to an important question. What problems can a perceptron solve? Recall that a sin-

gle-neuron perceptron is able to divide the input space into two regions. The boundary between the regions is defined by the equation

$$_1\mathbf{w}^T\mathbf{p} + b = 0 . \tag{4.77}$$

Linear Separability

This is a linear boundary (hyperplane). The perceptron can be used to classify input vectors that can be separated by a linear boundary. We call such vectors *linearly separable*. The logical AND gate example on page 4-7 illustrates a two-dimensional example of a linearly separable problem. The apple/orange recognition problem of Chapter 3 was a three-dimensional example.

Unfortunately, many problems are not linearly separable. The classic example is the XOR gate. The input/target pairs for the XOR gate are

$$\left\{\mathbf{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 1\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 1\right\}\left\{\mathbf{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 0\right\}.$$

This problem is illustrated graphically on the left side of Figure 4.6, which also shows two other linearly inseparable problems. Try drawing a straight line between the vectors with targets of 1 and those with targets of 0 in any of the diagrams of Figure 4.6.

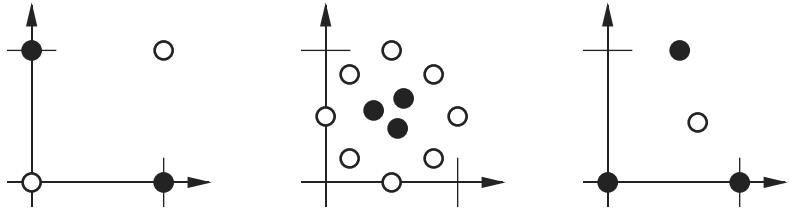
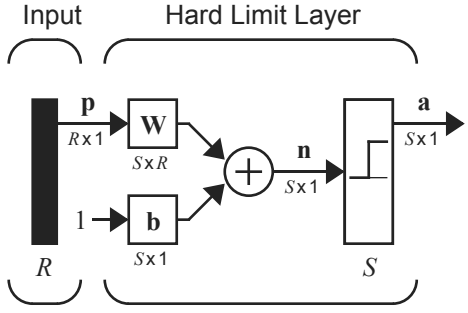


Figure 4.6 Linearly Inseparable Problems

It was the inability of the basic perceptron to solve such simple problems that led, in part, to a reduction in interest in neural network research during the 1970s. Rosenblatt had investigated more complex networks, which he felt would overcome the limitations of the basic perceptron, but he was never able to effectively extend the perceptron rule to such networks. In Chapter 11 we will introduce multilayer perceptrons, which can solve arbitrary classification problems, and will describe the backpropagation algorithm, which can be used to train them.

Summary of Results

Perceptron Architecture



$$\mathbf{a} = \mathbf{hardlim}(\mathbf{Wp}+\mathbf{b})$$

$$\mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ \vdots \\ {}_S\mathbf{w}^T \end{bmatrix}$$

$$\mathbf{a} = \mathbf{hardlim}(\mathbf{Wp} + \mathbf{b})$$

$$a_i = hardlim(n_i) = hardlim({}_i\mathbf{w}^T\mathbf{p} + b_i)$$

Decision Boundary

$${}_i\mathbf{w}^T\mathbf{p} + b_i = 0 .$$

The decision boundary is always orthogonal to the weight vector.

Single-layer perceptrons can only classify linearly separable vectors.

Perceptron Learning Rule

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$$

where $\mathbf{e} = \mathbf{t} - \mathbf{a}$.

Solved Problems

P4.1 Solve the three simple classification problems shown in Figure P4.1 by drawing a decision boundary. Find weight and bias values that result in single-neuron perceptrons with the chosen decision boundaries.

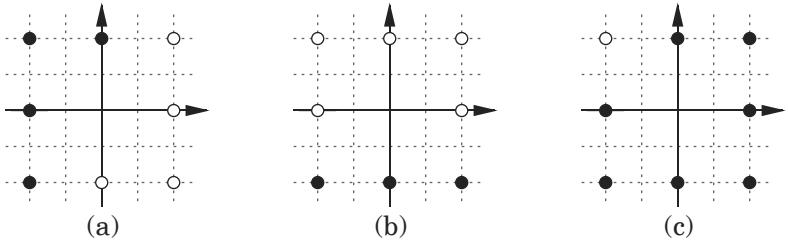
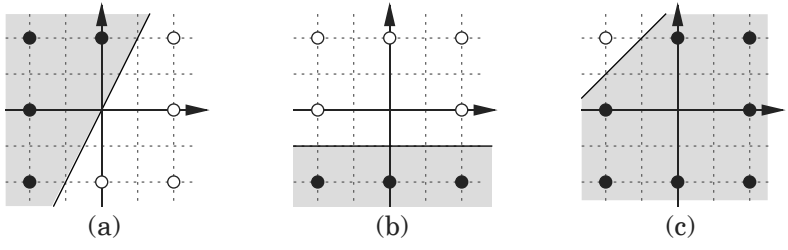
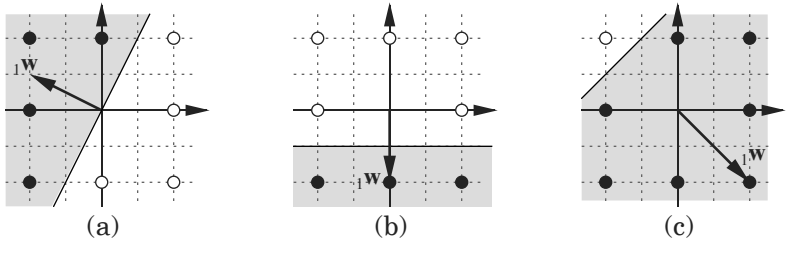


Figure P4.1 Simple Classification Problems

First we draw a line between each set of dark and light data points.



The next step is to find the weights and biases. The weight vectors must be orthogonal to the decision boundaries, and pointing in the direction of points to be classified as 1 (the dark points). The weight vectors can have any length we like.



Here is one set of choices for the weight vectors:

$$\text{(a) } {}_1\mathbf{w}^T = \begin{bmatrix} -2 & 1 \end{bmatrix}, \quad \text{(b) } {}_1\mathbf{w}^T = \begin{bmatrix} 0 & -2 \end{bmatrix}, \quad \text{(c) } {}_1\mathbf{w}^T = \begin{bmatrix} 2 & -2 \end{bmatrix}.$$

Now we find the bias values for each perceptron by picking a point on the decision boundary and satisfying Eq. (4.15).

$$
\begin{aligned}
{}_1\mathbf{w}^T\mathbf{p} + b &= 0 \\
b &= -{}_1\mathbf{w}^T\mathbf{p}
\end{aligned}
$$

This gives us the following three biases:

$$(a)\ b = -\begin{bmatrix}-2 & 1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} = 0,\ (b)\ b = -\begin{bmatrix}0 & -2\end{bmatrix}\begin{bmatrix}0\\-1\end{bmatrix} = -2,\ (c)\ b = -\begin{bmatrix}2 & -2\end{bmatrix}\begin{bmatrix}-2\\1\end{bmatrix} = 6$$

We can now check our solution against the original points. Here we test the first network on the input vector $\mathbf{p} = \begin{bmatrix}-2 & 2\end{bmatrix}^T$.

$$
\begin{aligned}
a &= hardlim({}_1\mathbf{w}^T\mathbf{p} + b) \\
&= hardlim\left(\begin{bmatrix}-2 & 1\end{bmatrix}\begin{bmatrix}-2\\2\end{bmatrix} + 0\right) \\
&= hardlim(6) \\
&= 1
\end{aligned}
$$

We can use MATLAB to automate the testing process and to try new points. Here the first network is used to classify a point that was not in the original problem.

```
w=[-2 1]; b = 0;
a = hardlim(w*[1;1]+b)
a =
       0
```

P4.2 Convert the classification problem defined below into an equivalent problem definition consisting of inequalities constraining weight and bias values.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}0\\2\end{bmatrix}, t_1 = 1\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}1\\0\end{bmatrix}, t_2 = 1\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}0\\-2\end{bmatrix}, t_3 = 0\right\}\left\{\mathbf{p}_4 = \begin{bmatrix}2\\0\end{bmatrix}, t_4 = 0\right\}$$

Each target t_i indicates whether or not the net input in response to $\mathbf{p}_i$ must be less than 0, or greater than or equal to 0. For example, since t_1 is 1, we know that the net input corresponding to $\mathbf{p}_1$ must be greater than or equal to 0. Thus we get the following inequality:

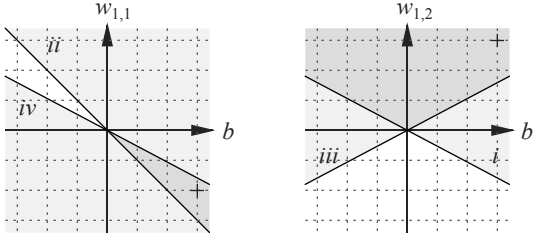
$$\begin{aligned} \mathbf{W}\mathbf{p}_1 + b &\geq 0 \\ 0w_{1,1} + 2w_{1,2} + b &\geq 0 \\ 2w_{1,2} + b &\geq 0. \end{aligned}$$

Applying the same procedure to the input/target pairs for $\{\mathbf{p}_2, t_2\}$, $\{\mathbf{p}_3, t_3\}$ and $\{\mathbf{p}_4, t_4\}$ results in the following set of inequalities.

$$\begin{aligned} 2w_{1,2} + b &\geq 0 \quad (i) \\ w_{1,1} + b &\geq 0 \quad (ii) \\ -2w_{1,2} + b &< 0 \quad (iii) \\ 2w_{1,1} + b &< 0 \quad (iv) \end{aligned}$$

Solving a set of inequalities is more difficult than solving a set of equalities. One added complexity is that there are often an infinite number of solutions (just as there are often an infinite number of linear decision boundaries that can solve a linearly separable classification problem).

However, because of the simplicity of this problem, we can solve it by graphing the solution spaces defined by the inequalities. Note that $w_{1,1}$ only appears in inequalities (*ii*) and (*iv*), and $w_{1,2}$ only appears in inequalities (*i*) and (*iii*). We can plot each pair of inequalities with two graphs.



Any weight and bias values that fall in both dark gray regions will solve the classification problem.

Here is one such solution:

$$\mathbf{W} = \begin{bmatrix} -2 & 3 \end{bmatrix} \qquad b = 3.$$

P4.3 We have a classification problem with four classes of input vector. The four classes are

$$\text{class 1:}\left\{\mathbf{p}_1 = \begin{bmatrix}1\\1\end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix}1\\2\end{bmatrix}\right\}, \text{class 2: } \left\{\mathbf{p}_3 = \begin{bmatrix}2\\-1\end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix}2\\0\end{bmatrix}\right\},$$

$$\text{class 3:}\left\{\mathbf{p}_5 = \begin{bmatrix}-1\\2\end{bmatrix}, \mathbf{p}_6 = \begin{bmatrix}-2\\1\end{bmatrix}\right\}, \text{class 4: } \left\{\mathbf{p}_7 = \begin{bmatrix}-1\\-1\end{bmatrix}, \mathbf{p}_8 = \begin{bmatrix}-2\\-2\end{bmatrix}\right\}.$$

Design a perceptron network to solve this problem.

To solve a problem with four classes of input vector we will need a perceptron with at least two neurons, since an S-neuron perceptron can categorize 2^S classes. The two-neuron perceptron is shown in Figure P4.2.

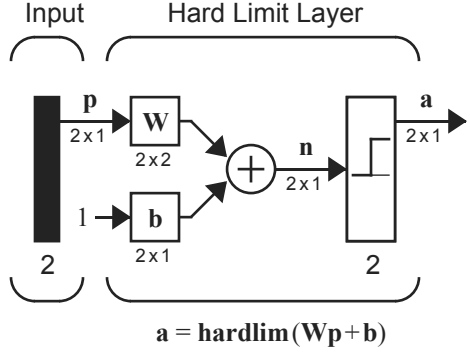


Figure P4.2 Two-Neuron Perceptron

Let's begin by displaying the input vectors, as in Figure P4.3. The light circles ○ indicate class 1 vectors, the light squares □ indicate class 2 vectors, the dark circles ● indicate class 3 vectors, and the dark squares ■ indicate class 4 vectors.

A two-neuron perceptron creates two decision boundaries. Therefore, to divide the input space into the four categories, we need to have one decision boundary divide the four classes into two sets of two. The remaining boundary must then isolate each class. Two such boundaries are illustrated in Figure P4.4. We now know that our patterns are linearly separable.

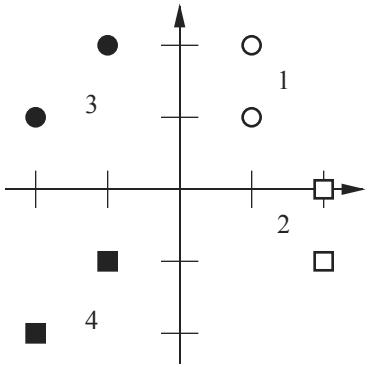


Figure P4.3 Input Vectors for Problem P4.3

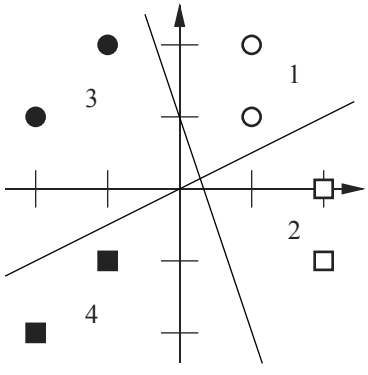


Figure P4.4 Tentative Decision Boundaries for Problem P4.3

The weight vectors should be orthogonal to the decision boundaries and should point toward the regions where the neuron outputs are 1. The next step is to decide which side of each boundary should produce a 1. One choice is illustrated in Figure P4.5, where the shaded areas represent outputs of 1. The darkest shading indicates that both neuron outputs are 1. Note that this solution corresponds to target values of

$$\text{class 1:}\left\{\mathbf{t}_1 = \begin{bmatrix}0\\0\end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix}0\\0\end{bmatrix}\right\}, \text{ class 2: } \left\{\mathbf{t}_3 = \begin{bmatrix}0\\1\end{bmatrix}, \mathbf{t}_4 = \begin{bmatrix}0\\1\end{bmatrix}\right\},$$

$$\text{class 3:}\left\{\mathbf{t}_5 = \begin{bmatrix}1\\0\end{bmatrix}, \mathbf{t}_6 = \begin{bmatrix}1\\0\end{bmatrix}\right\}, \text{ class 4: } \left\{\mathbf{t}_7 = \begin{bmatrix}1\\1\end{bmatrix}, \mathbf{t}_8 = \begin{bmatrix}1\\1\end{bmatrix}\right\}.$$

We can now select the weight vectors:

$$ {}_1\mathbf{w} = \begin{bmatrix} -3 \\ -1 \end{bmatrix} \text{ and } {}_2\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}. $$

Note that the lengths of the weight vectors is not important, only their directions. They must be orthogonal to the decision boundaries. Now we can calculate the bias by picking a point on a boundary and satisfying Eq. (4.15):

$$ b_1 = -{}_1\mathbf{w}^T\mathbf{p} = -\begin{bmatrix} -3 & -1 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix} = 1, $$

$$ b_2 = -{}_2\mathbf{w}^T\mathbf{p} = -\begin{bmatrix} 1 & -2 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 0. $$

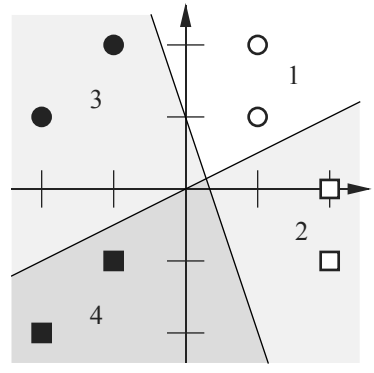


Figure P4.5 Decision Regions for Problem P4.3

In matrix form we have

$$ \mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \end{bmatrix} = \begin{bmatrix} -3 & -1 \\ 1 & -2 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, $$

which completes our design.

P4.4 Solve the following classification problem with the perceptron rule. Apply each input vector in order, for as many repetitions as it takes to ensure that the problem is solved. Draw a graph of the problem only after you have found a solution.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\2\end{bmatrix}, t_1 = 0\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}1\\-2\end{bmatrix}, t_2 = 1\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}-2\\2\end{bmatrix}, t_3 = 0\right\}\left\{\mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}, t_4 = 1\right\}$$

Use the initial weights and bias:

$$\mathbf{W}(0) = \begin{bmatrix}0 & 0\end{bmatrix} \qquad b(0) = 0.$$

We start by calculating the perceptron's output a for the first input vector $\mathbf{p}_1$, using the initial weights and bias.

$$a = hardlim(\mathbf{W}(0)\mathbf{p}_1 + b(0))$$

$$= hardlim\left(\begin{bmatrix}0 & 0\end{bmatrix}\begin{bmatrix}2\\2\end{bmatrix} + 0\right) = hardlim(0) = 1$$

The output a does not equal the target value t_1, so we use the perceptron rule to find new weights and biases based on the error.

$$e = t_1 - a = 0 - 1 = -1$$
$$\mathbf{W}(1) = \mathbf{W}(0) + e\mathbf{p}_1^T = \begin{bmatrix}0 & 0\end{bmatrix} + (-1)\begin{bmatrix}2 & 2\end{bmatrix} = \begin{bmatrix}-2 & -2\end{bmatrix}$$
$$b(1) = b(0) + e = 0 + (-1) = -1$$

We now apply the second input vector $\mathbf{p}_2$, using the updated weights and bias.

$$a = hardlim(\mathbf{W}(1)\mathbf{p}_2 + b(1))$$

$$= hardlim\left(\begin{bmatrix}-2 & -2\end{bmatrix}\begin{bmatrix}1\\-2\end{bmatrix} - 1\right) = hardlim(1) = 1$$

This time the output a is equal to the target t_2. Application of the perceptron rule will not result in any changes.

$$\mathbf{W}(2) = \mathbf{W}(1)$$
$$b(2) = b(1)$$

We now apply the third input vector.

$$a = hardlim(\mathbf{W}(2)\mathbf{p}_3 + b(2))$$

$$= hardlim\left(\begin{bmatrix}-2 & -2\end{bmatrix}\begin{bmatrix}-2\\2\end{bmatrix} - 1\right) = hardlim(-1) = 0$$

The output in response to input vector $\mathbf{p}_3$ is equal to the target t_3, so there will be no changes.

$$\mathbf{W}(3) = \mathbf{W}(2)$$
$$b(3) = b(2)$$

We now move on to the last input vector $\mathbf{p}_4$.

$$a = hardlim(\mathbf{W}(3)\mathbf{p}_4 + b(3))$$

$$= hardlim\left(\begin{bmatrix}-2 & -2\end{bmatrix}\begin{bmatrix}-1\\1\end{bmatrix} - 1\right) = hardlim(-1) = 0$$

This time the output a does not equal the appropriate target t_4. The perceptron rule will result in a new set of values for $\mathbf{W}$ and b.

$$e = t_4 - a = 1 - 0 = 1$$
$$\mathbf{W}(4) = \mathbf{W}(3) + e\mathbf{p}_4^T = \begin{bmatrix}-2 & -2\end{bmatrix} + (1)\begin{bmatrix}-1 & 1\end{bmatrix} = \begin{bmatrix}-3 & -1\end{bmatrix}$$
$$b(4) = b(3) + e = -1 + 1 = 0$$

We now must check the first vector $\mathbf{p}_1$ again. This time the output a is equal to the associated target t_1.

$$a = hardlim(\mathbf{W}(4)\mathbf{p}_1 + b(4))$$

$$= hardlim\left(\begin{bmatrix}-3 & -1\end{bmatrix}\begin{bmatrix}2\\2\end{bmatrix} + 0\right) = hardlim(-8) = 0$$

Therefore there are no changes.

$$\mathbf{W}(5) = \mathbf{W}(4)$$
$$b(5) = b(4)$$

The second presentation of $\mathbf{p}_2$ results in an error and therefore a new set of weight and bias values.

$$a = hardlim(\mathbf{W}(5)\mathbf{p}_2 + b(5))$$

$$= hardlim\left(\begin{bmatrix}-3 & -1\end{bmatrix}\begin{bmatrix}1\\-2\end{bmatrix} + 0\right) = hardlim(-1) = 0$$

Here are those new values:

$$e = t_2 - a = 1 - 0 = 1$$

$$\mathbf{W}(6) = \mathbf{W}(5) + e\mathbf{p}_2^T = \begin{bmatrix}-3 & -1\end{bmatrix} + (1)\begin{bmatrix}1 & -2\end{bmatrix} = \begin{bmatrix}-2 & -3\end{bmatrix}$$

$$b(6) = b(5) + e = 0 + 1 = 1.$$

Cycling through each input vector once more results in no errors.

$$a = hardlim(\mathbf{W}(6)\mathbf{p}_3 + b(6)) = hardlim\left(\begin{bmatrix}-2 & -3\end{bmatrix}\begin{bmatrix}-2\\2\end{bmatrix} + 1\right) = 0 = t_3$$

$$a = hardlim(\mathbf{W}(6)\mathbf{p}_4 + b(6)) = hardlim\left(\begin{bmatrix}-2 & -3\end{bmatrix}\begin{bmatrix}-1\\1\end{bmatrix} + 1\right) = 1 = t_4$$

$$a = hardlim(\mathbf{W}(6)\mathbf{p}_1 + b(6)) = hardlim\left(\begin{bmatrix}-2 & -3\end{bmatrix}\begin{bmatrix}2\\2\end{bmatrix} + 1\right) = 0 = t_1$$

$$a = hardlim(\mathbf{W}(6)\mathbf{p}_2 + b(6)) = hardlim\left(\begin{bmatrix}-2 & -3\end{bmatrix}\begin{bmatrix}1\\-2\end{bmatrix} + 1\right) = 1 = t_2$$

Therefore the algorithm has converged. The final solution is:

$$\mathbf{W} = \begin{bmatrix}-2 & -3\end{bmatrix} \qquad b = 1.$$

Now we can graph the training data and the decision boundary of the solution. The decision boundary is given by

$$n = \mathbf{W}\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b = -2p_1 - 3p_2 + 1 = 0.$$

To find the p_2 intercept of the decision boundary, set $p_1 = 0$:

$$p_2 = -\frac{b}{w_{1,2}} = -\frac{1}{-3} = \frac{1}{3} \qquad \text{if } p_1 = 0$$

.

To find the p_1 intercept, set $p_2 = 0$:

$$p_1 = -\frac{b}{w_{1,1}} = -\frac{1}{-2} = \frac{1}{2} \qquad \text{if } p_2 = 0$$

.

The resulting decision boundary is illustrated in Figure P4.6.

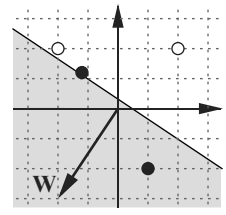


Figure P4.6 Decision Boundary for Problem P4.4

Note that the decision boundary falls across one of the training vectors. This is acceptable, given the problem definition, since the hard limit function returns 1 when given an input of 0, and the target for the vector in question is indeed 1.

P4.5 Consider again the four-class decision problem that we introduced in Problem P4.3. Train a perceptron network to solve this problem using the perceptron learning rule.

If we use the same target vectors that we introduced in Problem P4.3, the training set will be:

$$\left\{\mathbf{p}_1 = \begin{bmatrix}1\\1\end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix}0\\0\end{bmatrix}\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}1\\2\end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix}0\\0\end{bmatrix}\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}2\\-1\end{bmatrix}, \mathbf{t}_3 = \begin{bmatrix}0\\1\end{bmatrix}\right\}$$

$$\left\{\mathbf{p}_4 = \begin{bmatrix}2\\0\end{bmatrix}, \mathbf{t}_4 = \begin{bmatrix}0\\1\end{bmatrix}\right\}\left\{\mathbf{p}_5 = \begin{bmatrix}-1\\2\end{bmatrix}, \mathbf{t}_5 = \begin{bmatrix}1\\0\end{bmatrix}\right\}\left\{\mathbf{p}_6 = \begin{bmatrix}-2\\1\end{bmatrix}, \mathbf{t}_6 = \begin{bmatrix}1\\0\end{bmatrix}\right\}$$

$$\left\{\mathbf{p}_7 = \begin{bmatrix}-1\\-1\end{bmatrix}, \mathbf{t}_7 = \begin{bmatrix}1\\1\end{bmatrix}\right\}\left\{\mathbf{p}_8 = \begin{bmatrix}-2\\-2\end{bmatrix}, \mathbf{t}_8 = \begin{bmatrix}1\\1\end{bmatrix}\right\}.$$

Let's begin the algorithm with the following initial weights and biases:

$$\mathbf{W}(0) = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}, \mathbf{b}(0) = \begin{bmatrix}1\\1\end{bmatrix}.$$

The first iteration is

$$\mathbf{a} = hardlim\,(\mathbf{W}(0)\mathbf{p}_1 + \mathbf{b}(0)) = hardlim\,(\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}1\\ 1\end{bmatrix} + \begin{bmatrix}1\\ 1\end{bmatrix}) = \begin{bmatrix}1\\ 1\end{bmatrix},$$

$$\mathbf{e} = \mathbf{t}_1 - \mathbf{a} = \begin{bmatrix}0\\ 0\end{bmatrix} - \begin{bmatrix}1\\ 1\end{bmatrix} = \begin{bmatrix}-1\\ -1\end{bmatrix},$$

$$\mathbf{W}(1) = \mathbf{W}(0) + \mathbf{e}\mathbf{p}_1^T = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} + \begin{bmatrix}-1\\ -1\end{bmatrix}\begin{bmatrix}1 & 1\end{bmatrix} = \begin{bmatrix}0 & -1\\ -1 & 0\end{bmatrix},$$

$$\mathbf{b}(1) = \mathbf{b}(0) + \mathbf{e} = \begin{bmatrix}1\\ 1\end{bmatrix} + \begin{bmatrix}-1\\ -1\end{bmatrix} = \begin{bmatrix}0\\ 0\end{bmatrix}.$$

The second iteration is

$$\mathbf{a} = hardlim\,(\mathbf{W}(1)\mathbf{p}_2 + \mathbf{b}(1)) = hardlim\,(\begin{bmatrix}0 & -1\\ -1 & 0\end{bmatrix}\begin{bmatrix}1\\ 2\end{bmatrix} + \begin{bmatrix}0\\ 0\end{bmatrix}) = \begin{bmatrix}0\\ 0\end{bmatrix},$$

$$\mathbf{e} = \mathbf{t}_2 - \mathbf{a} = \begin{bmatrix}0\\ 0\end{bmatrix} - \begin{bmatrix}0\\ 0\end{bmatrix} = \begin{bmatrix}0\\ 0\end{bmatrix},$$

$$\mathbf{W}(2) = \mathbf{W}(1) + \mathbf{e}\mathbf{p}_2^T = \begin{bmatrix}0 & -1\\ -1 & 0\end{bmatrix} + \begin{bmatrix}0\\ 0\end{bmatrix}\begin{bmatrix}1 & 2\end{bmatrix} = \begin{bmatrix}0 & -1\\ -1 & 0\end{bmatrix},$$

$$\mathbf{b}(2) = \mathbf{b}(1) + \mathbf{e} = \begin{bmatrix}0\\ 0\end{bmatrix} + \begin{bmatrix}0\\ 0\end{bmatrix} = \begin{bmatrix}0\\ 0\end{bmatrix}.$$

The third iteration is

$$\mathbf{a} = hardlim\,(\mathbf{W}(2)\mathbf{p}_3 + \mathbf{b}(2)) = hardlim\,(\begin{bmatrix}0 & -1\\ -1 & 0\end{bmatrix}\begin{bmatrix}2\\ -1\end{bmatrix} + \begin{bmatrix}0\\ 0\end{bmatrix}) = \begin{bmatrix}1\\ 0\end{bmatrix},$$

$$\mathbf{e} = \mathbf{t}_3 - \mathbf{a} = \begin{bmatrix}0\\ 1\end{bmatrix} - \begin{bmatrix}1\\ 0\end{bmatrix} = \begin{bmatrix}-1\\ 1\end{bmatrix},$$

$$\mathbf{W}(3) = \mathbf{W}(2) + \mathbf{e}\mathbf{p}_3^T = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \end{bmatrix} \begin{bmatrix} 2 & -1 \end{bmatrix} = \begin{bmatrix} -2 & 0 \\ 1 & -1 \end{bmatrix},$$

$$\mathbf{b}(3) = \mathbf{b}(2) + \mathbf{e} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

Iterations four through eight produce no changes in the weights.

$$\mathbf{W}(8) = \mathbf{W}(7) = \mathbf{W}(6) = \mathbf{W}(5) = \mathbf{W}(4) = \mathbf{W}(3)$$

$$\mathbf{b}(8) = \mathbf{b}(7) = \mathbf{b}(6) = \mathbf{b}(5) = \mathbf{b}(4) = \mathbf{b}(3)$$

The ninth iteration produces

$$\mathbf{a} = hardlim\,(\mathbf{W}(8)\mathbf{p}_1 + \mathbf{b}(8)) = hardlim\,(\begin{bmatrix} -2 & 0 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \end{bmatrix}) = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

$$\mathbf{e} = \mathbf{t}_1 - \mathbf{a} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

$$\mathbf{W}(9) = \mathbf{W}(8) + \mathbf{e}\mathbf{p}_1^T = \begin{bmatrix} -2 & 0 \\ 1 & -1 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & 1 \end{bmatrix} = \begin{bmatrix} -2 & 0 \\ 0 & -2 \end{bmatrix},$$

$$\mathbf{b}(9) = \mathbf{b}(8) + \mathbf{e} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}.$$

At this point the algorithm has converged, since all input patterns will be correctly classified. The final decision boundaries are displayed in Figure P4.7. Compare this result with the network we designed in Problem P4.3.

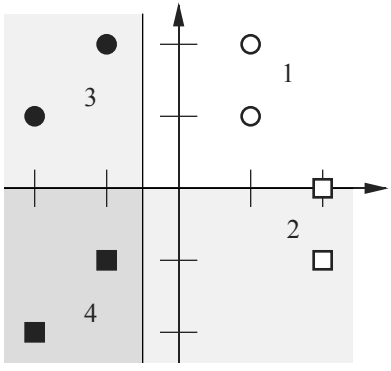


Figure P4.7 Final Decision Boundaries for Problem P4.5

Epilogue

In this chapter we have introduced our first learning rule — the perceptron learning rule. It is a type of learning called *supervised learning*, in which the learning rule is provided with a set of examples of proper network behavior. As each input is applied to the network, the learning rule adjusts the network parameters so that the network output will move closer to the target.

The perceptron learning rule is very simple, but it is also quite powerful. We have shown that the rule will always converge to a correct solution, if such a solution exists. The weakness of the perceptron network lies not with the learning rule, but with the structure of the network. The standard perceptron is only able to classify vectors that are linearly separable. We will see in Chapter 11 that the perceptron architecture can be generalized to multilayer perceptrons, which can solve arbitrary classification problems. The backpropagation learning rule, which is introduced in Chapter 11, can be used to train these networks.

In Chapters 3 and 4 we have used many concepts from the field of linear algebra, such as inner product, projection, distance (norm), etc. We will find in later chapters that a good foundation in linear algebra is essential to our understanding of all neural networks. In Chapters 5 and 6 we will review some of the key concepts from linear algebra that will be most important in our study of neural networks. Our objective will be to obtain a fundamental understanding of how neural networks work.

Further Reading

[BaSu83] A. Barto, R. Sutton and C. Anderson, "Neuron-like adaptive elements can solve difficult learning control problems," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 13, No. 5, pp. 834–846, 1983.

A classic paper in which a reinforcement learning algorithm is used to train a neural network to balance an inverted pendulum.

[Brog91] W. L. Brogan, *Modern Control Theory*, 3rd Ed., Englewood Cliffs, NJ: Prentice-Hall, 1991.

A well-written book on the subject of linear systems. The first half of the book is devoted to linear algebra. It also has good sections on the solution of linear differential equations and the stability of linear and nonlinear systems. It has many worked problems.

[McPi43] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, Vol. 5, pp. 115–133, 1943.

This article introduces the first mathematical model of a neuron, in which a weighted sum of input signals is compared to a threshold to determine whether or not the neuron fires.

[MiPa69] M. Minsky and S. Papert, *Perceptrons*, Cambridge, MA: MIT Press, 1969.

A landmark book that contains the first rigorous study devoted to determining what a perceptron network is capable of learning. A formal treatment of the perceptron was needed both to explain the perceptron's limitations and to indicate directions for overcoming them. Unfortunately, the book pessimistically predicted that the limitations of perceptrons indicated that the field of neural networks was a dead end. Although this was not true, it temporarily cooled research and funding for research for several years.

[Rose58] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, Vol. 65, pp. 386–408, 1958.

This paper presents the first practical artificial neural network — the perceptron.

[Rose61] F. Rosenblatt, *Principles of Neurodynamics*, Washington DC: Spartan Press, 1961.

One of the first books on neurocomputing.

[WhSo92] D. White and D. Sofge (Eds.), *Handbook of Intelligent Control*, New York: Van Nostrand Reinhold, 1992.

Collection of articles describing current research and applications of neural networks and fuzzy logic to control systems.

Exercises

E4.1 Consider the classification problem defined below:

$$\left\{\mathbf{p}_1 = \begin{bmatrix}-1\\1\end{bmatrix}, t_1 = 1\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}0\\0\end{bmatrix}, t_2 = 1\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}1\\-1\end{bmatrix}, t_3 = 1\right\}\left\{\mathbf{p}_4 = \begin{bmatrix}1\\0\end{bmatrix}, t_4 = 0\right\}$$
$$\left\{\mathbf{p}_5 = \begin{bmatrix}0\\1\end{bmatrix}, t_5 = 0\right\}.$$

i. Draw a diagram of the single-neuron perceptron you would use to solve this problem. How many inputs are required?

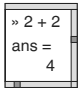
ii. Draw a graph of the data points, labeled according to their targets. Is this problem solvable with the network you defined in part (i)? Why or why not?

E4.2 Consider the classification problem defined below.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}-1\\1\end{bmatrix}, t_1 = 1\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}-1\\-1\end{bmatrix}, t_2 = 1\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}0\\0\end{bmatrix}, t_3 = 0\right\}\left\{\mathbf{p}_4 = \begin{bmatrix}1\\0\end{bmatrix}, t_4 = 0\right\}.$$

i. Design a single-neuron perceptron to solve this problem. Design the network graphically, by choosing weight vectors that are orthogonal to the decision boundaries.

ii. Test your solution with all four input vectors.



iii. Classify the following input vectors with your solution. You can either perform the calculations manually or with MATLAB.

$$\mathbf{p}_5 = \begin{bmatrix}-2\\0\end{bmatrix} \qquad \mathbf{p}_6 = \begin{bmatrix}1\\1\end{bmatrix} \qquad \mathbf{p}_7 = \begin{bmatrix}0\\1\end{bmatrix} \qquad \mathbf{p}_8 = \begin{bmatrix}-1\\-2\end{bmatrix}$$

iv. Which of the vectors in part (iii) will always be classified the same way, regardless of the solution values for $\mathbf{W}$ and b? Which may vary depending on the solution? Why?

E4.3 Solve the classification problem in Exercise E4.2 by solving inequalities (as in Problem P4.2), and repeat parts (ii) and (iii) with the new solution. (The solution is more difficult than Problem P4.2, since you can't isolate the weights and biases in a pairwise manner.)

E4.4 Solve the classification problem in Exercise E4.2 by applying the perceptron rule to the following initial parameters, and repeat parts (ii) and (iii) with the new solution.

$$\mathbf{W}(0) = \begin{bmatrix} 0 & 0 \end{bmatrix} \qquad b(0) = 0$$

E4.5 Prove mathematically (not graphically) that the following problem is unsolvable for a two-input/single-neuron perceptron.

$$\left\{\mathbf{p}_1 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_1 = 1\right\}\left\{\mathbf{p}_2 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, t_2 = 0\right\}\left\{\mathbf{p}_3 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, t_3 = 1\right\}\left\{\mathbf{p}_4 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, t_4 = 0\right\}$$

(Hint: start by rewriting the input/target requirements as inequalities that constrain the weight and bias values.)

E4.6 We have four categories of vectors.

$$\text{Category I: } \left\{\begin{bmatrix} -1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}\right\}, \text{ Category II: } \left\{\begin{bmatrix} 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \end{bmatrix}\right\}$$

$$\text{Category III: } \left\{\begin{bmatrix} 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}\right\}, \text{ Category IV: } \left\{\begin{bmatrix} 1 \\ -1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}\right\}$$

i. Design a two-neuron perceptron network (single layer) to recognize these four categories of vectors. Sketch the decision boundaries.

ii. Draw the network diagram.

iii. Suppose the following vector is to be added to Category I.

$$\begin{bmatrix} -1 \\ -3 \end{bmatrix}$$

Perform one iteration of the perceptron learning rule with this vector. (Start with the weights you determined in part i.) Draw the new decision boundaries.

E4.7 We have two categories of vectors. Category I consists of

$$\left\{\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right\}.$$

Category II consists of

$$\left\{ \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 2 \end{bmatrix}, \begin{bmatrix} -2 \\ 0 \end{bmatrix} \right\}.$$

i. Design a single-neuron perceptron network to recognize these two categories of vectors.

ii. Draw the network diagram.

iii. Sketch the decision boundary.

iv. If we add the following vector to Category I, will your network classify it correctly? Demonstrate by computing the network response.

$$\begin{bmatrix} -3 \\ 0 \end{bmatrix}$$

v. Can your weight matrix and bias be modified so your network can classify this new vector correctly (while continuing to classify the other vectors correctly)? Explain.

E4.8 We want to train a perceptron network with the following training set:

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, t_1 = 0 \right\} \left\{ \mathbf{p}_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, t_2 = 0 \right\} \left\{ \mathbf{p}_3 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_3 = 1 \right\}.$$

The initial weight matrix and bias are

$$\mathbf{W}(0) = \begin{bmatrix} 1 & 0 \end{bmatrix}, \; b(0) = 0.5 .$$

i. Plot the initial decision boundary, weight vector and input patterns. Which patterns are correctly classified using the initial weight and bias?

ii. Train the network with the perceptron rule. Present each input vector once, in the order shown.

iii. Plot the final decision boundary, and demonstrate graphically which patterns are correctly classified.

iv. Will the perceptron rule (given enough iterations) always learn to correctly classify the patterns in this training set, no matter what initial weights we use? Explain.

E4.9 We want to train a perceptron network using the following training set:

$$\left\{\mathbf{p}_1 = \begin{bmatrix}1\\0\end{bmatrix}, t_1 = 0\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}-1\\2\end{bmatrix}, t_2 = 0\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}1\\2\end{bmatrix}, t_3 = 1\right\},$$

starting from the initial conditions

$$\mathbf{W}(0) = \begin{bmatrix}0 & 1\end{bmatrix}, b(0) = \begin{bmatrix}1\end{bmatrix}.$$

i. Sketch the initial decision boundary, and show the weight vector and the three training input vectors, $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$. Indicate the class of each input vector, and show which ones are correctly classified by the initial decision boundary.

ii. Present the input $\mathbf{p}_1$ to the network, and perform one iteration of the perceptron learning rule.

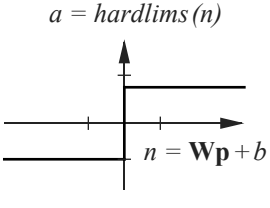
iii. Sketch the new decision boundary and weight vector, and again indicate which of the three input vectors are correctly classified.

iv. Present the input $\mathbf{p}_2$ to the network, and perform one more iteration of the perceptron learning rule.

v. Sketch the new decision boundary and weight vector, and again indicate which of the three input vectors are correctly classified.

vi. If you continued to use the perceptron learning rule, and presented all of the patterns many times, would the network eventually learn to correctly classify the patterns? Explain your answer. (This part does not require any calculations.)

E4.10 The symmetric hard limit function is sometimes used in perceptron networks, instead of the hard limit function. Target values are then taken from the set [-1, 1] instead of [0, 1].



i. Write a simple expression that maps numbers in the ordered set [0, 1] into the ordered set [-1, 1]. Write the expression that performs the inverse mapping.

ii. Consider two single-neuron perceptrons with the same weight and bias values. The first network uses the hard limit function ([0, 1] values), and the second network uses the symmetric hard limit function. If the two networks are given the same input $\mathbf{p}$, and updated with the perceptron learning rule, will their weights continue to have the same value?

iii. If the changes to the weights of the two neurons are different, how do they differ? Why?

iv. Given initial weight and bias values for a standard hard limit perceptron, create a method for initializing a symmetric hard limit perceptron so that the two neurons will always respond identically when trained on identical data.

E4.11 The vectors in the ordered set defined below were obtained by measuring the weight and ear lengths of toy rabbits and bears in the Fuzzy Wuzzy Animal Factory. The target values indicate whether the respective input vector was taken from a rabbit (0) or a bear (1). The first element of the input vector is the weight of the toy, and the second element is the ear length.

```
» 2 + 2
ans =
     4
```

$$\left\{\mathbf{p}_1 = \begin{bmatrix}1\\4\end{bmatrix}, t_1 = 0\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}1\\5\end{bmatrix}, t_2 = 0\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}2\\4\end{bmatrix}, t_3 = 0\right\}\left\{\mathbf{p}_4 = \begin{bmatrix}2\\5\end{bmatrix}, t_4 = 0\right\}$$

$$\left\{\mathbf{p}_5 = \begin{bmatrix}3\\1\end{bmatrix}, t_5 = 1\right\}\left\{\mathbf{p}_6 = \begin{bmatrix}3\\2\end{bmatrix}, t_6 = 1\right\}\left\{\mathbf{p}_7 = \begin{bmatrix}4\\1\end{bmatrix}, t_7 = 1\right\}\left\{\mathbf{p}_8 = \begin{bmatrix}4\\2\end{bmatrix}, t_8 = 1\right\}$$

i. Use MATLAB to initialize and train a network to solve this "practical" problem.

ii. Use MATLAB to test the resulting weight and bias values against the input vectors.

iii. Add input vectors to the training set to ensure that the decision boundary of any solution will not intersect one of the original input vectors (i.e., to ensure only robust solutions are found). Then retrain the network. Your method for adding the input vectors should be general purpose (not designed specifically for this problem).

E4.12 Consider again the four-category classification problem described in Problems P4.3 and P4.5. Suppose that we change the input vector $\mathbf{p}_3$ to

$$\mathbf{p}_3 = \begin{bmatrix}2\\2\end{bmatrix}.$$

i. Is the problem still linearly separable? Demonstrate your answer graphically.

```
» 2 + 2
ans =
     4
```

ii. Use MATLAB to initialize and train a network to solve this problem. Explain your results.

iii. If $\mathbf{p}_3$ is changed to

$$\mathbf{p}_3 = \begin{bmatrix}2\\1.5\end{bmatrix}$$

is the problem linearly separable?

iv. With the $\mathbf{p}_3$ from (iii), use MATLAB to initialize and train a network to solve this problem. Explain your results.

E4.13 One variation of the perceptron learning rule is

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha \mathbf{e}\mathbf{p}^T$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \alpha \mathbf{e}$$

where α is called the learning rate. Prove convergence of this algorithm. Does the proof require a limit on the learning rate? Explain.

5 Signal and Weight Vector Spaces

Objectives

It is clear from Chapters 3 and 4 that it is very useful to think of the inputs and outputs of a neural network, and the rows of a weight matrix, as vectors. In this chapter we want to examine these vector spaces in detail and to review those properties of vector spaces that are most helpful when analyzing neural networks. We will begin with general definitions and then apply these definitions to specific neural network problems. The concepts that are discussed in this chapter and in Chapter 6 will be used extensively throughout the remaining chapters of this book. They are critical to our understanding of why neural networks work.

Theory and Examples

Linear algebra is the core of the mathematics required for understanding neural networks. In Chapters 3 and 4 we saw the utility of representing the inputs and outputs of neural networks as vectors. In addition, we saw that it is often useful to think of the rows of a weight matrix as vectors in the same vector space as the input vectors.

Recall from Chapter 3 that in the Hamming network the rows of the weight matrix of the feedforward layer were equal to the prototype vectors. In fact, the purpose of the feedforward layer was to calculate the inner products between the prototype vectors and the input vector.

In the single neuron perceptron network we noted that the decision boundary was always orthogonal to the weight matrix (a row vector).

In this chapter we want to review the basic concepts of vector spaces (e.g., inner products, orthogonality) in the context of neural networks. We will begin with a general definition of vector spaces. Then we will present the basic properties of vectors that are most useful for neural network applications.

One comment about notation before we begin. All of the vectors we have discussed so far have been ordered n-tuples (columns) of real numbers and are represented by bold small letters, e.g.,

$$\mathbf{x} = \left[x_1\ x_2\ \ldots\ x_n\right]^T. \tag{5.1}$$

These are vectors in $\Re^n$, the standard n-dimensional Euclidean space. In this chapter we will also be talking about more general vector spaces than $\Re^n$. These more general vectors will be represented with a script typeface, as in $\mathcal{X}$. We will show in this chapter how these general vectors can often be represented by columns of numbers.

Linear Vector Spaces

What do we mean by a vector space? We will begin with a very general definition. While this definition may seem abstract, we will provide many concrete examples. By using a general definition we can solve a larger class of problems, and we can impart a deeper understanding of the concepts.

Vector Space

Definition. A linear *vector space*, X, is a set of elements (vectors) defined over a scalar field, F, that satisfies the following conditions:

1. An operation called vector addition is defined such that if $\mathcal{X} \in X$ ($\mathcal{X}$ is an element of X) and $\mathcal{Y} \in X$, then $\mathcal{X} + \mathcal{Y} \in X$.

2. $x + y = y + x$.

3. $(x + y) + z = x + (y + z)$.

4. There is a unique vector $0 \in X$, called the zero vector, such that $x + 0 = x$ for all $x \in X$.

5. For each vector $x \in X$ there is a unique vector in X, to be called $-x$, such that $x + (-x) = 0$.

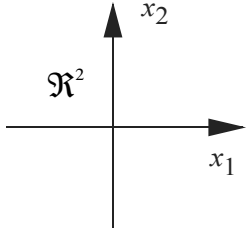
6. An operation, called multiplication, is defined such that for all scalars $a \in F$, and all vectors $x \in X$, $ax \in X$.

7. For any $x \in X$, $1x = x$ (for scalar 1).

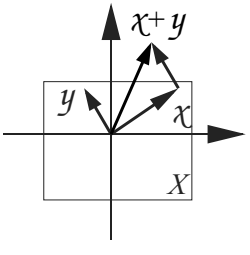
8. For any two scalars $a \in F$ and $b \in F$, and any $x \in X$, $a(bx) = (ab)x$.

9. $(a + b)x = ax + bx$.

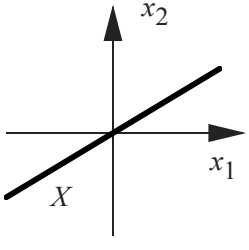
10. $a(x + y) = ax + ay$.



To illustrate these conditions, let's investigate a few sample sets and determine whether or not they are vector spaces. First consider the standard two-dimensional Euclidean space, $\Re^2$, shown in the upper left figure. This is clearly a vector space, and all ten conditions are satisfied for the standard definitions of vector addition and scalar multiplication.



What about subsets of $\Re^2$? What subsets of $\Re^2$ are also vector spaces (subspaces)? Consider the boxed area (X) in the center left figure. Does it satisfy all ten conditions? No. Clearly even condition 1 is not satisfied. The vectors x and y shown in the figure are in X, but $x + y$ is not. From this example it is clear that no bounded sets can be vector spaces.



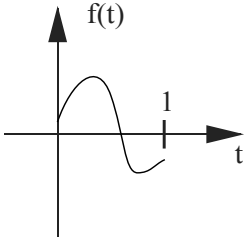
Are there any subsets of $\Re^2$ that are vector spaces? Consider the line (X) shown in the bottom left figure. (Assume that the line extends to infinity in both directions.) Is this line a vector space? We leave it to you to show that indeed all ten conditions are satisfied. Will any such infinite line satisfy the ten conditions? Well, any line that passes through the origin will work. If it does not pass through the origin then condition 4, for instance, would not be satisfied.

In addition to the standard Euclidean spaces, there are other sets that also satisfy the ten conditions of a vector space. Consider, for example, the set P^2 of all polynomials of degree less than or equal to 2. Two members of this set would be

$$x = 2 + t + 4t^2$$

$$y = 1 + 5t\,. \tag{5.2}$$

If you are used to thinking of vectors only as columns of numbers, these may seem to be strange vectors indeed. However, recall that to be a vector space, a set need only satisfy the ten conditions we presented. Are these conditions satisfied for the set P^2? If we add two polynomials of degree less than or equal to 2, the result will also be a polynomial of degree less than or equal to 2. Therefore condition 1 is satisfied. We can also multiply a polynomial by a scalar without changing the order of the polynomial. Therefore condition 6 is satisfied. It is not difficult to show that all ten conditions are satisfied, showing that P^2 is a vector space.



Consider the set $C_{[0,\,1]}$ of all continuous functions defined on the interval [0, 1]. Two members of this set would be

$$x = \sin(t)$$

$$y = e^{-2t}\,. \tag{5.3}$$

Another member of the set is shown in the figure to the left.

The sum of two continuous functions is also a continuous function, and a scalar times a continuous function is a continuous function. The set $C_{[0,\,1]}$ is also a vector space. This set is different than the other vector spaces we have discussed; it is infinite dimensional. We will define what we mean by dimension later in this chapter.

Linear Independence

Now that we have defined what we mean by a vector space, we will investigate some of the properties of vectors. The first properties are linear dependence and linear independence.

Consider n vectors $\{x_1, x_2, \ldots, x_n\}$. If there exist n scalars $a_1, a_2, \ldots, a_n$, at least one of which is nonzero, such that

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = 0, \tag{5.4}$$

then the $\{x_i\}$ are linearly dependent.

Linear Independence

The converse statement would be: If $a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = 0$ implies that each $a_i = 0$, then $\{x_i\}$ is a set of *linearly independent* vectors.

Note that these definitions are equivalent to saying that if a set of vectors is independent then no vector in the set can be written as a linear combination of the other vectors.

As an example of independence, consider the pattern recognition problem of Chapter 3. The two prototype patterns (*orange* and *apple*) were given by:

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, \ \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}. \tag{5.5}$$

Let $a_1\mathbf{p}_1 + a_2\mathbf{p}_2 = \mathbf{0}$, then

$$\begin{bmatrix} a_1 + a_2 \\ -a_1 + a_2 \\ -a_1 + (-a_2) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \tag{5.6}$$

but this can only be true if $a_1 = a_2 = 0$. Therefore $\mathbf{p}_1$ and $\mathbf{p}_2$ are linearly independent.

Consider vectors from the space P^2 of polynomials of degree less than or equal to 2. Three vectors from this space would be

$$\chi_1 = 1 + t + t^2, \ \chi_2 = 2 + 2t + t^2, \ \chi_3 = 1 + t. \tag{5.7}$$

Note that if we let $a_1 = 1$, $a_2 = -1$ and $a_3 = 1$, then

$$a_1\chi_1 + a_2\chi_2 + a_3\chi_3 = 0. \tag{5.8}$$

Therefore these three vectors are linearly dependent.

Spanning a Space

Next we want to define what we mean by the dimension (size) of a vector space. To do so we must first define the concept of a spanning set.

Let X be a linear vector space and let $\{u_1, u_2, \ldots, u_m\}$ be a subset of general vectors in X. This subset spans X if and only if for every vector $\chi \in X$ there exist scalars $x_1, x_2, \ldots, x_n$ such that $\chi = x_1u_1 + x_2u_2 + \cdots + x_mu_m$. In other words, a subset spans a space if every vector in the space can be written as a linear combination of the vectors in the subset.

The dimension of a vector space is determined by the minimum number of vectors it takes to span the space. This leads to the definition of a basis set.

Basis Set

A *basis set* for X is a set of linearly independent vectors that spans X. Any basis set contains the minimum number of vectors required to span the

space. The dimension of X is therefore equal to the number of elements in the basis set. Any vector space can have many basis sets, but each one must contain the same number of elements. (See [Stra80] for a proof of this fact.)

Take, for example, the linear vector space P^2. One possible basis for this space is

$$u_1 = 1,\ u_2 = t,\ u_3 = t^2. \tag{5.9}$$

Clearly any polynomial of degree two or less can be created by taking a linear combination of these three vectors. Note, however, that *any* three independent vectors from P^2 would form a basis for this space. One such alternate basis is:

$$u_1 = 1,\ u_2 = 1 + t,\ u_3 = 1 + t + t^2. \tag{5.10}$$

Inner Product

From our brief encounter with neural networks in Chapters 3 and 4, it is clear that the inner product is fundamental to the operation of many neural networks. Here we will introduce a general definition for inner products and then give several examples.

Inner Product

Any scalar function of x and y can be defined as an *inner product*, (x,y), provided that the following properties are satisfied:

1. $(x,y) = (y,x)$.
2. $(x,a y_1 + b y_2) = a(x,y_1) + b(x,y_2)$.
3. $(x,x) \geq 0$, where equality holds if and only if x is the zero vector.

The standard inner product for vectors in R^n is

$$\mathbf{x}^T\mathbf{y} = x_1y_1 + x_2y_2 + \cdots + x_ny_n, \tag{5.11}$$

but this is not the only possible inner product. Consider again the set $C_{[0,1]}$ of all continuous functions defined on the interval [0, 1]. Show that the following scalar function is man inner product (see Problem P5.6).

$$(x,y) = \int_0^1 x(t)y(t)dt \tag{5.12}$$

Norm

The next operation we need to define is the norm, which is based on the concept of vector length.

Norm

A scalar function $\|x\|$ is called a *norm* if it satisfies the following properties:

1. $\|x\| \geq 0$.
2. $\|x\| = 0$ if and only if $x = 0$.
3. $\|ax\| = |a|\|x\|$ for scalar a.
4. $\|x + y\| \leq \|x\| + \|y\|$.

There are many functions that would satisfy these conditions. One common norm is based on the inner product:

$$\|x\| = (x,x)^{1/2}. \tag{5.13}$$

For Euclidean spaces, $\Re^n$, this yields the norm with which we are most familiar:

$$\|\mathbf{x}\| = (\mathbf{x}^T\mathbf{x})^{1/2} = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}. \tag{5.14}$$

In neural network applications it is often useful to normalize the input vectors. This means that $\|\mathbf{p}_i\| = 1$ for each input vector.

Angle

Using the norm and the inner product we can generalize the concept of angle for vector spaces of dimension greater than two. The *angle* θ between two vectors x and y is defined by

$$\cos\theta = \frac{(x,y)}{\|x\|\|y\|}. \tag{5.15}$$

Orthogonality

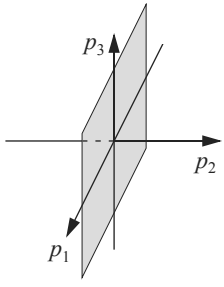
Now that we have defined the inner product operation, we can introduce the important concept of orthogonality.

Orthogonality

Two vectors $x, y \in X$ are said to be *orthogonal* if $(x,y) = 0$.

Orthogonality is an important concept in neural networks. We will see in Chapter 7 that when the prototype vectors of a pattern recognition problem are orthogonal and normalized, a linear associator neural network can be trained, using the Hebb rule, to achieve perfect recognition.

In addition to orthogonal vectors, we can also have orthogonal spaces. A vector $x \in X$ is orthogonal to a subspace X_1 if x is orthogonal to every vec-



tor in X_1. This is typically represented as $\chi \perp X_1$. A subspace X_1 is orthogonal to a subspace X_2 if every vector in X_1 is orthogonal to every vector in X_2. This is represented by $X_1 \perp X_2$.

The figure to the left illustrates the two orthogonal spaces that were used in the perceptron example of Chapter 3. (See Figure 3.4.) The p_1, p_3 plane is a subspace of $\Re^3$, which is orthogonal to the p_2 axis (which is another subspace of $\Re^3$). The p_1, p_3 plane was the decision boundary of a perceptron network. In Solved Problem P5.1 we will show that the perceptron decision boundary will be a vector space whenever the bias value is zero.

Gram-Schmidt Orthogonalization

There is a relationship between orthogonality and independence. It is possible to convert a set of independent vectors into a set of orthogonal vectors that spans the same vector space. The standard procedure to accomplish this is called Gram-Schmidt orthogonalization.

Assume that we have n independent vectors $y_1, y_2, \ldots, y_n$. From these vectors we want to obtain n orthogonal vectors $v_1, v_2, \ldots, v_n$. The first orthogonal vector is chosen to be the first independent vector:

$$v_1 = y_1 . \tag{5.16}$$

To obtain the second orthogonal vector we use y_2, but subtract off the portion of y_2 that is in the direction of v_1. This leads to the equation

$$v_2 = y_2 - a v_1 , \tag{5.17}$$

where a is chosen so that v_2 is orthogonal to v_1. This requires that

$$(v_1, v_2) = (v_1, y_2 - a v_1) = (v_1, y_2) - a(v_1, v_1) = 0 , \tag{5.18}$$

or

$$a = \frac{(v_1, y_2)}{(v_1, v_1)} . \tag{5.19}$$

Projection

Therefore to find the component of y_2 in the direction of v_1, $a v_1$, we need to find the inner product between the two vectors. We call $a v_1$ the *projection* of y_2 on the vector v_1.

If we continue this process, the kth step will be

$$v_k = y_k - \sum_{i=1}^{k-1} \frac{(v_i, y_k)}{(v_i, v_i)} v_i . \tag{5.20}$$

To illustrate this process, we consider the following independent vectors in $\Re^2$:

$$\mathbf{y}_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad \mathbf{y}_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \tag{5.21}$$

The first orthogonal vector would be

$$\mathbf{v}_1 = \mathbf{y}_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}. \tag{5.22}$$

The second orthogonal vector is calculated as follows:

$$\mathbf{v}_2 = \mathbf{y}_2 - \frac{\mathbf{v}_1^T \mathbf{y}_2}{\mathbf{v}_1^T \mathbf{v}_1} \mathbf{v}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} - \frac{\begin{bmatrix} 2 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix}}{\begin{bmatrix} 2 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix}} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} 1.6 \\ 0.8 \end{bmatrix} = \begin{bmatrix} -0.6 \\ 1.2 \end{bmatrix}. \tag{5.23}$$

See Figure 5.1 for a graphical representation of this process.

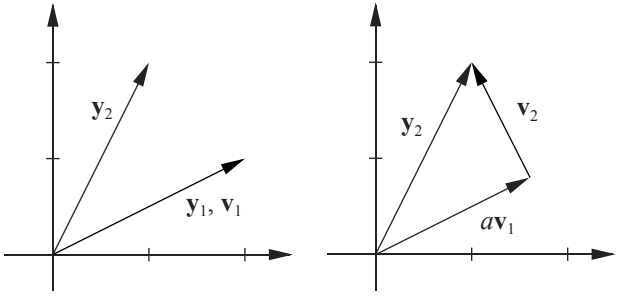


Figure 5.1 Gram-Schmidt Orthogonalization Example

Orthonormal

We could convert $\mathbf{v}_1$ and $\mathbf{v}_2$ to a set of *orthonormal* (orthogonal and normalized) vectors by dividing each vector by its norm.

To experiment with this orthogonalization process, use the Neural Network Design Demonstration Gram-Schmidt (**nnd5gs**).

Vector Expansions

Note that we have been using a script font ($\mathcal{x}$) to represent general vectors and bold type ($\mathbf{x}$) to represent vectors in $\Re^n$, which can be written as columns of numbers. In this section we will show that general vectors in finite

dimensional vector spaces can also be written as columns of numbers and therefore are in some ways equivalent to vectors in $\Re^n$.

Vector Expansion

If a vector space X has a basis set $\{v_1, v_2, \ldots, v_n\}$, then any $\chi \in X$ has a unique *vector expansion*:

$$\chi = \sum_{i=1}^{n} x_i v_i = x_1 v_1 + x_2 v_2 + \cdots + x_n v_n . \tag{5.24}$$

Therefore any vector in a finite dimensional vector space can be represented by a column of numbers:

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \ldots & x_n \end{bmatrix}^T . \tag{5.25}$$

This $\mathbf{x}$ is a representation of the general vector χ. Of course in order to interpret the meaning of $\mathbf{x}$ we need to know the basis set. If the basis set changes, $\mathbf{x}$ will change, even though it still represents the same general vector χ. We will discuss this in more detail in the next subsection.

If the vectors in the basis set are orthogonal ($(v_i, v_j) = 0$, $i \neq j$) it is very easy to compute the coefficients in the expansion. We simply take the inner product of v_j with both sides of Eq. (5.24):

$$(v_j, \chi) = (v_j, \sum_{i=1}^{n} x_i v_i) = \sum_{i=1}^{n} x_i (v_j, v_i) = x_j (v_j, v_j) . \tag{5.26}$$

Therefore the coefficients of the expansion are given by

$$x_j = \frac{(v_j, \chi)}{(v_j, v_j)} . \tag{5.27}$$

When the vectors in the basis set are not orthogonal, the computation of the coefficients in the vector expansion is more complex. This case is covered in the following subsection.

Reciprocal Basis Vectors

If a vector expansion is required and the basis set is not orthogonal, the reciprocal basis vectors are introduced. These are defined by the following equations:

$$\begin{aligned} (r_i, v_j) &= 0 \qquad i \neq j \\ &= 1 \qquad i = j , \end{aligned} \tag{5.28}$$

Reciprocal Basis Vectors

where the basis vectors are $\{v_1, v_2, \ldots, v_n\}$ and the *reciprocal basis vectors* are $\{r_1, r_2, \ldots, r_n\}$.

If the vectors have been represented by columns of numbers (through vector expansion), and the standard inner product is used

$$(r_i, v_j) = \mathbf{r}_i^T \mathbf{v}_j , \tag{5.29}$$

then Eq. (5.28) can be represented in matrix form as

$$\mathbf{R}^T \mathbf{B} = \mathbf{I} , \tag{5.30}$$

where

$$\mathbf{B} = \left[\mathbf{v}_1 \ \mathbf{v}_2 \ \ldots \ \mathbf{v}_n\right] , \tag{5.31}$$

$$\mathbf{R} = \left[\mathbf{r}_1 \ \mathbf{r}_2 \ \ldots \ \mathbf{r}_n\right] . \tag{5.32}$$

Therefore **R** can be found from

$$\mathbf{R}^T = \mathbf{B}^{-1} , \tag{5.33}$$

and the reciprocal basis vectors can be obtained from the columns of **R**.

Now consider again the vector expansion

$$x = x_1 v_1 + x_2 v_2 + \cdots + x_n v_n . \tag{5.34}$$

Taking the inner product of r_1 with both sides of Eq. (5.34) we obtain

$$(r_1, x) = x_1 (r_1, v_1) + x_2 (r_1, v_2) + \cdots + x_n (r_1, v_n) . \tag{5.35}$$

By definition

$$\begin{aligned} &(r_1, v_2) = (r_1, v_3) = \cdots = (r_1, v_n) = 0 \\ &(r_1, v_1) = 1 . \end{aligned} \tag{5.36}$$

Therefore the first coefficient of the expansion is

$$x_1 = (r_1, x) , \tag{5.37}$$

and in general

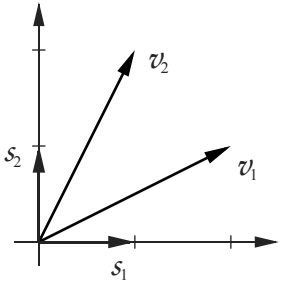
$$x_j = (r_j, x) . \tag{5.38}$$

As an example, consider the two basis vectors

$$\mathbf{v}_1^s = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \ \mathbf{v}_2^s = \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \tag{5.39}$$

Suppose that we want to expand the vector

$$\mathbf{x}^s = \begin{bmatrix} 0 \\ \frac{3}{2} \end{bmatrix} \tag{5.40}$$



in terms of the two basis vectors. (We are using the superscript s to indicate that these columns of numbers represent expansions of the vectors in terms of the standard basis in $\Re^2$. The elements of the standard basis are indicated in the adjacent figure as the vectors s_1 and s_2. We need to use this explicit notation in this example because we will be expanding the vectors in terms of two different basis sets.)

The first step in the vector expansion is to find the reciprocal basis vectors.

$$\mathbf{R}^T = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}^{-1} = \begin{bmatrix} \frac{2}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{2}{3} \end{bmatrix} \qquad \mathbf{r}_1 = \begin{bmatrix} \frac{2}{3} \\ -\frac{1}{3} \end{bmatrix} \qquad \mathbf{r}_2 = \begin{bmatrix} -\frac{1}{3} \\ \frac{2}{3} \end{bmatrix}. \tag{5.41}$$

Now we can find the coefficients in the expansion.

$$x_1^v = \mathbf{r}_1^T \mathbf{x}^s = \begin{bmatrix} \frac{2}{3} & -\frac{1}{3} \end{bmatrix} \begin{bmatrix} 0 \\ \frac{3}{2} \end{bmatrix} = -\frac{1}{2}$$

$$x_2^v = \mathbf{r}_2^T \mathbf{x}^s = \begin{bmatrix} -\frac{1}{3} & \frac{2}{3} \end{bmatrix} \begin{bmatrix} 0 \\ \frac{3}{2} \end{bmatrix} = 1 \tag{5.42}$$

or, in matrix form,

$$\mathbf{x}^v = \mathbf{R}^T \mathbf{x}^s = \mathbf{B}^{-1} \mathbf{x}^s = \begin{bmatrix} \frac{2}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{2}{3} \end{bmatrix} \begin{bmatrix} 0 \\ \frac{3}{2} \end{bmatrix} = \begin{bmatrix} -\frac{1}{2} \\ 1 \end{bmatrix}. \tag{5.43}$$

So that

$$\chi = -\frac{1}{2}v_1 + 1\,v_2, \tag{5.44}$$

as indicated in Figure 5.2.

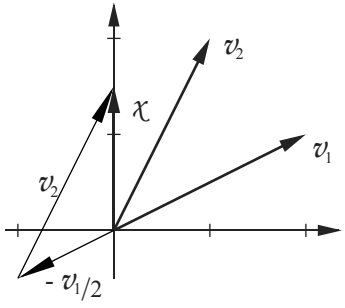


Figure 5.2 Vector Expansion

Note that we now have two different vector expansions for χ, represented by $\mathbf{x}^s$ and $\mathbf{x}^v$. In other words,

$$\chi = 0s_1 + \frac{3}{2}s_2 = -\frac{1}{2}v_1 + 1\,v_2. \tag{5.45}$$

When we represent a general vector as a column of numbers we need to know what basis set was used for the expansion. In this text, unless otherwise stated, assume the standard basis set was used.

Eq. (5.43) shows the relationship between the two different representations of χ, $\mathbf{x}^v = \mathbf{B}^{-1}\mathbf{x}^s$. This operation, called a change of basis, will become very important in later chapters for the performance analysis of certain neural networks.

To experiment with the vector expansion process, use the Neural Network Design Demonstration Reciprocal Basis (`nnd5rb`*).*

Summary of Results

Linear Vector Spaces

Definition. A linear vector space, X, is a set of elements (vectors) defined over a scalar field, F, that satisfies the following conditions:

1. An operation called vector addition is defined such that if $x \in X$ and $y \in X$, then $x + y \in X$.
2. $x + y = y + x$.
3. $(x + y) + z = x + (y + z)$.
4. There is a unique vector $0 \in X$, called the zero vector, such that $x + 0 = x$ for all $x \in X$.
5. For each vector $x \in X$ there is a unique vector in X, to be called $-x$, such that $x + (-x) = 0$.
6. An operation, called multiplication, is defined such that for all scalars $a \in F$, and all vectors $x \in X$, $ax \in X$.
7. For any $x \in X$, $1x = x$ (for scalar 1).
8. For any two scalars $a \in F$ and $b \in F$, and any $x \in X$, $a(bx) = (ab)x$.
9. $(a + b)x = ax + bx$.
10. $a(x + y) = ax + ay$.

Linear Independence

Consider n vectors $\{x_1, x_2, \ldots, x_n\}$. If there exist n scalars $a_1, a_2, \ldots, a_n$, at least one of which is nonzero, such that

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = 0,$$

then the $\{x_i\}$ are linearly dependent.

Spanning a Space

Let X be a linear vector space and let $\{u_1, u_2, \ldots, u_m\}$ be a subset of vectors in X. This subset spans X if and only if for every vector $\chi \in X$ there exist scalars $x_1, x_2, \ldots, x_n$ such that $\chi = x_1 u_1 + x_2 u_2 + \cdots + x_m u_m$.

Inner Product

Any scalar function of χ and y can be defined as an inner product, (χ, y), provided that the following properties are satisfied.

1. $(\chi, y) = (y, \chi)$.
2. $(\chi, a y_1 + b y_2) = a(\chi, y_1) + b(\chi, y_2)$.
3. $(\chi, \chi) \geq 0$, where equality holds if and only if χ is the zero vector.

Norm

A scalar function $\|\chi\|$ is called a norm if it satisfies the following properties:

1. $\|\chi\| \geq 0$.
2. $\|\chi\| = 0$ if and only if $\chi = 0$.
3. $\|a\chi\| = |a|\|\chi\|$ for scalar a.
4. $\|\chi + y\| \leq \|\chi\| + \|y\|$.

Angle

The angle θ between two vectors χ and y is defined by

$$\cos\theta = \frac{(\chi, y)}{\|\chi\|\|y\|}.$$

Orthogonality

Two vectors $\chi, y \in X$ are said to be orthogonal if $(\chi, y) = 0$.

Gram-Schmidt Orthogonalization

Assume that we have n independent vectors $y_1, y_2, \ldots, y_n$. From these vectors we will obtain n orthogonal vectors $v_1, v_2, \ldots, v_n$.

$$v_1 = y_1$$

$$v_k = y_k - \sum_{i=1}^{k-1} \frac{(v_i, y_k)}{(v_i, v_i)} v_i ,$$

where

$$\frac{(v_i, y_k)}{(v_i, v_i)} v_i$$

is the projection of y_k on v_i.

Vector Expansions

$$\chi = \sum_{i=1}^{n} x_i v_i = x_1 v_1 + x_2 v_2 + \cdots + x_n v_n .$$

For orthogonal vectors,

$$x_j = \frac{(v_j, \chi)}{(v_j, v_j)}$$

Reciprocal Basis Vectors

$$(r_i, v_j) = 0 \qquad i \neq j$$

$$= 1 \qquad i = j$$

$$x_j = (r_j, \chi) .$$

To compute the reciprocal basis vectors:

$$\mathbf{B} = \left[\mathbf{v}_1 \ \mathbf{v}_2 \ \ldots \ \mathbf{v}_n\right] ,$$

$$\mathbf{R} = \left[\mathbf{r}_1 \ \mathbf{r}_2 \ \ldots \ \mathbf{r}_n\right] ,$$

$$\mathbf{R}^T = \mathbf{B}^{-1} .$$

In matrix form:

$$\mathbf{x}^v = \mathbf{B}^{-1} \mathbf{x}^s .$$

Solved Problems

P5.1 Consider the single-neuron perceptron network shown in Figure P5.1. Recall from Chapter 3 (see Eq. (3.6)) that the decision boundary for this network is given by $\mathbf{W}\mathbf{p} + b = 0$. Show that the decision boundary is a vector space if $b = 0$.

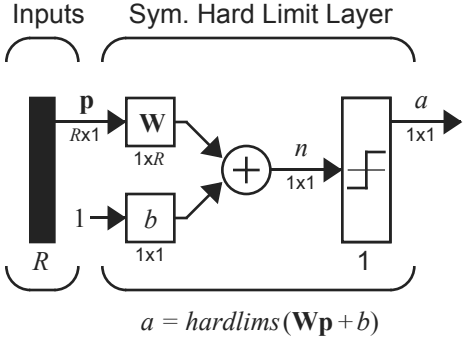


Figure P5.1 Single-Neuron Perceptron

To be a vector space the boundary must satisfy the ten conditions given at the beginning of this chapter. Condition 1 requires that when we add two vectors together the sum remains in the vector space. Let $\mathbf{p}_1$ and $\mathbf{p}_2$ be two vectors on the decision boundary. To be on the boundary they must satisfy

$$\mathbf{W}\mathbf{p}_1 = 0 \qquad \mathbf{W}\mathbf{p}_2 = 0 .$$

If we add these two equations together we find

$$\mathbf{W}(\mathbf{p}_1 + \mathbf{p}_2) = 0 .$$

Therefore the sum is also on the decision boundary.

Conditions 2 and 3 are clearly satisfied. Condition 4 requires that the zero vector be on the boundary. Since $\mathbf{W0} = 0$, the zero vector is on the decision boundary. Condition 5 implies that if $\mathbf{p}$ is on the boundary, then $-\mathbf{p}$ must also be on the boundary. If $\mathbf{p}$ is on the boundary, then

$$\mathbf{W}\mathbf{p} = 0 .$$

If we multiply both sides of this equation by -1 we find

$$\mathbf{W}(-\mathbf{p}) = 0 .$$

Therefore condition 5 is satisfied.

Condition 6 will be satisfied if for any $\mathbf{p}$ on the boundary $a\mathbf{p}$ is also on the boundary. This can be shown in the same way as condition 5. Just multiply both sides of the equation by a instead of by 1.

$$\mathbf{W}(a\mathbf{p}) = 0$$

Conditions 7 through 10 are clearly satisfied. Therefore the perceptron decision boundary is a vector space.

P5.2 Show that the set Y of nonnegative ($f(t) \geq 0$) continuous functions is not a vector space.

This set violates several of the conditions required of a vector space. For example, there are no negative vectors, so condition 5 cannot be satisfied. Also, consider condition 6. The function $f(t) = |t|$ is a member of Y. Let $a = -2$. Then

$$af(2) = -2|2| = -4 < 0 .$$

Therefore $af(t)$ is not a member of Y, and condition 6 is not satisfied.

P5.3 Which of the following sets of vectors are independent? Find the dimension of the vector space spanned by each set.

i. $\begin{bmatrix}1\\1\\1\end{bmatrix} \quad \begin{bmatrix}1\\0\\1\end{bmatrix} \quad \begin{bmatrix}1\\2\\1\end{bmatrix}$

ii. $\sin t \qquad \cos t \qquad 2\cos\left(t + \frac{\pi}{4}\right)$

iii. $\begin{bmatrix}1\\1\\1\\1\end{bmatrix} \quad \begin{bmatrix}1\\0\\1\\1\end{bmatrix} \quad \begin{bmatrix}1\\2\\1\\1\end{bmatrix}$

i. We can solve this problem several ways. First, let's assume that the vectors are dependent. Then we can write

$$a_1\begin{bmatrix}1\\1\\1\end{bmatrix} + a_2\begin{bmatrix}1\\0\\1\end{bmatrix} + a_3\begin{bmatrix}1\\2\\1\end{bmatrix} = \begin{bmatrix}0\\0\\0\end{bmatrix} .$$

If we can solve for the coefficients and they are not all zero, then the vectors are dependent. By inspection we can see that if we let $a_1 = 2$, $a_2 = -1$ and $a_3 = -1$, then the equation is satisfied. Therefore the vectors are dependent.

Another approach, when we have n vectors in $\Re^n$, is to write the above equation in matrix form:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 2 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

If the matrix in this equation has an inverse, then the solution will require that all coefficients be zero; therefore the vectors are independent. If the matrix is singular (has no inverse), then a nonzero set of coefficients will work, and the vectors are dependent. The test, then, is to create a matrix using the vectors as columns. If the determinant of the matrix is zero (singular matrix), then the vectors are dependent; otherwise they are independent. Using the Laplace expansion [Brog91] on the first column, the determinant of this matrix is

$$\begin{vmatrix} 1 & 1 & 1 \\ 1 & 0 & 2 \\ 1 & 1 & 1 \end{vmatrix} = 1 \begin{vmatrix} 0 & 2 \\ 1 & 1 \end{vmatrix} + (-1) \begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix} + 1 \begin{vmatrix} 1 & 1 \\ 0 & 2 \end{vmatrix} = -2 + 0 + 2 = 0$$

Therefore the vectors are dependent.

The dimension of the space spanned by the vectors is two, since any two of the vectors can be shown to be independent.

ii. By using some trigonometric identities we can write

$$\cos\left(t + \frac{\pi}{4}\right) = \frac{-1}{\sqrt{2}} \sin t + \frac{1}{\sqrt{2}} \cos t .$$

Therefore the vectors are dependent. The dimension of the space spanned by the vectors is two, since no linear combination of $\sin t$ and $\cos t$ is identically zero.

iii. This is similar to part (i), except that the number of vectors is less than the size of the vector space they are drawn from (three vectors in $\Re^4$). In this case the matrix made up of the vectors will not be square, so we will not be able to compute a determinant. However, we can use something called the Gramian [Brog91]. It is the determinant of a matrix whose i, j element is the inner product of vector i and vector j. The vectors are dependent if and only if the Gramian is zero.

For our problem the Gramian would be

$$G = \begin{vmatrix} (\mathbf{x}_1,\mathbf{x}_1) & (\mathbf{x}_1,\mathbf{x}_2) & (\mathbf{x}_1,\mathbf{x}_3) \\ (\mathbf{x}_2,\mathbf{x}_1) & (\mathbf{x}_2,\mathbf{x}_2) & (\mathbf{x}_2,\mathbf{x}_3) \\ (\mathbf{x}_3,\mathbf{x}_1) & (\mathbf{x}_3,\mathbf{x}_2) & (\mathbf{x}_3,\mathbf{x}_3) \end{vmatrix},$$

where

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \qquad \mathbf{x}_2 = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \qquad \mathbf{x}_3 = \begin{bmatrix} 1 \\ 2 \\ 1 \\ 1 \end{bmatrix}.$$

Therefore

$$G = \begin{vmatrix} 4 & 3 & 5 \\ 3 & 3 & 3 \\ 5 & 3 & 7 \end{vmatrix} = 4\begin{vmatrix} 3 & 3 \\ 3 & 7 \end{vmatrix} + (-3)\begin{vmatrix} 3 & 5 \\ 3 & 7 \end{vmatrix} + 5\begin{vmatrix} 3 & 5 \\ 3 & 3 \end{vmatrix} = 48 - 18 - 30 = 0.$$

We can also show that these vectors are dependent by noting

$$2\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} - 1\begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} - 1\begin{bmatrix} 1 \\ 2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

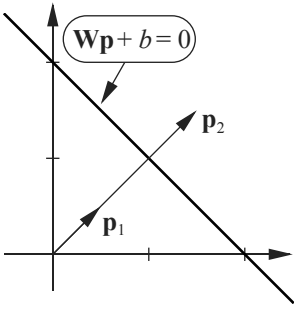
The dimension of the space must therefore be less than 3. We can show that $\mathbf{x}_1$ and $\mathbf{x}_2$ are independent, since

$$G = \begin{vmatrix} 4 & 3 \\ 3 & 3 \end{vmatrix} = 4 \neq 0.$$

Therefore the dimension of the space is 2.

P5.4 Recall from Chapters 3 and 4 that one-layer perceptrons can only be used to recognize patterns that are linearly separable (can be separated by a linear boundary — see Figure 3.3). If two patterns are linearly separable, are they always linearly independent?

No, these are two unrelated concepts. Take the following simple example. Consider the two input perceptron shown in Figure P5.2.



Suppose that we want to separate the two vectors

$$\mathbf{p}_1 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \qquad \mathbf{p}_2 = \begin{bmatrix} 1.5 \\ 1.5 \end{bmatrix}.$$

If we choose the weights and offsets to be $w_{11} = 1$, $w_{12} = 1$ and $b = -2$, then the decision boundary ($\mathbf{Wp} + b = 0$) is shown in the figure to the left. Clearly these two vectors are linearly separable. However, they are not linearly independent since $\mathbf{p}_2 = 3\mathbf{p}_1$.

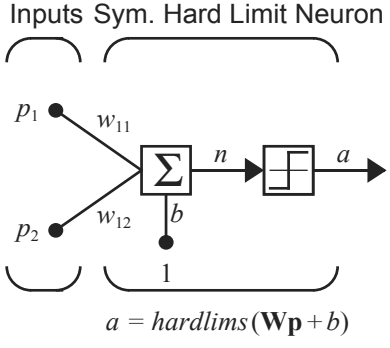


Figure P5.2 Two-Input Perceptron

P5.5 Using the following basis vectors, find an orthogonal set using Gram-Schmidt orthogonalization.

$$\mathbf{y}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \qquad \mathbf{y}_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{y}_3 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Step 1.

$$\mathbf{v}_1 = \mathbf{y}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Step 2.

$$\mathbf{v}_2 = \mathbf{y}_2 - \frac{\mathbf{v}_1^T\mathbf{y}_2}{\mathbf{v}_1^T\mathbf{v}_1}\mathbf{v}_1 = \begin{bmatrix}1\\0\\0\end{bmatrix} - \frac{\begin{bmatrix}1 & 1 & 1\end{bmatrix}\begin{bmatrix}1\\0\\0\end{bmatrix}}{\begin{bmatrix}1 & 1 & 1\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix}}\begin{bmatrix}1\\1\\1\end{bmatrix} = \begin{bmatrix}1\\0\\0\end{bmatrix} - \begin{bmatrix}1/3\\1/3\\1/3\end{bmatrix} = \begin{bmatrix}2/3\\-1/3\\-1/3\end{bmatrix}$$

Step 3.

$$\mathbf{v}_3 = \mathbf{y}_3 - \frac{\mathbf{v}_1^T\mathbf{y}_3}{\mathbf{v}_1^T\mathbf{v}_1}\mathbf{v}_1 - \frac{\mathbf{v}_2^T\mathbf{y}_3}{\mathbf{v}_2^T\mathbf{v}_2}\mathbf{v}_2$$

$$\mathbf{v}_3 = \begin{bmatrix}0\\1\\0\end{bmatrix} - \frac{\begin{bmatrix}1 & 1 & 1\end{bmatrix}\begin{bmatrix}0\\1\\0\end{bmatrix}}{\begin{bmatrix}1 & 1 & 1\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix}}\begin{bmatrix}1\\1\\1\end{bmatrix} - \frac{\begin{bmatrix}2/3 & -1/3 & -1/3\end{bmatrix}\begin{bmatrix}0\\1\\0\end{bmatrix}}{\begin{bmatrix}2/3 & -1/3 & -1/3\end{bmatrix}\begin{bmatrix}2/3\\-1/3\\-1/3\end{bmatrix}}\begin{bmatrix}2/3\\-1/3\\-1/3\end{bmatrix}$$

$$\mathbf{v}_3 = \begin{bmatrix}0\\1\\0\end{bmatrix} - \begin{bmatrix}1/3\\1/3\\1/3\end{bmatrix} - \begin{bmatrix}-1/3\\1/6\\1/6\end{bmatrix} = \begin{bmatrix}0\\1/2\\-1/2\end{bmatrix}$$

P5.6 Consider the vector space of all polynomials defined on the interval [-1, 1]. Show that $(x,y) = \int_{-1}^{1} x(t)y(t)dt$ is a valid inner product.

An inner product must satisfy the following properties.

1. $(x,y) = (y,x)$

$$(x,y) = \int_{-1}^{1} x(t)y(t)dt = \int_{-1}^{1} y(t)x(t)dt = (y,x)$$

2. $(x,ay_1 + by_2) = a(x,y_1) + b(x,y_2)$

$$(\chi, a y_1 + b y_2) = \int_{-1}^{1} \chi(t)(a y_1(t) + b y_2(t))dt = a\int_{-1}^{1} \chi(t) y_1(t)dt + b\int_{-1}^{1} \chi(t) y_2(t)dt$$
$$= a(\chi, y_1) + b(\chi, y_2)$$

3. $(\chi, \chi) \geq 0$, where equality holds if and only if χ is the zero vector.

$$(\chi, \chi) = \int_{-1}^{1} \chi(t)\chi(t)dt = \int_{-1}^{1} \chi^2(t)\ dt \geq 0$$

Equality holds here only if $\chi(t) = 0$ for $-1 \leq t \leq 1$, which is the zero vector.

P5.7 Two vectors from the vector space described in the previous problem (polynomials defined on the interval [-1, 1]) are $1 + t$ and $1 - t$. Find an orthogonal set of vectors based on these two vectors.

Step 1.

$$v_1 = y_1 = 1 + t$$

Step 2.

$$v_2 = y_2 - \frac{(v_1, y_2)}{(v_1, v_1)} v_1$$

where

$$(v_1, y_2) = \int_{-1}^{1} (1 + t)(1 - t)dt = \left(t - \frac{t^3}{3}\right)\Bigg|_{-1}^{1} = \left(\frac{2}{3}\right) - \left(-\frac{2}{3}\right) = \frac{4}{3}$$

$$(v_1, v_1) = \int_{-1}^{1} (1 + t)^2 dt = \frac{(1 + t)^3}{3}\Bigg|_{-1}^{1} = \left(\frac{8}{3}\right) - (0) = \frac{8}{3}.$$

Therefore

$$v_2 = (1 - t) - \frac{4/3}{8/3}(1 + t) = \frac{1}{2} - \frac{3}{2}t.$$

P5.8 Expand $\mathbf{x} = \begin{bmatrix} 6 & 9 & 9 \end{bmatrix}^T$ in terms of the following basis set.

$$\mathbf{v}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \qquad \mathbf{v}_2 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \qquad \mathbf{v}_3 = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}$$

The first step is to calculate the reciprocal basis vectors.

$$\mathbf{B} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 2 \end{bmatrix} \qquad \mathbf{B}^{-1} = \begin{bmatrix} \frac{5}{3} & -\frac{1}{3} & -\frac{1}{3} \\ -\frac{1}{3} & -\frac{1}{3} & \frac{2}{3} \\ -\frac{1}{3} & \frac{2}{3} & -\frac{1}{3} \end{bmatrix}$$

Therefore taking the rows of $\mathbf{B}^{-1}$,

$$\mathbf{r}_1 = \begin{bmatrix} 5/3 \\ -1/3 \\ -1/3 \end{bmatrix} \qquad \mathbf{r}_2 = \begin{bmatrix} -1/3 \\ -1/3 \\ 2/3 \end{bmatrix} \qquad \mathbf{r}_3 = \begin{bmatrix} -1/3 \\ 2/3 \\ -1/3 \end{bmatrix}.$$

The coefficients in the expansion are calculated

$$x_1^v = \mathbf{r}_1^T\mathbf{x} = \begin{bmatrix} \frac{5}{3} & \frac{-1}{3} & \frac{-1}{3} \end{bmatrix} \begin{bmatrix} 6 \\ 9 \\ 9 \end{bmatrix} = 4$$

$$x_2^v = \mathbf{r}_2^T\mathbf{x} = \begin{bmatrix} \frac{-1}{3} & \frac{-1}{3} & \frac{2}{3} \end{bmatrix} \begin{bmatrix} 6 \\ 9 \\ 9 \end{bmatrix} = 1$$

$$x_3^v = \mathbf{r}_3^T\mathbf{x} = \begin{bmatrix} \frac{-1}{3} & \frac{2}{3} & \frac{-1}{3} \end{bmatrix} \begin{bmatrix} 6 \\ 9 \\ 9 \end{bmatrix} = 1$$

,

and the expansion is written

$$\mathbf{x} = x_1^v \mathbf{v}_1 + x_2^v \mathbf{v}_2 + x_3^v \mathbf{v}_3 = 4\begin{bmatrix}1\\1\\1\end{bmatrix} + 1\begin{bmatrix}1\\2\\3\end{bmatrix} + 1\begin{bmatrix}1\\3\\2\end{bmatrix}.$$

We can represent the process in matrix form:

$$\mathbf{x}^v = \mathbf{B}^{-1}\mathbf{x} = \begin{bmatrix}\frac{5}{3} & -\frac{1}{3} & -\frac{1}{3}\\ -\frac{1}{3} & -\frac{1}{3} & \frac{2}{3}\\ -\frac{1}{3} & \frac{2}{3} & -\frac{1}{3}\end{bmatrix}\begin{bmatrix}6\\9\\9\end{bmatrix} = \begin{bmatrix}4\\1\\1\end{bmatrix}.$$

Recall that both $\mathbf{x}^v$ and $\mathbf{x}$ are representations of the same vector, but are expanded in terms of different basis sets. (It is assumed that $\mathbf{x}$ uses the standard basis set, unless otherwise indicated.)

Epilogue

This chapter has presented a few of the basic concepts of vector spaces, material that is critical to the understanding of how neural networks work. This subject of vector spaces is very large, and we have made no attempt to cover all its aspects. Instead, we have presented those concepts that we feel are most relevant to neural networks. The topics covered here will be revisited in almost every chapter that follows.

The next chapter will continue our investigation of the topics of linear algebra most relevant to neural networks. There we will concentrate on linear transformations and matrices.

Further Reading

[Brog91] W. L. Brogan, *Modern Control Theory*, 3rd Ed., Englewood Cliffs, NJ: Prentice-Hall, 1991.

This is a well-written book on the subject of linear systems. The first half of the book is devoted to linear algebra. It also has good sections on the solution of linear differential equations and the stability of linear and nonlinear systems. It has many worked problems.

[Stra76] G. Strang, *Linear Algebra and Its Applications*, New York: Academic Press, 1980.

Strang has written a good basic text on linear algebra. Many applications of linear algebra are integrated into the text.

Exercises

E5.1 Consider again the perceptron described in Problem P5.1. If $b \neq 0$, show that the decision boundary is not a vector space.

E5.2 What is the dimension of the vector space described in Problem P5.1?

E5.3 Consider the set of all continuous functions that satisfy the condition $f(0) = 0$. Show that this is a vector space.

E5.4 Show that the set of 2×2 matrices is a vector space.

E5.5 Consider a perceptron network, with the following weights and bias.

$$\mathbf{W} = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}, \; b = 0.$$

i. Write out the equation for the decision boundary.

ii. Show that the decision boundary is a vector space. (Demonstrate that the 10 criteria are satisfied for any point on the boundary.)

iii. What is the dimension of the vector space?

iv. Find a basis set for the vector space.

E5.6 The three parts to this question refer to subsets of the set of real-valued continuous functions defined on the interval [0,1]. Tell which of these subsets are vector spaces. If the subset is not a vector space, identify which of the 10 criteria are not satisfied.

i. All functions such that $f(0.5) = 2$.

ii. All functions such that $f(0.75) = 0$.

iii. All functions such that $f(0.5) = -f(0.75) - 3$.

E5.7 The next three questions refer to subsets of the set of real polynomials defined over the real line (e.g., $3 + 2t + 6t^2$). Tell which of these subsets are vector spaces. If the subset is not a vector space, identify which of the 10 criteria are not satisfied.

i. Polynomials of degree 5 or less.

ii. Polynomials that are positive for positive t.

iii. Polynomials that go to zero as t goes to zero.

E5.8 Which of the following sets of vectors are independent? Find the dimension of the vector space spanned by each set. (Verify your answers to parts (i) and (iv) using the MATLAB function `rank`.)

i. $\begin{bmatrix}1\\2\\3\end{bmatrix}$ $\begin{bmatrix}1\\0\\1\end{bmatrix}$ $\begin{bmatrix}1\\2\\1\end{bmatrix}$

ii. $\sin t$ $\cos t$ $\cos(2t)$

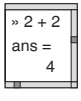
iii. $1+t$ $1-t$

iv. $\begin{bmatrix}1\\2\\2\\1\end{bmatrix}$ $\begin{bmatrix}1\\0\\0\\1\end{bmatrix}$ $\begin{bmatrix}3\\4\\4\\3\end{bmatrix}$

E5.9 Recall the apple and orange pattern recognition problem of Chapter 3. Find the angles between each of the prototype patterns (*orange* and *apple*) and the test input pattern (*oblong orange*). Verify that the angles make intuitive sense.

$$\mathbf{p}_1 = \begin{bmatrix}1\\-1\\-1\end{bmatrix}(orange) \qquad \mathbf{p}_2 = \begin{bmatrix}1\\1\\-1\end{bmatrix}(apple) \qquad \mathbf{p} = \begin{bmatrix}-1\\-1\\-1\end{bmatrix}$$

E5.10 Using the following basis vectors, find an orthogonal set using Gram-Schmidt orthogonalization. (Check your answer using MATLAB.)



$$\mathbf{y}_1 = \begin{bmatrix}1\\0\\0\end{bmatrix} \qquad \mathbf{y}_2 = \begin{bmatrix}1\\1\\0\end{bmatrix} \qquad \mathbf{y}_3 = \begin{bmatrix}1\\1\\1\end{bmatrix}$$

E5.11 Consider the vector space of all piecewise continuous functions on the interval [0, 1]. The set $\{f_1, f_2, f_3\}$, which is defined in Figure E15.1, contains three vectors from this vector space.

i. Show that this set is linearly independent.

ii. Generate an orthogonal set using the Gram-Schmidt procedure. The inner product is defined to be

$$(f,g) = \int_0^1 f(t)g(t)dt .$$

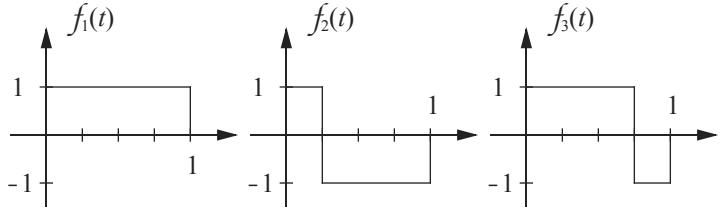


Figure E15.1 Basis Set for Exercise E5.11

E5.12 Consider the vector space of all piece wise continuous functions on the interval [0,1]. The set $\{f_1, f_2\}$, which is defined in Figure E15.2, contains two vectors from this vector space.

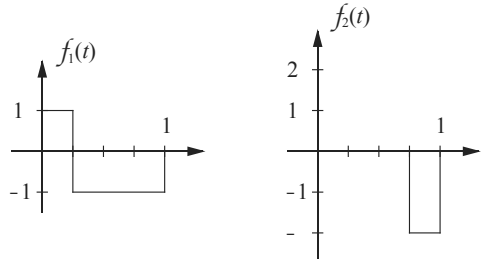


Figure E15.2 Basis Set for Exercise E5.12

i. Generate an orthogonal set using the Gram-Schmidt procedure. The inner product is defined to be

$$(f,g) = \int_0^1 f(t)g(t)dt .$$

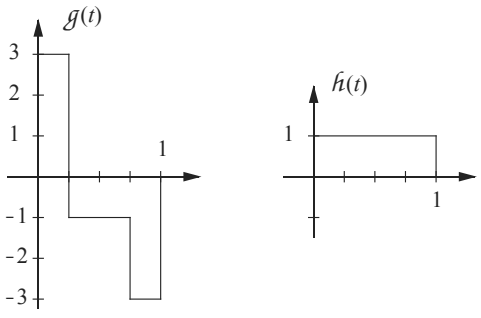


Figure E15.3 Vectors vectors g and h for Exercise E5.12 part ii.

ii. Expand the vectors g and h in Figure E15.3 in terms of the orthogonal set you created in Part 1. Explain any problems you find.

E5.13 Consider the set of polynomials of degree 1 or less. This is a linear vector space. One basis set for this space is

$$\{u_1 = 1, u_2 = t\}$$

Using this basis set, the polynomial $y = 2 + 4t$ can be represented as

$$\mathbf{y}^u = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

Consider the new basis set

$$\{v_1 = 1 + t, v_2 = 1 - t\}$$

Use reciprocal basis vectors to find the representation of y in terms of this new basis set.

E5.14 A vector x can be expanded in terms of the basis vectors $\{v_1, v_2\}$ as

$$x = 1\,v_1 + 1\,v_2$$

The vectors v_1 and v_2 can be expanded in terms of the basis vectors $\{s_1, s_2\}$ as

$$v_1 = 1\,s_1 - 1\,s_2$$
$$v_2 = 1\,s_1 + 1\,s_2$$

i. Find the expansion for x in terms of the basis vectors $\{s_1, s_2\}$.

ii. A vector y can be expanded in terms of the basis vectors $\{s_1, s_2\}$ as

$$y = 1s_1 + 1s_2 .$$

Find the expansion of y in terms of the basis vectors $\{v_1, v_2\}$.

E5.15 Consider the vector space of all continuous functions on the interval [0,1]. The set $\{f_1, f_2\}$, which is defined in the figure below, contains two vectors from this vector space.

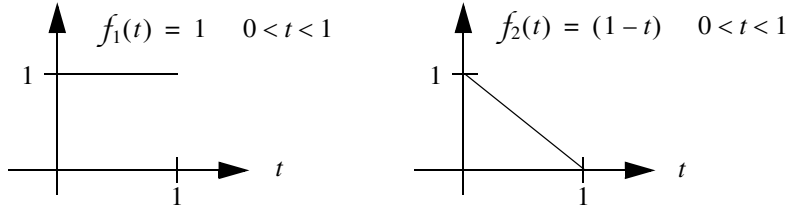


Figure E15.4 Independent Vectors for Exercise E5.15

i. From these two vectors, generate an orthogonal set $\{g_1, g_2\}$ using the Gram-Schmidt procedure. The inner product is defined to be

$$(f,g) = \int_0^1 f(t)g(t)dt .$$

Plot the two orthogonal vectors g_1 and g_2 as functions of time.

ii. Expand the following vector h in terms of the orthogonal set you created in part i., using Eq. (5.27). Demonstrate that the expansion is correct by reproducing h as a combination of g_1 and g_2 .

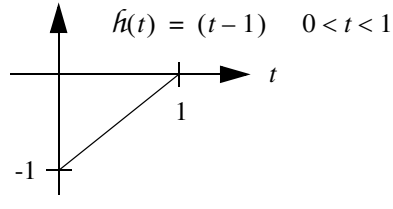


Figure E15.5 Vector h for Exercise E5.15

E5.16 Consider the set of all complex numbers. This can be considered a vector space, because it satisfies the ten defining properties. We can also define

an inner product for this vector space $(\chi, y) = Re(\chi)Re(y) + Im(\chi)Im(y)$, where $Re(\chi)$ is the real part of χ, and $Im(\chi)$ is the imaginary part of χ. This leads to the following definition for norm: $\|\chi\| = \sqrt{(\chi, y)}$.

i. Consider the following basis set for the vector space described above: $v_1 = 1 + 2j$, $v_2 = 2 + j$. Using the Gram-Schmidt method, find an orthogonal basis set.

ii. Using your orthogonal basis set from part i., find vector expansions for $u_1 = 1 - j$, $u_2 = 1 + j$, and $\chi = 3 + j$. This will allow you to write χ, u_1, and u_2 as a columns of numbers $\mathbf{x}$, $\mathbf{u}_1$ and $\mathbf{u}_2$.

iii. We now want to represent the vector χ using the basis set $\{u_1, u_2\}$. Use reciprocal basis vectors to find the expansion for χ in terms of the basis vectors $\{u_1, u_2\}$. This will allow you to write χ as a new column of numbers $\mathbf{x}^u$.

iv. Show that the representations for χ that you found in parts ii. and iii. are equivalent (the two columns of numbers $\mathbf{x}$ and $\mathbf{x}^u$ both represent the same vector χ).

E5.17 Consider the vectors defined in Figure E15.6. The set $\{s_1, s_2\}$ is the standard basis set. The set $\{u_1, u_2\}$ is an alternate basis set. The vector χ is a vector that we wish to represent with respect to the two basis sets.

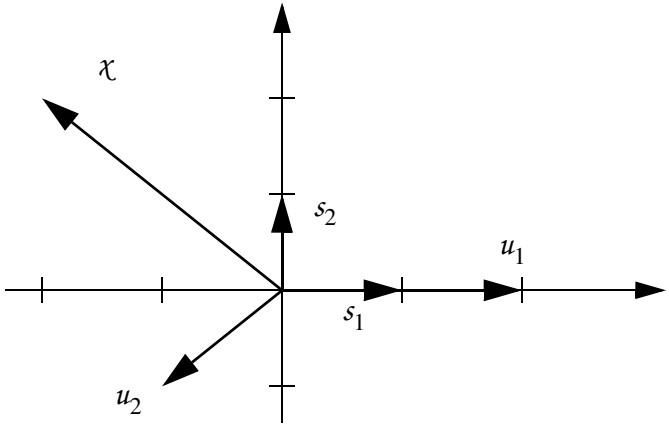


Figure E15.6 Vector Definitions for Exercise E5.17

i. Write the expansion for χ in terms of the standard basis $\{s_1, s_2\}$.

ii. Write the expansions for u_1 and u_2 in terms of the standard basis $\{s_1, s_2\}$

iii. Using reciprocal basis vectors, write the expansion for χ in terms of the basis $\{u_1, u_2\}$.

iv. Draw sketches, similar to Figure 5.2, that demonstrate that the expansions of part i. and part iii. are equivalent.

E5.18 Consider the set of all functions that can be written in the form $A\sin(t+\theta)$. This set can be considered a vector space, because it satisfies the ten defining properties.

i. Consider the following basis set for the vector space described above: $v_1 = \sin(t)$, $v_2 = \cos(t)$. Represent the vector $\chi = 2\sin(t) + 4\cos(t)$ as a column of numbers $\mathbf{x}^v$ (find the vector expansion), using this basis set.

ii. Using your basis set from part i., find vector expansions for $u_1 = 2\sin(t) + \cos(t)$, $u_2 = 3\sin(t)$.

iii. We now want to represent the vector χ of part i., using the basis set $\{u_1, u_2\}$. Use reciprocal basis vectors to find the expansion for χ in terms of the basis vectors $\{u_1, u_2\}$. This will allow you to write χ as a new column of numbers $\mathbf{x}^u$.

iv. Show that the representations for χ that you found in parts i. and iii. are equivalent (the two columns of numbers $\mathbf{x}^v$ and $\mathbf{x}^u$ both represent the same vector χ).

E5.19 Suppose that we have three vectors: $\chi, y, z \in X$. We want to add some multiple of y to χ, so that the resulting vector is orthogonal to z.

i. How would you determine the appropriate multiple of y to add to χ?

ii. Verify your results in part i. using the following vectors.

$$\mathbf{x} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} 1 \\ 0.5 \end{bmatrix} \qquad \mathbf{z} = \begin{bmatrix} 0.5 \\ 1 \end{bmatrix}$$

iii. Use a sketch to illustrate your results from part ii.

E5.20 Expand $\mathbf{x} = \begin{bmatrix} 1 & 2 & 2 \end{bmatrix}^T$ in terms of the following basis set. (Verify your answer using MATLAB.)

```
» 2 + 2
ans =
     4
```

$$\mathbf{v}_1 = \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix} \qquad \mathbf{v}_2 = \begin{bmatrix} 1 \\ 1 \\ -2 \end{bmatrix} \qquad \mathbf{v}_3 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

E5.21 Find the value of a that makes $\|x - ay\|$ a minimum. (Use $\|x\| = (x,x)^{1/2}$.) Show that for this value of a the vector $z = x - ay$ is orthogonal to y and that

$$\|x - ay\|^2 + \|ay\|^2 = \|x\|^2 .$$

(The vector ay is the projection of x on y.) Draw a diagram for the case where x and y are two-dimensional. Explain how this concept is related to Gram-Schmidt orthogonalization.

6 Linear Transformations for Neural Networks

Objectives

This chapter will continue the work of Chapter 5 in laying out the mathematical foundations for our analysis of neural networks. In Chapter 5 we reviewed vector spaces; in this chapter we investigate linear transformations as they apply to neural networks.

As we have seen in previous chapters, the multiplication of an input vector by a weight matrix is one of the key operations that is performed by neural networks. This operation is an example of a linear transformation. We want to investigate general linear transformations and determine their fundamental characteristics. The concepts covered in this chapter, such as eigenvalues, eigenvectors and change of basis, will be critical to our understanding of such key neural network topics as performance learning (including the Widrow-Hoff rule and backpropagation) and Hopfield network convergence.

Theory and Examples

Recall the Hopfield network that was discussed in Chapter 3. (See Figure 6.1.) The output of the network is updated synchronously according to the equation

$$\mathbf{a}(t+1) = satlin(\mathbf{W}\mathbf{a}(t) + \mathbf{b}) . \tag{6.1}$$

Notice that at each iteration the output of the network is again multiplied by the weight matrix **W**. What is the effect of this repeated operation? Can we determine whether or not the output of the network will converge to some steady state value, go to infinity, or oscillate? In this chapter we will lay the foundation for answering these questions, along with many other questions about neural networks discussed in this book.

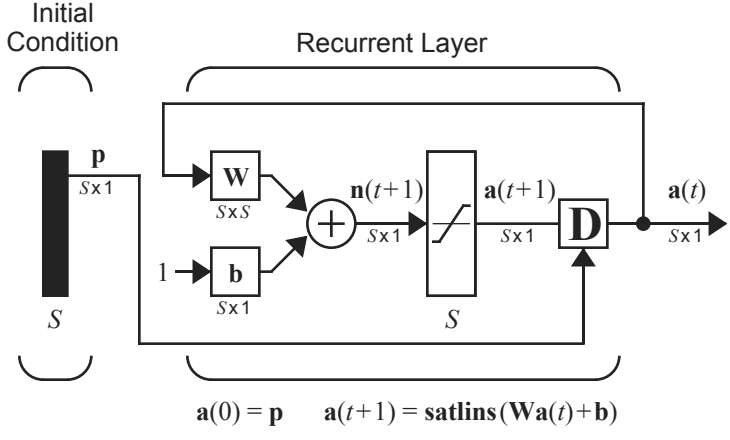


Figure 6.1 Hopfield Network

Linear Transformations

We begin with some general definitions.

Transformation

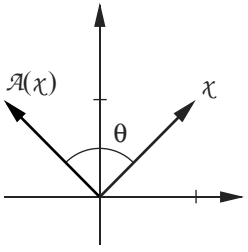
A *transformation* consists of three parts:

1. a set of elements $X = \{x_i\}$, called the domain,
2. a set of elements $Y = \{y_i\}$, called the range, and
3. a rule relating each $x_i \in X$ to an element $y_i \in Y$.

Linear Transformation

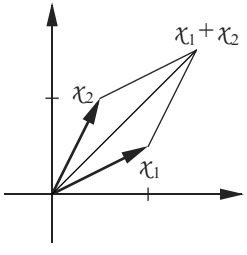
A transformation $\mathcal{A}$ is *linear* if:

1. for all $\chi_1, \chi_2 \in X$, $\mathcal{A}(\chi_1 + \chi_2) = \mathcal{A}(\chi_1) + \mathcal{A}(\chi_2)$,
2. for all $\chi \in X$, $a \in R$, $\mathcal{A}(a\chi) = a\mathcal{A}(\chi)$.



Consider, for example, the transformation obtained by rotating vectors in $\Re^2$ by an angle θ, as shown in the figure to the left. The next two figures illustrate that property 1 is satisfied for rotation. They show that if you want to rotate a sum of two vectors, you can rotate each vector first and then sum them. The fourth figure illustrates property 2. If you want to rotate a scaled vector, you can rotate it first and then scale it. Therefore rotation is a linear operation.

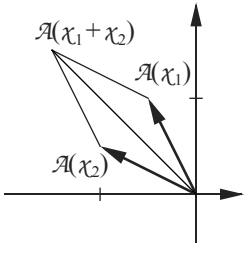
Matrix Representations



As we mentioned at the beginning of this chapter, matrix multiplication is an example of a linear transformation. We can also show that any linear transformation between two finite-dimensional vector spaces can be represented by a matrix (just as in the last chapter we showed that any general vector in a finite-dimensional vector space can be represented by a column of numbers). To show this we will use most of the concepts covered in the previous chapter.

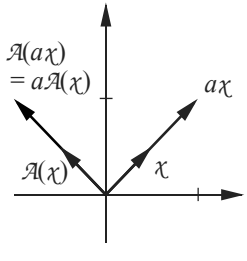
Let $\{v_1, v_2, \ldots, v_n\}$ be a basis for vector space X, and let $\{u_1, u_2, \ldots, u_m\}$ be a basis for vector space Y. This means that for any two vectors $\chi \in X$ and $y \in Y$

$$\chi = \sum_{i=1}^{n} x_i v_i \text{ and } y = \sum_{i=1}^{m} y_i u_i . \tag{6.2}$$



Let $\mathcal{A}$ be a linear transformation with domain X and range Y ($\mathcal{A}: X \to Y$). Then

$$\mathcal{A}(\chi) = y \tag{6.3}$$



can be written

$$\mathcal{A}\left(\sum_{j=1}^{n} x_j v_j\right) = \sum_{i=1}^{m} y_i u_i . \tag{6.4}$$

Since $\mathcal{A}$ is a linear operator, Eq. (6.4) can be written

$$\sum_{j=1}^{n} x_j \mathcal{A}(v_j) = \sum_{i=1}^{m} y_i u_i . \tag{6.5}$$

Since the vectors $\mathcal{A}(v_j)$ are elements of Y, they can be written as linear combinations of the basis vectors for Y:

$$\mathcal{A}(v_j) = \sum_{i=1}^{m} a_{ij} u_i . \tag{6.6}$$

(Note that the notation used for the coefficients of this expansion, a_{ij}, was not chosen by accident.) If we substitute Eq. (6.6) into Eq. (6.5) we obtain

$$\sum_{j=1}^{n} x_j \sum_{i=1}^{m} a_{ij} u_i = \sum_{i=1}^{m} y_i u_i . \tag{6.7}$$

The order of the summations can be reversed, to produce

$$\sum_{i=1}^{m} u_i \sum_{j=1}^{n} a_{ij} x_j = \sum_{i=1}^{m} y_i u_i . \tag{6.8}$$

This equation can be rearranged, to obtain

$$\sum_{i=1}^{m} u_i \left(\sum_{j=1}^{n} a_{ij} x_j - y_i \right) = 0. \tag{6.9}$$

Recall that since the u_i form a basis set they must be independent. This means that each coefficient that multiplies u_i in Eq. (6.9) must be identically zero (see Eq. (5.4)), therefore

$$\sum_{j=1}^{n} a_{ij} x_j = y_i . \tag{6.10}$$

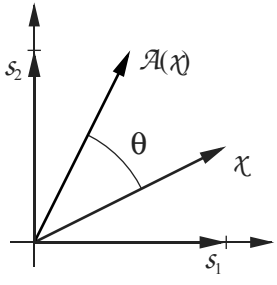
This is just matrix multiplication, as in

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \tag{6.11}$$

We can summarize these results: *For any linear transformation between two finite-dimensional vector spaces there is a matrix representation. When we multiply the matrix times the vector expansion for the domain vector x, we obtain the vector expansion for the transformed vector y.*

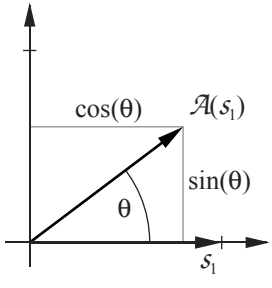
Keep in mind that the matrix representation is not unique (just as the representation of a general vector by a column of numbers is not unique — see Chapter 5). If we change the basis set for the domain or for the range, the matrix representation will also change. We will use this fact to our advantage in later chapters.

As an example of a matrix representation, consider the rotation transformation. Let's find a matrix representation for that transformation. The key step is given in Eq. (6.6). We must transform each basis vector for the domain and then expand it in terms of the basis vectors of the range. In this example the domain and the range are the same ($X = Y = \Re^2$), so to keep things simple we will use the standard basis for both ($u_i = v_i = s_i$), as shown in the adjacent figure.



The first step is to transform the first basis vector and expand the resulting transformed vector in terms of the basis vectors. If we rotate s_1 counterclockwise by the angle θ we obtain

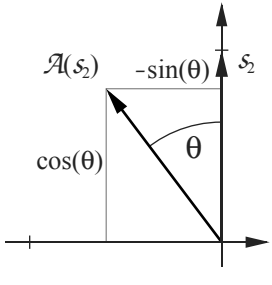
$$\mathcal{A}(s_1) = \cos(\theta)s_1 + \sin(\theta)s_2 = \sum_{i=1}^{2} a_{i1}s_i = a_{11}s_1 + a_{21}s_2, \qquad (6.12)$$



as can be seen in the middle left figure. The two coefficients in this expansion make up the first column of the matrix representation.

The next step is to transform the second basis vector. If we rotate s_2 counterclockwise by the angle θ we obtain

$$\mathcal{A}(s_2) = -\sin(\theta)s_1 + \cos(\theta)s_2 = \sum_{i=1}^{2} a_{i2}s_i = a_{12}s_1 + a_{22}s_2, \qquad (6.13)$$



as can be seen in the lower left figure. From this expansion we obtain the second column of the matrix representation. The complete matrix representation is thus given by

$$\mathbf{A} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}. \qquad (6.14)$$

Verify for yourself that when you multiply a vector by the matrix of Eq. (6.14), the vector is rotated by an angle θ.

In summary, to obtain the matrix representation of a transformation we use Eq. (6.6). We transform each basis vector for the domain and expand it in terms of the basis vectors of the range. The coefficients of each expansion produce one column of the matrix.

To graphically investigate the process of creating a matrix representation, use the Neural Network Design Demonstration Linear Transformations (`nnd6lt`).

Change of Basis

We notice from the previous section that the matrix representation of a linear transformation is not unique. The representation will depend on what basis sets are used for the domain and the range of the transformation. In this section we will illustrate exactly how a matrix representation changes as the basis sets are changed.

Consider a linear transformation $\mathcal{A}:X \rightarrow Y$. Let $\{v_1, v_2, \ldots, v_n\}$ be a basis for vector space X, and let $\{u_1, u_2, \ldots, u_m\}$ be a basis for vector space Y. Therefore, any vector $\chi \in X$ can be written

$$\chi = \sum_{i=1}^{n} x_i v_i , \tag{6.15}$$

and any vector $y \in Y$ can be written

$$y = \sum_{i=1}^{m} y_i u_i . \tag{6.16}$$

So if

$$\mathcal{A}(\chi) = y \tag{6.17}$$

the matrix representation will be

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}, \tag{6.18}$$

or

$$\mathbf{A}\mathbf{x} = \mathbf{y} . \tag{6.19}$$

Now suppose that we use different basis sets for X and Y. Let $\{t_1, t_2, \ldots, t_n\}$ be the new basis for X, and let $\{w_1, w_2, \ldots, w_m\}$ be the new basis for Y. With the new basis sets, the vector $\chi \in X$ is written

$$x = \sum_{i=1}^{n} x'_i t_i , \tag{6.20}$$

and the vector $y \in Y$ is written

$$y = \sum_{i=1}^{m} y'_i w_i . \tag{6.21}$$

This produces a new matrix representation:

$$\begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n} \\ a'_{21} & a'_{22} & \cdots & a'_{2n} \\ \vdots & \vdots & & \vdots \\ a'_{m1} & a'_{m2} & \cdots & a'_{mn} \end{bmatrix} \begin{bmatrix} x'_1 \\ x'_2 \\ \vdots \\ x'_n \end{bmatrix} = \begin{bmatrix} y'_1 \\ y'_2 \\ \vdots \\ y'_m \end{bmatrix} , \tag{6.22}$$

or

$$\mathbf{A}'\mathbf{x}' = \mathbf{y}' . \tag{6.23}$$

What is the relationship between $\mathbf{A}$ and $\mathbf{A}'$? To find out, we need to find the relationship between the two basis sets. First, since each t_i is an element of X, they can be expanded in terms of the original basis for X:

$$t_i = \sum_{j=1}^{n} t_{ji} v_j . \tag{6.24}$$

Next, since each w_i is an element of Y, they can be expanded in terms of the original basis for Y:

$$w_i = \sum_{j=1}^{m} w_{ji} u_j . \tag{6.25}$$

Therefore, the basis vectors can be written as columns of numbers:

$$\mathbf{t}_i = \begin{bmatrix} t_{1i} \\ t_{2i} \\ \vdots \\ t_{ni} \end{bmatrix} \qquad \mathbf{w}_i = \begin{bmatrix} w_{1i} \\ w_{2i} \\ \vdots \\ w_{mi} \end{bmatrix} . \tag{6.26}$$

Define a matrix whose columns are the $\mathbf{t}_i$:

$$\mathbf{B}_t = \left[\mathbf{t}_1\ \mathbf{t}_2\ \ldots\ \mathbf{t}_n\right]. \tag{6.27}$$

Then we can write Eq. (6.20) in matrix form:

$$\mathbf{x} = x'_1\mathbf{t}_1 + x'_2\mathbf{t}_2 + \cdots + x'_n\mathbf{t}_n = \mathbf{B}_t\mathbf{x}'. \tag{6.28}$$

This equation demonstrates the relationships between the two different representations for the vector χ. (Note that this is effectively the same as Eq. (5.43). You may want to revisit our discussion of reciprocal basis vectors in Chapter 5.)

Now define a matrix whose columns are the $\mathbf{w}_i$:

$$\mathbf{B}_w = \left[\mathbf{w}_1\ \mathbf{w}_2\ \ldots\ \mathbf{w}_m\right]. \tag{6.29}$$

This allows us to write Eq. (6.21) in matrix form,

$$\mathbf{y} = \mathbf{B}_w\mathbf{y}', \tag{6.30}$$

which then demonstrates the relationships between the two different representations for the vector *y*.

Now substitute Eq. (6.28) and Eq. (6.30) into Eq. (6.19):

$$\mathbf{A}\mathbf{B}_t\mathbf{x}' = \mathbf{B}_w\mathbf{y}'. \tag{6.31}$$

If we multiply both sides of this equation by $\mathbf{B}_w^{-1}$ we obtain

$$[\mathbf{B}_w^{-1}\mathbf{A}\mathbf{B}_t]\mathbf{x}' = \mathbf{y}'. \tag{6.32}$$

Change of Basis

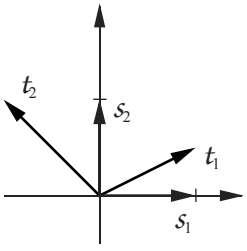
A comparison of Eq. (6.32) and Eq. (6.23) yields the following operation for a *change of basis*:

$$\mathbf{A}' = [\mathbf{B}_w^{-1}\mathbf{A}\mathbf{B}_t]. \tag{6.33}$$

Similarity Transform

This key result, which describes the relationship between any two matrix representations of a given linear transformation, is called a *similarity transform* [Brog91]. It will be of great use to us in later chapters. It turns out that with the right choice of basis vectors we can obtain a matrix representation that reveals the key characteristics of the linear transformation it represents. This will be discussed in the next section.

As an example of changing basis sets, let's revisit the vector rotation example of the previous section. In that section a matrix representation was developed using the standard basis set $\{s_1, s_2\}$. Now let's find a new representation using the basis $\{t_1, t_2\}$, which is shown in the adjacent fig-



ure. (Note that in this example the same basis set is used for both the domain and the range.)

The first step is to expand t_1 and t_2 in terms of the standard basis set, as in Eq. (6.24) and Eq. (6.25). By inspection of the adjacent figure we find:

$$t_1 = s_1 + 0.5\,s_2\,, \tag{6.34}$$

$$t_2 = -s_1 + s_2\,. \tag{6.35}$$

Therefore we can write

$$\mathbf{t}_1 = \begin{bmatrix} 1 \\ 0.5 \end{bmatrix} \qquad \mathbf{t}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}. \tag{6.36}$$

Now we can form the matrix

$$\mathbf{B}_t = \begin{bmatrix} \mathbf{t}_1 & \mathbf{t}_2 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0.5 & 1 \end{bmatrix}, \tag{6.37}$$

and, because we are using the same basis set for both the domain and the range of the transformation,

$$\mathbf{B}_w = \mathbf{B}_t = \begin{bmatrix} 1 & -1 \\ 0.5 & 1 \end{bmatrix}. \tag{6.38}$$

We can now compute the new matrix representation from Eq. (6.33):

$$\begin{aligned} \mathbf{A}' = [\mathbf{B}_w^{-1}\mathbf{A}\mathbf{B}_t] &= \begin{bmatrix} 2/3 & 2/3 \\ -1/3 & 2/3 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0.5 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1/3\sin\theta + \cos\theta & -4/3\sin\theta \\ \dfrac{5}{6}\sin\theta & -1/3\sin\theta + \cos\theta \end{bmatrix}. \end{aligned} \tag{6.39}$$

Take, for example, the case where $\theta = 30°$.

$$\mathbf{A}' = \begin{bmatrix} 1.033 & -0.667 \\ 0.417 & 0.699 \end{bmatrix}, \tag{6.40}$$

and

$$\mathbf{A} = \begin{bmatrix} 0.866 & -0.5 \\ 0.5 & 0.866 \end{bmatrix}. \tag{6.41}$$

To check that these matrices are correct, let's try a test vector

$$\mathbf{x} = \begin{bmatrix} 1 \\ 0.5 \end{bmatrix}, \text{ which corresponds to } \mathbf{x}' = \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \tag{6.42}$$

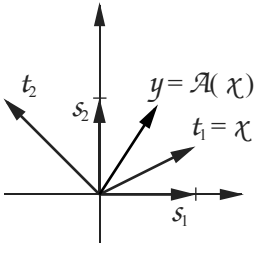
(Note that the vector represented by $\mathbf{x}$ and $\mathbf{x}'$ is t_1, a member of the second basis set.) The transformed test vector would be

$$\mathbf{y} = \mathbf{A}\mathbf{x} = \begin{bmatrix} 0.866 & -0.5 \\ 0.5 & 0.866 \end{bmatrix} \begin{bmatrix} 1 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0.616 \\ 0.933 \end{bmatrix}, \tag{6.43}$$

which should correspond to

$$\mathbf{y}' = \mathbf{A}'\mathbf{x}' = \begin{bmatrix} 1.033 & -0.667 \\ 0.416 & 0.699 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1.033 \\ 0.416 \end{bmatrix}. \tag{6.44}$$

How can we test to see if $\mathbf{y}'$ does correspond to $\mathbf{y}$? Both should be representations of the same vector, y, in terms of two different basis sets; $\mathbf{y}$ uses the basis $\{s_1, s_2\}$ and $\mathbf{y}'$ uses the basis $\{t_1, t_2\}$. In Chapter 5 we used the reciprocal basis vectors to transform from one representation to another (see Eq. (5.43)). Using that concept we have



$$\mathbf{y}' = \mathbf{B}^{-1}\mathbf{y} = \begin{bmatrix} 1 & -1 \\ 0.5 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0.616 \\ 0.933 \end{bmatrix} = \begin{bmatrix} 2/3 & 2/3 \\ -1/3 & 2/3 \end{bmatrix} \begin{bmatrix} 0.616 \\ 0.933 \end{bmatrix} = \begin{bmatrix} 1.033 \\ 0.416 \end{bmatrix}, \tag{6.45}$$

which verifies our previous result. The vectors are displayed in the figure to the left. Verify graphically that the two representations, $\mathbf{y}$ and $\mathbf{y}'$, given by Eq. (6.43) and Eq. (6.44), are reasonable.

Eigenvalues and Eigenvectors

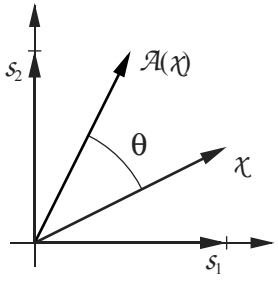
In this final section we want to discuss two key properties of linear transformations: eigenvalues and eigenvectors. Knowledge of these properties will allow us to answer some key questions about neural network performance, such as the question we posed at the beginning of this chapter, concerning the stability of Hopfield networks.

Eigenvalues
Eigenvectors

Let's first define what we mean by *eigenvalues* and *eigenvectors*. Consider a linear transformation $\mathcal{A}: X \to X$. (The domain is the same as the range.) Those vectors $z \in X$ that are not equal to zero and those scalars λ that satisfy

$$\mathcal{A}(z) = \lambda z \tag{6.46}$$

are called eigenvectors (z) and eigenvalues (λ), respectively. Notice that the term eigenvector is a little misleading, since it is not really a vector but a vector space, since if z satisfies Eq. (6.46), then az will also satisfy it.



Therefore an eigenvector of a given transformation represents a direction, such that any vector in that direction, when transformed, will continue to point in the same direction, but will be scaled by the eigenvalue. As an example, consider again the rotation example used in the previous sections. Is there any vector that, when rotated by 30°, continues to point in the same direction? No; this is a case where there are no real eigenvalues. (If we allow complex scalars, then two eigenvalues exist, as we will see later.)

How can we compute the eigenvalues and eigenvectors? Suppose that a basis has been chosen for the n-dimensional vector space X. Then the matrix representation for Eq. (6.46) can be written

$$\mathbf{A}\mathbf{z} = \lambda\mathbf{z}, \tag{6.47}$$

or

$$[\mathbf{A} - \lambda\mathbf{I}]\mathbf{z} = \mathbf{0}. \tag{6.48}$$

This means that the columns of $[\mathbf{A} - \lambda\mathbf{I}]$ are dependent, and therefore the determinant of this matrix must be zero:

$$|[\mathbf{A} - \lambda\mathbf{I}]| = 0. \tag{6.49}$$

This determinant is an nth-order polynomial. Therefore Eq. (6.49) always has n roots, some of which may be complex and some of which may be repeated.

As an example, let's revisit the rotation example. If we use the standard basis set, the matrix of the transformation is

$$\mathbf{A} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}. \tag{6.50}$$

We can then write Eq. (6.49) as

$$\left| \begin{bmatrix} \cos\theta - \lambda & -\sin\theta \\ \sin\theta & \cos\theta - \lambda \end{bmatrix} \right| = 0, \tag{6.51}$$

or

$$\lambda^2 - 2\lambda\cos\theta + ((\cos\theta)^2 + (\sin\theta)^2) = \lambda^2 - 2\lambda\cos\theta + 1 = 0. \tag{6.52}$$

The roots of this equation are

$$\lambda_1 = \cos\theta + j\sin\theta \qquad \lambda_2 = \cos\theta - j\sin\theta . \tag{6.53}$$

Therefore, as we predicted, this transformation has no real eigenvalues (if $\sin\theta \neq 0$). This means that when any real vector is transformed, it will point in a new direction.

Consider another matrix:

$$\mathbf{A} = \begin{bmatrix} -1 & 1 \\ 0 & -2 \end{bmatrix} . \tag{6.54}$$

To find the eigenvalues we must solve

$$\left| \begin{bmatrix} -1-\lambda & 1 \\ 0 & -2-\lambda \end{bmatrix} \right| = 0 , \tag{6.55}$$

or

$$\lambda^2 + 3\lambda + 2 = (\lambda + 1)(\lambda + 2) = 0 , \tag{6.56}$$

and the eigenvalues are

$$\lambda_1 = -1 \qquad \lambda_2 = -2 . \tag{6.57}$$

To find the eigenvectors we must solve Eq. (6.48), which in this example becomes

$$\begin{bmatrix} -1-\lambda & 1 \\ 0 & -2-\lambda \end{bmatrix} \mathbf{z} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} . \tag{6.58}$$

We will solve this equation twice, once using λ_1 and once using λ_2. Beginning with λ_1 we have

$$\begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \mathbf{z}_1 = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} z_{11} \\ z_{21} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{6.59}$$

or

$$z_{21} = 0 , \text{ no constraint on } z_{11} . \tag{6.60}$$

Therefore the first eigenvector will be

$$\mathbf{z}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \tag{6.61}$$

or any scalar multiple. For the second eigenvector we use λ_2:

$$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \mathbf{z}_2 = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} z_{12} \\ z_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \tag{6.62}$$

or

$$z_{22} = -z_{12}. \tag{6.63}$$

Therefore the second eigenvector will be

$$\mathbf{z}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \tag{6.64}$$

or any scalar multiple.

To verify our results we consider the following:

$$\mathbf{A}\mathbf{z}_1 = \begin{bmatrix} -1 & 1 \\ 0 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix} = (-1) \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \lambda_1 \mathbf{z}_1, \tag{6.65}$$

$$\mathbf{A}\mathbf{z}_2 = \begin{bmatrix} -1 & 1 \\ 0 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} -2 \\ 2 \end{bmatrix} = (-2) \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \lambda_2 \mathbf{z}_2. \tag{6.66}$$

To test your understanding of eigenvectors, use the Neural Network Design Demonstration Eigenvector Game (**nnd6eg**).

Diagonalization

Whenever we have n distinct eigenvalues we are guaranteed that we can find n independent eigenvectors [Brog91]. Therefore the eigenvectors make up a basis set for the vector space of the transformation. Let's find the matrix of the previous transformation (Eq. (6.54)) using the eigenvectors as the basis vectors. From Eq. (6.33) we have

$$\mathbf{A}' = [\mathbf{B}^{-1}\mathbf{A}\mathbf{B}] = \begin{bmatrix} 1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} -1 & 1 \\ 0 & -2 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & -2 \end{bmatrix}. \tag{6.67}$$

Note that this is a diagonal matrix, with the eigenvalues on the diagonal. This is not a coincidence. Whenever we have distinct eigenvalues we can diagonalize the matrix representation by using the eigenvectors as the ba-

Diagonalization

sis vectors. This *diagonalization* process is summarized in the following. Let

$$\mathbf{B} = \left[\mathbf{z}_1 \ \mathbf{z}_2 \ \ldots \ \mathbf{z}_n\right], \tag{6.68}$$

where $\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n\}$ are the eigenvectors of a matrix **A**. Then

$$[\mathbf{B}^{-1}\mathbf{A}\mathbf{B}] = \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & \lambda_n \end{bmatrix}, \tag{6.69}$$

where $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ are the eigenvalues of the matrix **A**.

This result will be very helpful as we analyze the performance of several neural networks in later chapters.

Summary of Results

Transformations

A *transformation* consists of three parts:

1. a set of elements $X = \{\chi_i\}$, called the domain,
2. a set of elements $Y = \{y_i\}$, called the range, and
3. a rule relating each $\chi_i \in X$ to an element $y_i \in Y$.

Linear Transformations

A transformation $\mathcal{A}$ is *linear* if:

1. for all $\chi_1, \chi_2 \in X$, $\mathcal{A}(\chi_1 + \chi_2) = \mathcal{A}(\chi_1) + \mathcal{A}(\chi_2)$,
2. for all $\chi \in X$, $a \in R$, $\mathcal{A}(a\chi) = a\mathcal{A}(\chi)$.

Matrix Representations

Let $\{v_1, v_2, \ldots, v_n\}$ be a basis for vector space X, and let $\{u_1, u_2, \ldots, u_m\}$ be a basis for vector space Y. Let $\mathcal{A}$ be a linear transformation with domain X and range Y:

$$\mathcal{A}(\chi) = y.$$

The coefficients of the matrix representation are obtained from

$$\mathcal{A}(v_j) = \sum_{i=1}^{m} a_{ij} u_i .$$

Change of Basis

$$\mathbf{B}_t = \left[\mathbf{t}_1 \ \mathbf{t}_2 \ \ldots \ \mathbf{t}_n\right]$$

$$\mathbf{B}_w = \left[\mathbf{w}_1 \ \mathbf{w}_2 \ \ldots \ \mathbf{w}_m\right]$$

$$\mathbf{A}' = [\mathbf{B}_w^{-1}\mathbf{A}\mathbf{B}_t]$$

Eigenvalues and Eigenvectors

$$\mathbf{A}\mathbf{z} = \lambda\mathbf{z}$$

$$|[\mathbf{A} - \lambda\mathbf{I}]| = 0$$

Diagonalization

$$\mathbf{B} = \left[\mathbf{z}_1\ \mathbf{z}_2\ \ldots\ \mathbf{z}_n\right],$$

where $\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n\}$ are the eigenvectors of a square matrix $\mathbf{A}$.

$$[\mathbf{B}^{-1}\mathbf{A}\mathbf{B}] = \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & \lambda_n \end{bmatrix}$$

Solved Problems

P6.1 Consider the single-layer network shown in Figure P6.1, which has a linear transfer function. Is the transformation from the input vector to the output vector a linear transformation?

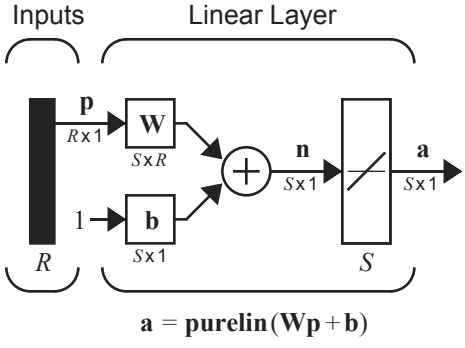


Figure P6.1 Single-Neuron Perceptron

The network equation is

$$\mathbf{a} = \mathcal{A}(\mathbf{p}) = \mathbf{W}\mathbf{p} + \mathbf{b}.$$

In order for this transformation to be linear it must satisfy

1. $\mathcal{A}(\mathbf{p}_1 + \mathbf{p}_2) = \mathcal{A}(\mathbf{p}_1) + \mathcal{A}(\mathbf{p}_2)$,
2. $\mathcal{A}(a\mathbf{p}) = a\mathcal{A}(\mathbf{p})$.

Let's test condition 1 first.

$$\mathcal{A}(\mathbf{p}_1 + \mathbf{p}_2) = \mathbf{W}(\mathbf{p}_1 + \mathbf{p}_2) + \mathbf{b} = \mathbf{W}\mathbf{p}_1 + \mathbf{W}\mathbf{p}_2 + \mathbf{b}.$$

Compare this with

$$\mathcal{A}(\mathbf{p}_1) + \mathcal{A}(\mathbf{p}_2) = \mathbf{W}\mathbf{p}_1 + \mathbf{b} + \mathbf{W}\mathbf{p}_2 + \mathbf{b} = \mathbf{W}\mathbf{p}_1 + \mathbf{W}\mathbf{p}_2 + 2\mathbf{b}.$$

Clearly these two expressions will be equal only if $\mathbf{b} = \mathbf{0}$. Therefore this network performs a nonlinear transformation, even though it has a linear transfer function. This particular type of nonlinearity is called an affine transformation.

P6.2 We discussed projections in Chapter 5. Is a projection a linear transformation?

The projection of a vector χ onto a vector v is computed as

$$y = \mathcal{A}(\chi) = \frac{(\chi,v)}{(v,v)}v,$$

where (χ,v) is the inner product of χ with v.

We need to check to see if this transformation satisfies the two conditions for linearity. Let's start with condition 1:

$$\mathcal{A}(\chi_1 + \chi_2) = \frac{(\chi_1 + \chi_2,v)}{(v,v)}v = \frac{(\chi_1,v) + (\chi_2,v)}{(v,v)}v = \frac{(\chi_1,v)}{(v,v)}v + \frac{(\chi_2,v)}{(v,v)}v$$

$$= \mathcal{A}(\chi_1) + \mathcal{A}(\chi_2).$$

(Here we used linearity properties of inner products.) Checking condition 2:

$$\mathcal{A}(a\chi) = \frac{(a\chi,v)}{(v,v)}v = \frac{a(\chi,v)}{(v,v)}v = a\mathcal{A}(\chi).$$

Therefore projection is a linear operation.

P6.3 Consider the transformation $\mathcal{A}$ created by reflecting a vector χ in $\Re^2$ about the line $x_1 + x_2 = 0$, as shown in Figure P6.2. Find the matrix of this transformation relative to the standard basis in $\Re^2$.

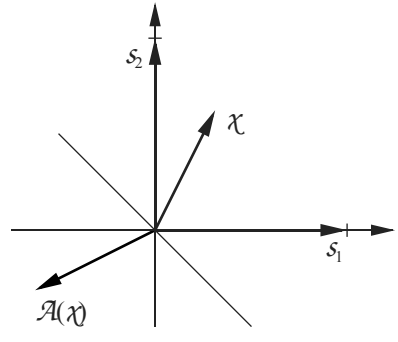
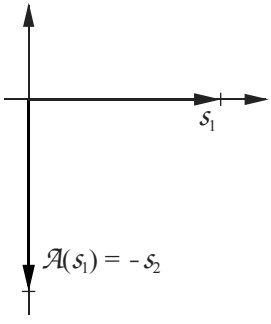


Figure P6.2 Reflection Transformation

The key to finding the matrix of a transformation is given in Eq. (6.6):

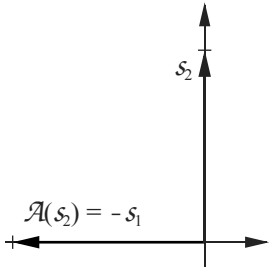
$$\mathcal{A}(v_j) = \sum_{i=1}^{m} a_{ij} u_i .$$



We need to transform each basis vector of the domain and then expand the result in terms of the basis vectors for the range. Each time we do the expansion we get one column of the matrix representation. In this case the basis set for both the domain and the range is $\{s_1, s_2\}$. So let's transform s_1 first. If we reflect s_1 about the line $x_1 + x_2 = 0$, we find

$$\mathcal{A}(s_1) = -s_2 = \sum_{i=1}^{2} a_{i1} s_i = a_{11} s_1 + a_{21} s_2 = 0 s_1 + (-1) s_2$$

(as shown in the top left figure), which gives us the first column of the matrix. Next we transform s_2:



$$\mathcal{A}(s_2) = -s_1 = \sum_{i=1}^{2} a_{i2} s_i = a_{12} s_1 + a_{22} s_2 = (-1) s_1 + 0 s_2$$

(as shown in the second figure on the left), which gives us the second column of the matrix. The final result is

$$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}.$$

Let's test our result by transforming the vector $\mathbf{x} = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \end{bmatrix}.$$

This is indeed the reflection of $\mathbf{x}$ about the line $x_1 + x_2 = 0$, as we can see in Figure P6.3.

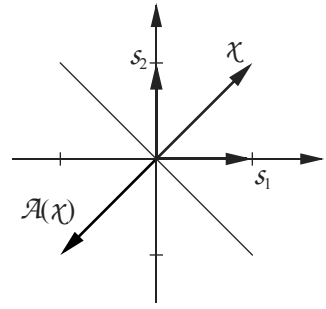


Figure P6.3 Test of Reflection Operation

(Can you guess the eigenvalues and eigenvectors of this transformation? Use the *Neural Network Design Demonstration Linear Transformations* (**nnd6lt**) to investigate this graphically. Compute the eigenvalues and eigenvectors, using the MATLAB function **eig**, and check your guess.)

P6.4 Consider the space of complex numbers. Let this be the vector space X, and let the basis for X be $\{1+j, 1-j\}$. Let $\mathcal{A}: X \rightarrow X$ be the conjugation operator (i.e., $\mathcal{A}(x) = x^*$).

i. Find the matrix of the transformation $\mathcal{A}$ relative to the basis set given above.

ii. Find the eigenvalues and eigenvectors of the transformation.

iii. Find the matrix representation for $\mathcal{A}$ relative to the eigenvectors as the basis vectors.

i. To find the matrix of the transformation, transform each of the basis vectors (by finding their conjugate):

$$\mathcal{A}(v_1) = \mathcal{A}(1+j) = 1-j = v_2 = a_{11}v_1 + a_{21}v_2 = 0v_1 + 1v_2,$$

$$\mathcal{A}(v_2) = \mathcal{A}(1-j) = 1+j = v_1 = a_{12}v_1 + a_{22}v_2 = 1v_1 + 0v_2.$$

This gives us the matrix representation

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

ii. To find the eigenvalues, we need to use Eq. (6.49):

$$|[\mathbf{A} - \lambda\mathbf{I}]| = \left|\begin{bmatrix} -\lambda & 1 \\ 1 & -\lambda \end{bmatrix}\right| = \lambda^2 - 1 = (\lambda - 1)(\lambda + 1) = 0 .$$

So the eigenvalues are: $\lambda_1 = 1$, $\lambda_2 = -1$. To find the eigenvectors, use Eq. (6.48):

$$[\mathbf{A} - \lambda\mathbf{I}]\mathbf{z} = \begin{bmatrix} -\lambda & 1 \\ 1 & -\lambda \end{bmatrix}\mathbf{z} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

For $\lambda = \lambda_1 = 1$ this gives us

$$\begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}\mathbf{z}_1 = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} z_{11} \\ z_{21} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

or

$$z_{11} = z_{21} .$$

Therefore the first eigenvector will be

$$\mathbf{z}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

or any scalar multiple. For the second eigenvector we use $\lambda = \lambda_2 = -1$:

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\mathbf{z}_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} z_{12} \\ z_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

or

$$z_{12} = -z_{22} .$$

Therefore the second eigenvector is

$$\mathbf{z}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix},$$

or any scalar multiple.

Note that while these eigenvectors can be represented as columns of numbers, in reality they are complex numbers. For example:

$$z_1 = 1v_1 + 1v_2 = (1 + j) + (1 - j) = 2 ,$$

$$z_2 = 1v_1 + (-1)v_2 = (1+j) - (1-j) = 2j.$$

Checking that these are indeed eigenvectors:

$$\mathcal{A}(z_1) = (2)^* = 2 = \lambda_1 z_1,$$

$$\mathcal{A}(z_2) = (2j)^* = -2j = \lambda_2 z_2.$$

iii. To perform a change of basis we need to use Eq. (6.33):

$$\mathbf{A}' = [\mathbf{B}_w^{-1}\mathbf{A}\mathbf{B}_t] = [\mathbf{B}^{-1}\mathbf{A}\mathbf{B}],$$

where

$$\mathbf{B} = \begin{bmatrix}\mathbf{z}_1 & \mathbf{z}_2\end{bmatrix} = \begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}.$$

(We are using the same basis set for the range and the domain.) Therefore we have

$$\mathbf{A}' = \begin{bmatrix}0.5 & 0.5\\ 0.5 & -0.5\end{bmatrix}\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} = \begin{bmatrix}\lambda_1 & 0\\ 0 & \lambda_2\end{bmatrix}.$$

As expected from Eq. (6.69), we have diagonalized the matrix representation.

P6.5 Diagonalize the following matrix:

$$\mathbf{A} = \begin{bmatrix}2 & -2\\ -1 & 3\end{bmatrix}.$$

The first step is to find the eigenvalues:

$$|[\mathbf{A} - \lambda\mathbf{I}]| = \left|\begin{bmatrix}2-\lambda & -2\\ -1 & 3-\lambda\end{bmatrix}\right| = \lambda^2 - 5\lambda + 4 = (\lambda - 1)(\lambda - 4) = 0,$$

so the eigenvalues are $\lambda_1 = 1$, $\lambda_2 = 4$. To find the eigenvectors,

$$[\mathbf{A} - \lambda\mathbf{I}]\mathbf{z} = \begin{bmatrix}2-\lambda & -2\\ -1 & 3-\lambda\end{bmatrix}\mathbf{z} = \begin{bmatrix}0\\ 0\end{bmatrix}.$$

For $\lambda = \lambda_1 = 1$

$$\begin{bmatrix} 1 & -2 \\ -1 & 2 \end{bmatrix} \mathbf{z}_1 = \begin{bmatrix} 1 & -2 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} z_{11} \\ z_{21} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

or

$$z_{11} = 2z_{21}.$$

Therefore the first eigenvector will be

$$\mathbf{z}_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix},$$

or any scalar multiple.

For $\lambda = \lambda_2 = 4$

$$\begin{bmatrix} -2 & -2 \\ -1 & -1 \end{bmatrix} \mathbf{z}_1 = \begin{bmatrix} -2 & -2 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} z_{12} \\ z_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

or

$$z_{12} = -z_{22}.$$

Therefore the second eigenvector will be

$$\mathbf{z}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix},$$

or any scalar multiple.

To diagonalize the matrix we use Eq. (6.69):

$$\mathbf{A}' = [\mathbf{B}^{-1}\mathbf{A}\mathbf{B}],$$

where

$$\mathbf{B} = \begin{bmatrix} \mathbf{z}_1 & \mathbf{z}_2 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & -1 \end{bmatrix}.$$

Therefore we have

$$\mathbf{A}' = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & -\frac{2}{3} \end{bmatrix} \begin{bmatrix} 2 & -2 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}.$$

P6.6 Consider a transformation $\mathcal{A}: R^3 \to R^2$ whose matrix representation relative to the standard basis sets is

$$\mathbf{A} = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Find the matrix for this transformation relative to the basis sets:

$$T = \left\{ \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -2 \\ 3 \end{bmatrix} \right\} \qquad W = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -2 \end{bmatrix} \right\}.$$

The first step is to form the matrices

$$\mathbf{B}_t = \begin{bmatrix} 2 & 0 & 0 \\ 0 & -1 & -2 \\ 1 & 0 & 3 \end{bmatrix} \qquad \mathbf{B}_w = \begin{bmatrix} 1 & 0 \\ 0 & -2 \end{bmatrix}.$$

Now we use Eq. (6.33) to form the new matrix representation:

$$\mathbf{A}' = [\mathbf{B}_w^{-1} \mathbf{A} \mathbf{B}_t],$$

$$\mathbf{A}' = \begin{bmatrix} 1 & 0 \\ 0 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} 3 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & -1 & -2 \\ 1 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 6 & 1 & 2 \\ -\frac{1}{2} & 0 & -\frac{3}{2} \end{bmatrix}.$$

Therefore this is the matrix of the transformation with respect to the basis sets T and W.

P6.7 Consider a transformation $\mathcal{A}: \Re^2 \to \Re^2$. One basis set for $\Re^2$ is given as $V = \{v_1, v_2\}$.

i. Find the matrix of the transformation $\mathcal{A}$ relative to the basis set V if it is given that

$$\mathcal{A}(v_1) = v_1 + 2v_2,$$
$$\mathcal{A}(v_2) = v_1 + v_2.$$

ii. Consider a new basis set $W = \{w_1, w_2\}$. Find the matrix of the transformation $\mathcal{A}$ relative to the basis set W if it is given that

$$w_1 = v_1 + v_2,$$
$$w_2 = v_1 - v_2.$$

i. Each of the two equations gives us one column of the matrix, as defined in Eq. (6.6). Therefore the matrix is

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix}.$$

ii. We can represent the W basis vectors as columns of numbers in terms of the V basis vectors:

$$\mathbf{w}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \mathbf{w}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

We can now form the basis matrix that we need to perform the similarity transform:

$$\mathbf{B}_w = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

The new matrix representation can then be obtained from Eq. (6.33):

$$\mathbf{A}' = [\mathbf{B}_w^{-1}\mathbf{A}\mathbf{B}_w],$$

$$\mathbf{A}' = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} \frac{5}{2} & \frac{1}{2} \\ -\frac{1}{2} & -\frac{1}{2} \end{bmatrix}.$$

P6.8 Consider the vector space P^2 of all polynomials of degree less than or equal to 2. One basis for this vector space is $V = \{1, t, t^2\}$. Consider the differentiation transformation $\mathcal{D}$.

i. Find the matrix of this transformation relative to the basis set V.

ii. Find the eigenvalues and eigenvectors of the transformation.

i. The first step is to transform each of the basis vectors:

$$\mathcal{D}(1) = 0 = (0)1 + (0)t + (0)t^2,$$

$$\mathcal{D}(t) = 1 = (1)1 + (0)t + (0)t^2,$$

$$\mathcal{D}(t^2) = 2t = (0)1 + (2)t + (0)t^2.$$

The matrix of the transformation is then given by

$$\mathbf{D} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}.$$

ii. To find the eigenvalues we must solve

$$|[\mathbf{D} - \lambda\mathbf{I}]| = \left| \begin{bmatrix} -\lambda & 1 & 0 \\ 0 & -\lambda & 2 \\ 0 & 0 & -\lambda \end{bmatrix} \right| = -\lambda^3 = 0.$$

Therefore all three eigenvalues are zero. To find the eigenvectors we need to solve

$$[\mathbf{D} - \lambda\mathbf{I}]\mathbf{z} = \begin{bmatrix} -\lambda & 1 & 0 \\ 0 & -\lambda & 2 \\ 0 & 0 & -\lambda \end{bmatrix} \mathbf{z} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

For $\lambda = 0$ we have

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

This means that

$$z_2 = z_3 = 0 .$$

Therefore we have a single eigenvector:

$$\mathbf{z} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} .$$

Therefore the only polynomial whose derivative is a scaled version of itself is a constant (a zeroth-order polynomial).

P6.9 Consider a transformation $\mathcal{A}: R^2 \to R^2$. Two examples of transformed vectors are given in Figure P6.4. Find the matrix representation of this transformation relative to the standard basis set.

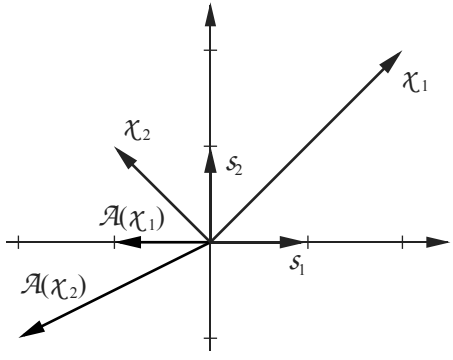


Figure P6.4 Transformation for Problem P6.9

For this problem we do not know how the basis vectors are transformed, so we cannot use Eq. (6.6) to find the matrix representation. However, we do know how two vectors are transformed, and we do know how those vectors can be represented in terms of the standard basis set. From Figure P6.4 we can write the following equations:

$$\mathbf{A}\begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \ \mathbf{A}\begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ -1 \end{bmatrix} .$$

We then put these two equations together to form

$$\mathbf{A}\begin{bmatrix} 2 & -1 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} -1 & -2 \\ 0 & -1 \end{bmatrix} .$$

So that

$$\mathbf{A} = \begin{bmatrix} -1 & -2 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 2 & -1 \\ 2 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} -1 & -2 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} \frac{1}{4} & \frac{1}{4} \\ -\frac{1}{2} & \frac{1}{2} \end{bmatrix} = \begin{bmatrix} \frac{3}{4} & -\frac{5}{4} \\ \frac{1}{2} & -\frac{1}{2} \end{bmatrix}.$$

This is the matrix representation of the transformation with respect to the standard basis set.

This procedure is used in the *Neural Network Design Demonstration Linear Transformations* (**nnd6lt**).

Epilogue

In this chapter we have reviewed those properties of linear transformations and matrices that are most important to our study of neural networks. The concepts of eigenvalues, eigenvectors, change of basis (similarity transformation) and diagonalization will be used again and again throughout the remainder of this text. Without this linear algebra background our study of neural networks could only be superficial.

In the next chapter we will use linear algebra to analyze the operation of one of the first neural network training algorithms — the Hebb rule.

Further Reading

[Brog91] W. L. Brogan, *Modern Control Theory*, 3rd Ed., Englewood Cliffs, NJ: Prentice-Hall, 1991.

This is a well-written book on the subject of linear systems. The first half of the book is devoted to linear algebra. It also has good sections on the solution of linear differential equations and the stability of linear and nonlinear systems. It has many worked problems.

[Stra76] G. Strang, *Linear Algebra and Its Applications*, New York: Academic Press, 1980.

Strang has written a good basic text on linear algebra. Many applications of linear algebra are integrated into the text.

Exercises

E6.1 Is the operation of transposing a matrix a linear transformation?

E6.2 Consider again the neural network shown in Figure P6.1. Show that if the bias vector **b** is equal to zero then the network performs a linear operation.

E6.3 Consider the linear transformation illustrated in Figure E6.1.

i. Find the matrix representation of this transformation relative to the standard basis set.

ii. Find the matrix of this transformation relative to the basis set $\{v_1, v_2\}$.

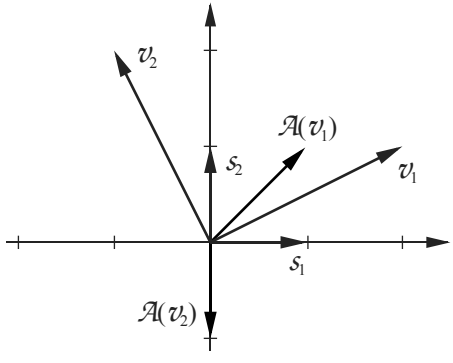


Figure E6.1 Example Transformation for Exercise E6.3

E6.4 Consider the space of complex numbers. Let this be the vector space X, and let the basis for X be $\{1+j, 1-j\}$. Let $\mathcal{A}: X \to X$ be the operation of multiplication by $(1+j)$ (i.e., $\mathcal{A}(x) = (1+j)x$).

i. Find the matrix of the transformation $\mathcal{A}$ relative to the basis set given above.

ii. Find the eigenvalues and eigenvectors of the transformation.

iii. Find the matrix representation for $\mathcal{A}$ relative to the eigenvectors as the basis vectors.

```
» 2 + 2
ans =
     4
```

iv. Check your answers to parts (ii) and (iii) using MATLAB.

E6.5 Consider a transformation $\mathcal{A}: P^2 \rightarrow P^3$, from the space of second-order polynomials to the space of third-order polynomials, which is defined by the following:

$$\chi = a_0 + a_1 t + a_2 t^2,$$

$$\mathcal{A}(\chi) = a_0(t+1) + a_1(t+1)^2 + a_2(t+1)^3.$$

Find the matrix representation of this transformation relative to the basis sets $V^2 = \{1, t, t^2\}$, $V^3 = \{1, t, t^2, t^3\}$.

E6.6 Consider the vector space of polynomials of degree two or less. These polynomials have the form $f(t) = a_0 + a_1 t + a_2 t^2$. Now consider the transformation in which the variable t is replaced by $t+1$. (for example, $t^2 + 2t + 3 \Rightarrow (t+1)^2 + 2(t+1) + 3 = t^2 + 4t + 6$)

- **i.** Find the matrix of this transformation with respect to the basis set $\{1, t-1, t^2\}$.
- **ii.** Find the eigenvalues and eigenvectors of the transformation. Show the eigenvectors as columns of numbers and as functions of time (polynomials).

E6.7 Consider the space of functions of the form $\alpha \sin(t + \phi)$. One basis set for this space is $V = \{\sin t, \cos t\}$. Consider the differentiation transformation $\mathcal{D}$.

- **i.** Find the matrix of the transformation $\mathcal{D}$ relative to the basis set V.
- **ii.** Find the eigenvalues and eigenvectors of the transformation. Show the eigenvectors as columns of numbers and as functions of t.
- **iii.** Find the matrix of the transformation relative to the eigenvectors as basis vectors.

E6.8 Consider the vector space of functions of the form $\alpha + \beta e^{2t}$. One basis set for this vector space is $V = \{1 + e^{2t}, 1 - e^{2t}\}$. Consider the differentiation transformation $\mathcal{D}$.

- **i.** Find the matrix of the transformation $\mathcal{D}$ relative to the basis set V, using Eq. (6.6).
- **ii.** Verify the operation of the matrix on the function $2e^{2t}$.
- **iii.** Find the eigenvalues and eigenvectors of the transformation. Show the eigenvectors as columns of numbers (with respect to the basis set V) and as functions of t.

iv. Find the matrix of the transformation relative to the eigenvectors as basis vectors.

E6.9 Consider the set of all 2x2 matrices. This set is a vector space, which we will call X (yes, matrices can be vectors). If $\mathbf{M}$ is an element of this vector space, define the transformation $\mathcal{A}:X \to X$, such that $\mathcal{A}(\mathbf{M}) = \mathbf{M} + \mathbf{M}^T$. Consider the following basis set for the vector space X.

$$v_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \; v_2 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \; v_3 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \; v_4 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

i. Find the matrix representation of the transformation $\mathcal{A}$ relative to the basis set $\{v_1, v_2, v_3, v_4\}$ (for both domain and range) (using Eq. (6.6)).

ii. Verify the operation of the matrix representation from part i. on the element of X given below. (Verify that the matrix multiplication produces the same result as the transformation.)

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$$

iii. Find the eigenvalues and eigenvectors of the transformation. You do not need to use the matrix representation that you found in part i. You can find the eigenvalues and eigenvectors directly from the definition of the transformation. Your eigenvectors should be 2x2 matrices (elements of the vector space X). This does not require much computation. Use the definition of eigenvector in Eq. (6.46).

E6.10 Consider a transformation $\mathcal{A}:P^1 \to P^2$, from the space of first degree polynomials into the space of second degree polynomials. The transformation is defined as follows

$$\mathcal{A}(a + bt) = at + \frac{b}{2}t^2$$

(e.g., $\mathcal{A}(2 + 6t) = 2t + 3t^2$). One basis set for P^1 is $U = \{1, t\}$. One basis for P^2 is $V = \{1, t, t^2\}$.

i. Find the matrix representation of the transformation A relative to the basis sets U and V, using Eq. (6.6).

ii. Verify the operation of the matrix on the polynomial $6 + 8t$. (Verify that the matrix multiplication produces the same result as the transformation.)

iii. Using a similarity transform, find the matrix of the transformation with respect to the basis sets $S = \{1 + t, 1 - t\}$ and V.

E6.11 Let $\mathcal{D}$ be the differentiation operator ($\mathcal{D}(f) = df/dt$), and use the basis set

$$\{u_1, u_2\} = \{e^{5t}, te^{5t}\}$$

for both the domain and the range of the transformation $\mathcal{D}$.

i. Show that the transformation $\mathcal{D}$ is linear.,

ii. Find the matrix of this transformation relative to the basis shown above.

iii. Find the eigenvalues and eigenvectors of the transformation $\mathcal{D}$.

E6.12 A certain linear transformation has the following eigenvalues and eigenvectors (represented in terms of the standard basis set):

$$\left\{\mathbf{z}_1 = \begin{bmatrix}1\\2\end{bmatrix}, \lambda_1 = 1\right\}, \left\{\mathbf{z}_2 = \begin{bmatrix}-1\\2\end{bmatrix}, \lambda_2 = 2\right\}$$

i. Find the matrix representation of the transformation, relative to the standard basis set.

ii. Find the matrix representation of the transformation relative to the eigenvectors as the basis vectors.

E6.13 Consider a transformation $\mathcal{A}: \Re^2 \rightarrow \Re^2$. In the figure below, we show a set of basis vectors $V = \{v_1, v_2\}$ and the transformed basis vectors.

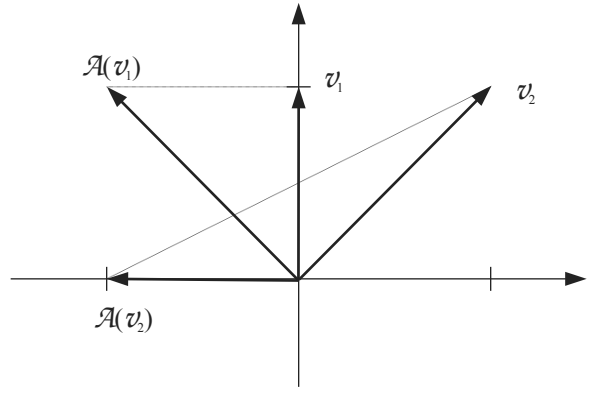


Figure E6.2 Definition of Transformation for Exercise E6.13

i. Find the matrix representation of this transformation with respect to the basis vectors $V = \{v_1, v_2\}$.

ii. Find the matrix representation of this transformation with respect to the standard basis vectors.

iii. Find the eigenvalues and eigenvectors of this transformation. Sketch the eigenvectors and their transformations.

iv. Find the matrix representation of this transformation with respect to the eigenvectors as the basis vectors.

E6.14 Consider the vector spaces P^2 and P^3 of second-order and third-order polynomials. Find the matrix representation of the integration transformation $I: P^2 \rightarrow P^3$, relative to the basis sets $V^2 = \{1, t, t^2\}$, $V^3 = \{1, t, t^2, t^3\}$.

E6.15 A certain linear transformation $\mathcal{A}: \Re^2 \rightarrow \Re^2$ has a matrix representation relative to the standard basis set of

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}.$$

Find the matrix representation of this transformation relative to the new basis set:

$$V = \left\{ \begin{bmatrix} 1 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 5 \end{bmatrix} \right\}.$$

E6.16 We know that a certain linear transformation $\mathcal{A}: R^2 \rightarrow R^2$ has eigenvalues and eigenvectors given by

$$\lambda_1 = 1 \qquad \mathbf{z}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \lambda_2 = 2 \qquad \mathbf{z}_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

(The eigenvectors are represented relative to the standard basis set.)

i. Find the matrix representation of the transformation $\mathcal{A}$ relative to the standard basis set.

ii. Find the matrix representation relative to the new basis

$$V = \left\{ \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\}.$$

E6.17 Consider the transformation $\mathcal{A}$ created by projecting a vector x onto the line shown in Figure E6.3. An example of the transformation is shown in the figure.

i. Using Eq. (6.6), find the matrix representation of this transformation relative to the standard basis set $\{s_1, s_2\}$.

ii. Using your answer to part i, find the matrix representation of this transformation relative to the basis set $\{v_1, v_2\}$ shown in Figure E6.3.

iii. What are the eigenvalues and eigenvectors of this transformation? Sketch the eigenvectors and their transformations.

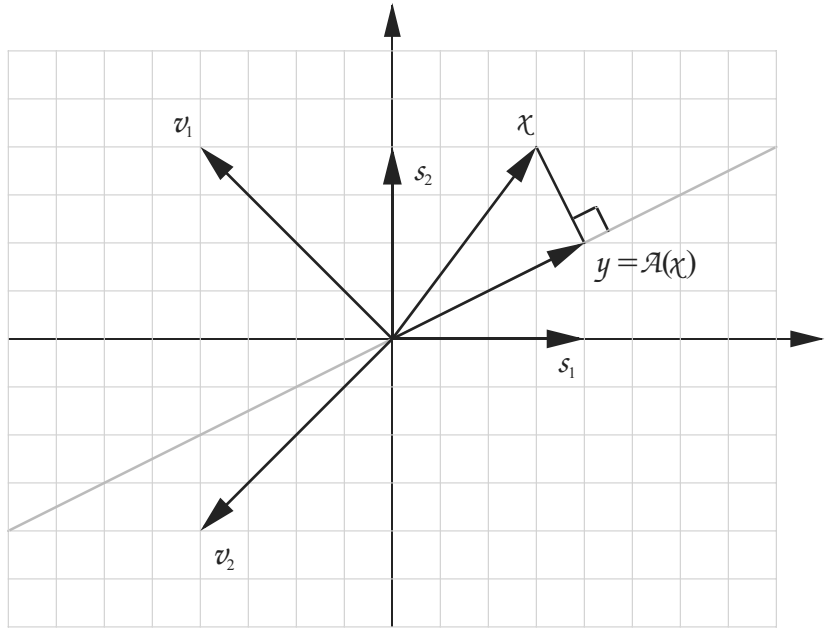


Figure E6.3 Definition of Transformation for Exercise E6.17

E6.18 Consider the following basis set for $\Re^2$:

$$V = \{\mathbf{v}_1, \mathbf{v}_2\} = \left\{ \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 \\ -2 \end{bmatrix} \right\}.$$

(The basis vectors are represented relative to the standard basis set.)

i. Find the reciprocal basis vectors for this basis set.

ii. Consider a transformation $\mathcal{A}: \Re^2 \to \Re^2$. The matrix representation for $\mathcal{A}$ relative to the standard basis in $\Re^2$ is

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix}.$$

Find the expansion of $\mathbf{A}\mathbf{v}_1$ in terms of the basis set V. (Use the reciprocal basis vectors.)

iii. Find the expansion of $\mathbf{A}\mathbf{v}_2$ in terms of the basis set V.

iv. Find the matrix representation for $\mathcal{A}$ relative to the basis V. (This step should require no further computation.)

7 Supervised Hebbian Learning

Objectives

The Hebb rule was one of the first neural network learning laws. It was proposed by Donald Hebb in 1949 as a possible mechanism for synaptic modification in the brain and since then has been used to train artificial neural networks.

In this chapter we will use the linear algebra concepts of the previous two chapters to explain why Hebbian learning works. We will also show how the Hebb rule can be used to train neural networks for pattern recognition.

Theory and Examples

Donald O. Hebb was born in Chester, Nova Scotia, just after the turn of the century. He originally planned to become a novelist, and obtained a degree in English from Dalhousie University in Halifax in 1925. Since every first-rate novelist needs to have a good understanding of human nature, he began to study Freud after graduation and became interested in psychology. He then pursued a master's degree in psychology at McGill University, where he wrote a thesis on Pavlovian conditioning. He received his Ph.D. from Harvard in 1936, where his dissertation investigated the effects of early experience on the vision of rats. Later he joined the Montreal Neurological Institute, where he studied the extent of intellectual changes in brain surgery patients. In 1942 he moved to the Yerkes Laboratories of Primate Biology in Florida, where he studied chimpanzee behavior.

In 1949 Hebb summarized his two decades of research in *The Organization of Behavior* [Hebb49]. The main premise of this book was that behavior could be explained by the action of neurons. This was in marked contrast to the behaviorist school of psychology (with proponents such as B. F. Skinner), which emphasized the correlation between stimulus and response and discouraged the use of any physiological hypotheses. It was a confrontation between a top-down philosophy and a bottom-up philosophy. Hebb stated his approach: "The method then calls for learning as much as one can about what the parts of the brain do (primarily the physiologist's field), and relating the behavior as far as possible to this knowledge (primarily for the psychologist); then seeing what further information is to be had about how the total brain works, from the discrepancy between (1) actual behavior and (2) the behavior that would be predicted from adding up what is known about the action of the various parts."

The most famous idea contained in *The Organization of Behavior* was the postulate that came to be known as Hebbian learning:

Hebb's Postulate

"*When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.*"

This postulate suggested a physical mechanism for learning at the cellular level. Although Hebb never claimed to have firm physiological evidence for his theory, subsequent research has shown that some cells do exhibit Hebbian learning. Hebb's theories continue to influence current research in neuroscience.

As with most historic ideas, Hebb's postulate was not completely new, as he himself emphasized. It had been foreshadowed by several others, including Freud. Consider, for example, the following principle of association stated by psychologist and philosopher William James in 1890: "When two

brain processes are active together or in immediate succession, one of them, on reoccurring tends to propagate its excitement into the other."

Linear Associator

Linear Associator

Hebb's learning law can be used in combination with a variety of neural network architectures. We will use a very simple architecture for our initial presentation of Hebbian learning. In this way we can concentrate on the learning law rather than the architecture. The network we will use is the *linear associator*, which is shown in Figure 7.1. (This network was introduced independently by James Anderson [Ande72] and Teuvo Kohonen [Koho72].)

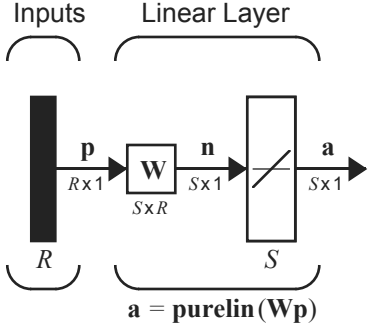


Figure 7.1 Linear Associator

The output vector $\mathbf{a}$ is determined from the input vector $\mathbf{p}$ according to:

$$\mathbf{a} = \mathbf{W}\mathbf{p}, \tag{7.1}$$

or

$$a_i = \sum_{j=1}^{R} w_{ij} p_j . \tag{7.2}$$

Associative Memory

The linear associator is an example of a type of neural network called an *associative memory*. The task of an associative memory is to learn Q pairs of prototype input/output vectors:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\} . \tag{7.3}$$

In other words, if the network receives an input $\mathbf{p} = \mathbf{p}_q$ then it should produce an output $\mathbf{a} = \mathbf{t}_q$, for $q = 1, 2, \ldots, Q$. In addition, if the input is changed slightly (i.e., $\mathbf{p} = \mathbf{p}_q + \delta$) then the output should only be changed slightly (i.e., $\mathbf{a} = \mathbf{t}_q + \varepsilon$).

The Hebb Rule

How can we interpret Hebb's postulate mathematically, so that we can use it to train the weight matrix of the linear associator? First, let's rephrase the postulate: If two neurons on either side of a synapse are activated simultaneously, the strength of the synapse will increase. Notice from Eq. (7.2) that the connection (synapse) between input p_j and output a_i is the weight w_{ij}. Therefore Hebb's postulate would imply that if a positive p_j produces a positive a_i then w_{ij} should increase. This suggests that one mathematical interpretation of the postulate could be

The Hebb Rule

$$w_{ij}^{new} = w_{ij}^{old} + \alpha f_i(a_{iq}) g_j(p_{jq}), \tag{7.4}$$

where p_{jq} is the jth element of the qth input vector $\mathbf{p}_q$; a_{iq} is the ith element of the network output when the qth input vector is presented to the network; and α is a positive constant, called the learning rate. This equation says that the change in the weight w_{ij} is proportional to a product of functions of the activities on either side of the synapse. For this chapter we will simplify Eq. (7.4) to the following form

$$w_{ij}^{new} = w_{ij}^{old} + \alpha a_{iq} p_{jq}. \tag{7.5}$$

Note that this expression actually extends Hebb's postulate beyond its strict interpretation. The change in the weight is proportional to a product of the activity on either side of the synapse. Therefore, not only do we increase the weight when both p_j and a_i are positive, but we also increase the weight when they are both negative. In addition, this implementation of the Hebb rule will decrease the weight whenever p_j and a_i have opposite sign.

The Hebb rule defined in Eq. (7.5) is an *unsupervised* learning rule. It does not require any information concerning the target output. In this chapter we are interested in using the Hebb rule for supervised learning, in which the target output is known for each input vector. (We will revisit the unsupervised Hebb rule in Chapter 13.) For the *supervised* Hebb rule we substitute the target output for the actual output. In this way, we are telling the algorithm what the network *should* do, rather than what it is currently doing. The resulting equation is

$$w_{ij}^{new} = w_{ij}^{old} + t_{iq} p_{jq}, \tag{7.6}$$

where t_{iq} is the ith element of the qth target vector $\mathbf{t}_q$. (We have set the learning rate α to one, for simplicity.)

Notice that Eq. (7.6) can be written in vector notation:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{t}_q \mathbf{p}_q^T. \tag{7.7}$$

If we assume that the weight matrix is initialized to zero and then each of the Q input/output pairs are applied once to Eq. (7.7), we can write

$$\mathbf{W} = \mathbf{t}_1\mathbf{p}_1^T + \mathbf{t}_2\mathbf{p}_2^T + \cdots + \mathbf{t}_Q\mathbf{p}_Q^T = \sum_{q=1}^{Q} \mathbf{t}_q\mathbf{p}_q^T. \tag{7.8}$$

This can be represented in matrix form:

$$\mathbf{W} = \begin{bmatrix} \mathbf{t}_1 & \mathbf{t}_2 & \cdots & \mathbf{t}_Q \end{bmatrix} \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \vdots \\ \mathbf{p}_Q^T \end{bmatrix} = \mathbf{T}\mathbf{P}^T, \tag{7.9}$$

where

$$\mathbf{T} = \begin{bmatrix} \mathbf{t}_1 & \mathbf{t}_2 & \cdots & \mathbf{t}_Q \end{bmatrix}, \quad \mathbf{P} = \begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \cdots & \mathbf{p}_Q \end{bmatrix}. \tag{7.10}$$

Performance Analysis

Let's analyze the performance of Hebbian learning for the linear associator. First consider the case where the $\mathbf{p}_q$ vectors are orthonormal (orthogonal and unit length). If $\mathbf{p}_k$ is input to the network, then the network output can be computed

$$\mathbf{a} = \mathbf{W}\mathbf{p}_k = \left(\sum_{q=1}^{Q} \mathbf{t}_q\mathbf{p}_q^T \right)\mathbf{p}_k = \sum_{q=1}^{Q} \mathbf{t}_q(\mathbf{p}_q^T\mathbf{p}_k). \tag{7.11}$$

Since the $\mathbf{p}_q$ are orthonormal,

$$\begin{aligned} (\mathbf{p}_q^T\mathbf{p}_k) &= 1 \qquad q = k \\ &= 0 \qquad q \neq k. \end{aligned} \tag{7.12}$$

Therefore Eq. (7.11) can be rewritten

$$\mathbf{a} = \mathbf{W}\mathbf{p}_k = \mathbf{t}_k. \tag{7.13}$$

The output of the network is equal to the target output. This shows that, if the input prototype vectors are orthonormal, the Hebb rule will produce the correct output for each input.

But what about non-orthogonal prototype vectors? Let's assume that each $\mathbf{p}_q$ vector is unit length, but that they are not orthogonal. Then Eq. (7.11) becomes

$$\mathbf{a} = \mathbf{W}\mathbf{p}_k = \mathbf{t}_k + \overbrace{\sum_{q \neq k} \mathbf{t}_q(\mathbf{p}_q^T\mathbf{p}_k)}^{Error}. \tag{7.14}$$

Because the vectors are not orthogonal, the network will not produce the correct output. The magnitude of the error will depend on the amount of correlation between the prototype input patterns.

As an example, suppose that the prototype input/output vectors are

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 0.5 \\ -0.5 \\ 0.5 \\ -0.5 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}\right\} \qquad \left\{\mathbf{p}_2 = \begin{bmatrix} 0.5 \\ 0.5 \\ -0.5 \\ -0.5 \end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}\right\}. \tag{7.15}$$

(Check that the two input vectors are orthonormal.)

The weight matrix would be

$$\mathbf{W} = \mathbf{T}\mathbf{P}^T = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} 0.5 & -0.5 & 0.5 & -0.5 \\ 0.5 & 0.5 & -0.5 & -0.5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix}. \tag{7.16}$$

If we test this weight matrix on the two prototype inputs we find

$$\mathbf{W}\mathbf{p}_1 = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix}\begin{bmatrix} 0.5 \\ -0.5 \\ 0.5 \\ -0.5 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \tag{7.17}$$

and

$$\mathbf{W}\mathbf{p}_2 = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix}\begin{bmatrix} 0.5 \\ 0.5 \\ -0.5 \\ -0.5 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}. \tag{7.18}$$

Success!! The outputs of the network are equal to the targets.

Now let's revisit the *apple* and *orange* recognition problem described in Chapter 3. Recall that the prototype inputs were

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} (orange) \qquad \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} (apple) . \tag{7.19}$$

(Note that they are not orthogonal.) If we normalize these inputs and choose as desired outputs -1 and 1, we obtain

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 0.5774 \\ -0.5774 \\ -0.5774 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} -1 \end{bmatrix} \right\} \qquad \left\{ \mathbf{p}_2 = \begin{bmatrix} 0.5774 \\ 0.5774 \\ -0.5774 \end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix} 1 \end{bmatrix} \right\} . \tag{7.20}$$

Our weight matrix becomes

$$\mathbf{W} = \mathbf{T}\mathbf{P}^T = \begin{bmatrix} -1 & 1 \end{bmatrix} \begin{bmatrix} 0.5774 & -0.5774 & -0.5774 \\ 0.5774 & 0.5774 & -0.5774 \end{bmatrix} = \begin{bmatrix} 0 & 1.1548 & 0 \end{bmatrix} . \tag{7.21}$$

So, if we use our two prototype patterns,

$$\mathbf{W}\mathbf{p}_1 = \begin{bmatrix} 0 & 1.1548 & 0 \end{bmatrix} \begin{bmatrix} 0.5774 \\ -0.5774 \\ -0.5774 \end{bmatrix} = \begin{bmatrix} -0.6668 \end{bmatrix} , \tag{7.22}$$

$$\mathbf{W}\mathbf{p}_2 = \begin{bmatrix} 0 & 1.1548 & 0 \end{bmatrix} \begin{bmatrix} 0.5774 \\ 0.5774 \\ -0.5774 \end{bmatrix} = \begin{bmatrix} 0.6668 \end{bmatrix} . \tag{7.23}$$

The outputs are close, but do not quite match the target outputs.

Pseudoinverse Rule

When the prototype input patterns are not orthogonal, the Hebb rule produces some errors. There are several procedures that can be used to reduce these errors. In this section we will discuss one of those procedures, the pseudoinverse rule.

Recall that the task of the linear associator was to produce an output of $\mathbf{t}_q$ for an input of $\mathbf{p}_q$. In other words,

$$\mathbf{W}\mathbf{p}_q = \mathbf{t}_q \qquad q = 1, 2, \ldots , Q . \tag{7.24}$$

If it is not possible to choose a weight matrix so that these equations are exactly satisfied, then we want them to be approximately satisfied. One approach would be to choose the weight matrix to minimize the following performance index:

$$F(\mathbf{W}) = \sum_{q=1}^{Q} \|\mathbf{t}_q - \mathbf{W}\mathbf{p}_q\|^2 . \tag{7.25}$$

If the prototype input vectors $\mathbf{p}_q$ are orthonormal and we use the Hebb rule to find $\mathbf{W}$, then $F(\mathbf{W})$ will be zero. When the input vectors are not orthogonal and we use the Hebb rule, then $F(\mathbf{W})$ will be not be zero, and it is not clear that $F(\mathbf{W})$ will be minimized. It turns out that the weight matrix that will minimize $F(\mathbf{W})$ is obtained by using the pseudoinverse matrix, which we will define next.

First, let's rewrite Eq. (7.24) in matrix form:

$$\mathbf{W}\mathbf{P} = \mathbf{T}, \tag{7.26}$$

where

$$\mathbf{T} = \left[\mathbf{t}_1\ \mathbf{t}_2\ \cdots\ \mathbf{t}_Q\right], \mathbf{P} = \left[\mathbf{p}_1\ \mathbf{p}_2\ \cdots\ \mathbf{p}_Q\right] . \tag{7.27}$$

Then Eq. (7.25) can be written

$$F(\mathbf{W}) = \|\mathbf{T} - \mathbf{W}\mathbf{P}\|^2 = \|\mathbf{E}\|^2, \tag{7.28}$$

where

$$\mathbf{E} = \mathbf{T} - \mathbf{W}\mathbf{P}, \tag{7.29}$$

and

$$\|\mathbf{E}\|^2 = \sum_i \sum_j e_{ij}^2 . \tag{7.30}$$

Note that $F(\mathbf{W})$ can be made zero if we can solve Eq. (7.26). If the $\mathbf{P}$ matrix has an inverse, the solution is

$$\mathbf{W} = \mathbf{T}\mathbf{P}^{-1} . \tag{7.31}$$

However, this is rarely possible. Normally the $\mathbf{p}_q$ vectors (the columns of $\mathbf{P}$) will be independent, but R (the dimension of $\mathbf{p}_q$) will be larger than Q (the number of $\mathbf{p}_q$ vectors). Therefore, $\mathbf{P}$ will not be a square matrix, and no exact inverse will exist.

Pseudoinverse Rule

It has been shown [Albe72] that the weight matrix that minimizes Eq. (7.25) is given by the *pseudoinverse rule*:

$$\mathbf{W} = \mathbf{T}\mathbf{P}^{+}, \tag{7.32}$$

where $\mathbf{P}^{+}$ is the Moore-Penrose pseudoinverse. The pseudoinverse of a real matrix $\mathbf{P}$ is the unique matrix that satisfies

$$\begin{aligned} \mathbf{P}\mathbf{P}^{+}\mathbf{P} &= \mathbf{P}, \\ \mathbf{P}^{+}\mathbf{P}\mathbf{P}^{+} &= \mathbf{P}^{+}, \\ \mathbf{P}^{+}\mathbf{P} &= (\mathbf{P}^{+}\mathbf{P})^{T}, \\ \mathbf{P}\mathbf{P}^{+} &= (\mathbf{P}\mathbf{P}^{+})^{T}. \end{aligned} \tag{7.33}$$

When the number, R, of rows of $\mathbf{P}$ is greater than the number of columns, Q, of $\mathbf{P}$, and the columns of $\mathbf{P}$ are independent, then the pseudoinverse can be computed by

$$\mathbf{P}^{+} = (\mathbf{P}^{T}\mathbf{P})^{-1}\mathbf{P}^{T}. \tag{7.34}$$

To test the pseudoinverse rule (Eq. (7.32)), consider again the apple and orange recognition problem. Recall that the input/output prototype vectors are

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} -1 \end{bmatrix}\right\} \qquad \left\{\mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix} 1 \end{bmatrix}\right\}. \tag{7.35}$$

(Note that we do not need to normalize the input vectors when using the pseudoinverse rule.)

The weight matrix is calculated from Eq. (7.32):

$$\mathbf{W} = \mathbf{T}\mathbf{P}^{+} = \begin{bmatrix} -1 & 1 \end{bmatrix}\left(\begin{bmatrix} 1 & 1 \\ -1 & 1 \\ -1 & -1 \end{bmatrix}\right)^{+}, \tag{7.36}$$

where the pseudoinverse is computed from Eq. (7.34):

$$\mathbf{P}^{+} = (\mathbf{P}^{T}\mathbf{P})^{-1}\mathbf{P}^{T} = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}^{-1}\begin{bmatrix} 1 & -1 & -1 \\ 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 0.25 & -0.5 & -0.25 \\ 0.25 & 0.5 & -0.25 \end{bmatrix}. \tag{7.37}$$

This produces the following weight matrix:

$$\mathbf{W} = \mathbf{TP}^{+} = \begin{bmatrix} -1 & 1 \end{bmatrix} \begin{bmatrix} 0.25 & -0.5 & -0.25 \\ 0.25 & 0.5 & -0.25 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}. \tag{7.38}$$

Let's try this matrix on our two prototype patterns.

$$\mathbf{W}\mathbf{p}_1 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} -1 \end{bmatrix} \tag{7.39}$$

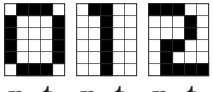
$$\mathbf{W}\mathbf{p}_2 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \end{bmatrix} \tag{7.40}$$

The network outputs exactly match the desired outputs. Compare this result with the performance of the Hebb rule. As you can see from Eq. (7.22) and Eq. (7.23), the Hebbian outputs are only close, while the pseudoinverse rule produces exact results.

Application

Autoassociative Memory

Now let's see how we might use the Hebb rule on a practical, although greatly oversimplified, pattern recognition problem. For this problem we will use a special type of associative memory — the autoassociative memory. In an *autoassociative memory* the desired output vector is equal to the input vector (i.e., $\mathbf{t}_q = \mathbf{p}_q$). We will use an autoassociative memory to store a set of patterns and then to recall these patterns, even when corrupted patterns are provided as input.



$\mathbf{p}_1,\mathbf{t}_1$ $\mathbf{p}_2,\mathbf{t}_2$ $\mathbf{p}_3,\mathbf{t}_3$

The patterns we want to store are shown to the left. (Since we are designing an autoassociative memory, these patterns represent the input vectors and the targets.) They represent the digits {0, 1, 2} displayed in a 6X5 grid. We need to convert these digits to vectors, which will become the prototype patterns for our network. Each white square will be represented by a "-1", and each dark square will be represented by a "1". Then, to create the input vectors, we will scan each 6X5 grid one column at a time. For example, the first prototype pattern will be

$$\mathbf{p}_1 = \begin{bmatrix} -1 & 1 & 1 & 1 & 1 & -1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & -1 & \ldots & 1 & -1 \end{bmatrix}^T. \tag{7.41}$$

The vector $\mathbf{p}_1$ corresponds to the digit "0", $\mathbf{p}_2$ to the digit "1", and $\mathbf{p}_3$ to the digit "2". Using the Hebb rule, the weight matrix is computed

$$\mathbf{W} = \mathbf{p}_1\mathbf{p}_1^T + \mathbf{p}_2\mathbf{p}_2^T + \mathbf{p}_3\mathbf{p}_3^T . \tag{7.42}$$

(Note that $\mathbf{p}_q$ replaces $\mathbf{t}_q$ in Eq. (7.8), since this is autoassociative memory.)

Because there are only two allowable values for the elements of the prototype vectors, we will modify the linear associator so that its output elements can only take on values of "-1" or "1". We can do this by replacing the linear transfer function with a symmetrical hard limit transfer function. The resulting network is displayed in Figure 7.2.

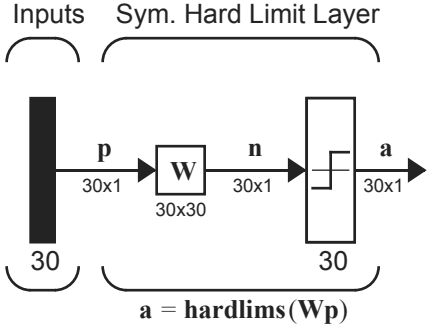


Figure 7.2 Autoassociative Network for Digit Recognition

Now let's investigate the operation of this network. We will provide the network with corrupted versions of the prototype patterns and then check the network output. In the first test, which is shown in Figure 7.3, the network is presented with a prototype pattern in which the lower half of the pattern is occluded. In each case the correct pattern is produced by the network.

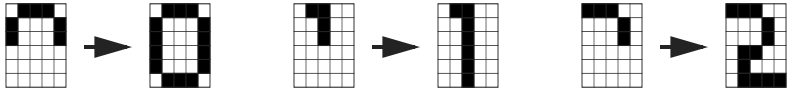


Figure 7.3 Recovery of 50% Occluded Patterns

In the next test we remove even more of the prototype patterns. Figure 7.4 illustrates the result of removing the lower two-thirds of each pattern. In this case only the digit "1" is recovered correctly. The other two patterns produce results that do not correspond to any of the prototype patterns. This is a common problem in associative memories. We would like to design networks so that the number of such spurious patterns would be minimized. We will come back to this topic again in Chapter 18, when we discuss recurrent associative memories.

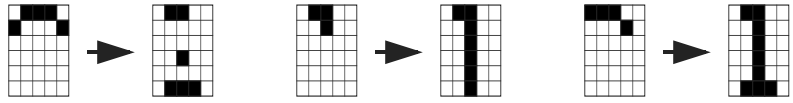


Figure 7.4 Recovery of 67% Occluded Patterns

In our final test we will present the autoassociative network with noisy versions of the prototype pattern. To create the noisy patterns we will randomly change seven elements of each pattern. The results are shown in Figure 7.5. For these examples all of the patterns were correctly recovered.

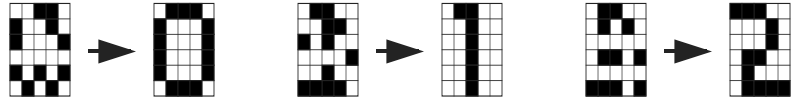


Figure 7.5 Recovery of Noisy Patterns

To experiment with this type of pattern recognition problem, use the Neural Network Design Demonstration Supervised Hebb (**nnd7sh**).

Variations of Hebbian Learning

There have been a number of variations on the basic Hebb rule. In fact, many of the learning laws that will be discussed in the remainder of this text have some relationship to the Hebb rule.

One of the problems of the Hebb rule is that it can lead to weight matrices having very large elements if there are many prototype patterns in the training set. Consider again the basic rule:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{t}_q\mathbf{p}_q^T. \tag{7.43}$$

A positive parameter α, called the learning rate, can be used to limit the amount of increase in the weight matrix elements, if the learning rate is less than one, as in:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha\mathbf{t}_q\mathbf{p}_q^T. \tag{7.44}$$

We can also add a decay term, so that the learning rule behaves like a smoothing filter, remembering the most recent inputs more clearly:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha\mathbf{t}_q\mathbf{p}_q^T - \gamma\mathbf{W}^{old} = (1-\gamma)\mathbf{W}^{old} + \alpha\mathbf{t}_q\mathbf{p}_q^T, \tag{7.45}$$

where γ is a positive constant less than one. As γ approaches zero, the learning law becomes the standard rule. As γ approaches one, the learning

law quickly forgets old inputs and remembers only the most recent patterns. This keeps the weight matrix from growing without bound.

The idea of filtering the weight changes and of having an adjustable learning rate are important ones, and we will discuss them again in Chapters 10, 12, 15, 16, 18 and 19.

If we modify Eq. (7.44) by replacing the desired output with the difference between the desired output and the actual output, we get another important learning rule:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha(\mathbf{t}_q - \mathbf{a}_q)\mathbf{p}_q^T. \tag{7.46}$$

This is sometimes known as the delta rule, since it uses the difference between desired and actual output. It is also known as the Widrow-Hoff algorithm, after the researchers who introduced it. The delta rule adjusts the weights so as to minimize the mean square error (see Chapter 10). For this reason it will produce the same results as the pseudoinverse rule, which minimizes the sum of squares of errors (see Eq. (7.25)). The advantage of the delta rule is that it can update the weights after each new input pattern is presented, whereas the pseudoinverse rule computes the weights in one step, after all of the input/target pairs are known. This sequential updating allows the delta rule to adapt to a changing environment. The delta rule will be discussed in detail in Chapter 10.

The basic Hebb rule will be discussed again, in a different context, in Chapter 13. In the present chapter we have used a supervised form of the Hebb rule. We have assumed that the desired outputs of the network, $\mathbf{t}_q$, are known, and can be used in the learning rule. In the unsupervised Hebb rule, which is discussed in Chapter 13, the *actual* network output is used instead of the *desired* network output, as in:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha\mathbf{a}_q\mathbf{p}_q^T, \tag{7.47}$$

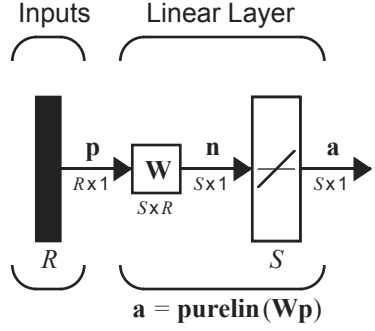
where $\mathbf{a}_q$ is the output of the network when $\mathbf{p}_q$ is given as the input (see also Eq. (7.5)). This unsupervised form of the Hebb rule, which does not require knowledge of the desired output, is actually a more direct interpretation of Hebb's postulate than is the supervised form discussed in this chapter.

Summary of Results

Hebb's Postulate

"*When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.*"

Linear Associator



The Hebb Rule

$$w_{ij}^{new} = w_{ij}^{old} + t_{qi}p_{qj}$$

$$\mathbf{W} = \mathbf{t}_1\mathbf{p}_1^T + \mathbf{t}_2\mathbf{p}_2^T + \cdots + \mathbf{t}_Q\mathbf{p}_Q^T$$

$$\mathbf{W} = \begin{bmatrix}\mathbf{t}_1 & \mathbf{t}_2 & \cdots & \mathbf{t}_Q\end{bmatrix}\begin{bmatrix}\mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \vdots \\ \mathbf{p}_Q^T\end{bmatrix} = \mathbf{T}\mathbf{P}^T$$

Pseudoinverse Rule

$$\mathbf{W} = \mathbf{T}\mathbf{P}^+$$

When the number, R, of rows of $\mathbf{P}$ is greater than the number of columns, Q, of $\mathbf{P}$ and the columns of $\mathbf{P}$ are independent, then the pseudoinverse can be computed by

$$\mathbf{P}^{+} = (\mathbf{P}^{T}\mathbf{P})^{-1}\mathbf{P}^{T}.$$

Variations of Hebbian Learning

Filtered Learning

(See Chapter 14)

$$\mathbf{W}^{new} = (1-\gamma)\mathbf{W}^{old} + \alpha\mathbf{t}_q\mathbf{p}_q^T$$

Delta Rule

(See Chapter 10)

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha(\mathbf{t}_q - \mathbf{a}_q)\mathbf{p}_q^T$$

Unsupervised Hebb

(See Chapter 13)

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha\mathbf{a}_q\mathbf{p}_q^T$$

Solved Problems

P7.1 Consider the linear associator shown in Figure P7.1.

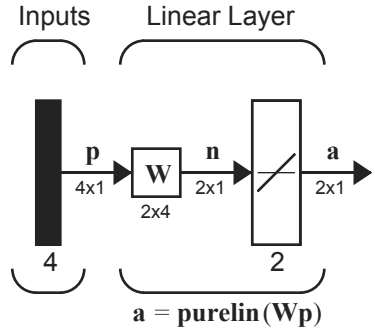


Figure P7.1 Single-Neuron Perceptron

Let the input/output prototype vectors be

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}\right\} \qquad \left\{\mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}\right\}.$$

i. Use the Hebb rule to find the appropriate weight matrix for this linear associator.

ii. Repeat part (i) using the pseudoinverse rule.

iii. Apply the input $\mathbf{p}_1$ to the linear associator using the weight matrix of part (i), then using the weight matrix of part (ii).

i. The first step is to create the **P** and **T** matrices of Eq. (7.10):

$$\mathbf{P} = \begin{bmatrix} 1 & 1 \\ -1 & 1 \\ 1 & -1 \\ -1 & -1 \end{bmatrix}, \qquad \mathbf{T} = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}.$$

Then the weight matrix can be computed using Eq. (7.9):

$$\mathbf{W}^h = \mathbf{T}\mathbf{P}^T = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & -2 \\ 0 & 2 & -2 & 0 \end{bmatrix}.$$

ii. For the pseudoinverse rule we use Eq. (7.32):

$$\mathbf{W} = \mathbf{T}\mathbf{P}^+.$$

Since the number of rows of $\mathbf{P}$, four, is greater than the number of columns of $\mathbf{P}$, two, and the columns of $\mathbf{P}$ are independent, then the pseudoinverse can be computed by Eq. (7.34):

$$\mathbf{P}^+ = (\mathbf{P}^T\mathbf{P})^{-1}\mathbf{P}^T.$$

$$\mathbf{P}^+ = \left(\begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ -1 & 1 \\ 1 & -1 \\ -1 & -1 \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \end{bmatrix} = \left(\begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{4} & 0 \\ 0 & \frac{1}{4} \end{bmatrix} \begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \end{bmatrix}$$

The weight matrix can now be computed:

$$\mathbf{W}^p = \mathbf{T}\mathbf{P}^+ = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & -\frac{1}{2} \\ 0 & \frac{1}{2} & -\frac{1}{2} & 0 \end{bmatrix}.$$

iii. We now test the two weight matrices.

$$\mathbf{W}^h\mathbf{p}_1 = \begin{bmatrix} 2 & 0 & 0 & -2 \\ 0 & 2 & -2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 4 \\ -4 \end{bmatrix} \neq \mathbf{t}_1$$

$$\mathbf{W}^p\mathbf{p}_1 = \begin{bmatrix} \frac{1}{2} & 0 & 0 & -\frac{1}{2} \\ 0 & \frac{1}{2} & -\frac{1}{2} & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \mathbf{t}_1$$

Why didn't the Hebb rule produce the correct results? Well, consider again Eq. (7.11). Since $\mathbf{p}_1$ and $\mathbf{p}_2$ are orthogonal (check that they are) this equation can be written

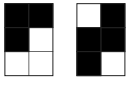
$$\mathbf{W}^h\mathbf{p}_1 = \mathbf{t}_1(\mathbf{p}_1^T\mathbf{p}_1),$$

but the $\mathbf{p}_1$ vector is not normalized, so $(\mathbf{p}_1^T\mathbf{p}_1) \neq 1$. Therefore the output of the network will not be $\mathbf{t}_1$.

The pseudoinverse rule, on the other hand, is guaranteed to minimize

$$\sum_{q=1}^{2} \|\mathbf{t}_q - \mathbf{W}\mathbf{p}_q\|^2,$$

which in this case can be made equal to zero.



$\mathbf{p}_1$ $\mathbf{p}_2$

P7.2 Consider the prototype patterns shown to the left.

i. Are these patterns orthogonal?

ii. Design an autoassociator for these patterns. Use the Hebb rule.

$\mathbf{p}_t$

iii. What response does the network give to the test input pattern, $\mathbf{p}_t$, shown to the left?

i. The first thing we need to do is to convert the patterns into vectors. Let's assign any solid square the value 1 and any open square the value -1. Then to convert from the two-dimensional pattern to a vector we will scan the pattern column by column. (We could use rows if we wished.) The two prototype vectors then become:

$$\mathbf{p}_1 = \begin{bmatrix} 1 & 1 & -1 & 1 & -1 & -1 \end{bmatrix}^T \qquad \mathbf{p}_2 = \begin{bmatrix} -1 & 1 & 1 & 1 & 1 & -1 \end{bmatrix}^T.$$

To test orthogonality we take the inner product of $\mathbf{p}_1$ and $\mathbf{p}_2$:

$$\mathbf{p}_1^T\mathbf{p}_2 = \begin{bmatrix} 1 & 1 & -1 & 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \\ 1 \\ -1 \end{bmatrix} = 0 \cdot$$

Therefore they are orthogonal. (Although they are not normalized since

$$\mathbf{p}_1^T\mathbf{p}_1 = \mathbf{p}_2^T\mathbf{p}_2 = 6 .)$$

ii. We will use an autoassociator like the one in Figure 7.2, except that the number of inputs and outputs to the network will be six. To find the weight matrix we use the Hebb rule:

$$\mathbf{W} = \mathbf{T}\mathbf{P}^T ,$$

where

$$\mathbf{P} = \mathbf{T} = \begin{bmatrix} 1 & -1 \\ 1 & 1 \\ -1 & 1 \\ 1 & 1 \\ -1 & 1 \\ -1 & -1 \end{bmatrix} .$$

Therefore the weight matrix is

$$\mathbf{W} = \mathbf{T}\mathbf{P}^T = \begin{bmatrix} 1 & -1 \\ 1 & 1 \\ -1 & 1 \\ 1 & 1 \\ -1 & 1 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 1 & -1 & 1 & -1 & -1 \\ -1 & 1 & 1 & 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & -2 & 0 & -2 & 0 \\ 0 & 2 & 0 & 2 & 0 & -2 \\ -2 & 0 & 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 & 0 & -2 \\ -2 & 0 & 2 & 0 & 2 & 0 \\ 0 & -2 & 0 & -2 & 0 & 2 \end{bmatrix} .$$

iii. To apply the test pattern to the network we convert it to a vector:

$$\mathbf{p}_t = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & -1 \end{bmatrix}^T .$$

The network response is then

$$\mathbf{a} = \mathbf{hardlims}(\mathbf{W}\mathbf{p}_t) = \mathbf{hardlims}\left(\begin{bmatrix} 2 & 0 & -2 & 0 & -2 & 0 \\ 0 & 2 & 0 & 2 & 0 & -2 \\ -2 & 0 & 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 & 0 & -2 \\ -2 & 0 & 2 & 0 & 2 & 0 \\ 0 & -2 & 0 & -2 & 0 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ -1 \end{bmatrix}\right)$$

$$\mathbf{a} = \mathbf{hardlims}\left(\begin{bmatrix} -2 \\ 6 \\ 2 \\ 6 \\ 2 \\ -6 \end{bmatrix}\right) = \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \\ 1 \\ -1 \end{bmatrix} = \mathbf{p}_2 .$$

Is this a satisfactory response? How would we want the network to respond to this input pattern? The network should produce the prototype pattern that is closest to the input pattern. In this case the test input pattern, $\mathbf{p}_t$, has a Hamming distance of 1 from $\mathbf{p}_2$, and a distance of 2 from $\mathbf{p}_1$. Therefore the network did produce the correct response. (See Chapter 3 for a discussion of Hamming distance.)

Note that in this example the prototype vectors were not normalized. This did not cause the same problem with network performance that we saw in Problem P7.1, because of the *hardlims* nonlinearity. It forces the network output to be either 1 or -1. In fact, most of the interesting and useful properties of neural networks are due to the effects of nonlinearities.

P7.3 Consider an autoassociation problem in which there are three prototype patterns (shown below as $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$). Design autoassociative networks to recognize these patterns, using both the Hebb rule and the pseudoinverse rule. Check their performance on the test pattern $\mathbf{p}_t$ shown below.

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \\ 1 \\ -1 \\ 1 \end{bmatrix} \quad \mathbf{p}_3 = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \\ 1 \\ -1 \\ 1 \end{bmatrix} \quad \mathbf{p}_t = \begin{bmatrix} -1 \\ 1 \\ -1 \\ -1 \\ 1 \\ -1 \\ 1 \end{bmatrix}$$

This problem is a little tedious to work out by hand, so let's use MATLAB. First we create the prototype vectors.

```
p1=[ 1  1 -1 -1  1  1  1]';

p2=[ 1  1  1 -1  1  -1  1]';

p3=[-1  1 -1  1  1 -1  1]';

P=[p1 p2 p3];
```

Now we can compute the weight matrix using the Hebb rule.

```
wh=P*P';
```

To check the network we create the test vector.

```
pt=[-1  1 -1 -1  1 -1  1]';
```

The network response is then calculated.

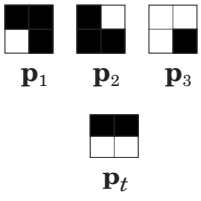
```
ah=hardlims(wh*pt);

ah'

ans =

     1     1    -1    -1     1    -1     1
```

Notice that this response does not match any of the prototype vectors. This is not surprising since the prototype patterns are not orthogonal. Let's try the pseudoinverse rule.

```
pseu=inv(P'*P)*P';

wp=P*pseu;

ap=hardlims(wp*pt);

ap'

ans =

    -1     1    -1     1     1    -1     1
```

Note that the network response is equal to $\mathbf{p}_3$. Is this the correct response? As usual, we want the response to be the prototype pattern closest to the input pattern. In this case $\mathbf{p}_t$ is a Hamming distance of 2 from both $\mathbf{p}_1$ and $\mathbf{p}_2$, but only a distance of 1 from $\mathbf{p}_3$. Therefore the pseudoinverse rule produces the correct response.

Try other test inputs to see if there are additional cases where the pseudoinverse rule produces better results than the Hebb rule.



P7.4 Consider the three prototype patterns shown to the left.

i. Use the Hebb rule to design a perceptron network that will recognize these three patterns.

ii. Find the response of the network to the pattern $\mathbf{p}_t$ shown to the left. Is the response correct?

i. We can convert the patterns to vectors, as we did in previous problems, to obtain:

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix} \quad \mathbf{p}_3 = \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix} \quad \mathbf{p}_t = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}.$$

We now need to choose the desired output vectors for each prototype input vector. Since there are three prototype vectors that we need to distinguish, we will need two elements in the output vector. We can choose the three desired outputs to be:

$$\mathbf{t}_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \quad \mathbf{t}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \quad \mathbf{t}_3 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

(Note that this choice was arbitrary. Any distinct combination of 1 and -1 could have been chosen for each vector.)

The resulting network is shown in Figure P7.2.

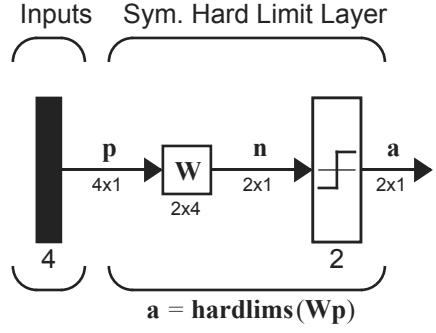


Figure P7.2 Perceptron Network for Problem P7.4

The next step is to determine the weight matrix using the Hebb rule.

$$\mathbf{W} = \mathbf{T}\mathbf{P}^T = \begin{bmatrix} -1 & -1 & 1 \\ -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ -1 & -1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} -3 & -1 & -1 & -1 \\ 1 & 3 & -1 & -1 \end{bmatrix}$$

ii. The response of the network to the test input pattern is calculated as follows.

$$\mathbf{a} = \mathbf{hardlims}(\mathbf{W}\mathbf{p}_t) = \mathbf{hardlims}\left(\begin{bmatrix} -3 & -1 & -1 & -1 \\ 1 & 3 & -1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \right)$$

$$= \mathbf{hardlims}\left(\begin{bmatrix} -2 \\ -2 \end{bmatrix} \right) = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \rightarrow \mathbf{p}_1 .$$

So the response of the network indicates that the test input pattern is closest to $\mathbf{p}_1$. Is this correct? Yes, the Hamming distance to $\mathbf{p}_1$ is 1, while the distance to $\mathbf{p}_2$ and $\mathbf{p}_3$ is 3.

P7.5 Suppose that we have a linear autoassociator that has been designed for Q orthogonal prototype vectors of length R using the Hebb rule. The vector elements are either 1 or -1.

i. Show that the Q prototype patterns are eigenvectors of the weight matrix.

ii. What are the other (R - Q) eigenvectors of the weight matrix?

i. Suppose the prototype vectors are:

$$\mathbf{p}_1, \mathbf{p}_2, \cdots, \mathbf{p}_Q .$$

Since this is an autoassociator, these are both the input vectors and the desired output vectors. Therefore

$$\mathbf{T} = \begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \cdots & \mathbf{p}_Q \end{bmatrix} \qquad \mathbf{P} = \begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \cdots & \mathbf{p}_Q \end{bmatrix} .$$

If we then use the Hebb rule to calculate the weight matrix we find

$$\mathbf{W} = \mathbf{T}\mathbf{P}^T = \sum_{q=1}^{Q} \mathbf{p}_q \mathbf{p}_q^T ,$$

from Eq. (7.8). Now, if we apply one of the prototype vectors as input to the network we obtain

$$\mathbf{a} = \mathbf{W}\mathbf{p}_k = \left(\sum_{q=1}^{Q} \mathbf{p}_q \mathbf{p}_q^T \right) \mathbf{p}_k = \sum_{q=1}^{Q} \mathbf{p}_q (\mathbf{p}_q^T \mathbf{p}_k) .$$

Because the patterns are orthogonal, this reduces to

$$\mathbf{a} = \mathbf{p}_k (\mathbf{p}_k^T \mathbf{p}_k) .$$

And since every element of $\mathbf{p}_k$ must be either -1 or 1, we find that

$$\mathbf{a} = \mathbf{p}_k R .$$

To summarize the results:

$$\mathbf{W}\mathbf{p}_k = R\mathbf{p}_k ,$$

which implies that $\mathbf{p}_k$ is an eigenvector of $\mathbf{W}$ and R is the corresponding eigenvalue. Each prototype vector is an eigenvector with the same eigenvalue.

ii. Note that the repeated eigenvalue R has a Q-dimensional eigenspace associated with it: the subspace spanned by the Q prototype vectors. Now consider the subspace that is orthogonal to this eigenspace. Every vector in this subspace should be orthogonal to each prototype vector. The dimension of the orthogonal subspace will be $R - Q$. Consider the following arbitrary basis set for this orthogonal space:

$$\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_{R-Q} .$$

If we apply any one of these basis vectors to the network we obtain:

$$\mathbf{a} = \mathbf{W}\mathbf{z}_k = \left(\sum_{q=1}^{Q} \mathbf{p}_q \mathbf{p}_q^T \right) \mathbf{z}_k = \sum_{q=1}^{Q} \mathbf{p}_q (\mathbf{p}_q^T \mathbf{z}_k) = 0 ,$$

since each $\mathbf{z}_k$ is orthogonal to every $\mathbf{p}_q$. This implies that each $\mathbf{z}_k$ is an eigenvector of $\mathbf{W}$ with eigenvalue 0.

To summarize, the weight matrix $\mathbf{W}$ has two eigenvalues, R and 0. This means that any vector in the space spanned by the prototype vectors will be amplified by R, whereas any vector that is orthogonal to the prototype vectors will be set to 0. We will revisit this concept when we discuss the performance of the Hopfield network in Chapter 18.

P7.6 The networks we have used so far in this chapter have not included a bias vector. Consider the problem of designing a perceptron network (Figure P7.3) to recognize the following patterns:

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \mathbf{p}_2 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}.$$

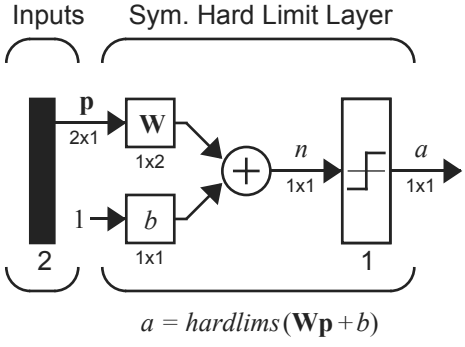


Figure P7.3 Single-Neuron Perceptron

i. Why is a bias required to solve this problem?

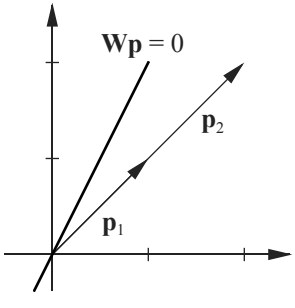
ii. Use the pseudoinverse rule to design a network with bias to solve this problem.

i. Recall from Chapters 3 and 4 that the decision boundary for the perceptron network is the line defined by:

$$\mathbf{W}\mathbf{p} + b = 0.$$

If there is no bias, then $b = 0$ and the boundary is defined by:

$$\mathbf{W}\mathbf{p} = 0,$$



which is a line that must pass through the origin. Now consider the two vectors, $\mathbf{p}_1$ and $\mathbf{p}_2$, which are given in this problem. They are shown graphically in the figure to the left, along with an arbitrary decision boundary that passes through the origin. It is clear that no decision boundary that passes through the origin could separate these two vectors. Therefore a bias is required to solve this problem.

ii. To use the pseudoinverse rule (or the Hebb rule) when there is a bias term, we should treat the bias as another weight, with an input of 1 (as is shown in all of the network figures). We then augment the input vectors with a 1 as the last element:

$$\mathbf{p}'_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \qquad \mathbf{p}'_2 = \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}.$$

Let's choose the desired outputs to be

$$t_1 = 1 \qquad t_2 = -1,$$

so that

$$\mathbf{P} = \begin{bmatrix} 1 & 2 \\ 1 & 2 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{T} = \begin{bmatrix} 1 & -1 \end{bmatrix}.$$

We now form the pseudoinverse matrix:

$$\mathbf{P}^+ = \left(\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 1 & 2 \\ 1 & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 5 \\ 5 & 9 \end{bmatrix}^{-1} \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 1 \end{bmatrix} = \begin{bmatrix} -0.5 & -0.5 & 2 \\ 0.5 & 0.5 & -1 \end{bmatrix}.$$

The augmented weight matrix is then computed:

$$\mathbf{W}' = \mathbf{T}\mathbf{P}^+ = \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} -0.5 & -0.5 & 2 \\ 0.5 & 0.5 & -1 \end{bmatrix} = \begin{bmatrix} -1 & -1 & 3 \end{bmatrix}.$$

We can then pull out the standard weight matrix and bias:

$$\mathbf{W} = \begin{bmatrix} -1 & -1 \end{bmatrix} \qquad b = 3.$$

The decision boundary for this weight and bias is shown in the Figure P7.4. This boundary does separate the two prototype vectors.

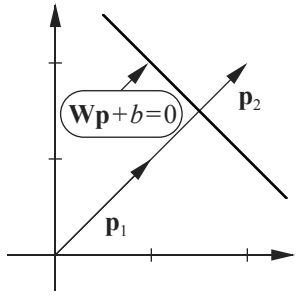


Figure P7.4 Decision Boundary for Solved Problem P7.6

P7.7 In all of our pattern recognition examples thus far, we have represented patterns as vectors by using "1" and "-1" to represent dark and light pixels (picture elements), respectively. What if we were to use "1" and "0" instead? How should the Hebb rule be changed?

First, let's introduce some notation to distinguish the two different representations (usually referred to as the bipolar {-1, 1} representation and the binary {0, 1} representation). The bipolar representation of the prototype input/output vectors will be denoted

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\},$$

and the binary representation will be denoted

$$\{\mathbf{p}'_1, \mathbf{t}'_1\}, \{\mathbf{p}'_2, \mathbf{t}'_2\}, \ldots, \{\mathbf{p}'_Q, \mathbf{t}'_Q\}.$$

The relationship between the two representations is given by:

$$\mathbf{p}'_q = \frac{1}{2}\mathbf{p}_q + \frac{1}{2}\mathbf{1} \qquad \mathbf{p}_q = 2\mathbf{p}'_q - \mathbf{1},$$

where $\mathbf{1}$ is a vector of ones.

Next, we determine the form of the binary associative network. We will use the network shown in Figure P7.5. It is different than the bipolar associative network, as shown in Figure 7.2, in two ways. First, it uses the *hardlim* nonlinearity rather than *hardlims*, since the output should be either 0 or 1. Secondly, it uses a bias vector. It requires a bias vector because all binary vectors will fall into one quadrant of the vector space, so a boundary that passes through the origin will not always be able to divide the patterns. (See Problem P7.6.)

The next step is to determine the weight matrix and the bias vector for this network. If we want the binary network of Figure P7.5 to have the same

effective response as a bipolar network (as in Figure 7.2), then the net input, $\mathbf{n}$, should be the same for both networks:

$$\mathbf{W}'\mathbf{p}' + \mathbf{b} = \mathbf{W}\mathbf{p}.$$

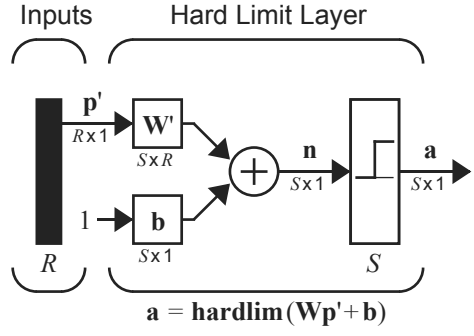


Figure P7.5 Binary Associative Network

This will guarantee that whenever the bipolar network produces a “1” the binary network will produce a “1”, and whenever the bipolar network produces a “-1” the binary network will produce a “0”.

If we then substitute for $\mathbf{p}'$ as a function of $\mathbf{p}$ we find:

$$\mathbf{W}'\left(\frac{1}{2}\mathbf{p} + \frac{1}{2}\mathbf{1}\right) + \mathbf{b} = \frac{1}{2}\mathbf{W}'\mathbf{p} + \frac{1}{2}\mathbf{W}'\mathbf{1} + \mathbf{b} = \mathbf{W}\mathbf{p}.$$

Therefore, to produce the same results as the bipolar network, we should choose

$$\mathbf{W}' = 2\mathbf{W} \qquad \mathbf{b} = -\mathbf{W}\mathbf{1},$$

where $\mathbf{W}$ is the bipolar weight matrix.

Epilogue

We had two main objectives for this chapter. First, we wanted to introduce one of the most influential neural network learning rules: the Hebb rule. This was one of the first neural learning rules ever proposed, and yet it continues to influence even the most recent developments in network learning theory. Second, we wanted to show how the performance of this learning rule could be explained using the linear algebra concepts discussed in the two preceding chapters. This is one of the key objectives of this text. We want to show how certain important mathematical concepts underlie the operation of all artificial neural networks. We plan to continue to weave together the mathematical ideas with the neural network applications, and hope in the process to increase our understanding of both.

We will again revisit the Hebb rule in Chapters 15 and 21. In Chapter 21 we will use the Hebb rule in the design of a *recurrent* associative memory network — the Hopfield network.

The next two chapters introduce some mathematics that are critical to our understanding of the two learning laws covered in Chapters 10 and 11. Those learning laws fall under a subheading called *performance* learning, because they attempt to optimize the performance of the network. In order to understand these performance learning laws, we need to introduce some basic concepts in optimization. As with the material on the Hebb rule, our understanding of these topics in optimization will be greatly aided by our previous work in linear algebra.

Further Reading

[Albe72] A. Albert, *Regression and the Moore-Penrose Pseudoinverse*, New York: Academic Press, 1972.

Albert's text is the major reference for the theory and basic properties of the pseudoinverse. Proofs are included for all major pseudoinverse theorems.

[Ande72] J. Anderson, "A simple neural network generating an interactive memory," *Mathematical Biosciences*, vol. 14, pp. 197–220, 1972.

Anderson proposed a "linear associator" model for associative memory. The model was trained, using a generalization of the Hebb postulate, to learn an association between input and output vectors. The physiological plausibility of the network was emphasized. Kohonen published a closely related paper at the same time [Koho72], although the two researchers were working independently.

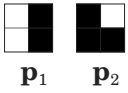
[Hebb49] D. O. Hebb, *The Organization of Behavior*, New York: Wiley, 1949.

The main premise of this seminal book is that behavior can be explained by the action of neurons. In it, Hebb proposes one of the first learning laws, which postulated a mechanism for learning at the cellular level.

[Koho72] T. Kohonen, "Correlation matrix memories," *IEEE Transactions on Computers*, vol. 21, pp. 353–359, 1972.

Kohonen proposed a correlation matrix model for associative memory. The model was trained, using the outer product rule (also known as the Hebb rule), to learn an association between input and output vectors. The mathematical structure of the network was emphasized. Anderson published a closely related paper at the same time [Ande72], although the two researchers were working independently.

Exercises



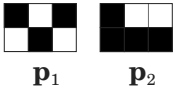
E7.1 Consider the prototype patterns given to the left.

i. Are $\mathbf{p}_1$ and $\mathbf{p}_2$ orthogonal?

ii. Use the Hebb rule to design an autoassociator network for these patterns.

iii. Test the operation of the network using the test input pattern $\mathbf{p}_t$ shown to the left. Does the network perform as you expected? Explain.

E7.2 Repeat Exercise E7.1 using the pseudoinverse rule.



E7.3 Use the Hebb rule to determine the weight matrix for a perceptron network (shown in Figure E7.1) to recognize the patterns shown to the left.

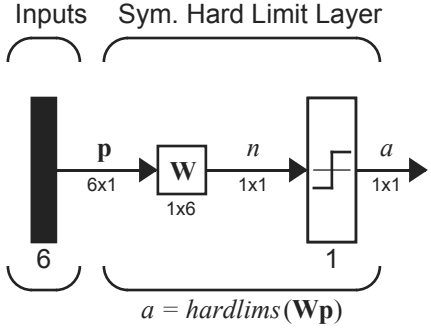


Figure E7.1 Perceptron Network for Exercise E7.3

E7.4 In Problem P7.7 we demonstrated how networks can be trained using the Hebb rule when the prototype vectors are given in binary (as opposed to bipolar) form. Repeat Exercise E7.1 using the binary representation for the prototype vectors. Show that the response of this binary network is equivalent to the response of the original bipolar network.

E7.5 Show that an autoassociator network will continue to perform if we zero the diagonal elements of a weight matrix that has been determined by the Hebb rule. In other words, suppose that the weight matrix is determined from:

$$\mathbf{W} = \mathbf{P}\mathbf{P}^T - Q\mathbf{I},$$

where Q is the number of prototype vectors. (Hint: show that the prototype vectors continue to be eigenvectors of the new weight matrix.)

E7.6 We have three input/output prototype vector pairs:

$$\left\{\mathbf{p}_1 = \begin{bmatrix}1\\0\end{bmatrix}, t_1 = 1\right\}, \left\{\mathbf{p}_2 = \begin{bmatrix}1\\1\end{bmatrix}, t_2 = -1\right\}, \left\{\mathbf{p}_3 = \begin{bmatrix}0\\1\end{bmatrix}, t_3 = 1\right\}.$$

i. Show that this problem cannot be solved unless the network uses a bias.

ii. Use the pseudoinverse rule to design a network for these prototype vectors. Verify that the network correctly transforms the prototype vectors.

E7.7 Consider the reference patterns and targets given below. We want to use these data to train a linear associator network.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\4\end{bmatrix}, t_1 = \begin{bmatrix}26\end{bmatrix}\right\} \qquad \left\{\mathbf{p}_2 = \begin{bmatrix}4\\2\end{bmatrix}, t_2 = \begin{bmatrix}26\end{bmatrix}\right\} \qquad \left\{\mathbf{p}_3 = \begin{bmatrix}-2\\-2\end{bmatrix}, t_3 = \begin{bmatrix}-26\end{bmatrix}\right\}$$

i. Use the Hebb rule to find the weights of the network.

ii. Find and sketch the decision boundary for the network with the Hebb rule weights.

iii. Use the pseudo-inverse rule to find the weights of the network. Because the number, R, of rows of $\mathbf{P}$ is less than the number of columns, Q, of $\mathbf{P}$, the pseudoinverse can be computed by $\mathbf{P}^+ = \mathbf{P}^T(\mathbf{P}\mathbf{P}^T)^{-1}$.

iv. Find and sketch the decision boundary for the network with the pseudo-inverse rule weights.

v. Compare (discuss) the decision boundaries and weights for each of the methods (Hebb and pseudo-inverse).

E7.8 Consider the three prototype patterns shown in Figure E7.2.

i. Are these patterns orthogonal? Demonstrate.

ii. Use the Hebb rule to determine the weight matrix for a linear autoassociator to recognize these patterns.

iii. Draw the network diagram.

iv. Find the eigenvalues and eigenvectors of the weight matrix. (Do **not** solve the equation $|\mathbf{W} - \lambda\mathbf{I}| = 0$. Use an analysis of the Hebb rule.)

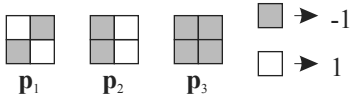


Figure E7.2 Prototype Patterns for Exercise E7.8

E7.9 Suppose that we have the following three reference patterns and their targets.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}3\\6\end{bmatrix}, t_1 = \begin{bmatrix}75\end{bmatrix}\right\} \qquad \left\{\mathbf{p}_2 = \begin{bmatrix}6\\3\end{bmatrix}, t_2 = \begin{bmatrix}75\end{bmatrix}\right\} \qquad \left\{\mathbf{p}_3 = \begin{bmatrix}-6\\3\end{bmatrix}, t_3 = \begin{bmatrix}-75\end{bmatrix}\right\}$$

i. Draw the network diagram for a linear associator network that could be trained on these patterns.

ii. Use the Hebb rule to find the weights of the network.

iii. Find and sketch the decision boundary for the network with the Hebb rule weights. Does the boundary separate the patterns? Demonstrate.

iv. Use the pseudo-inverse rule to find the weights of the network. Describe the difference between this boundary and the Hebb rule boundary.

E7.10 We have the following input/output pairs:

$$\left\{\mathbf{p}_1 = \begin{bmatrix}1\\1\end{bmatrix}, t_1 = \begin{bmatrix}1\end{bmatrix}\right\} \qquad \left\{\mathbf{p}_2 = \begin{bmatrix}1\\-1\end{bmatrix}, t_2 = \begin{bmatrix}-1\end{bmatrix}\right\}$$

i. Use the Hebb rule to determine the weight matrix for the perceptron network shown in Figure E7.3.

ii. Plot the resulting decision boundary. Is this a “good” decision boundary? Explain.

iii. Repeat part i. using the Pseudoinverse rule.

iv. Will there be any difference in the operation of the network if the Pseudoinverse weight matrix is used? Explain.

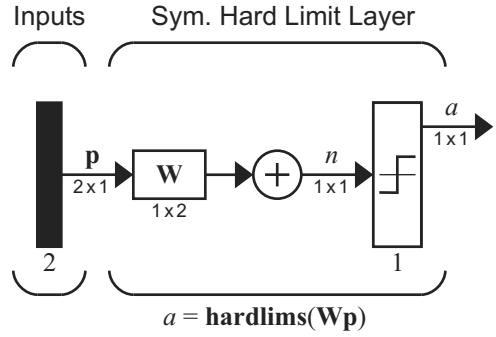


Figure E7.3 Network for Exercise E7.10

E7.11 One question we might ask about the Hebb and pseudoinverse rules is: How many prototype patterns can be stored in one weight matrix? Test this experimentally using the digit recognition problem that was discussed on page 7-10. Begin with the digits “0” and “1”. Add one digit at a time up to “6”, and test how often the correct digit is reconstructed after randomly changing 2, 4 and 6 pixels.

i. First use the Hebb rule to create the weight matrix for the digits “0” and “1”. Then randomly change 2 pixels of each digit and apply the noisy digits to the network. Repeat this process 10 times, and record the percentage of times in which the correct pattern (without noise) is produced at the output of the network. Repeat as 4 and 6 pixels of each digit are modified. The entire process is then repeated when the digits “0”, “1” and “2” are used. This continues, one digit at a time, until you test the network when all of the digits “0” through “6” are used. When you have completed all of the tests, you will be able to plot three curves showing percentage error versus number of digits stored, one curve each for 2, 4 and 6 pixel errors.

ii. Repeat part (i) using the pseudoinverse rule, and compare the results of the two rules.

8 Performance Surfaces and Optimum Points

Objectives

This chapter lays the foundation for a type of neural network training technique called performance learning. There are several different classes of network learning laws, including associative learning (as in the Hebbian learning of Chapter 7) and competitive learning (which we will discuss in Chapter 16). Performance learning is another important class of learning law, in which the network parameters are adjusted to optimize the performance of the network. In the next two chapters we will lay the groundwork for the development of performance learning, which will then be presented in detail in Chapters 10–14. The main objective of the present chapter is to investigate performance surfaces and to determine conditions for the existence of minima and maxima of the performance surface. Chapter 9 will follow this up with a discussion of procedures to locate the minima or maxima.

Theory and Examples

Performance Learning

There are several different learning laws that fall under the category of *performance learning*. Two of these will be presented in this text. These learning laws are distinguished by the fact that during training the network parameters (weights and biases) are adjusted in an effort to optimize the "performance" of the network.

Performance Index

There are two steps involved in this optimization process. The first step is to define what we mean by "performance." In other words, we must find a quantitative measure of network performance, called the *performance index*, which is small when the network performs well and large when the network performs poorly. In this chapter, and in Chapter 9, we will assume that the performance index is given. In Chapters 10, 11 and 13 we will discuss the choice of performance index.

The second step of the optimization process is to search the parameter space (adjust the network weights and biases) in order to reduce the performance index. In this chapter we will investigate the characteristics of performance surfaces and set some conditions that will guarantee that a surface does have a minimum point (the optimum we are searching for). Thus, in this chapter we will obtain some understanding of what performance surfaces look like. Then, in Chapter 9 we will develop procedures for locating the optimum points.

Taylor Series

Taylor Series Expansion

Let us say that the performance index that we want to minimize is represented by $F(x)$, where x is the scalar parameter we are adjusting. We will assume that the performance index is an analytic function, so that all of its derivatives exist. Then it can be represented by its *Taylor series expansion* about some nominal point x^*:

$$F(x) = F(x^*) + \frac{d}{dx}F(x)\Big|_{x=x^*}(x-x^*) + \frac{1}{2}\frac{d^2}{dx^2}F(x)\Big|_{x=x^*}(x-x^*)^2 + \cdots + \frac{1}{n!}\frac{d^n}{dx^n}F(x)\Big|_{x=x^*}(x-x^*)^n + \cdots \tag{8.1}$$

We will use the Taylor series expansion to approximate the performance index, by limiting the expansion to a finite number of terms. For example, let

$$F(x) = \cos(x). \tag{8.2}$$

The Taylor series expansion for $F(x)$ about the point $x^* = 0$ is

$$\begin{aligned} F(x) = \cos(x) &= \cos(0) - \sin(0)(x-0) - \frac{1}{2}\cos(0)(x-0)^2 \\ &\quad + \frac{1}{6}\sin(0)(x-0)^3 + \cdots \\ &= 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4 + \cdots \end{aligned} \tag{8.3}$$

The zeroth-order approximation of $F(x)$ (using only the zeroth power of x) is

$$F(x) \approx F_0(x) = 1. \tag{8.4}$$

The second-order approximation is

$$F(x) \approx F_2(x) = 1 - \frac{1}{2}x^2. \tag{8.5}$$

(Note that in this case the first-order approximation is the same as the zeroth-order approximation, since the first derivative is zero.)

The fourth-order approximation is

$$F(x) \approx F_4(x) = 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4. \tag{8.6}$$

A graph showing $F(x)$ and these three approximations is shown in Figure 8.1.

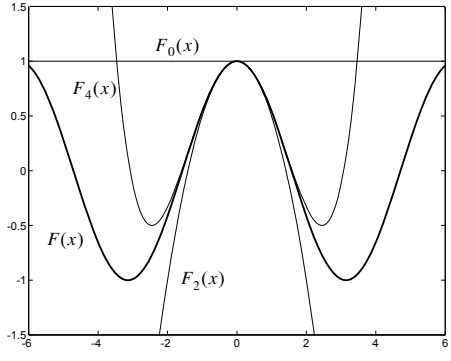


Figure 8.1 Cosine Function and Taylor Series Approximations

From the figure we can see that all three approximations are accurate if x is very close to $x^* = 0$. However, as x moves farther away from x^* only the higher-order approximations are accurate. The second-order approximation is accurate over a wider range than the zeroth-order approximation, and the fourth-order approximation is accurate over a wider range than the second-order approximation. An investigation of Eq. (8.1) explains this behavior. Each succeeding term in the series involves a higher power of $(x - x^*)$. As x gets closer to x^*, these terms will become geometrically smaller.

We will use the Taylor series approximations of the performance index to investigate the shape of the performance index in the neighborhood of possible optimum points.

To experiment with Taylor series expansions of the cosine function, use the MATLAB® Neural Network Design Demonstration Taylor Series (**nnd8ts1**).

Vector Case

Of course the neural network performance index will not be a function of a scalar x. It will be a function of all of the network parameters (weights and biases), of which there may be a very large number. Therefore, we need to extend the Taylor series expansion to functions of many variables. Consider the following function of n variables:

$$F(\mathbf{x}) = F(x_1, x_2, \ldots, x_n). \tag{8.7}$$

The Taylor series expansion for this function, about the point x^*, will be

$$\begin{aligned} F(\mathbf{x}) = F(\mathbf{x}^*) &+ \frac{\partial}{\partial x_1} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_1 - x_1^*) + \frac{\partial}{\partial x_2} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_2 - x_2^*) \\ &+ \cdots + \frac{\partial}{\partial x_n} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_n - x_n^*) + \frac{1}{2}\frac{\partial^2}{\partial x_1^2} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_1 - x_1^*)^2 \\ &+ \frac{1}{2}\frac{\partial^2}{\partial x_1 \partial x_2} F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(x_1 - x_1^*)(x_2 - x_2^*) + \cdots \end{aligned} \tag{8.8}$$

This notation is a bit cumbersome. It is more convenient to write it in matrix form, as in:

$$\begin{aligned} F(\mathbf{x}) = F(\mathbf{x}^*) &+ \nabla F(\mathbf{x})^T\Big|_{\mathbf{x}=\mathbf{x}^*}(\mathbf{x} - \mathbf{x}^*) \\ &+ \frac{1}{2}(\mathbf{x} - \mathbf{x}^*)^T \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(\mathbf{x} - \mathbf{x}^*) + \cdots \end{aligned} \tag{8.9}$$

Gradient

where $\nabla F(\mathbf{x})$ is the *gradient*, and is defined as

$$\nabla F(\mathbf{x}) = \left[\frac{\partial}{\partial x_1}F(\mathbf{x})\ \frac{\partial}{\partial x_2}F(\mathbf{x})\ \ldots\ \frac{\partial}{\partial x_n}F(\mathbf{x})\right]^T, \tag{8.10}$$

Hessian

and $\nabla^2 F(\mathbf{x})$ is the *Hessian*, and is defined as:

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2}{\partial x_1^2}F(\mathbf{x}) & \frac{\partial^2}{\partial x_1 \partial x_2}F(\mathbf{x}) & \ldots & \frac{\partial^2}{\partial x_1 \partial x_n}F(\mathbf{x}) \\ \frac{\partial^2}{\partial x_2 \partial x_1}F(\mathbf{x}) & \frac{\partial^2}{\partial x_2^2}F(\mathbf{x}) & \ldots & \frac{\partial^2}{\partial x_2 \partial x_n}F(\mathbf{x}) \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2}{\partial x_n \partial x_1}F(\mathbf{x}) & \frac{\partial^2}{\partial x_n \partial x_2}F(\mathbf{x}) & \ldots & \frac{\partial^2}{\partial x_n^2}F(\mathbf{x}) \end{bmatrix}. \tag{8.11}$$

The gradient and the Hessian are very important to our understanding of performance surfaces. In the next section we discuss the practical meaning of these two concepts.

To experiment with Taylor series expansions of a function of two variables, use the MATLAB® Neural Network Design Demonstration Vector Taylor Series(`nnd8ts2`).

Directional Derivatives

The ith element of the gradient, $\partial F(\mathbf{x})/\partial x_i$, is the first derivative of the performance index F along the x_i axis. The ith element of the diagonal of the Hessian matrix, $\partial^2 F(\mathbf{x})/\partial x_i^2$, is the second derivative of the performance index F along the x_i axis. What if we want to know the derivative of the function in an arbitrary direction? We let $\mathbf{p}$ be a vector in the direction along which we wish to know the derivative. This *directional derivative* can be computed from

Directional Derivative

$$\frac{\mathbf{p}^T \nabla F(\mathbf{x})}{\|\mathbf{p}\|}. \tag{8.12}$$

The second derivative along $\mathbf{p}$ can also be computed:

$$\frac{\mathbf{p}^T \nabla^2 F(\mathbf{x}) \mathbf{p}}{\|\mathbf{p}\|^2}. \tag{8.13}$$

To illustrate these concepts, consider the function

$$F(\mathbf{x}) = x_1^2 + 2x_2^2 . \tag{8.14}$$

Suppose that we want to know the derivative of the function at the point $\mathbf{x}^* = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}^T$ in the direction $\mathbf{p} = \begin{bmatrix} 2 & -1 \end{bmatrix}^T$. First we evaluate the gradient at $\mathbf{x}^*$:

$$\nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*} = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix}\Bigg|_{\mathbf{x}=\mathbf{x}^*} = \begin{bmatrix} 2x_1 \\ 4x_2 \end{bmatrix}\Bigg|_{\mathbf{x}=\mathbf{x}^*} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \tag{8.15}$$

The derivative in the direction $\mathbf{p}$ can then be computed:

$$\frac{\mathbf{p}^T \nabla F(\mathbf{x})}{\|\mathbf{p}\|} = \frac{\begin{bmatrix} 2 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix}}{\left\| \begin{bmatrix} 2 \\ -1 \end{bmatrix} \right\|} = \frac{\begin{bmatrix} 0 \end{bmatrix}}{\sqrt{5}} = 0 . \tag{8.16}$$

Therefore the function has zero slope in the direction $\mathbf{p}$ from the point $\mathbf{x}^*$. Why did this happen? What can we say about those directions that have zero slope? If we consider the definition of directional derivative in Eq. (8.12), we can see that the numerator is an inner product between the direction vector and the gradient. Therefore any direction that is orthogonal to the gradient will have zero slope.

Which direction has the greatest slope? The maximum slope will occur when the inner product of the direction vector and the gradient is a maximum. This happens when the direction vector is the same as the gradient. (Notice that the magnitude of the direction vector has no effect, since we normalize by its magnitude.) This effect is illustrated in Figure 8.2, which shows a contour plot and a 3-D plot of $F(\mathbf{x})$. On the contour plot we see five vectors starting from our nominal point $\mathbf{x}^*$ and pointing in different directions. At the end of each vector the first directional derivative is displayed. The maximum derivative occurs in the direction of the gradient. The zero derivative is in the direction orthogonal to the gradient (tangent to the contour line).

To experiment with directional derivatives, use the MATLAB® Neural Network Design Demonstration Directional Derivatives (**nnd8dd**).

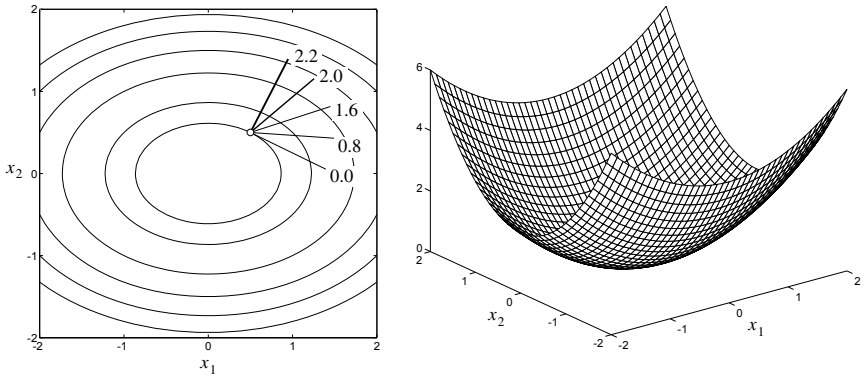


Figure 8.2 Quadratic Function and Directional Derivatives

Minima

Recall that the objective of performance learning will be to optimize the network performance index. In this section we want to define what we mean by an optimum point. We will assume that the optimum point is a minimum of the performance index. The definitions can be easily modified for maximization problems.

Strong Minimum

Strong Minimum

The point $\mathbf{x}^*$ is a strong minimum of $F(\mathbf{x})$ if a scalar $\delta > 0$ exists, such that $F(\mathbf{x}^*) < F(\mathbf{x}^* + \Delta\mathbf{x})$ for all $\Delta\mathbf{x}$ such that $\delta > \|\Delta\mathbf{x}\| > 0$.

In other words, if we move away from a strong minimum a small distance in *any* direction the function will increase.

Global Minimum

Global Minimum

The point $\mathbf{x}^*$ is a unique global minimum of $F(\mathbf{x})$ if $F(\mathbf{x}^*) < F(\mathbf{x}^* + \Delta\mathbf{x})$ for all $\Delta\mathbf{x} \neq \mathbf{0}$.

For a simple strong minimum, $\mathbf{x}^*$, the function may be smaller than $F(\mathbf{x}^*)$ at some points outside a small neighborhood of $\mathbf{x}^*$. Therefore this is sometimes called a local minimum. For a global minimum the function will be larger than the minimum point at every other point in the parameter space.

Weak Minimum

Weak Minimum

The point $\mathbf{x}^*$ is a weak minimum of $F(\mathbf{x})$ if it is not a strong minimum, and a scalar $\delta > 0$ exists, such that $F(\mathbf{x}^*) \leq F(\mathbf{x}^* + \Delta\mathbf{x})$ for all $\Delta\mathbf{x}$ such that $\delta > \|\Delta\mathbf{x}\| > 0$.

No matter which direction we move away from a weak minimum, the function cannot decrease, although there may be some directions in which the function does not change.

As an example of local and global minimum points, consider the following scalar function:

$$F(x) = 3x^4 - 7x^2 - \frac{1}{2}x + 6 . \tag{8.17}$$

This function is displayed in Figure 8.3. Notice that it has two strong minimum points: at approximately -1.1 and 1.1. For both of these points the function increases in a local neighborhood. The minimum at 1.1 is a global minimum, since there is no other point for which the function is as small.

There is no weak minimum for this function. We will show a two-dimensional example of a weak minimum later.

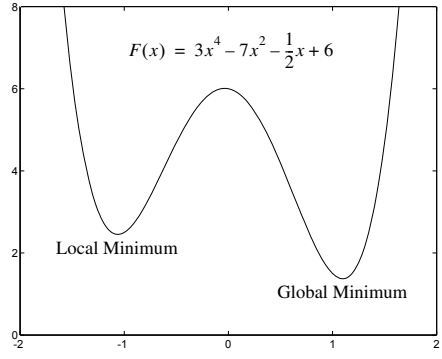


Figure 8.3 Scalar Example of Local and Global Minima

Now let's consider some vector cases. First, consider the following function:

$$F(\mathbf{x}) = (x_2 - x_1)^4 + 8x_1x_2 - x_1 + x_2 + 3 . \tag{8.18}$$

Contour Plot

In Figure 8.4 we have a *contour plot* (a series of curves along which the function value remains constant) and a 3-D surface plot for this function (for function values less than 12). We can see that the function has two strong local minimum points: one at (-0.42, 0.42), and the other at (0.55, -0.55). The global minimum point is at (0.55, -0.55).

Saddle Point

There is also another interesting feature of this function at (-0.13, 0.13). It is called a *saddle point* because of the shape of the surface in the neighborhood of the point. It is characterized by the fact that along the line $x_1 = -x_2$ the saddle point is a local maximum, but along a line orthogonal to that line it is a local minimum. We will investigate this example in more detail in Problems P8.2 and P8.5.

This function is used in the MATLAB® Neural Network Design Demonstration Vector Taylor Series (**nnd8ts2**).

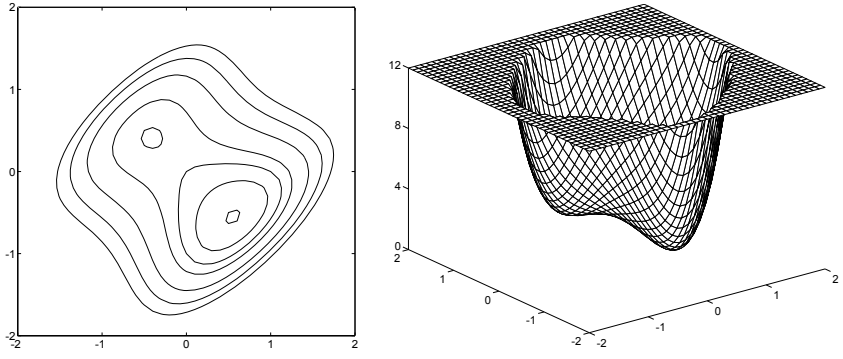


Figure 8.4 Vector Example of Minima and Saddle Point

As a final example, consider the function defined in Eq. (8.19):

$$F(\mathbf{x}) = (x_1^2 - 1.5x_1x_2 + 2x_2^2)x_1^2 \tag{8.19}$$

The contour and 3-D plots of this function are given in Figure 8.5. Here we can see that any point along the line $x_1 = 0$ is a weak minimum.

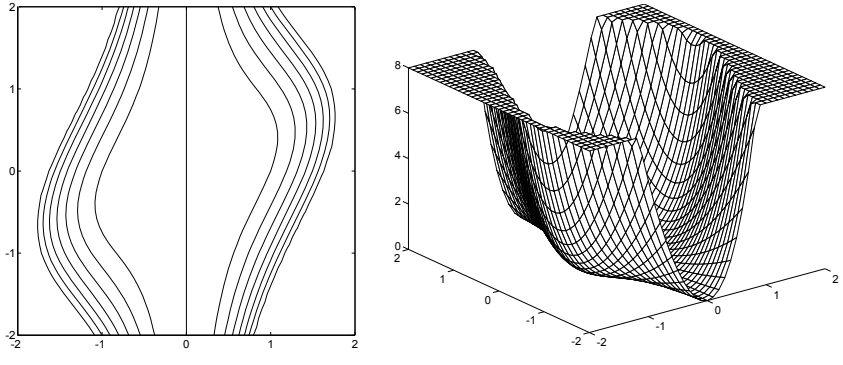


Figure 8.5 Weak Minimum Example

Necessary Conditions for Optimality

Now that we have defined what we mean by an optimum (minimum) point, let's identify some conditions that would have to be satisfied by such a point. We will again use the Taylor series expansion to derive these conditions:

$$F(\mathbf{x}) = F(\mathbf{x}^* + \Delta\mathbf{x}) = F(\mathbf{x}^*) + \nabla F(\mathbf{x})^T\Big|_{\mathbf{x}=\mathbf{x}^*}\Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T\nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}\Delta\mathbf{x} + \cdots , \tag{8.20}$$

where

$$\Delta\mathbf{x} = \mathbf{x} - \mathbf{x}^* . \tag{8.21}$$

First-Order Conditions

If $\|\Delta\mathbf{x}\|$ is very small then the higher order terms in Eq. (8.20) will be negligible and we can approximate the function as

$$F(\mathbf{x}^* + \Delta\mathbf{x}) \cong F(\mathbf{x}^*) + \nabla F(\mathbf{x})^T\Big|_{\mathbf{x}=\mathbf{x}^*}\Delta\mathbf{x} . \tag{8.22}$$

The point $\mathbf{x}^*$ is a candidate minimum point, which means that the function should go up (or at least not go down) if $\Delta\mathbf{x}$ is not zero. For this to happen the second term in Eq. (8.22) should not be negative. In other words

$$\nabla F(\mathbf{x})^T\Big|_{\mathbf{x}=\mathbf{x}^*}\Delta\mathbf{x} \geq 0 . \tag{8.23}$$

However, if this term is positive,

$$\nabla F(\mathbf{x})^T\Big|_{\mathbf{x}=\mathbf{x}^*}\Delta\mathbf{x} > 0 , \tag{8.24}$$

then this would imply that

$$F(\mathbf{x}^* - \Delta\mathbf{x}) \cong F(\mathbf{x}^*) - \nabla F(\mathbf{x})^T\Big|_{\mathbf{x}=\mathbf{x}^*}\Delta\mathbf{x} < F(\mathbf{x}^*) . \tag{8.25}$$

But this is a contradiction, since $\mathbf{x}^*$ should be a minimum point. Therefore, since Eq. (8.23) must be true, and Eq. (8.24) must be false, the only alternative must be that

$$\nabla F(\mathbf{x})^T\Big|_{\mathbf{x}=\mathbf{x}^*}\Delta\mathbf{x} = 0 . \tag{8.26}$$

Since this must be true for any $\Delta\mathbf{x}$, we have

$$\nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*} = \mathbf{0} . \tag{8.27}$$

Therefore the gradient must be zero at a minimum point. This is a first-order, necessary (but not sufficient) condition for $\mathbf{x}^*$ to be a local minimum point. Any points that satisfy Eq. (8.27) are called *stationary points*.

Stationary Points

Second-Order Conditions

Assume that we have a stationary point $\mathbf{x}^*$. Since the gradient of $F(\mathbf{x})$ is zero at all stationary points, the Taylor series expansion will be

$$F(\mathbf{x}^* + \Delta\mathbf{x}) = F(\mathbf{x}^*) + \frac{1}{2}\Delta\mathbf{x}^T\nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}\Delta\mathbf{x} + \cdots . \tag{8.28}$$

As before, we will consider only those points in a small neighborhood of $\mathbf{x}^*$, so that $\|\Delta\mathbf{x}\|$ is small and $F(\mathbf{x})$ can be approximated by the first two terms in Eq. (8.28). Therefore a strong minimum will exist at $\mathbf{x}^*$ if

$$\Delta\mathbf{x}^T\nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}\Delta\mathbf{x} > 0 . \tag{8.29}$$

Positive Definite Matrix

For this to be true for arbitrary $\Delta\mathbf{x} \neq \mathbf{0}$ requires that the Hessian matrix be positive definite. (By definition, a matrix $\mathbf{A}$ is *positive definite* if

$$\mathbf{z}^T\mathbf{A}\mathbf{z} > 0 \tag{8.30}$$

Positive Semidefinite

for any vector $\mathbf{z} \neq \mathbf{0}$. It is *positive semidefinite* if

$$\mathbf{z}^T\mathbf{A}\mathbf{z} \geq 0 \tag{8.31}$$

for any vector $\mathbf{z}$. We can check these conditions by testing the eigenvalues of the matrix. If all eigenvalues are positive, then the matrix is positive definite. If all eigenvalues are nonnegative, then the matrix is positive semidefinite.)

Sufficient Condition

A positive definite Hessian matrix is a second-order, *sufficient* condition for a strong minimum to exist. It is not a necessary condition. A minimum can still be strong if the second-order term of the Taylor series is zero, but the third-order term is positive. Therefore the second-order, *necessary* condition for a strong minimum is that the Hessian matrix be positive semi-definite.

To illustrate these conditions, consider the following function of two variables:

$$F(\mathbf{x}) = x_1^4 + x_2^2 . \tag{8.32}$$

First, we want to locate any stationary points, so we need to evaluate the gradient:

$$\nabla F(\mathbf{x}) = \begin{bmatrix} 4x_1^3 \\ 2x_2 \end{bmatrix} = \mathbf{0} . \tag{8.33}$$

Therefore the only stationary point is the point $\mathbf{x}^* = \mathbf{0}$. We now need to test the second-order condition, which requires the Hessian matrix:

$$\nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{0}} = \begin{bmatrix} 12x_1^2 & 0 \\ 0 & 2 \end{bmatrix}\Bigg|_{\mathbf{x}=\mathbf{0}} = \begin{bmatrix} 0 & 0 \\ 0 & 2 \end{bmatrix}. \tag{8.34}$$

This matrix is positive semidefinite, which is a necessary condition for $\mathbf{x}^* = \mathbf{0}$ to be a strong minimum point. We cannot guarantee from first-order and second-order conditions that it is a minimum point, but we have not eliminated it as a possibility. Actually, even though the Hessian matrix is only positive semidefinite, $\mathbf{x}^* = \mathbf{0}$ is a strong minimum point, but we cannot prove it from the conditions we have discussed.

Just to summarize, the necessary conditions for $\mathbf{x}^*$ to be a minimum, strong or weak, of $F(\mathbf{x})$ are:

$$\nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*} = \mathbf{0} \text{ and } \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*} \text{ positive semidefinite.}$$

The sufficient conditions for $\mathbf{x}^*$ to be a strong minimum point of $F(\mathbf{x})$ are:

$$\nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*} = \mathbf{0} \text{ and } \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*} \text{ positive definite.}$$

Quadratic Functions

We will find throughout this text that one type of performance index is universal — the quadratic function. This is true because there are many applications in which the quadratic function appears, but also because many functions can be approximated by quadratic functions in small neighborhoods, especially near local minimum points. For this reason we want to spend a little time investigating the characteristics of the quadratic function.

Quadratic Function

The general form of a *quadratic function* is

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{d}^T\mathbf{x} + c, \tag{8.35}$$

where the matrix $\mathbf{A}$ is symmetric. (If the matrix is not symmetric it can be replaced by a symmetric matrix that produces the same $F(\mathbf{x})$. Try it!)

To find the gradient for this function, we will use the following useful properties of the gradient:

$$\nabla(\mathbf{h}^T\mathbf{x}) = \nabla(\mathbf{x}^T\mathbf{h}) = \mathbf{h}, \tag{8.36}$$

where $\mathbf{h}$ is a constant vector, and

$$\nabla \mathbf{x}^T \mathbf{Q}\mathbf{x} = \mathbf{Q}\mathbf{x} + \mathbf{Q}^T\mathbf{x} = 2\mathbf{Q}\mathbf{x} \text{ (for symmetric } \mathbf{Q}). \tag{8.37}$$

We can now compute the gradient of $F(\mathbf{x})$:

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d}, \tag{8.38}$$

and in a similar way we can find the Hessian:

$$\nabla^2 F(\mathbf{x}) = \mathbf{A}. \tag{8.39}$$

All higher derivatives of the quadratic function are zero. Therefore the first three terms of the Taylor series expansion (as in Eq. (8.20)) give an exact representation of the function. We can also say that all analytic functions behave like quadratics over a small neighborhood (i.e., when $\|\Delta\mathbf{x}\|$ is small).

Eigensystem of the Hessian

We now want to investigate the general shape of the quadratic function. It turns out that we can tell a lot about the shape by looking at the eigenvalues and eigenvectors of the Hessian matrix. Consider a quadratic function that has a stationary point at the origin, and whose value there is zero:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}. \tag{8.40}$$

The shape of this function can be seen more clearly if we perform a change of basis (see Chapter 6). We want to use the eigenvectors of the Hessian matrix, $\mathbf{A}$, as the new basis vectors. Since $\mathbf{A}$ is symmetric, its eigenvectors will be mutually orthogonal. (See [Brog91].) This means that if we make up a matrix with the eigenvectors as the columns, as in Eq. (6.68):

$$\mathbf{B} = \left[\mathbf{z}_1\ \mathbf{z}_2\ \ldots\ \mathbf{z}_n\right], \tag{8.41}$$

the inverse of the matrix will be the same as the transpose:

$$\mathbf{B}^{-1} = \mathbf{B}^T. \tag{8.42}$$

(This assumes that we have normalized the eigenvectors.)

If we now perform a change of basis, so that the eigenvectors are the basis vectors (as in Eq. (6.69)), the new $\mathbf{A}$ matrix will be

$$\mathbf{A}' = [\mathbf{B}^T\mathbf{A}\mathbf{B}] = \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & \lambda_n \end{bmatrix} = \Lambda, \tag{8.43}$$

where the λ_i are the eigenvalues of $\mathbf{A}$. We can also write this equation as

$$\mathbf{A} = \mathbf{B}\Lambda\mathbf{B}^T. \tag{8.44}$$

We will now use the concept of the directional derivative to explain the physical meaning of the eigenvalues and eigenvectors of $\mathbf{A}$, and to explain how they determine the shape of the surface of the quadratic function.

Recall from Eq. (8.13) that the second derivative of a function $F(\mathbf{x})$ in the direction of a vector $\mathbf{p}$ is given by

$$\frac{\mathbf{p}^T\nabla^2 F(\mathbf{x})\mathbf{p}}{\|\mathbf{p}\|^2} = \frac{\mathbf{p}^T\mathbf{A}\mathbf{p}}{\|\mathbf{p}\|^2}. \tag{8.45}$$

Now define

$$\mathbf{p} = \mathbf{B}\mathbf{c}, \tag{8.46}$$

where $\mathbf{c}$ is the representation of the vector $\mathbf{p}$ with respect to the eigenvectors of $\mathbf{A}$. (See Eq. (6.28) and the discussion that follows.) With this definition, and Eq. (8.44), we can rewrite Eq. (8.45):

$$\frac{\mathbf{p}^T\mathbf{A}\mathbf{p}}{\|\mathbf{p}\|^2} = \frac{\mathbf{c}^T\mathbf{B}^T(\mathbf{B}\Lambda\mathbf{B}^T)\mathbf{B}\mathbf{c}}{\mathbf{c}^T\mathbf{B}^T\mathbf{B}\mathbf{c}} = \frac{\mathbf{c}^T\Lambda\mathbf{c}}{\mathbf{c}^T\mathbf{c}} = \frac{\sum_{i=1}^{n}\lambda_i c_i^2}{\sum_{i=1}^{n} c_i^2}. \tag{8.47}$$

This result tells us several useful things. First, note that this second derivative is just a weighted average of the eigenvalues. Therefore it can never be larger than the largest eigenvalue, or smaller than the smallest eigenvalue. In other words,

$$\lambda_{min} \le \frac{\mathbf{p}^T\mathbf{A}\mathbf{p}}{\|\mathbf{p}\|^2} \le \lambda_{max}. \tag{8.48}$$

Under what condition, if any, will this second derivative be equal to the largest eigenvalue? What if we choose

$$\mathbf{p} = \mathbf{z}_{max}, \tag{8.49}$$

where $\mathbf{z}_{max}$ is the eigenvector associated with the largest eigenvalue, λ_{max}? For this case the $\mathbf{c}$ vector will be

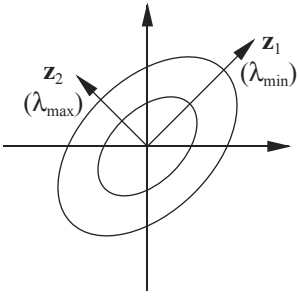
$$\mathbf{c} = \mathbf{B}^T\mathbf{p} = \mathbf{B}^T\mathbf{z}_{max} = \begin{bmatrix}0 & 0 & \dots & 0 & 1 & 0 & \dots & 0\end{bmatrix}^T, \tag{8.50}$$

where the one occurs only in the position that corresponds to the largest eigenvalue (i.e., $c_{max} = 1$). This is because the eigenvectors are orthonormal.

If we now substitute $\mathbf{z}_{max}$ for $\mathbf{p}$ in Eq. (8.47) we obtain

$$\frac{\mathbf{z}_{max}^T \mathbf{A} \mathbf{z}_{max}}{\|\mathbf{z}_{max}\|^2} = \frac{\sum_{i=1}^{n} \lambda_i c_i^2}{\sum_{i=1}^{n} c_i^2} = \lambda_{max}. \tag{8.51}$$

So the maximum second derivative occurs in the direction of the eigenvector that corresponds to the largest eigenvalue. In fact, in each of the eigenvector directions the second derivatives will be equal to the corresponding eigenvalue. In other directions the second derivative will be a weighted average of the eigenvalues. The eigenvalues are the second derivatives in the directions of the eigenvectors.



The eigenvectors define a new coordinate system in which the quadratic cross terms vanish. The eigenvectors are known as the principal axes of the function contours. The figure to the left illustrates these concepts in two dimensions. This figure illustrates the case where the first eigenvalue is smaller than the second eigenvalue. Therefore the minimum curvature (second derivative) will occur in the direction of the first eigenvector. This means that we will cross contour lines more slowly in this direction. The maximum curvature will occur in the direction of the second eigenvector, therefore we will cross contour lines more quickly in that direction.

One caveat about this figure: it is only valid when both eigenvalues have the same sign, so that we have either a strong minimum or a strong maximum. For these cases the contour lines are always elliptical. We will provide examples later where the eigenvalues have opposite signs and where one of the eigenvalues is zero.

For our first example, consider the following function:

$$F(\mathbf{x}) = x_1^2 + x_2^2 = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \mathbf{x}. \tag{8.52}$$

The Hessian matrix and its eigenvalues and eigenvectors are

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}, \lambda_1 = 2, \mathbf{z}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \lambda_2 = 2, \mathbf{z}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{8.53}$$

(Actually, any two independent vectors could be the eigenvectors in this case. There is a repeated eigenvalue, and its eigenvector is the plane.) Since all the eigenvalues are equal, the curvature should be the same in all directions, and therefore the function should have circular contours. Figure 8.6 shows the contour and 3-D plots for this function, a circular hollow.

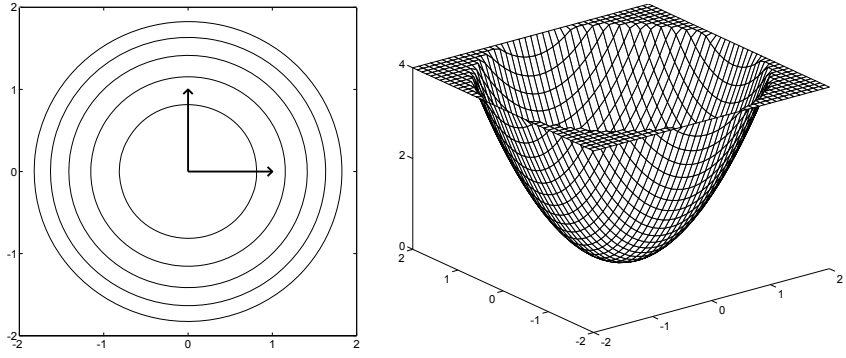


Figure 8.6 Circular Hollow

Let's try an example with distinct eigenvalues. Consider the following quadratic function:

$$F(\mathbf{x}) = x_1^2 + x_1 x_2 + x_2^2 = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \mathbf{x} \tag{8.54}$$

The Hessian matrix and its eigenvalues and eigenvectors are

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}, \lambda_1 = 1, \mathbf{z}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \lambda_2 = 3, \mathbf{z}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}. \tag{8.55}$$

(As we discussed in Chapter 6, the eigenvectors are not unique, they can be multiplied by any scalar.) In this case the maximum curvature is in the direction of $\mathbf{z}_2$ so we should cross contour lines more quickly in that direction. Figure 8.7 shows the contour and 3-D plots for this function, an elliptical hollow.

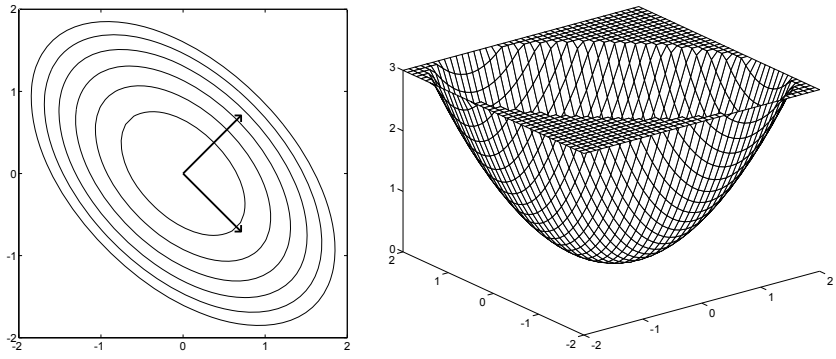


Figure 8.7 Elliptical Hollow

What happens when the eigenvalues have opposite signs? Consider the following function:

$$F(\mathbf{x}) = -\frac{1}{4}x_1^2 - \frac{3}{2}x_1x_2 - \frac{1}{4}x_2^2 = \frac{1}{2}\mathbf{x}^T\begin{bmatrix}-0.5 & -1.5\\ -1.5 & -0.5\end{bmatrix}\mathbf{x}. \tag{8.56}$$

The Hessian matrix and its eigenvalues and eigenvectors are

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix}-0.5 & -1.5\\ -1.5 & -0.5\end{bmatrix},\ \lambda_1 = 1,\ \mathbf{z}_1 = \begin{bmatrix}-1\\ 1\end{bmatrix},\ \lambda_2 = -2,\ \mathbf{z}_2 = \begin{bmatrix}-1\\ -1\end{bmatrix}. \tag{8.57}$$

The first eigenvalue is positive, so there is positive curvature in the direction of $\mathbf{z}_1$. The second eigenvalue is negative, so there is negative curvature in the direction of $\mathbf{z}_2$. Also, since the magnitude of the second eigenvalue is greater than the magnitude of the first eigenvalue, we will cross contour lines faster in the direction of $\mathbf{z}_2$.

Figure 8.8 shows the contour and 3-D plots for this function, an elongated saddle. Note that the stationary point,

$$\mathbf{x}^* = \begin{bmatrix}0\\ 0\end{bmatrix}, \tag{8.58}$$

is no longer a strong minimum point, since the Hessian matrix is not positive definite. Since the eigenvalues are of opposite sign, we know that the Hessian is indefinite (see [Brog91]). The stationary point is therefore a saddle point. It is a minimum of the function along the first eigenvector (positive eigenvalue), but it is a maximum of the function along the second eigenvector (negative eigenvalue).

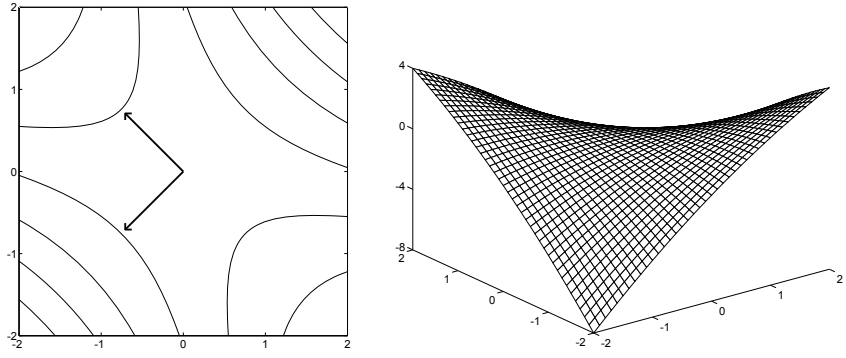


Figure 8.8 Elongated Saddle

As a final example, let's try a case where one of the eigenvalues is zero. An example of this is given by the following function:

$$F(\mathbf{x}) = \frac{1}{2}x_1^2 - x_1x_2 + \frac{1}{2}x_2^2 = \frac{1}{2}\mathbf{x}^T\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\mathbf{x}. \tag{8.59}$$

The Hessian matrix and its eigenvalues and eigenvectors are

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, \lambda_1 = 2, \mathbf{z}_1 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \lambda_2 = 0, \mathbf{z}_2 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}. \tag{8.60}$$

The second eigenvalue is zero, so we would expect to have zero curvature along $\mathbf{z}_2$. Figure 8.9 shows the contour and 3-D plots for this function, a stationary valley. In this case the Hessian matrix is positive semidefinite, and we have a weak minimum along the line

$$x_1 = x_2, \tag{8.61}$$

corresponding to the second eigenvector.

For quadratic functions the Hessian matrix must be positive definite in order for a strong minimum to exist. For higher-order functions it is possible to have a strong minimum with a positive semidefinite Hessian matrix, as we discussed previously in the section on minima.

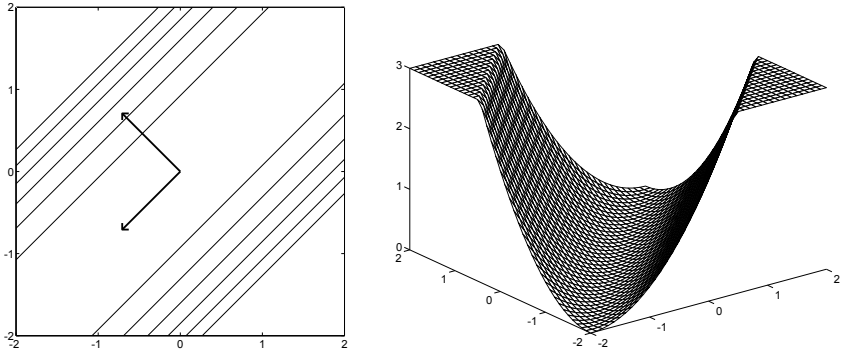


Figure 8.9 Stationary Valley

To experiment with other quadratic functions, use the MATLAB® Neural Network Design Demonstration Quadratic Function (**nnd8qf**).

At this point we can summarize some characteristics of the quadratic function.

1. If the eigenvalues of the Hessian matrix are all positive, the function will have a single strong minimum.

2. If the eigenvalues are all negative, the function will have a single strong maximum.

3. If some eigenvalues are positive and other eigenvalues are negative, the function will have a single saddle point.

4. If the eigenvalues are all nonnegative, but some eigenvalues are zero, then the function will either have a weak minimum (as in Figure 8.9) or will have no stationary point (see Solved Problem P8.7).

5. If the eigenvalues are all nonpositive, but some eigenvalues are zero, then the function will either have a weak maximum or will have no stationary point.

We should note that in this discussion we have assumed, for simplicity, that the stationary point of the quadratic function was at the origin, and that it had a zero value there. This requires that the terms $\mathbf{d}$ and c in Eq. (8.35) both be zero. If c is nonzero then the function is simply increased in magnitude by c at every point. The shape of the contours do not change. When $\mathbf{d}$ is nonzero, and $\mathbf{A}$ is invertible, the shape of the contours are not changed, but the stationary point of the function moves to

$$\mathbf{x}^* = -\mathbf{A}^{-1}\mathbf{d}\,. \tag{8.62}$$

If $\mathbf{A}$ is not invertible (has some zero eigenvalues) and $\mathbf{d}$ is nonzero then stationary points may not exist (see Solved Problem P8.7).

Summary of Results

Taylor Series

$$F(\mathbf{x}) = F(\mathbf{x}^*) + \nabla F(\mathbf{x})^T\Big|_{\mathbf{x}=\mathbf{x}^*}(\mathbf{x}-\mathbf{x}^*) + \frac{1}{2}(\mathbf{x}-\mathbf{x}^*)^T\nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^*}(\mathbf{x}-\mathbf{x}^*) + \cdots$$

Gradient

$$\nabla F(\mathbf{x}) = \left[\frac{\partial}{\partial x_1}F(\mathbf{x})\ \frac{\partial}{\partial x_2}F(\mathbf{x})\ \cdots\ \frac{\partial}{\partial x_n}F(\mathbf{x})\right]^T$$

Hessian Matrix

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2}{\partial x_1^2}F(\mathbf{x}) & \frac{\partial^2}{\partial x_1 \partial x_2}F(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_1 \partial x_n}F(\mathbf{x}) \\ \frac{\partial^2}{\partial x_2 \partial x_1}F(\mathbf{x}) & \frac{\partial^2}{\partial x_2^2}F(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_2 \partial x_n}F(\mathbf{x}) \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2}{\partial x_n \partial x_1}F(\mathbf{x}) & \frac{\partial^2}{\partial x_n \partial x_2}F(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_n^2}F(\mathbf{x}) \end{bmatrix}$$

Directional Derivatives

First Directional Derivative

$$\frac{\mathbf{p}^T\nabla F(\mathbf{x})}{\|\mathbf{p}\|}$$

Second Directional Derivative

$$\frac{\mathbf{p}^T\nabla^2 F(\mathbf{x})\mathbf{p}}{\|\mathbf{p}\|^2}$$

Minima

Strong Minimum

The point $\mathbf{x}^*$ is a strong minimum of $F(\mathbf{x})$ if a scalar $\delta > 0$ exists, such that $F(\mathbf{x}) < F(\mathbf{x} + \Delta\mathbf{x})$ for all $\Delta\mathbf{x}$ such that $\delta > \|\Delta\mathbf{x}\| > 0$.

Global Minimum

The point $\mathbf{x}^*$ is a unique global minimum of $F(\mathbf{x})$ if $F(\mathbf{x}) < F(\mathbf{x} + \Delta\mathbf{x})$ for all $\Delta\mathbf{x} \neq \mathbf{0}$.

Weak Minimum

The point $\mathbf{x}^*$ is a weak minimum of $F(\mathbf{x})$ if it is not a strong minimum, and a scalar $\delta > 0$ exists, such that $F(\mathbf{x}) \leq F(\mathbf{x} + \Delta\mathbf{x})$ for all $\Delta\mathbf{x}$ such that $\delta > \|\Delta\mathbf{x}\| > 0$.

Necessary Conditions for Optimality

First-Order Condition

$$\nabla F(\mathbf{x})\big|_{\mathbf{x} = \mathbf{x}^*} = \mathbf{0} \quad \text{(Stationary Points)}$$

Second-Order Condition

$$\nabla^2 F(\mathbf{x})\big|_{\mathbf{x} = \mathbf{x}^*} \geq 0 \quad \text{(Positive Semidefinite Hessian Matrix)}$$

Quadratic Functions

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{d}^T\mathbf{x} + c$$

Gradient

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d}$$

Hessian

$$\nabla^2 F(\mathbf{x}) = \mathbf{A}$$

Directional Derivatives

$$\lambda_{min} \leq \frac{\mathbf{p}^T\mathbf{A}\mathbf{p}}{\|\mathbf{p}\|^2} \leq \lambda_{max}$$

Solved Problems

P8.1 In Figure 8.1 we illustrated 3 approximations to the cosine function about the point $x^* = 0$. Repeat that procedure about the point $x^* = \pi/2$.

The function we want to approximate is

$$F(x) = \cos(x).$$

The Taylor series expansion for $F(x)$ about the point $x^* = \pi/2$ is

$$F(x) = \cos(x) = \cos\left(\frac{\pi}{2}\right) - \sin\left(\frac{\pi}{2}\right)\left(x-\frac{\pi}{2}\right) - \frac{1}{2}\cos\left(\frac{\pi}{2}\right)\left(x-\frac{\pi}{2}\right)^2 + \frac{1}{6}\sin\left(\frac{\pi}{2}\right)\left(x-\frac{\pi}{2}\right)^3 + \cdots$$

$$= -\left(x-\frac{\pi}{2}\right) + \frac{1}{6}\left(x-\frac{\pi}{2}\right)^3 - \frac{1}{120}\left(x-\frac{\pi}{2}\right)^5 + \cdots$$

The zeroth-order approximation of $F(x)$ is

$$F(x) \approx F_0(x) = 0.$$

The first-order approximation is

$$F(\mathbf{x}) \approx F_1(x) = -\left(x-\frac{\pi}{2}\right) = \frac{\pi}{2} - x.$$

(Note that in this case the second-order approximation is the same as the first-order approximation, since the second derivative is zero.)

The third-order approximation is

$$F(\mathbf{x}) \approx F_3(x) = -\left(x-\frac{\pi}{2}\right) + \frac{1}{6}\left(x-\frac{\pi}{2}\right)^3.$$

A graph showing $F(x)$ and these three approximations is shown in Figure P8.1. Note that in this case the zeroth-order approximation is very poor, while the first-order approximation is accurate over a reasonably wide range. Compare this result with Figure 8.1. In that case we were expanding about a local maximum point, $x^* = 0$, so the first derivative was zero.

Check the Taylor series expansions at other points using the Neural Network Design Demonstration Taylor Series (`nnd8ts1`).

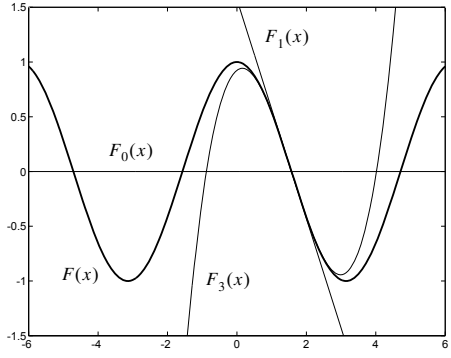


Figure P8.1 Cosine Approximation About $x = \pi/2$

P8.2 Recall the function that is displayed in Figure 8.4, on page 8-9. We know that this function has two strong minima. Find the second-order Taylor series expansions for this function about the two minima.

The equation for this function is

$$F(\mathbf{x}) = (x_2 - x_1)^4 + 8x_1x_2 - x_1 + x_2 + 3\,.$$

To find the second-order Taylor series expansion, we need to find the gradient and the Hessian for $F(\mathbf{x})$. For the gradient we have

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \dfrac{\partial}{\partial x_1} F(\mathbf{x}) \\ \dfrac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} -4(x_2 - x_1)^3 + 8x_2 - 1 \\ 4(x_2 - x_1)^3 + 8x_1 + 1 \end{bmatrix},$$

and the Hessian matrix is

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} \dfrac{\partial^2}{\partial x_1^2} F(\mathbf{x}) & \dfrac{\partial^2}{\partial x_1 \partial x_2} F(\mathbf{x}) \\ \dfrac{\partial^2}{\partial x_2 \partial x_1} F(\mathbf{x}) & \dfrac{\partial^2}{\partial x_2^2} F(\mathbf{x}) \end{bmatrix}$$

$$= \begin{bmatrix} 12(x_2 - x_1)^2 & -12(x_2 - x_1)^2 + 8 \\ -12(x_2 - x_1)^2 + 8 & 12(x_2 - x_1)^2 \end{bmatrix}$$

One strong minimum occurs at $\mathbf{x}^1 = \begin{bmatrix}-0.42 & 0.42\end{bmatrix}^T$, and the other at $\mathbf{x}^2 = \begin{bmatrix}0.55 & -0.55\end{bmatrix}^T$. If we perform the second-order Taylor series expansion of $F(\mathbf{x})$ about these two points we obtain:

$$F^1(\mathbf{x}) = F(\mathbf{x}^1) + \nabla F(\mathbf{x})^T\Big|_{\mathbf{x}=\mathbf{x}^1}(\mathbf{x}-\mathbf{x}^1) + \frac{1}{2}(\mathbf{x}-\mathbf{x}^1)^T\nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^1}(\mathbf{x}-\mathbf{x}^1)$$

$$= 2.93 + \frac{1}{2}\left(\mathbf{x} - \begin{bmatrix}-0.42\\ 0.42\end{bmatrix}\right)^T \begin{bmatrix}8.42 & -0.42\\ -0.42 & 8.42\end{bmatrix}\left(\mathbf{x} - \begin{bmatrix}-0.42\\ 0.42\end{bmatrix}\right).$$

If we simplify this expression we find

$$F^1(\mathbf{x}) = 4.49 - \begin{bmatrix}-3.7128 & 3.7128\end{bmatrix}\mathbf{x} + \frac{1}{2}\mathbf{x}^T\begin{bmatrix}8.42 & -0.42\\ -0.42 & 8.42\end{bmatrix}\mathbf{x}.$$

Repeating this process for $\mathbf{x}^2$ results in

$$F^2(\mathbf{x}) = 7.41 - \begin{bmatrix}11.781 & -11.781\end{bmatrix}\mathbf{x} + \frac{1}{2}\mathbf{x}^T\begin{bmatrix}14.71 & -6.71\\ -6.71 & 14.71\end{bmatrix}\mathbf{x}.$$

The original function and the two approximations are plotted in the following figures.

Check the Taylor series expansions at other points using the Neural Network Design Demonstration Vector Taylor Series (**nnd8ts2**).

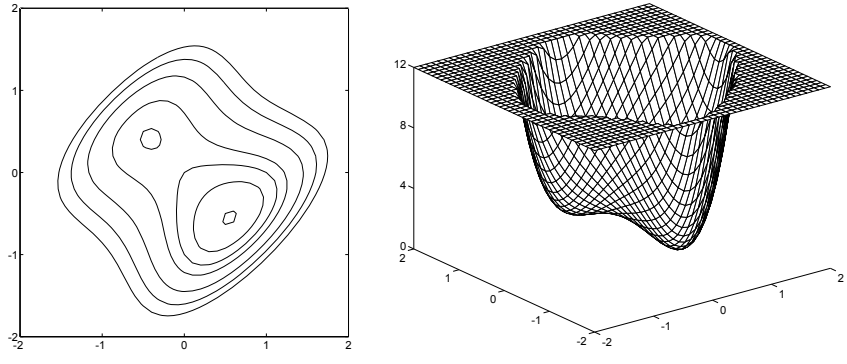


Figure P8.2 Function $F(\mathbf{x})$ for Problem P8.2

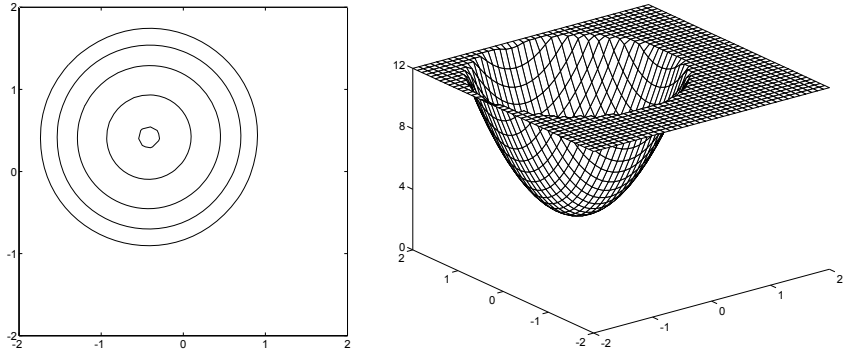


Figure P8.3 Function $F^1(\mathbf{x})$ for Problem P8.2

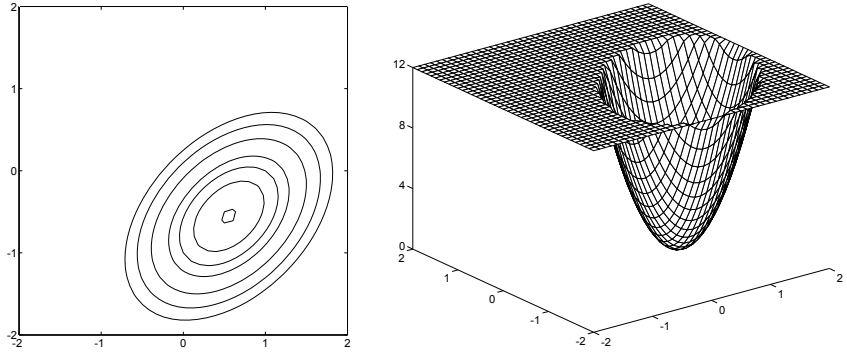


Figure P8.4 Function $F^2(\mathbf{x})$ for Problem P8.2

P8.3 For the function $F(\mathbf{x})$ given below, find the equation for the line that is tangent to the contour line at $\mathbf{x} = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$.

$$F(\mathbf{x}) = (2 + x_1)^2 + 5(1 - x_1 - x_2^2)^2$$

To solve this problem we can use the directional derivative. What is the derivative of $F(\mathbf{x})$ along a line that is tangent to a contour line? Since the contour is a line along which the function does not change, the derivative of $F(\mathbf{x})$ should be zero in the direction of the contour. So we can get the equation for the tangent to the contour line by setting the directional derivative equal to zero.

First we need to find the gradient:

$$\nabla F(\mathbf{x}) = \begin{bmatrix} 2(2+x_1) + 10(1-x_1-x_2^2)(-1) \\ 10(1-x_1-x_2^2)(-2x_2) \end{bmatrix} = \begin{bmatrix} -6 + 12x_1 + 10x_2^2 \\ -20x_2 + 20x_1x_2 + 20x_2^3 \end{bmatrix}.$$

If we evaluate this at $\mathbf{x}^* = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$, we obtain

$$\nabla F(\mathbf{x}^*) = \begin{bmatrix} -6 \\ 0 \end{bmatrix}.$$

Now recall that the equation for the derivative of $F(\mathbf{x})$ in the direction of a vector $\mathbf{p}$ is

$$\frac{\mathbf{p}^T \nabla F(\mathbf{x})}{\|\mathbf{p}\|}.$$

Therefore if we want the equation for the line that passes through $\mathbf{x}^* = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$ and along which the derivative is zero, we can set the numerator of the directional derivative in the direction of $\Delta\mathbf{x}$ to zero:

$$\Delta\mathbf{x}^T \nabla F(\mathbf{x}^*) = 0,$$

where $\Delta\mathbf{x} = \mathbf{x} - \mathbf{x}^*$. For this case we have

$$\mathbf{x}^T \begin{bmatrix} -6 \\ 0 \end{bmatrix} = 0, \text{ or } x_1 = 0.$$

This result is illustrated in Figure P8.5.

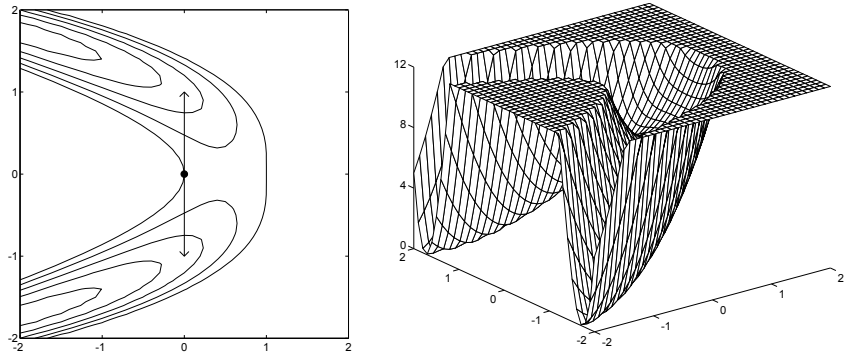


Figure P8.5 Plot of $F(\mathbf{x})$ for Problem P8.3

P8.4 Consider the following fourth-order polynomial:

$$F(x) = x^4 - \frac{2}{3}x^3 - 2x^2 + 2x + 4 .$$

Find any stationary points and test them to see if they are minima.

To find the stationary points we set the derivative of $F(x)$ to zero:

$$\frac{d}{dx}F(x) = 4x^3 - 2x^2 - 4x + 2 = 0 .$$

We can use MATLAB to find the roots of this polynomial:

```
coef=[4 -2 -4 2];
stapoints=roots(coef);
stapoints'
ans =
   1.0000   -1.0000    0.5000
```

Now we need to check the second derivative at each of these points. The second derivative of $F(x)$ is

$$\frac{d^2}{dx^2}F(x) = 12x^2 - 4x - 4 .$$

If we evaluate this at each of the stationary points we find

$$\left(\frac{d^2}{dx^2}F(1) = 4\right), \left(\frac{d^2}{dx^2}F(-1) = 12\right), \left(\frac{d^2}{dx^2}F(0.5) = -3\right) .$$

Therefore we should have strong local minima at 1 and -1 (since the second derivatives were positive), and a strong local maximum at 0.5 (since the second derivative was negative). To find the global minimum we would have to evaluate the function at the two local minima:

$$(F(1) = 4.333), (F(-1) = 1.667) .$$

Therefore the global minimum occurs at -1. But are we sure that this is a global minimum? What happens to the function as $x \to \infty$ or $x \to -\infty$? In this case, because the highest power of x has a positive coefficient and is an even power (x^4), the function goes to ∞ at both limits. So we can safely say that the global minimum occurs at -1. The function is plotted in Figure P8.6.

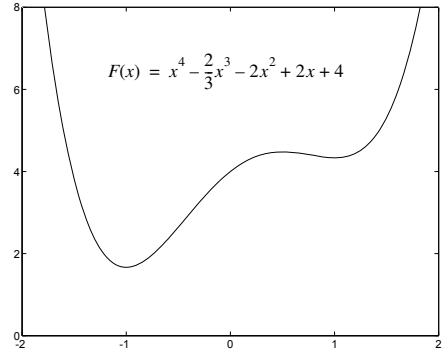


Figure P8.6 Graph of $F(x)$ for Problem P8.4

P8.5 Look back to the function of Problem P8.2. This function has three stationary points:

$$\mathbf{x}^1 = \begin{bmatrix} -0.41878 \\ 0.41878 \end{bmatrix}, \ \mathbf{x}^2 = \begin{bmatrix} -0.134797 \\ 0.134797 \end{bmatrix}, \ \mathbf{x}^3 = \begin{bmatrix} 0.55358 \\ -0.55358 \end{bmatrix}.$$

Test whether or not any of these points could be local minima.

From Problem P8.2 we know that the Hessian matrix for the function is

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} 12(x_2 - x_1)^2 & -12(x_2 - x_1)^2 + 8 \\ -12(x_2 - x_1)^2 + 8 & 12(x_2 - x_1)^2 \end{bmatrix}.$$

To test the definiteness of this matrix we can check the eigenvalues. If the eigenvalues are all positive, the Hessian is positive definite, which guarantees a strong minimum. If the eigenvalues are nonnegative, the Hessian is positive semidefinite, which is consistent with either a strong or a weak minimum. If one eigenvalue is positive and the other eigenvalue is negative, the Hessian is indefinite, which would signal a saddle point.

If we evaluate the Hessian at $\mathbf{x}^1$, we find

$$\nabla^2 F(\mathbf{x}^1) = \begin{bmatrix} 8.42 & -0.42 \\ -0.42 & 8.42 \end{bmatrix}.$$

The eigenvalues of this matrix are

$$\lambda_1 = 8.84, \ \lambda_2 = 8.0,$$

therefore $\mathbf{x}^1$ must be a strong minimum point.

If we evaluate the Hessian at $\mathbf{x}^2$, we find

$$\nabla^2 F(\mathbf{x}^2) = \begin{bmatrix} 0.87 & 7.13 \\ 7.13 & 0.87 \end{bmatrix}.$$

The eigenvalues of this matrix are

$$\lambda_1 = -6.26, \ \lambda_2 = 8.0,$$

therefore $\mathbf{x}^2$ must be a saddle point. In one direction the curvature is negative, and in another direction the curvature is positive. The negative curvature is in the direction of the first eigenvector, and the positive curvature is in the direction of the second eigenvector. The eigenvectors are

$$\mathbf{z}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \text{ and } \mathbf{z}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

(Note that this is consistent with our previous discussion of this function on page 8-8.)

If we evaluate the Hessian at $\mathbf{x}^3$, we find

$$\nabla^2 F(\mathbf{x}^3) = \begin{bmatrix} 14.7 & -6.71 \\ -6.71 & 14.7 \end{bmatrix}.$$

The eigenvalues of this matrix are

$$\lambda_1 = 21.42, \ \lambda_2 = 8.0,$$

therefore $\mathbf{x}^3$ must be a strong minimum point.

Check these results using the Neural Network Design Demonstration Vector Taylor Series (`nnd8ts2`).

P8.6 Let's apply the concepts in this chapter to a neural network problem. Consider the linear network shown in Figure P8.7. Suppose that the desired inputs/outputs for the network are

$$\{(p_1 = 2), (t_1 = 0.5)\}, \{(p_2 = -1), (t_2 = 0)\}.$$

Sketch the following performance index for this network:

$$F(\mathbf{x}) = (t_1 - a_1(\mathbf{x}))^2 + (t_2 - a_2(\mathbf{x}))^2.$$