Neural Network Design



2nd Edition

Hagan
Demuth
Beale
De Jesús

TESLA 1024
Tesla, Inc. v. Autonomous Devices LLC
IPR2024-00745

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



Copyright by Martin T. Hagan and Howard B. Demuth. All rights reserved. No part of the book may be reproduced, stored in a retrieval system, or transcribed in any form or by any means - electronic, mechanical, photocopying, recording or otherwise - without the prior permission of Hagan and Demuth.

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

Preface

Introduction

Objectives 1-1
History 1-2
Applications 1-5
Biological Inspiration 1-8
Further Reading 1-10

2 

Neuron Model and Network Architectures

Objectives 2-1
Theory and Examples 2-2
- Notation 2-2
- Neuron Model 2-2
  - Single-Input Neuron 2-2
  - Transfer Functions 2-3
  - Multiple-Input Neuron 2-7
- Network Architectures 2-9
  - A Layer of Neurons 2-9
  - Multiple Layers of Neurons 2-10
  - Recurrent Networks 2-13

Summary of Results 2-16
Solved Problems 2-20
Epilogue 2-22
Exercises 2-23



An Illustrative Example



Objectives	3-1
Theory and Examples	3-2
Problem Statement	3-2
Perceptron	3-3
Two-Input Case	3-4
Pattern Recognition Example	3-5
Hamming Network	3-8
Feedforward Layer	3-8
Recurrent Layer	3-9
Hopfield Network	3-12
Epilogue	3-15
Exercises	3-16



Perceptron Learning Rule



Objectives	4-1
Theory and Examples	4-2
Learning Rules	4-2
Perceptron Architecture	4-3
Single-Neuron Perceptron	4-5
Multiple-Neuron Perceptron	4-8
Perceptron Learning Rule	4-8
Test Problem	4-9
Constructing Learning Rules	4-10
Unified Learning Rule	4-12
Training Multiple-Neuron Perceptrons	4-13
Proof of Convergence	4-15
Notation	4-15
Proof	4-16
Limitations	4-18
Summary of Results	4-20
Solved Problems	4-21
Epilogue	4-33
Further Reading	4-34
Exercises	4-36

5 **Signal and Weight Vector Spaces**



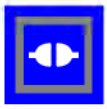
Objectives 5-1
Theory and Examples 5-2
Linear Vector Spaces 5-2
Linear Independence 5-4
Spanning a Space 5-5
Inner Product 5-6
Norm 5-7
Orthogonality 5-7
Gram-Schmidt Orthogonalization 5-8
Vector Expansions 5-9
Reciprocal Basis Vectors 5-10
Summary of Results 5-14
Solved Problems 5-17
Epilogue 5-26
Further Reading 5-27
Exercises 5-28

6 **Linear Transformations for Neural Networks**



Objectives 6-1
Theory and Examples 6-2
Linear Transformations 6-2
Matrix Representations 6-3
Change of Basis 6-6
Eigenvalues and Eigenvectors 6-10
Diagonalization 6-13
Summary of Results 6-15
Solved Problems 6-17
Epilogue 6-28
Further Reading 6-29
Exercises 6-30

7 Supervised Hebbian Learning



Objectives	7-1
Theory and Examples	7-2
Linear Associator	7-3
The Hebb Rule	7-4
Performance Analysis	7-5
Pseudoinverse Rule	7-7
Application	7-10
Variations of Hebbian Learning	7-12
Summary of Results	17-4
Solved Problems	7-16
Epilogue	7-29
Further Reading	7-30
Exercises	7-31

8 Performance Surfaces and Optimum Points



Objectives	8-1
Theory and Examples	8-2
Taylor Series	8-2
Vector Case	8-4
Directional Derivatives	8-5
Minima	8-7
Necessary Conditions for Optimality	8-9
First-Order Conditions	8-10
Second-Order Conditions	8-11
Quadratic Functions	8-12
Eigensystem of the Hessian	8-13
Summary of Results	8-20
Solved Problems	8-22
Epilogue	8-34
Further Reading	8-35
Exercises	8-36

9 **Performance Optimization**



Objectives 9-1
Theory and Examples 9-2
Steepest Descent 9-2
Stable Learning Rates 9-6
Minimizing Along a Line 9-8
Newton's Method 9-10
Conjugate Gradient 9-15
Summary of Results 9-21
Solved Problems 9-23
Epilogue 9-37
Further Reading 9-38
Exercises 9-39

10 **Widrow-Hoff Learning**



Objectives 10-1
Theory and Examples 10-2
ADALINE Network 10-2
Single ADALINE 10-3
Mean Square Error 10-4
LMS Algorithm 10-7
Analysis of Convergence 10-9
Adaptive Filtering 10-13
Adaptive Noise Cancellation 10-15
Echo Cancellation 10-21
Summary of Results 10-22
Solved Problems 10-24
Epilogue 10-40
Further Reading 10-41
Exercises 10-42



Backpropagation



Objectives	11-1
Theory and Examples	11-2
Multilayer Perceptrons	11-2
Pattern Classification	11-3
Function Approximation	11-4
The Backpropagation Algorithm	11-7
Performance Index	11-8
Chain Rule	11-9
Backpropagating the Sensitivities	11-11
Summary	11-13
Example	11-14
Batch vs. Incremental Training	11-17
Using Backpropagation	11-18
Choice of Network Architecture	11-18
Convergence	11-20
Generalization	11-22
Summary of Results	11-25
Solved Problems	11-27
Epilogue	11-41
Further Reading	11-42
Exercises	11-44



Variations on Backpropagation



Objectives	12-1
Theory and Examples	12-2
Drawbacks of Backpropagation	12-3
Performance Surface Example	12-3
Convergence Example	12-7
Heuristic Modifications of Backpropagation	12-9
Momentum	12-9
Variable Learning Rate	12-12
Numerical Optimization Techniques	12-14
Conjugate Gradient	12-14
Levenberg-Marquardt Algorithm	12-19
Summary of Results	12-28
Solved Problems	12-32
Epilogue	12-46
Further Reading	12-47
Exercises	12-50



Generalization



Objectives	13-1
Theory and Examples	13-2
Problem Statement	13-2
Methods for Improving Generalization	13-5
Estimating Generalization Error	13-6
Early Stopping	13-6
Regularization	13-8
Bayesian Analysis	13-10
Bayesian Regularization	13-12
Relationship Between Early Stopping and Regularization	13-19
Summary of Results	13-29
Solved Problems	13-32
Epilogue	13-44
Further Reading	13-45
Exercises	13-47



Dynamic Networks



Objectives	14-1
Theory and Examples	14-2
Layered Digital Dynamic Networks	14-3
Example Dynamic Networks	14-5
Principles of Dynamic Learning	14-8
Dynamic Backpropagation	14-12
Preliminary Definitions	14-12
Real Time Recurrent Learning	14-12
Backpropagation-Through-Time	14-22
Summary and Comments on Dynamic Training	14-30
Summary of Results	14-34
Solved Problems	14-37
Epilogue	14-46
Further Reading	14-47
Exercises	14-48



Associative Learning



Objectives	15-1
Theory and Examples	15-2
Simple Associative Network	15-3
Unsupervised Hebb Rule	15-5
Hebb Rule with Decay	15-7
Simple Recognition Network	15-9
Instar Rule	15-11
Kohonen Rule	15-15
Simple Recall Network	15-16
Outstar Rule	15-17
Summary of Results	15-21
Solved Problems	15-23
Epilogue	15-34
Further Reading	15-35
Exercises	15-37



Competitive Networks



Objectives	16-1
Theory and Examples	16-2
Hamming Network	16-3
Layer 1	16-3
Layer 2	16-4
Competitive Layer	16-5
Competitive Learning	16-7
Problems with Competitive Layers	16-9
Competitive Layers in Biology	16-10
Self-Organizing Feature Maps	16-12
Improving Feature Maps	16-15
Learning Vector Quantization	16-16
LVQ Learning	16-18
Improving LVQ Networks (LVQ2)	16-21
Summary of Results	16-22
Solved Problems	16-24
Epilogue	16-37
Further Reading	16-38
Exercises	16-39

17 Radial Basis Networks



Objectives	17-1
Theory and Examples	17-2
Radial Basis Network	17-2
Function Approximation	17-4
Pattern Classification	17-6
Global vs. Local	17-9
Training RBF Networks	17-10
Linear Least Squares	17-11
Orthogonal Least Squares	17-18
Clustering	17-23
Nonlinear Optimization	17-25
Other Training Techniques	17-26
Summary of Results	17-27
Solved Problems	17-30
Epilogue	17-35
Further Reading	17-36
Exercises	17-38

18 Grossberg Network



Objectives	18-1
Theory and Examples	18-2
Biological Motivation: Vision	18-3
Illusions	18-4
Vision Normalization	18-8
Basic Nonlinear Model	18-9
Two-Layer Competitive Network	18-12
Layer 1	18-13
Layer 2	18-17
Choice of Transfer Function	18-20
Learning Law	18-22
Relation to Kohonen Law	18-24
Summary of Results	18-26
Solved Problems	18-30
Epilogue	18-42
Further Reading	18-43
Exercises	18-45

19 Adaptive Resonance Theory

Objectives 19-1
Theory and Examples 19-2
Overview of Adaptive Resonance 19-2
Layer 1 19-4
Steady State Analysis 19-6
Layer 2 19-10
Orienting Subsystem 19-13
Learning Law: L1-L2 19-17
Subset/Superset Dilemma 19-17
Learning Law 19-18
Learning Law: L2-L1 19-20
ART1 Algorithm Summary 19-21
Initialization 19-21
Algorithm 19-21
Other ART Architectures 19-23
Summary of Results 19-25
Solved Problems 19-30
Epilogue 19-45
Further Reading 19-46
Exercises 19-48

20 Stability

Objectives 20-1
Theory and Examples 20-2
Recurrent Networks 20-2
Stability Concepts 20-3
Definitions 20-4
Lyapunov Stability Theorem 20-5
Pendulum Example 20-6
LaSalle's Invariance Theorem 20-12
Definitions 20-12
Theorem 20-13
Example 20-14
Comments 20-18
Summary of Results 20-19
Solved Problems 20-21
Epilogue 20-28
Further Reading 20-29
Exercises 30

21 Hopfield Network

Objectives 21-1
Theory and Examples 21-2
Hopfield Model 21-3
Lyapunov Function 21-5
Invariant Sets 21-7
Example 21-7
Hopfield Attractors 21-11
Effect of Gain 21-12
Hopfield Design 21-16
Content-Addressable Memory 21-16
Hebb Rule 21-18
Lyapunov Surface 21-22
Summary of Results 21-24
Solved Problems 21-26
Epilogue 21-36
Further Reading 21-37
Exercises 21-40

22 Practical Training Issues

Objectives 22-1
Theory and Examples 22-2
Pre-Training Steps 22-3
Selection of Data 22-3
Data Preprocessing 22-5
Choice of Network Architecture 22-8
Training the Network 22-13
Weight Initialization 22-13
Choice of Training Algorithm 22-14
Stopping Criteria 22-14
Choice of Performance Function 22-16
Committees of Networks 22-18
Post-Training Analysis 22-18
Fitting 22-18
Pattern Recognition 22-21
Clustering 22-23
Prediction 22-24
Overfitting and Extrapolation 22-27
Sensitivity Analysis 22-28
Epilogue 22-30
Further Reading 22-31

23 **Case Study 1:Function Approximation**



Objectives 23-1
Theory and Examples 23-2
Description of the Smart Sensor System 23-2
Data Collection and Preprocessing 23-3
Selecting the Architecture 23-4
Training the Network 23-5
Validation 23-7
Data Sets 23-10
Epilogue 23-11
Further Reading 23-12

24 **Case Study 2:Probability Estimation**



Objectives 24-1
Theory and Examples 24-2
Description of the CVD Process 24-2
Data Collection and Preprocessing 24-3
Selecting the Architecture 24-5
Training the Network 24-7
Validation 24-9
Data Sets 24-12
Epilogue 24-13
Further Reading 24-14

25 **Case Study 3:Pattern Recognition**



Objectives 25-1
Theory and Examples 25-2
Description of Myocardial Infarction Recognition 25-2
Data Collection and Preprocessing 25-3
Selecting the Architecture 25-6
Training the Network 25-7
Validation 25-7
Data Sets 25-10
Epilogue 25-11
Further Reading 25-12

26 Case Study 4: Clustering

Objectives 26-1
Theory and Examples 26-2
Description of the Forest Cover Problem 26-2
Data Collection and Preprocessing 26-4
Selecting the Architecture 26-5
Training the Network 26-6
Validation 26-7
Data Sets 26-11
Epilogue 26-12
Further Reading 26-13

27 Case Study 5: Prediction

Objectives 27-1
Theory and Examples 27-2
Description of the Magnetic Levitation System 27-2
Data Collection and Preprocessing 27-3
Selecting the Architecture 27-4
Training the Network 27-6
Validation 27-8
Data Sets 27-13
Epilogue 27-14
Further Reading 27-15

Appendices

A Bibliography

B Notation

C Software

I Index

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.

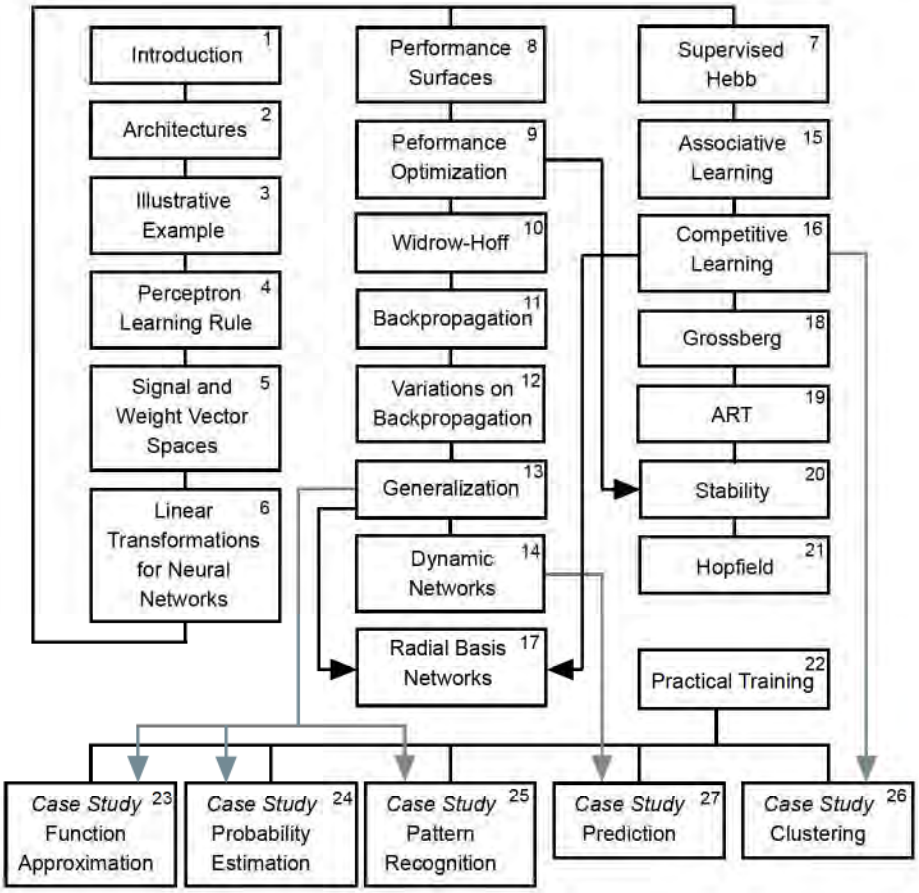
Since this is a book on the design of neural networks, our choice of topics was guided by two principles. First, we wanted to present the most useful and practical neural network architectures, learning rules and training techniques. Second, we wanted the book to be complete in itself and to flow easily 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	1-1
History	1-2
Applications	1-5
Biological Inspiration	1-8
Further Reading	1-10

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.

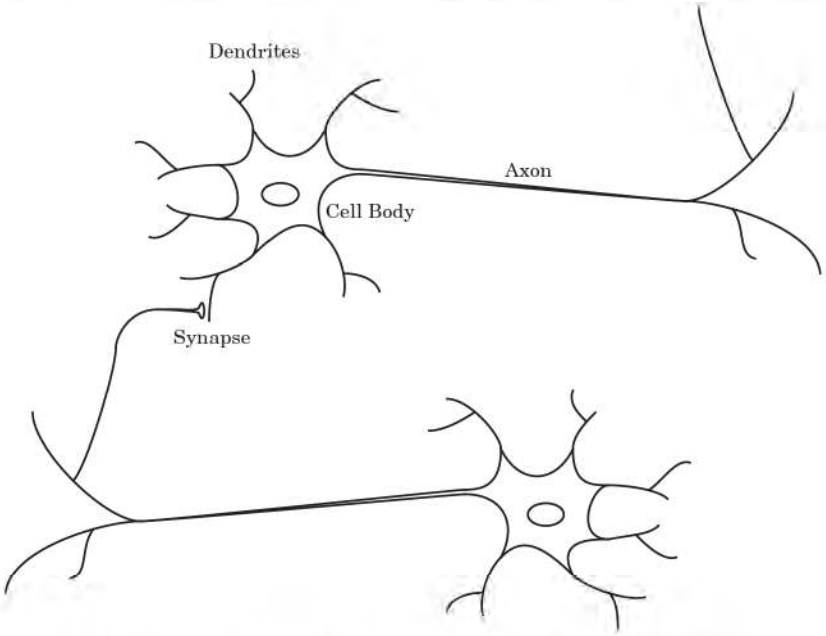


Figure 1.1 Schematic Drawing of Biological Neurons

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	2-1
Theory and Examples	2-2
Notation	2-2
Neuron Model	2-2
Single-Input Neuron	2-2
Transfer Functions	2-3
Multiple-Input Neuron	2-7
Network Architectures	2-9
A Layer of Neurons	2-9
Multiple Layers of Neurons	2-10
Recurrent Networks	2-13
Summary of Results	2-16
Solved Problems	2-20
Epilogue	2-22
Exercises	2-23

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: $\mathbf{a},\mathbf{b},\mathbf{c}$

Matrices — capital **BOLD** nonitalic letters: $\mathbf{A},\mathbf{B},\mathbf{C}$

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.

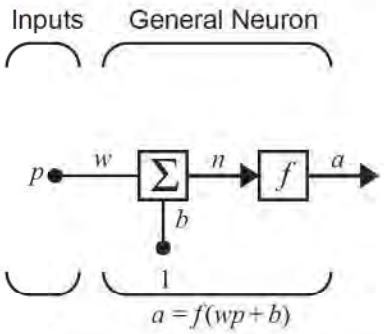


Figure 2.1 Single-Input Neuron

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.

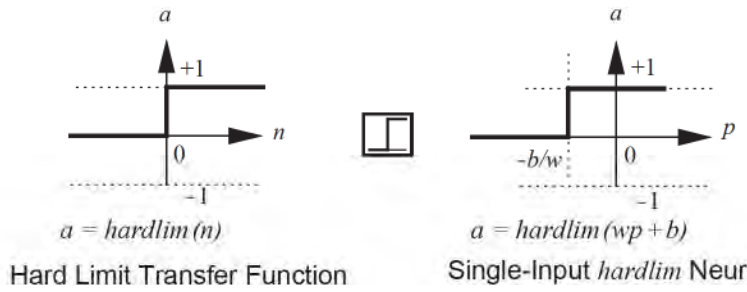


Figure 2.2 Hard Limit Transfer Function

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.

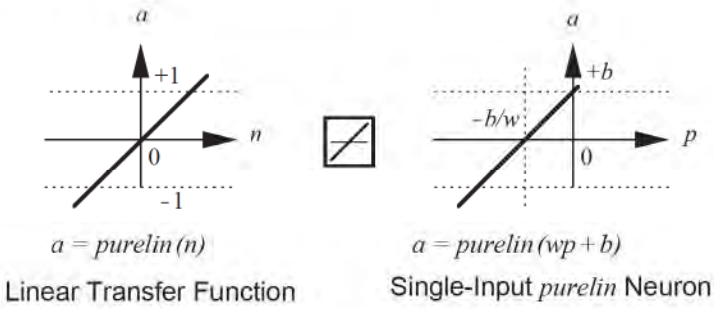


Figure 2.3 Linear Transfer Function

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.

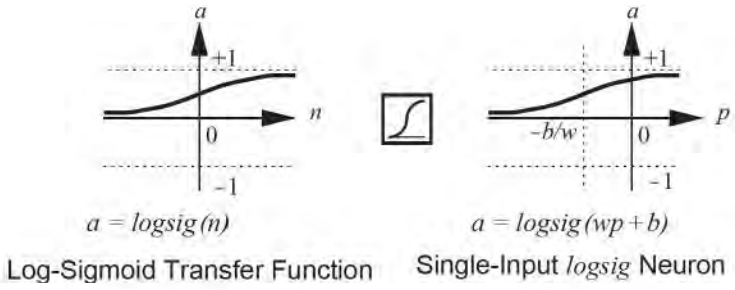


Figure 2.4 Log-Sigmoid Transfer Function

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

Weight Matrix

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* $\mathbf{W}$.

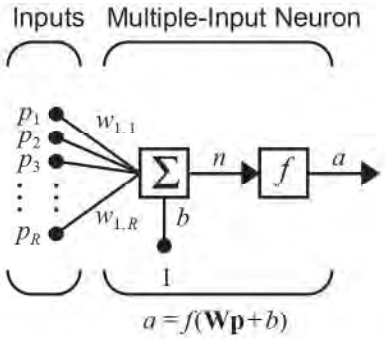


Figure 2.5 Multiple-Input Neuron

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.

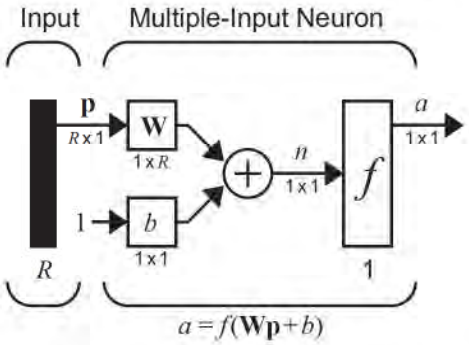


Figure 2.6 Neuron with R Inputs, Abbreviated Notation

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.

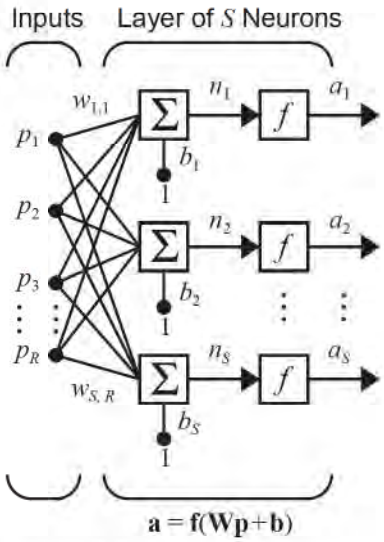


Figure 2.7 Layer of S Neurons

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

Each element of the input vector **p** is connected to each neuron through the weight matrix **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 **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.

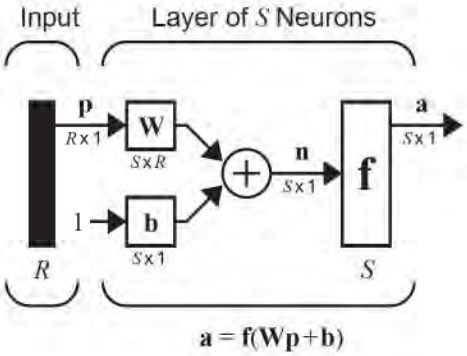


Figure 2.8 Layer of S Neurons, Abbreviated Notation

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.

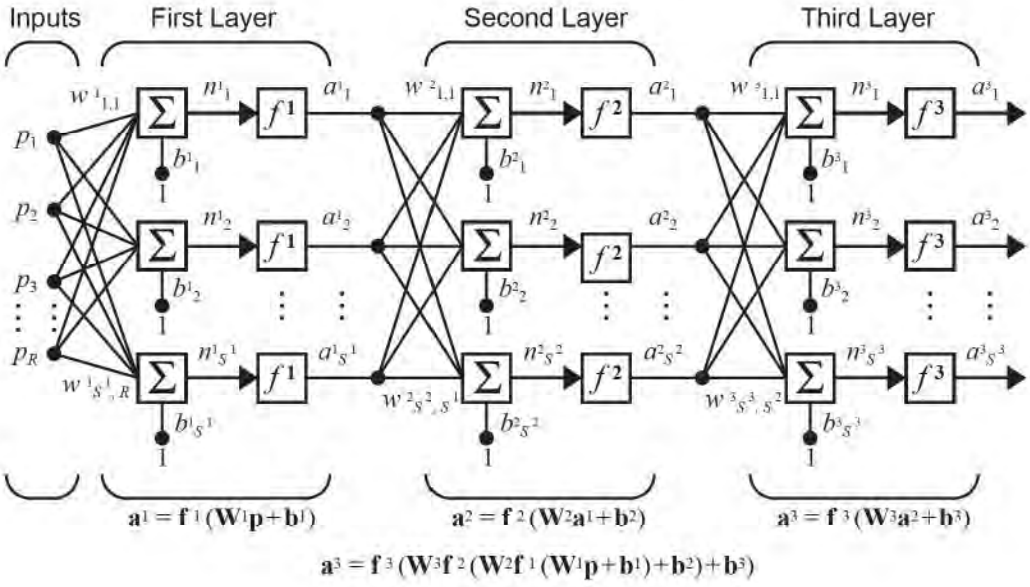


Figure 2.9 Three-Layer Network

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.

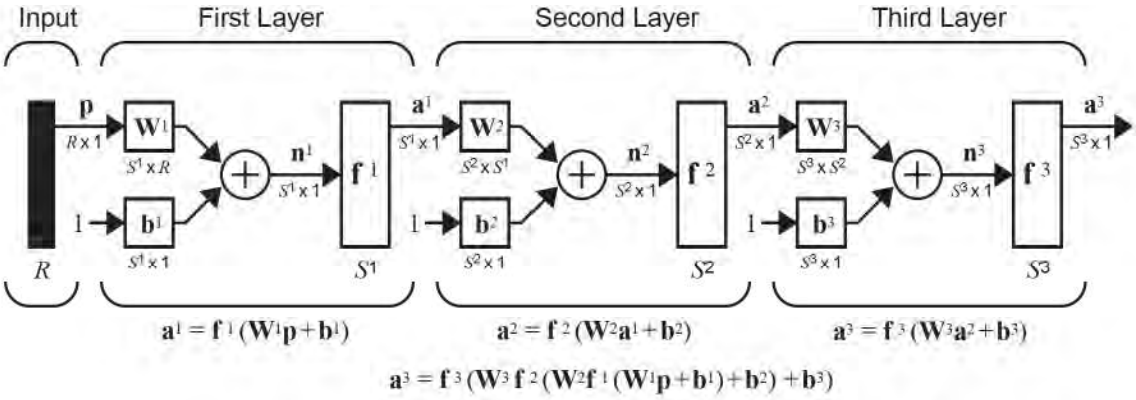


Figure 2.10 Three-Layer Network, Abbreviated Notation

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.

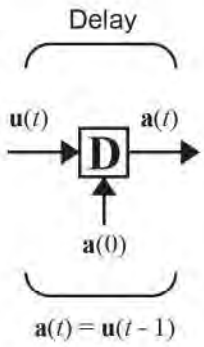


Figure 2.11 Delay Block

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.

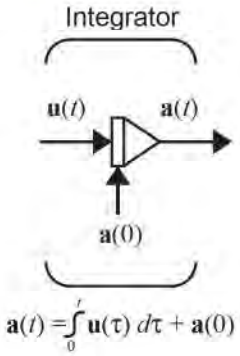


Figure 2.12 Integrator Block

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.

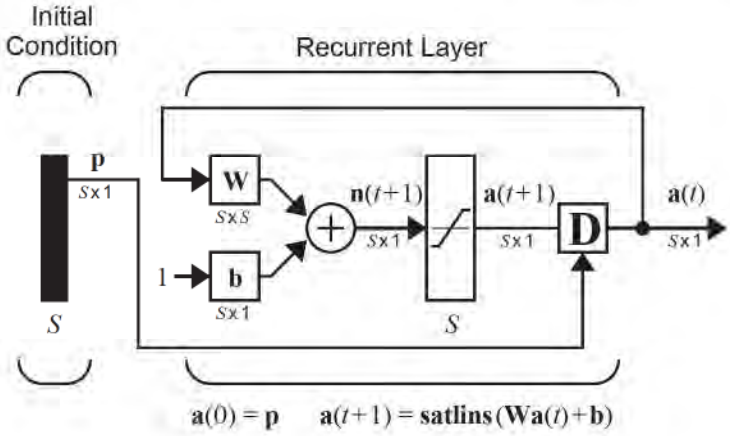


Figure 2.13 Recurrent Network

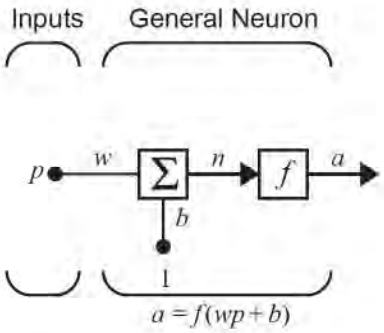
In this particular network the vector **p** supplies the initial conditions (i.e., $\mathbf{a}(0) = \mathbf{p}$). Then future outputs of the network are computed from previous outputs:

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

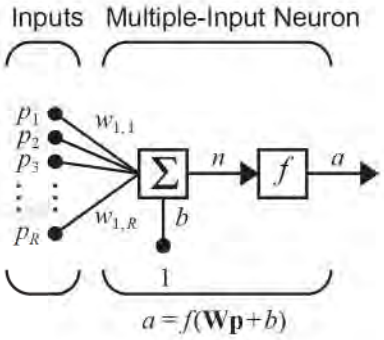
Recurrent networks are potentially more powerful than feedforward networks and can exhibit temporal behavior. These types of networks are discussed in Chapters 3, 14 and 18–21.

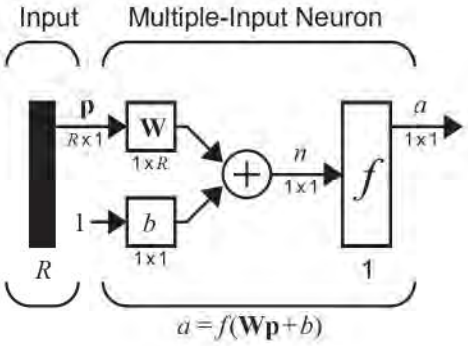
Summary of Results

Single-Input Neuron



Multiple-Input Neuron

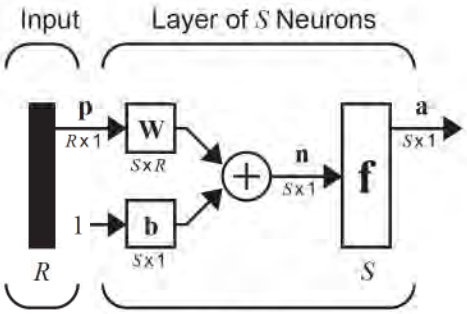




Transfer Functions

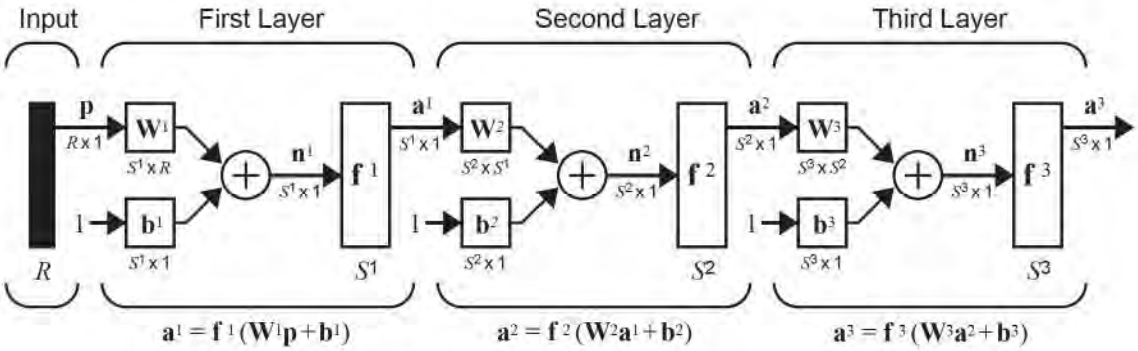
Name	Input/Output Relation	Icon	MATLAB Function
Hard Limit	$a = 0 \quad n < 0$ $a = 1 \quad n \geq 0$		hardlim
Symmetrical Hard Limit	$a = -1 \quad n < 0$ $a = +1 \quad n \geq 0$		hardlims
Linear	$a = n$		purelin
Saturating Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n \leq 1$ $a = 1 \quad n > 1$		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



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

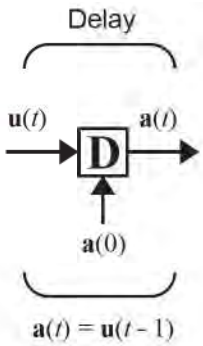
Three Layers of Neurons



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

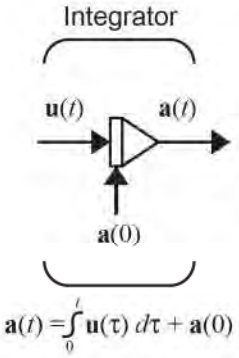
$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$$

Delay

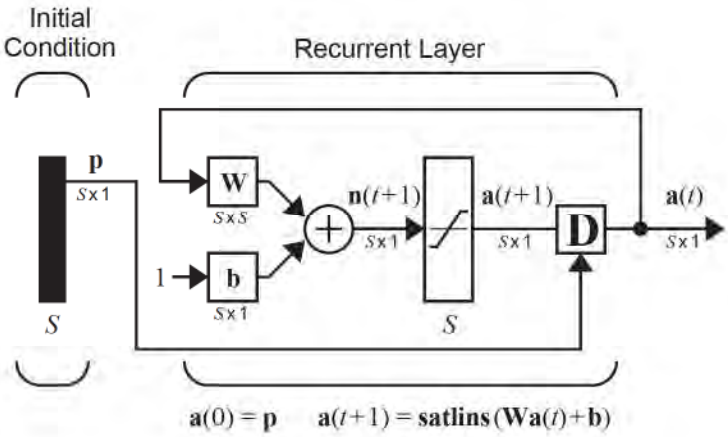


$$\mathbf{a}(t) = \mathbf{u}(t-1)$$

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 **Wp** 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.

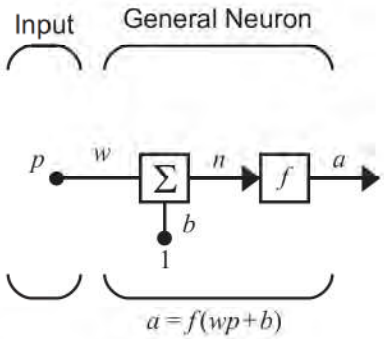


Figure P15.1 General Neuron

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

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

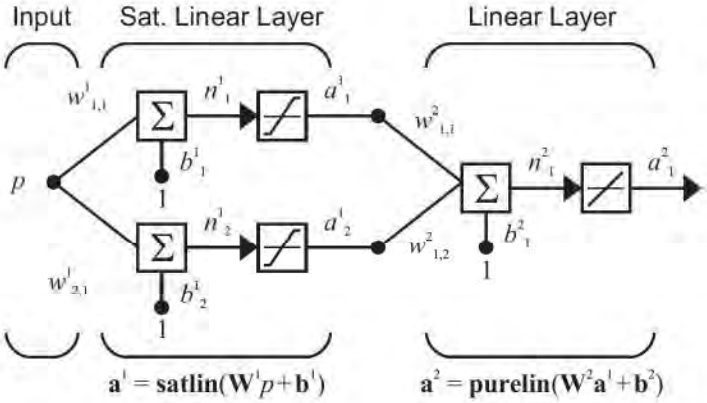
ii. $w = -1, b = 1, f = hardlims$.

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	3-1
Theory and Examples	3-2
Problem Statement	3-2
Perceptron	3-3
Two-Input Case	3-4
Pattern Recognition Example	3-5
Hamming Network	3-8
Feedforward Layer	3-9
Recurrent Layer	3-10
Hopfield Network	3-12
Epilogue	3-15
Exercises	3-16

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.

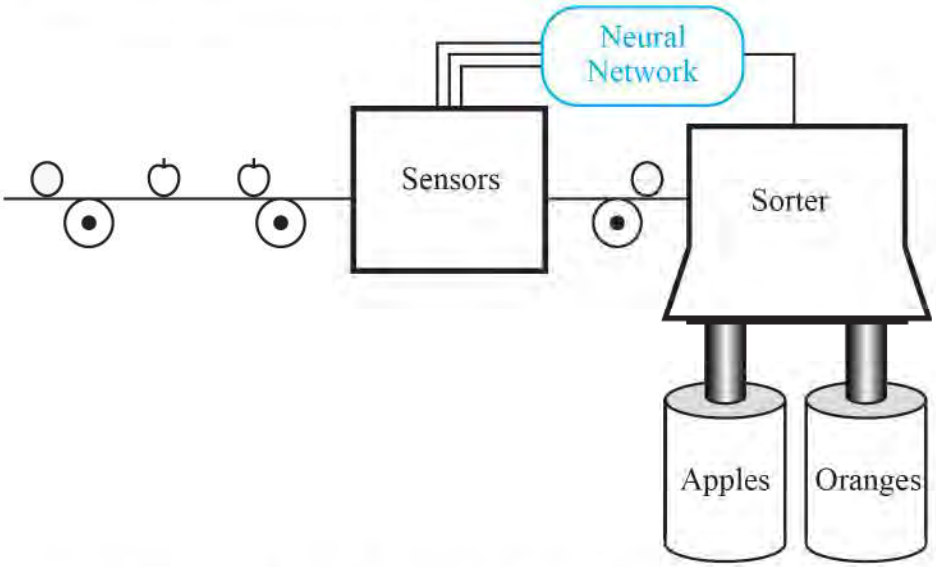
The three networks presented in this chapter are representative of the types of networks discussed in the remaining chapters: feedforward networks (represented here by the perceptron), competitive networks (represented here by the Hamming network) and recurrent associative memory networks (represented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

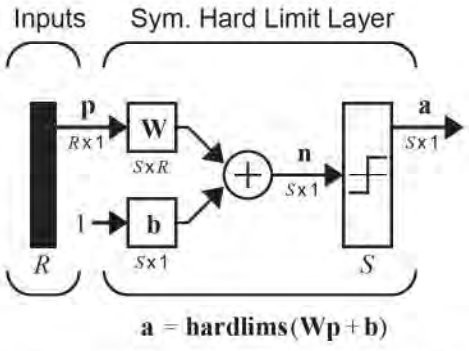


Figure 3.1 Single-Layer Perceptron

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.

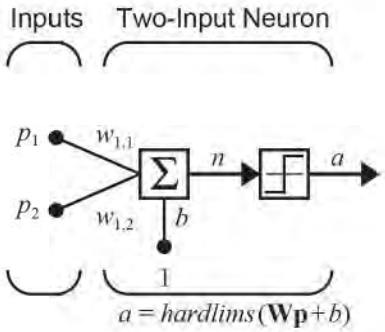


Figure 3.2 Two-Input/Single-Neuron Perceptron

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.

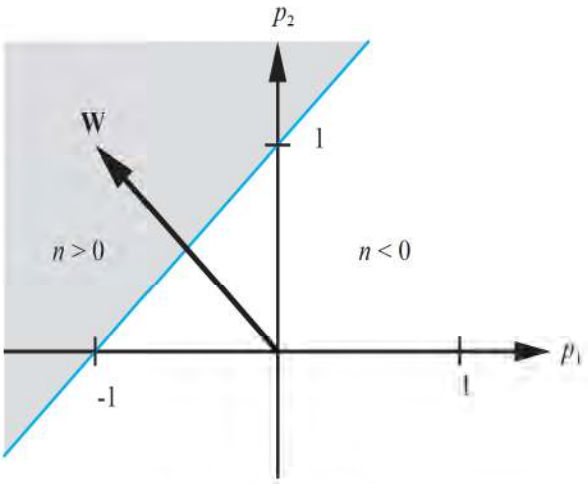


Figure 3.3 Perceptron Decision Boundary

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.

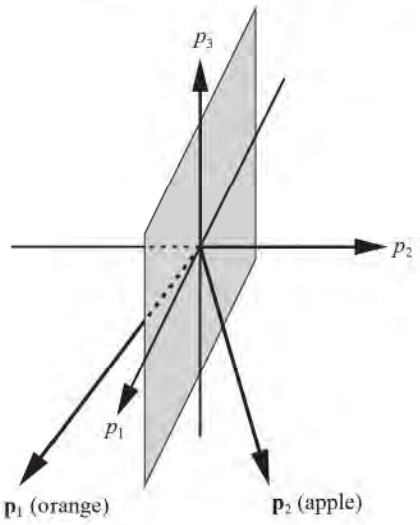


Figure 3.4 Prototype Vectors

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.

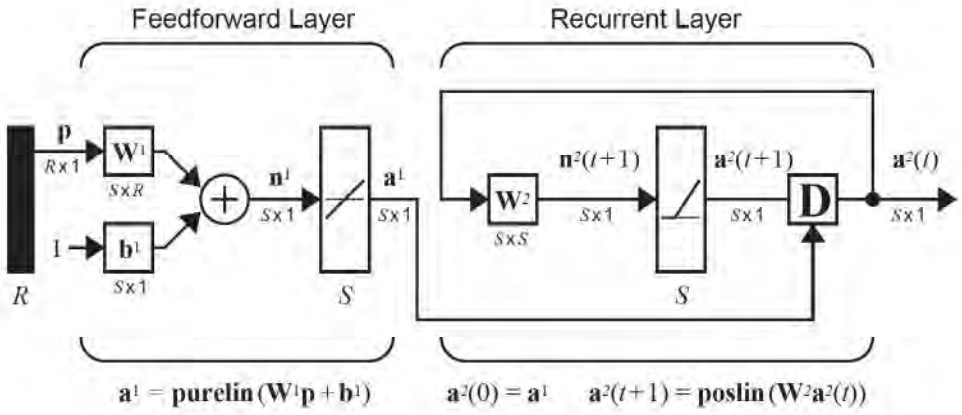


Figure 3.5 Hamming Network

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.

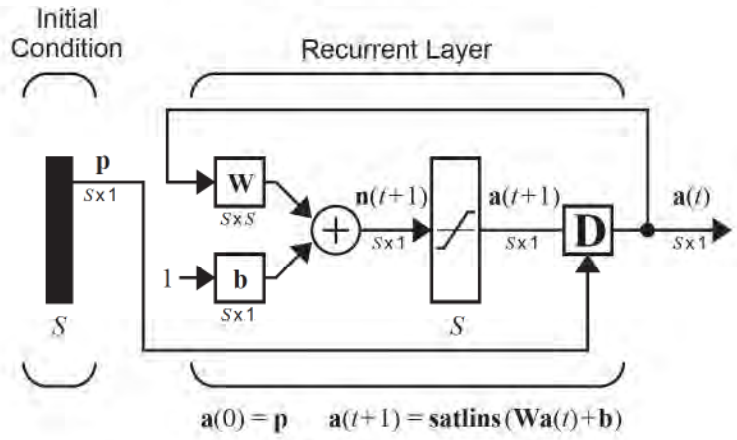


Figure 3.6 Hopfield Network

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.

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

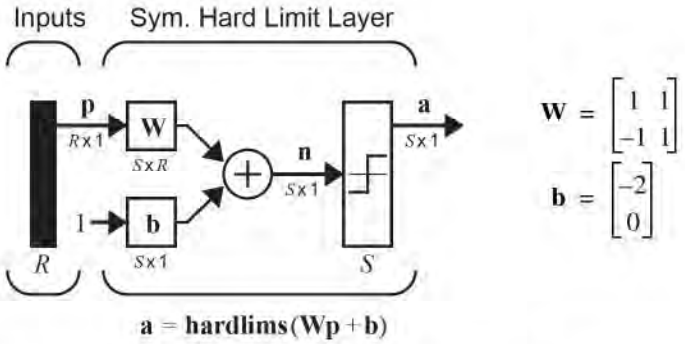
i. The following input (initial condition) is applied to the network. Find the network response (show the network output at each iteration until the network converges).

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

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.



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	4-1
Theory and Examples	4-2
Learning Rules	4-2
Perceptron Architecture	4-3
Single-Neuron Perceptron	4-5
Multiple-Neuron Perceptron	4-8
Perceptron Learning Rule	4-8
Test Problem	4-9
Constructing Learning Rules	4-10
Unified Learning Rule	4-12
Training Multiple-Neuron Perceptrons	4-13
Proof of Convergence	4-15
Notation	4-15
Proof	4-16
Limitations	4-18
Summary of Results	4-20
Solved Problems	4-21
Epilogue	4-34
Further Reading	4-35
Exercises	4-37

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.)

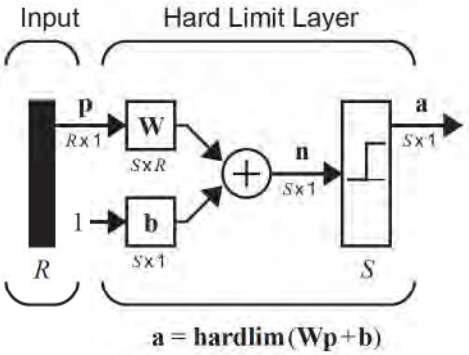


Figure 4.1 Perceptron Network

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 $\mathbf{W}$:

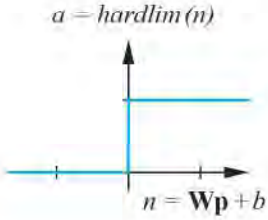
$${}_i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix}. \tag{4.4}$$

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.

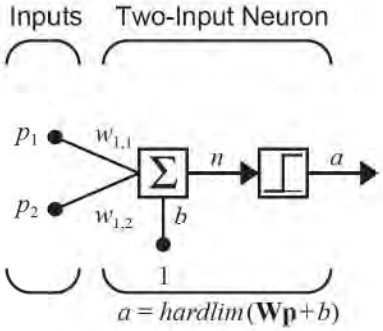


Figure 4.2 Two-Input/Single-Output Perceptron

The output of this network is determined by

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

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.

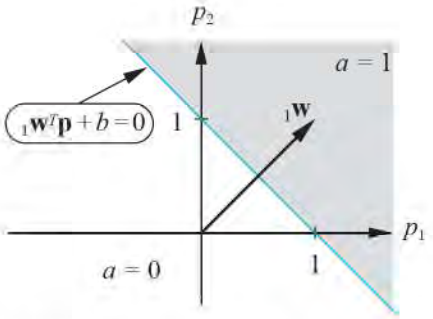
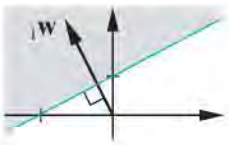


Figure 4.3 Decision Boundary for Two-Input Perceptron



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

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



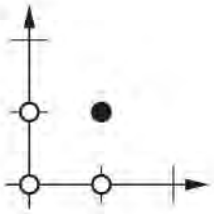
For all points on the boundary, the inner product of the input vector with the weight vector is the same. This implies that these input vectors will all have the same projection onto the weight vector, so they must lie on a line orthogonal to the weight vector. (These concepts will be covered in more 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

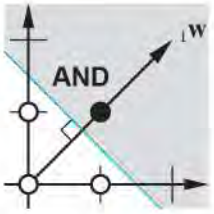
$$\left\{\mathbf{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 0\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 0\right\}\left\{\mathbf{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 1\right\}.$$



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

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

$$ a = hardlim(-1) = 0, \tag{4.18} $$

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:

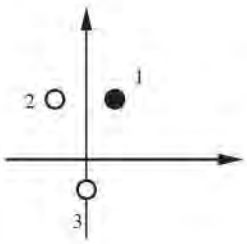
$$ \{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}, \tag{4.20} $$

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.

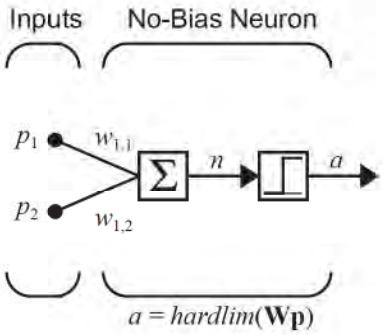
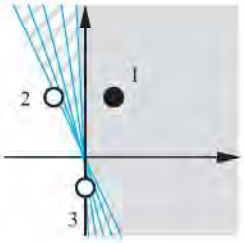
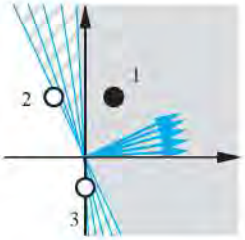


Figure 4.4 Test Problem Network



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

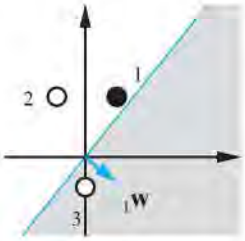
Training begins by assigning some initial values for the network parameters. In this case we are training a two-input/single-output network without a bias, so we only have to initialize its two weights. Here we set the elements of the weight vector, ${}_1\mathbf{w}$, to the following randomly generated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

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

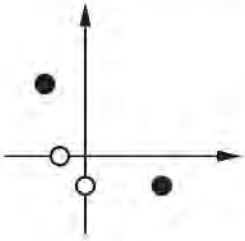
$$a = hardlim(-0.6) = 0. \tag{4.22}$$



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.

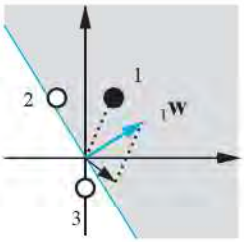


Another possibility would be to add $\mathbf{p}_1$ to ${}_1\mathbf{w}$. Adding $\mathbf{p}_1$ to ${}_1\mathbf{w}$ would make ${}_1\mathbf{w}$ point more in the direction of $\mathbf{p}_1$. Repeated presentations of $\mathbf{p}_1$ would cause the direction of ${}_1\mathbf{w}$ to asymptotically approach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

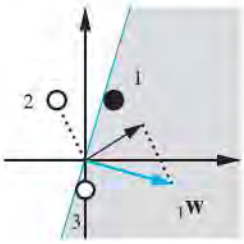
$$ \begin{aligned} a &= hardlim({}_1\mathbf{w}^T\mathbf{p}_2) = hardlim\left(\begin{bmatrix} 2.0 & 1.2 \end{bmatrix}\begin{bmatrix} -1 \\ 2 \end{bmatrix}\right) \\ &= hardlim(0.4) = 1 . \end{aligned} \tag{4.25} $$

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:



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

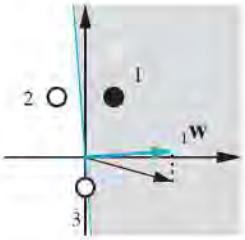
which is illustrated in the adjacent figure.

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 ith row of the weight matrix use:

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

To update the ith 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.

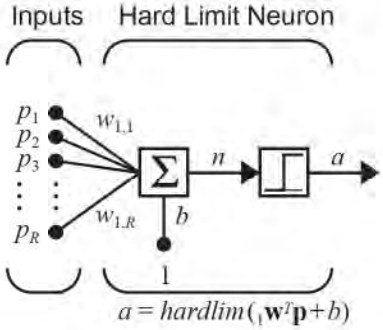


Figure 4.5 Single-Neuron Perceptron

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.

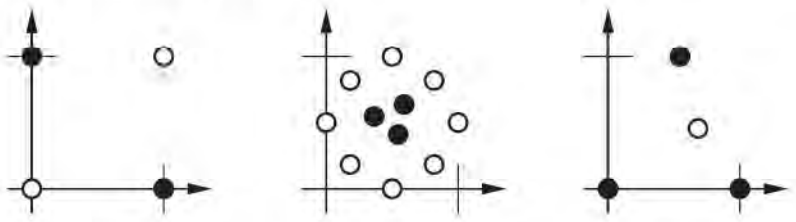
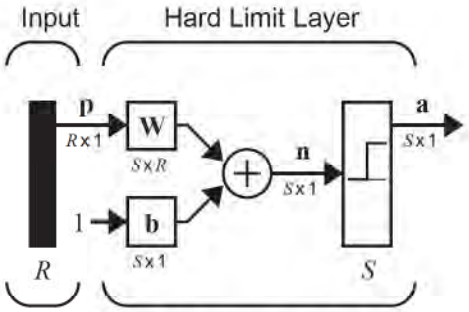


Figure 4.6 Linearly Inseparable Problems

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.

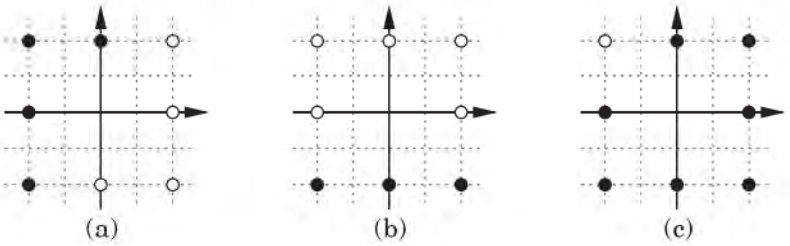
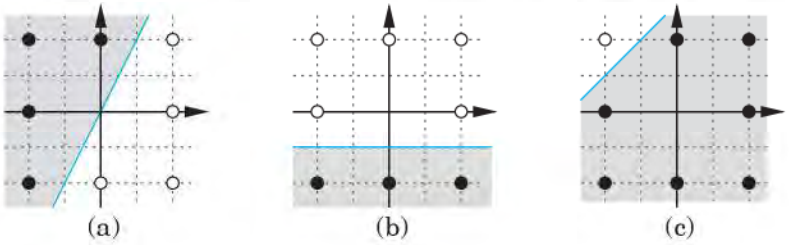
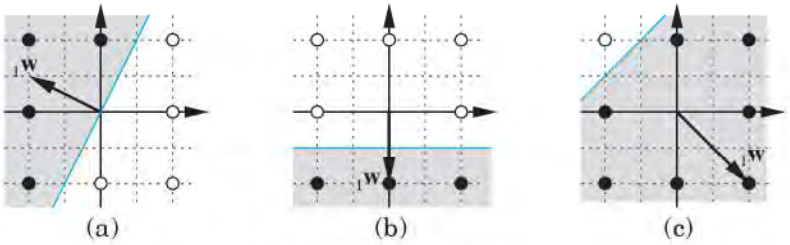


Figure P4.1 Simple Classification Problems

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:

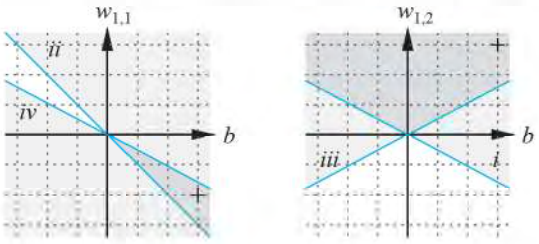
$$\begin{aligned} \mathbf{W}\mathbf{p}_1 + b &\geq 0 \\ 0w_{1,1} + 2w_{1,2} + b &\geq 0 \\ 2w_{1,2} + b &\geq 0. \end{aligned}$$

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.

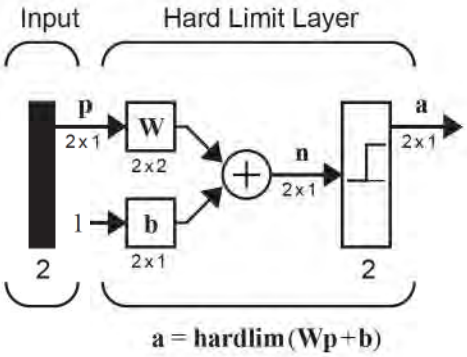


Figure P4.2 Two-Neuron Perceptron

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.

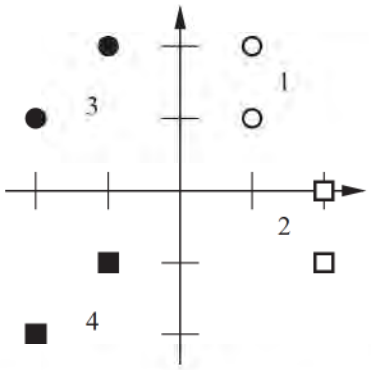


Figure P4.3 Input Vectors for Problem P4.3

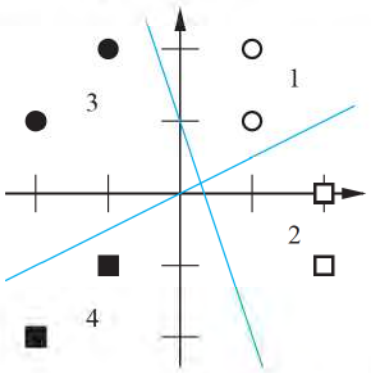


Figure P4.4 Tentative Decision Boundaries for Problem P4.3

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):

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

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

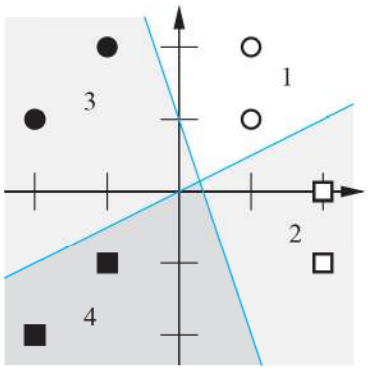


Figure P4.5 Decision Regions for Problem P4.3

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 $p_1 = 0$:

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

.

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

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

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

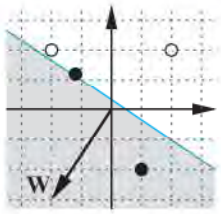


Figure P4.6 Decision Boundary for Problem P4.4

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.

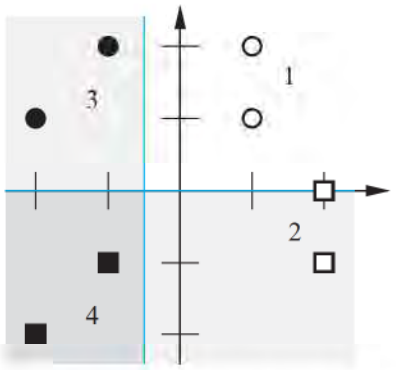


Figure P4.7 Final Decision Boundaries for Problem P4.5

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.

```
» 2 + 2
ans =
     4
```

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

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

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.

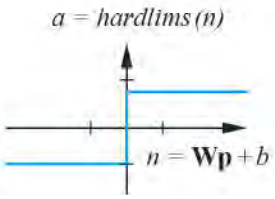
iii. Sketch the new decision boundary and weight vector, and again indicate which of the three input vectors are correctly classified.

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.



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

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

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.



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	5-1
Theory and Examples	5-2
Linear Vector Spaces	5-2
Linear Independence	5-4
Spanning a Space	5-5
Inner Product	5-6
Norm	5-7
Orthogonality	5-7
Gram-Schmidt Orthogonalization	5-8
Vector Expansions	5-9
Reciprocal Basis Vectors	5-10
Summary of Results	5-14
Solved Problems	5-17
Epilogue	5-26
Further Reading	5-27
Exercises	5-28

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.,

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}^T. \tag{5.1}$$

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 $x + (-x) = 0$.

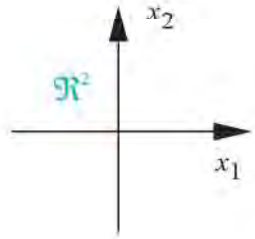
6. An operation, called multiplication, is defined such that for all scalars $a \in F$, and all vectors $x \in X$, $ax \in X$.

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

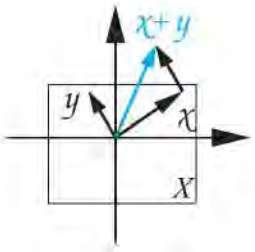
8. For any two scalars $a \in F$ and $b \in F$, and any $x \in X$, $a(bx) = (ab)x$.

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

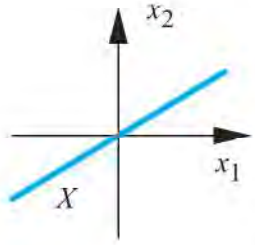
10. $a(x + y) = ax + ay$.



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.



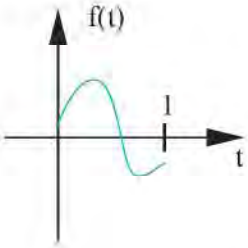
Are there any subsets of $\Re^2$ that are vector spaces? Consider the line (X) shown in the bottom left figure. (Assume that the line extends to infinity in both directions.) Is this line a vector space? We leave it to you to show that indeed all ten conditions are satisfied. Will any such infinite line 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.

Basis Set

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. 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. $(x,ay_1 + by_2) = a(x,y_1) + b(x,y_2)$.
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

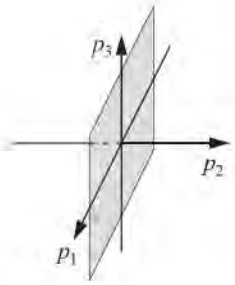
Now that we have defined the inner product operation, we can introduce the important concept of orthogonality.

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 $\Re^2$:

$$\mathbf{y}_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \mathbf{y}_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \tag{5.21}$$

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.

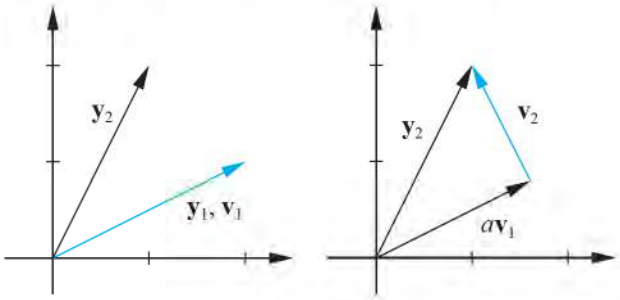


Figure 5.1 Gram-Schmidt Orthogonalization Example

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 $\mathbf{R}$ 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 $\mathbf{R}$.

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

$$x_j = (r_j, x) . \tag{5.38}$$

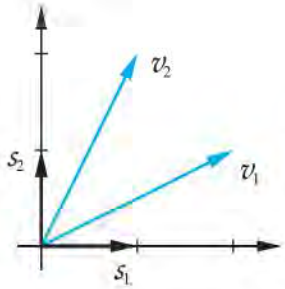


As an example, consider the two basis vectors

$$\mathbf{v}_1^s = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \; \mathbf{v}_2^s = \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \tag{5.39}$$

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.

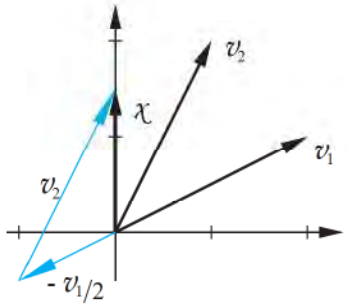


Figure 5.2 Vector Expansion

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 $b = 0$.

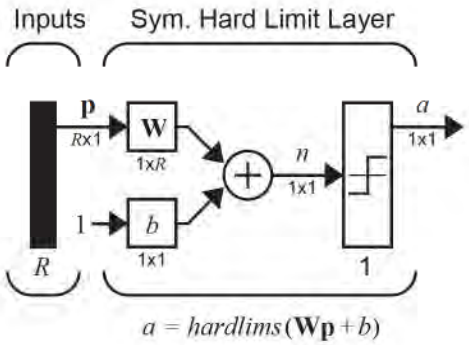


Figure P5.1 Single-Neuron Perceptron

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

$$2\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} - 1\begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} - 1\begin{bmatrix} 1 \\ 2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

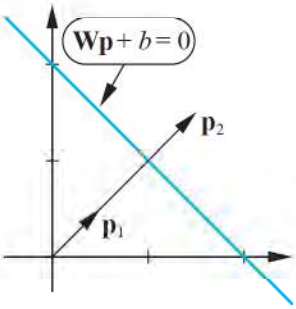
The dimension of the space must therefore be less than 3. We can show that $\mathbf{x}_1$ and $\mathbf{x}_2$ are independent, since

$$G = \begin{vmatrix} 4 & 3 \\ 3 & 3 \end{vmatrix} = 4 \neq 0.$$

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 $\mathbf{p}_2 = 3\mathbf{p}_1$.

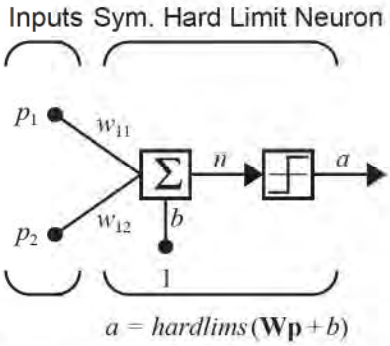


Figure P5.2 Two-Input Perceptron

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

$$(v_1, y_2) = \int_{-1}^{1} (1 + t)(1 - t)dt = \left(t - \frac{t^3}{3}\right)\Big|_{-1}^{1} = \left(\frac{2}{3}\right) - \left(-\frac{2}{3}\right) = \frac{4}{3}$$

$$(v_1, v_1) = \int_{-1}^{1} (1 + t)^2 dt = \frac{(1 + t)^3}{3}\Big|_{-1}^{1} = \left(\frac{8}{3}\right) - (0) = \frac{8}{3}.$$

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`.)

```
» 2+2
ans =
     4
```

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.)

```
» 2+2
ans =
     4
```

$$\mathbf{y}_1 = \begin{bmatrix}1\\0\\0\end{bmatrix} \qquad \mathbf{y}_2 = \begin{bmatrix}1\\1\\0\end{bmatrix} \qquad \mathbf{y}_3 = \begin{bmatrix}1\\1\\1\end{bmatrix}$$

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

$$(f,g) = \int_0^1 f(t)g(t)dt .$$

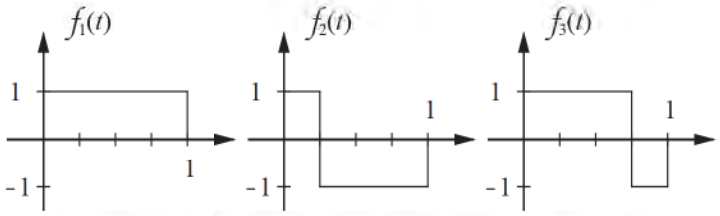


Figure E15.1 Basis Set for Exercise E5.11

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.

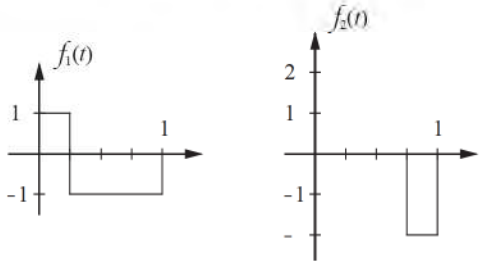


Figure E15.2 Basis Set for Exercise E5.12

i. Generate an orthogonal set using the Gram-Schmidt procedure. The inner product is defined to be

$$(f,g) = \int_0^1 f(t)g(t)dt .$$

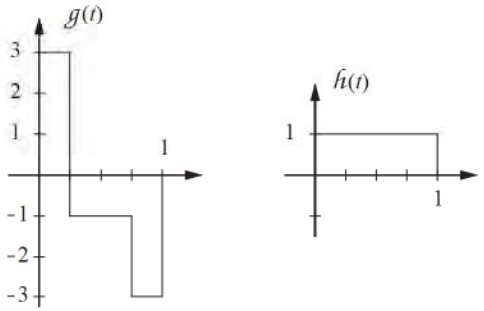


Figure E15.3 Vectors vectors g and h for Exercise E5.12 part ii.

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 χ can be expanded in terms of the basis vectors $\{v_1, v_2\}$ as

$$\chi = 1\,v_1 + 1\,v_2$$

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.

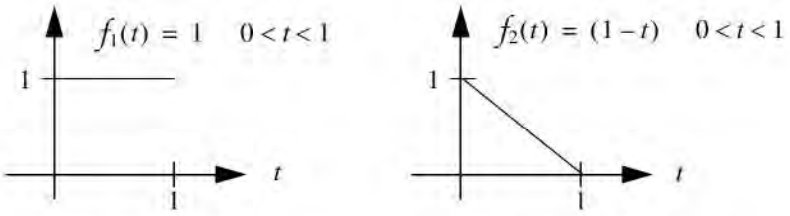


Figure E15.4 Independent Vectors for Exercise E5.15

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 .

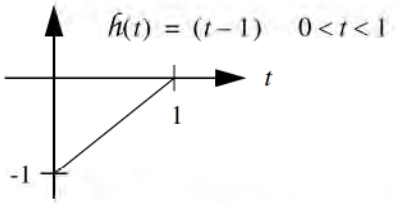


Figure E15.5 Vector h for Exercise E5.15

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.

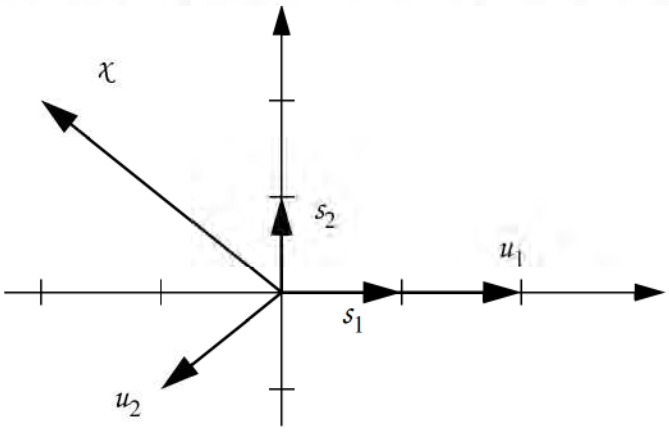


Figure E15.6 Vector Definitions for Exercise E5.17

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.)



$$\mathbf{v}_1 = \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix} \qquad \mathbf{v}_2 = \begin{bmatrix} 1 \\ 1 \\ -2 \end{bmatrix} \qquad \mathbf{v}_3 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

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	6-1
Theory and Examples	6-2
Linear Transformations	6-2
Matrix Representations	6-3
Change of Basis	6-6
Eigenvalues and Eigenvectors	6-10
Diagonalization	6-13
Summary of Results	6-15
Solved Problems	6-17
Epilogue	6-29
Further Reading	6-30
Exercises	6-31

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.

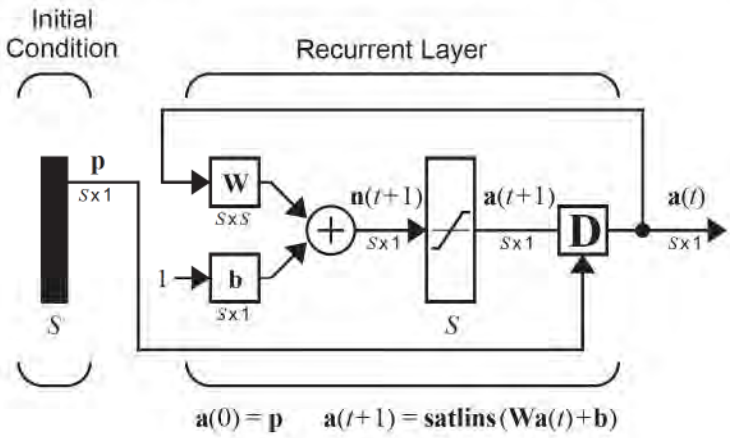


Figure 6.1 Hopfield Network

Linear Transformations

We begin with some general definitions.

Transformation

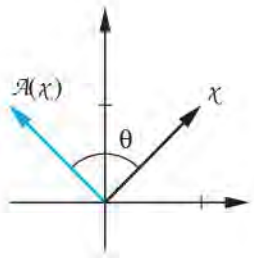
A *transformation* consists of three parts:

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

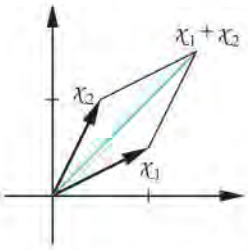
A transformation $\mathcal{A}$ is *linear* if:

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 $\chi \in X$, $a \in R$, $\mathcal{A}(a\chi) = a\mathcal{A}(\chi)$.

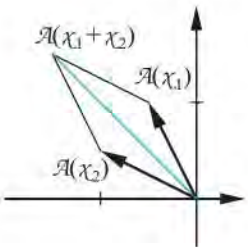


Consider, for example, the transformation obtained by rotating vectors in $\Re^2$ by an angle θ, as shown in the figure to the left. The next two figures illustrate that property 1 is satisfied for rotation. They show that if you want to rotate a sum of two vectors, you can rotate each vector first and 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.

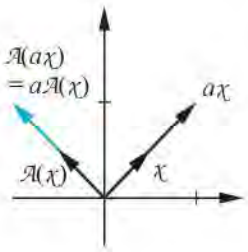


Let $\{v_1, v_2, \ldots, v_n\}$ be a basis for vector space X, and let $\{u_1, u_2, \ldots, u_m\}$ be a basis for vector space Y. This means that for any two vectors $\chi \in X$ and $y \in Y$

$$\chi = \sum_{i=1}^{n} x_i v_i \text{ and } y = \sum_{i=1}^{m} y_i u_i . \tag{6.2}$$

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

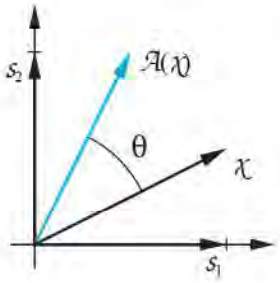
$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \tag{6.11}$$

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.

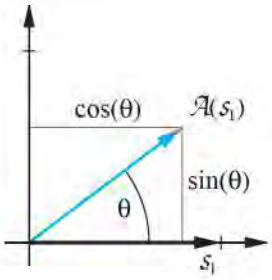


As an example of a matrix representation, consider the rotation transformation. Let's find a matrix representation for that transformation. The key step is given in Eq. (6.6). We must transform each basis vector for the domain and then expand it in terms of the basis vectors of the range. In this example 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 ($u_j = v_i = s_i$), 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

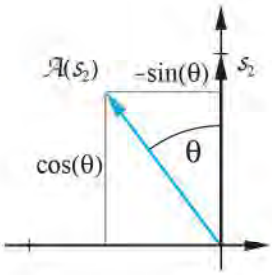
$$\mathcal{A}(s_1) = \cos(\theta)s_1 + \sin(\theta)s_2 = \sum_{i=1}^{2} a_{i1}s_i = a_{11}s_1 + a_{21}s_2, \qquad (6.12)$$



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 $\mathbf{t}_i$:

$$\mathbf{B}_t = \left[\mathbf{t}_1 \ \mathbf{t}_2 \ \cdots \ \mathbf{t}_n\right]. \tag{6.27}$$

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 $\mathbf{w}_i$:

$$\mathbf{B}_w = \left[\mathbf{w}_1 \ \mathbf{w}_2 \ \cdots \ \mathbf{w}_m\right]. \tag{6.29}$$

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*:

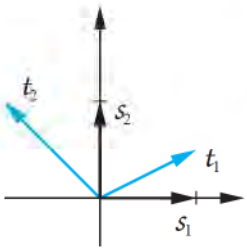
$$\mathbf{A}' = [\mathbf{B}_w^{-1}\mathbf{A}\mathbf{B}_t]. \tag{6.33}$$

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):

$$\mathbf{A}' = [\mathbf{B}_w^{-1} \mathbf{A} \mathbf{B}_t] = \begin{bmatrix} 2/3 & 2/3 \\ -1/3 & 2/3 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0.5 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1/3\sin\theta + \cos\theta & -4/3\sin\theta \\ \frac{5}{6}\sin\theta & -1/3\sin\theta + \cos\theta \end{bmatrix} . \tag{6.39}$$

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

$$\mathbf{x} = \begin{bmatrix} 1 \\ 0.5 \end{bmatrix}, \text{ which corresponds to } \mathbf{x}' = \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \tag{6.42}$$

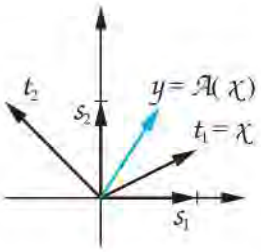
(Note that the vector represented by $\mathbf{x}$ and $\mathbf{x}'$ is t_1, a member of the second basis set.) The transformed test vector would be

$$\mathbf{y} = \mathbf{A}\mathbf{x} = \begin{bmatrix} 0.866 & -0.5 \\ 0.5 & 0.866 \end{bmatrix} \begin{bmatrix} 1 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0.616 \\ 0.933 \end{bmatrix}, \tag{6.43}$$

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

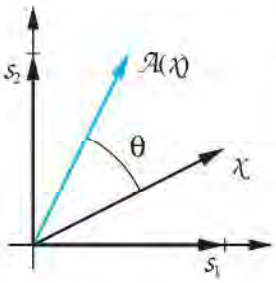
In this final section we want to discuss two key properties of linear transformations: eigenvalues and eigenvectors. Knowledge of these properties will allow us to answer some key questions about neural network performance, such as the question we posed at the beginning of this chapter, concerning the 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 $\mathbf{A}$. 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 $\mathbf{A}$.

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 $X = \{x_i\}$, 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 Transformations

A transformation $\mathcal{A}$ is *linear* if:

1. for all $x_1, x_2 \in X$, $\mathcal{A}(x_1 + x_2) = \mathcal{A}(x_1) + \mathcal{A}(x_2)$,
2. for all $x \in X$, $a \in R$, $\mathcal{A}(ax) = a\mathcal{A}(x)$.

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:

$$\mathcal{A}(x) = y.$$

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?

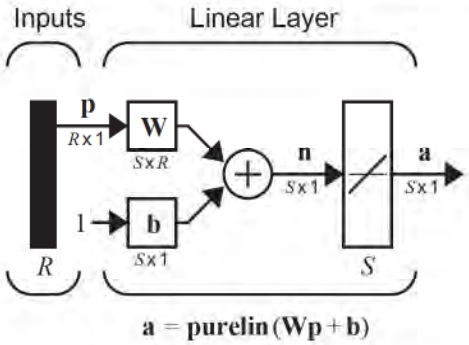


Figure P6.1 Single-Neuron Perceptron

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 $\Re^2$.

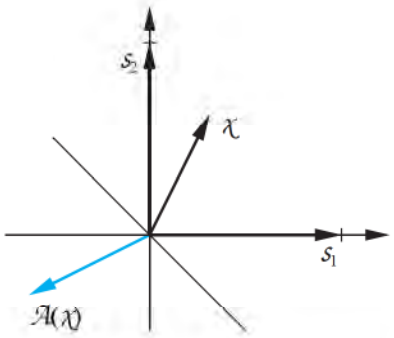
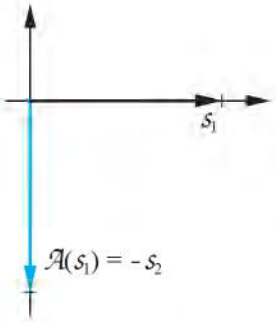


Figure P6.2 Reflection Transformation

The key to finding the matrix of a transformation is given in Eq. (6.6):

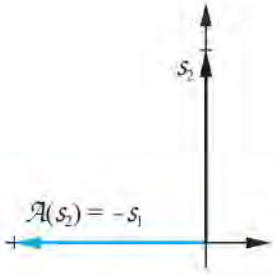
$$\mathcal{A}(v_j) = \sum_{i=1}^{m} a_{ij} u_i .$$



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 $\mathbf{x} = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$:

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

This is indeed the reflection of $\mathbf{x}$ about the line $x_1 + x_2 = 0$, as we can see in Figure P6.3.

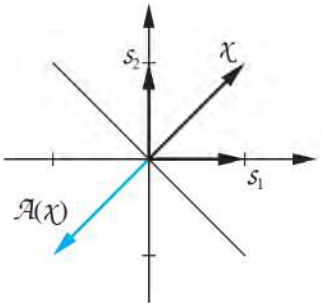


Figure P6.3 Test of Reflection Operation



(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 $\mathcal{A}: X \to X$ be the conjugation operator (i.e., $\mathcal{A}(\chi) = \chi^*$).

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):

$$\mathbf{A}' = [\mathbf{B}_w^{-1}\mathbf{A}\mathbf{B}_I] = [\mathbf{B}^{-1}\mathbf{A}\mathbf{B}],$$

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

$$\mathbf{B} = [\mathbf{z}_1 \ \mathbf{z}_2] = \begin{bmatrix} 2 & 1 \\ 1 & -1 \end{bmatrix}.$$

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.

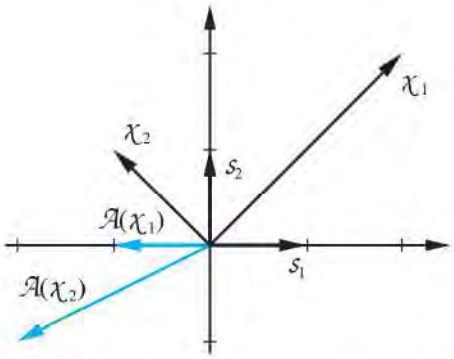


Figure P6.4 Transformation for Problem P6.9

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:

$$\mathbf{A}\begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \quad \mathbf{A}\begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ -1 \end{bmatrix} .$$

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 $\{v_1, v_2\}$.

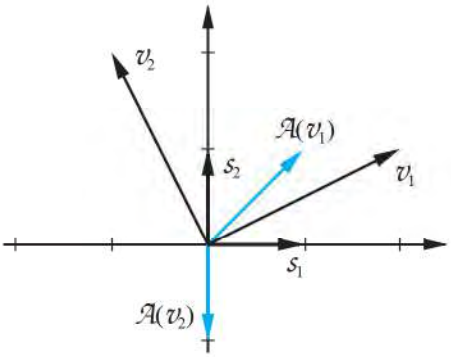


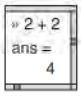
Figure E6.1 Example Transformation for Exercise E6.3

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.



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.

$$v_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \quad v_2 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad v_3 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \quad v_4 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

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 $\mathcal{A}:\Re^2 \to \Re^2$. In the figure below, we show a set of basis vectors $V = \{v_1, v_2\}$ and the transformed basis vectors.

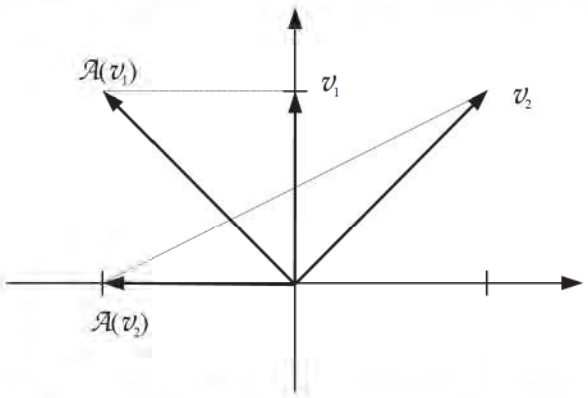


Figure E6.2 Definition of Transformation for Exercise E6.13

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 $I: P^2 \to P^3$, 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 $\mathcal{A}: \Re^2 \to \Re^2$ 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 $\mathcal{A}: R^2 \to R^2$ 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 χ 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.

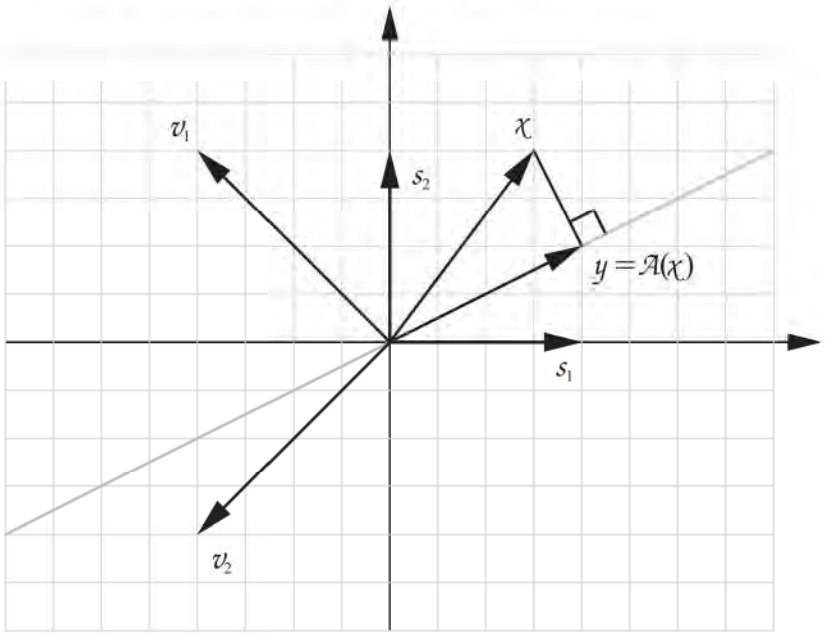


Figure E6.3 Definition of Transformation for Exercise E6.17

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	7-1
Theory and Examples	7-2
Linear Associator	7-3
The Hebb Rule	7-4
Performance Analysis	7-5
Pseudoinverse Rule	7-7
Application	7-10
Variations of Hebbian Learning	7-12
Summary of Results	7-14
Solved Problems	7-16
Epilogue	7-29
Further Reading	7-30
Exercises	7-31

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].)

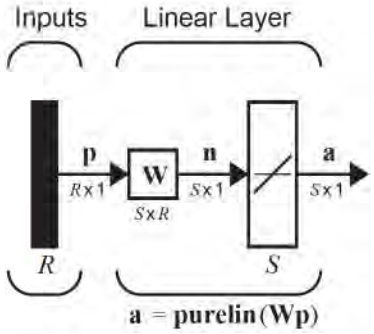


Figure 7.1 Linear Associator

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

$$\mathbf{a} = \mathbf{W}\mathbf{p}_k = \mathbf{t}_k + \underbrace{\sum_{q \neq k} \mathbf{t}_q(\mathbf{p}_q^T\mathbf{p}_k)}_{\textit{Error}}. \tag{7.14}$$

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:

$$\mathbf{W} = \mathbf{T}\mathbf{P}^{+} = \begin{bmatrix} -1 & 1 \end{bmatrix} \begin{bmatrix} 0.25 & -0.5 & -0.25 \\ 0.25 & 0.5 & -0.25 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}. \tag{7.38}$$

Let's try this matrix on our two prototype patterns.

$$\mathbf{W}\mathbf{p}_1 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} -1 \end{bmatrix} \tag{7.39}$$

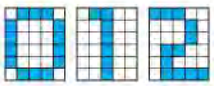
$$\mathbf{W}\mathbf{p}_2 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \end{bmatrix} \tag{7.40}$$

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.

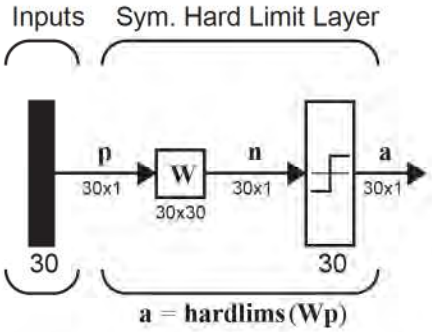


Figure 7.2 Autoassociative Network for Digit Recognition

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.

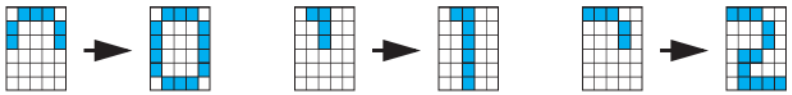


Figure 7.3 Recovery of 50% Occluded Patterns

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.

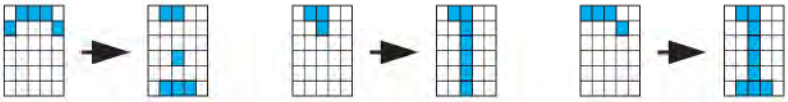


Figure 7.4 Recovery of 67% Occluded Patterns

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.

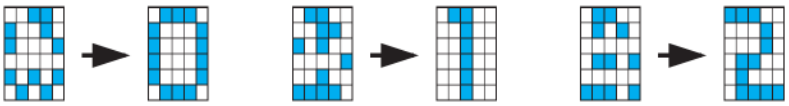


Figure 7.5 Recovery of Noisy Patterns



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:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha\mathbf{a}_q\mathbf{p}_q^T, \tag{7.47}$$

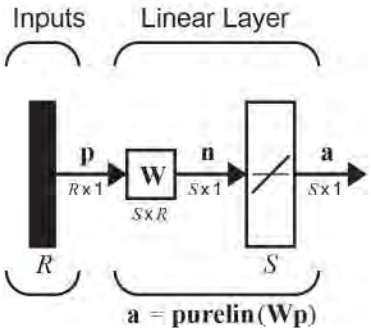
where $\mathbf{a}_q$ is the output of the network when $\mathbf{p}_q$ is given as the input (see also Eq. (7.5)). This unsupervised form of the Hebb rule, which does not require knowledge of the desired output, is actually a more direct interpretation of Hebb's postulate than is the supervised form discussed 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.

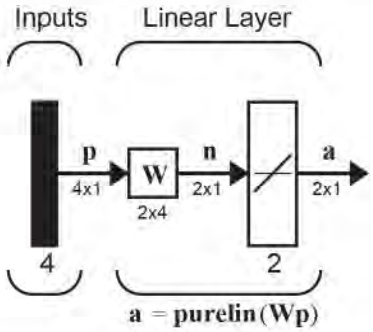


Figure P7.1 Single-Neuron Perceptron

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.



$\mathbf{p}_1$ $\mathbf{p}_2$



$\mathbf{p}_t$

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.

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.

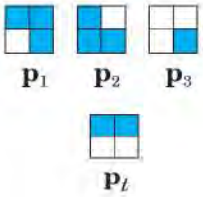
```
ah=hardlims(wh*pt);
ah'
ans =
     1     1    -1    -1     1    -1     1
```

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.

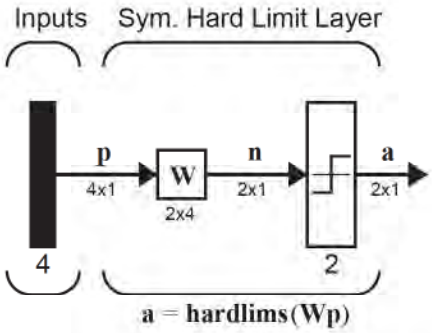


Figure P7.2 Perceptron Network for Problem P7.4

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:

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \mathbf{p}_2 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}.$$

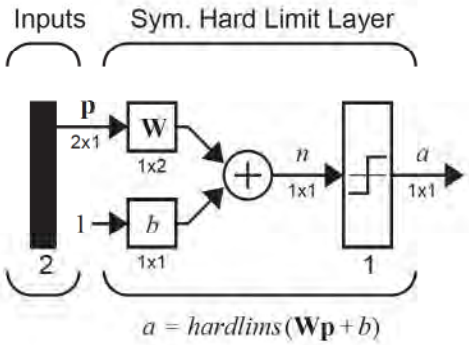


Figure P7.3 Single-Neuron Perceptron

i. Why is a bias required to solve this problem?

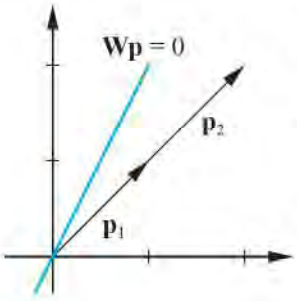
ii. Use the pseudoinverse rule to design a network with bias to solve this problem.

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.

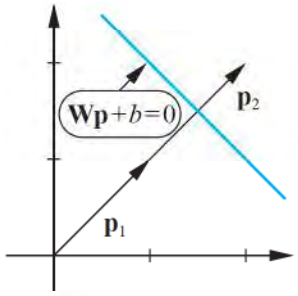


Figure P7.4 Decision Boundary for Solved Problem P7.6

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:

$$\mathbf{W}'\mathbf{p}' + \mathbf{b} = \mathbf{W}\mathbf{p}.$$

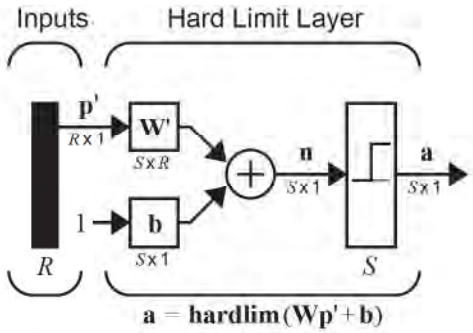


Figure P7.5 Binary Associative Network

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.

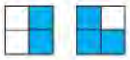
[Hebb49] D. O. Hebb, *The Organization of Behavior*, New York: Wiley, 1949.

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


$\mathbf{p}_1$ $\mathbf{p}_2$

E7.1 Consider the prototype patterns given to the left.

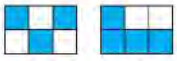
i. Are $\mathbf{p}_1$ and $\mathbf{p}_2$ orthogonal?

ii. Use the Hebb rule to design an autoassociator network for these patterns.


$\mathbf{p}_t$

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.


$\mathbf{p}_1$ $\mathbf{p}_2$

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.

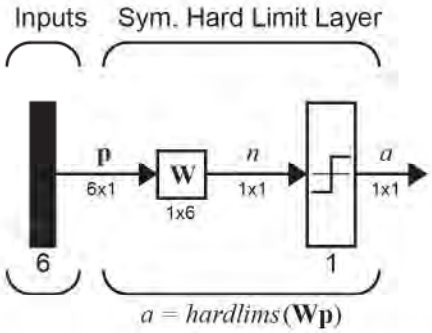


Figure E7.1 Perceptron Network for Exercise E7.3

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.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\4\end{bmatrix}, t_1 = \begin{bmatrix}26\end{bmatrix}\right\} \quad \left\{\mathbf{p}_2 = \begin{bmatrix}4\\2\end{bmatrix}, t_2 = \begin{bmatrix}26\end{bmatrix}\right\} \quad \left\{\mathbf{p}_3 = \begin{bmatrix}-2\\-2\end{bmatrix}, t_3 = \begin{bmatrix}-26\end{bmatrix}\right\}$$

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.)

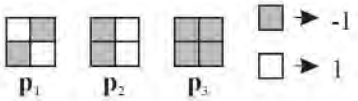


Figure E7.2 Prototype Patterns for Exercise E7.8

E7.9 Suppose that we have the following three reference patterns and their targets.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}3\\6\end{bmatrix}, t_1 = [75]\right\} \qquad \left\{\mathbf{p}_2 = \begin{bmatrix}6\\3\end{bmatrix}, t_2 = [75]\right\} \qquad \left\{\mathbf{p}_3 = \begin{bmatrix}-6\\3\end{bmatrix}, t_3 = [-75]\right\}$$

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:

$$\left\{\mathbf{p}_1 = \begin{bmatrix}1\\1\end{bmatrix}, t_1 = [1]\right\} \qquad \left\{\mathbf{p}_2 = \begin{bmatrix}1\\-1\end{bmatrix}, t_2 = [-1]\right\}$$

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.

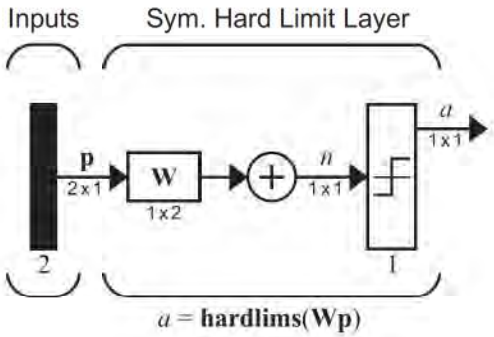


Figure E7.3 Network for Exercise E7.10



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	8-1
Theory and Examples	8-2
Taylor Series	8-2
Vector Case	8-4
Directional Derivatives	8-5
Minima	8-7
Necessary Conditions for Optimality	8-9
First-Order Conditions	8-10
Second-Order Conditions	8-11
Quadratic Functions	8-12
Eigensystem of the Hessian	8-13
Summary of Results	8-20
Solved Problems	8-22
Epilogue	8-34
Further Reading	8-35
Exercises	8-36

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

$$F(x) = \cos(x) = \cos(0) - \sin(0)(x-0) - \frac{1}{2}\cos(0)(x-0)^2 + \frac{1}{6}\sin(0)(x-0)^3 + \cdots$$

$$= 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4 + \cdots \tag{8.3}$$

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.

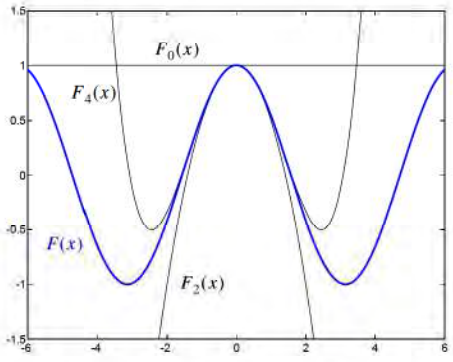


Figure 8.1 Cosine Function and Taylor Series Approximations

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**).

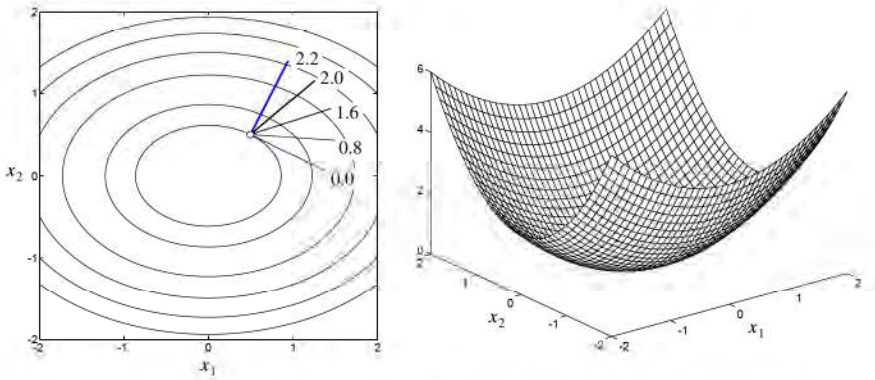


Figure 8.2 Quadratic Function and Directional Derivatives

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.

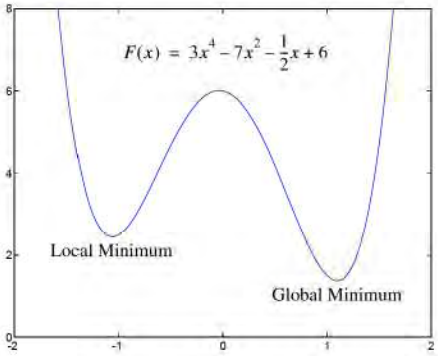


Figure 8.3 Scalar Example of Local and Global Minima



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`).

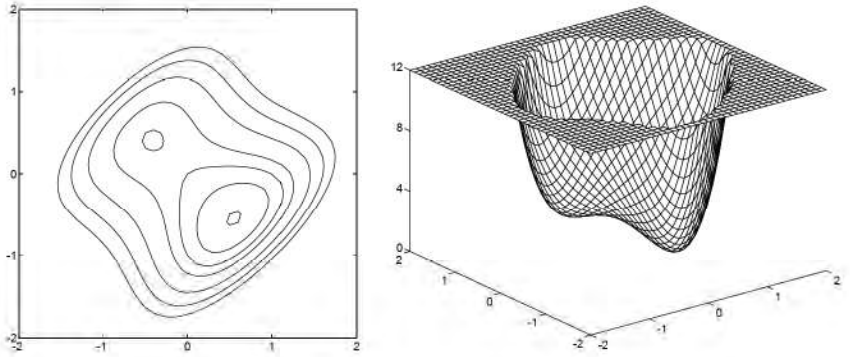


Figure 8.4 Vector Example of Minima and Saddle Point



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.

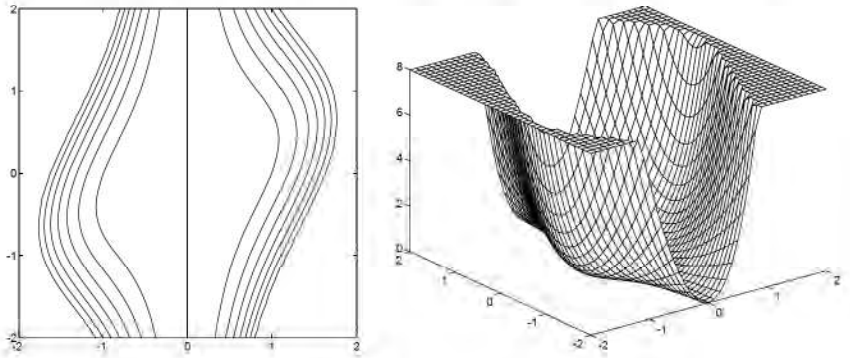


Figure 8.5 Weak Minimum Example

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:

$$\nabla^2 F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{0}} = \left[\begin{matrix} 12x_1^2 & 0 \\ 0 & 2 \end{matrix}\right]\Bigg|_{\mathbf{x}=\mathbf{0}} = \left[\begin{matrix} 0 & 0 \\ 0 & 2 \end{matrix}\right]. \tag{8.34}$$

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

$$\nabla \mathbf{x}^T \mathbf{Q}\mathbf{x} = \mathbf{Q}\mathbf{x} + \mathbf{Q}^T\mathbf{x} = 2\mathbf{Q}\mathbf{x} \quad \text{(for symmetric } \mathbf{Q}\text{)}. \tag{8.37}$$

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

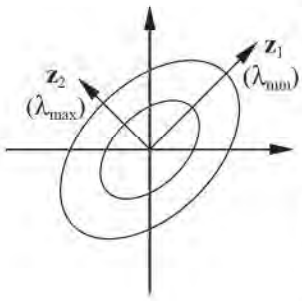
$$\mathbf{c} = \mathbf{B}^T\mathbf{p} = \mathbf{B}^T\mathbf{z}_{max} = \begin{bmatrix}0 & 0 & \dots & 0 & 1 & 0 & \dots & 0\end{bmatrix}^T, \tag{8.50}$$

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.

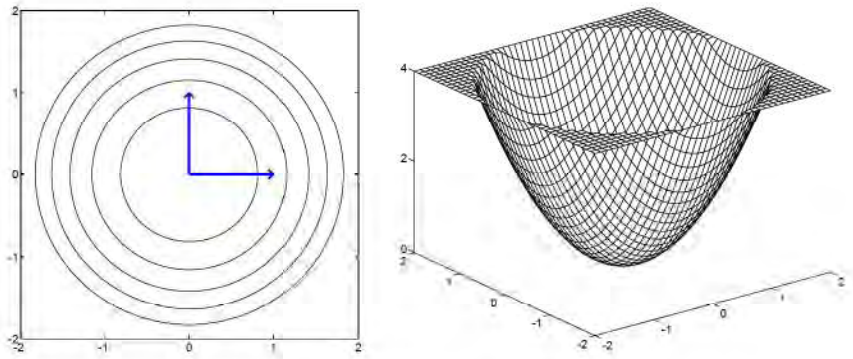


Figure 8.6 Circular Hollow



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.

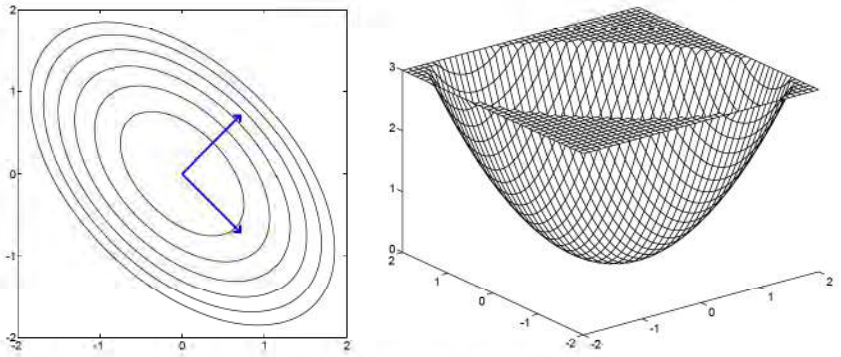


Figure 8.7 Elliptical Hollow

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).

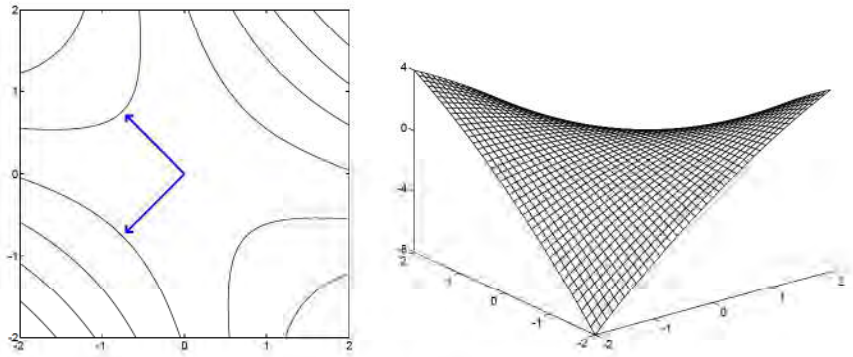


Figure 8.8 Elongated Saddle

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.

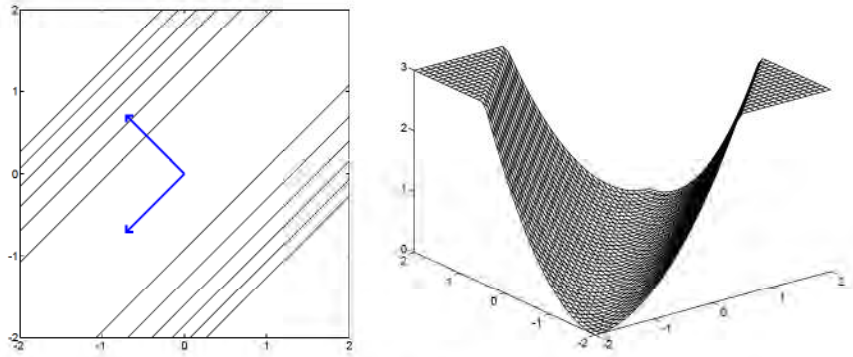


Figure 8.9 Stationary Valley



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

$$\nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}^*} = \mathbf{0} \text{ (Stationary Points)}$$

Second-Order Condition

$$\nabla^2 F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}^*} \geq 0 \text{ (Positive Semidefinite Hessian Matrix)}$$

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`).

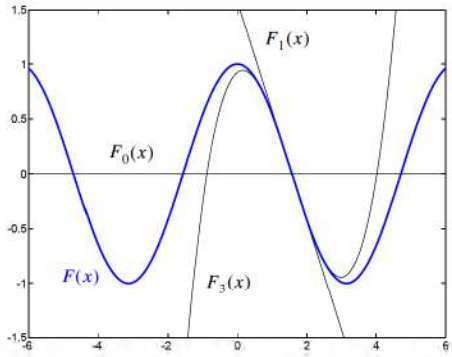


Figure P8.1 Cosine Approximation About $x = \pi/2$

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

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} -4(x_2 - x_1)^3 + 8x_2 - 1 \\ 4(x_2 - x_1)^3 + 8x_1 + 1 \end{bmatrix},$$

and the Hessian matrix is

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2}{\partial x_1^2} F(\mathbf{x}) & \frac{\partial^2}{\partial x_1 \partial x_2} F(\mathbf{x}) \\ \frac{\partial^2}{\partial x_2 \partial x_1} F(\mathbf{x}) & \frac{\partial^2}{\partial x_2^2} F(\mathbf{x}) \end{bmatrix}$$

$$= \begin{bmatrix} 12(x_2 - x_1)^2 & -12(x_2 - x_1)^2 + 8 \\ -12(x_2 - x_1)^2 + 8 & 12(x_2 - x_1)^2 \end{bmatrix}$$

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`).

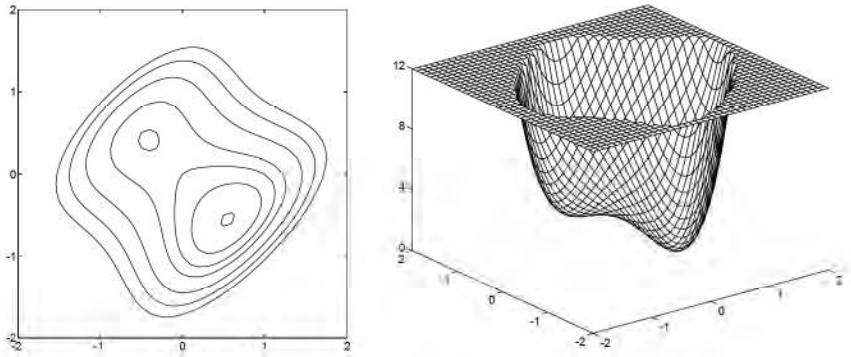


Figure P8.2 Function $F(\mathbf{x})$ for Problem P8.2

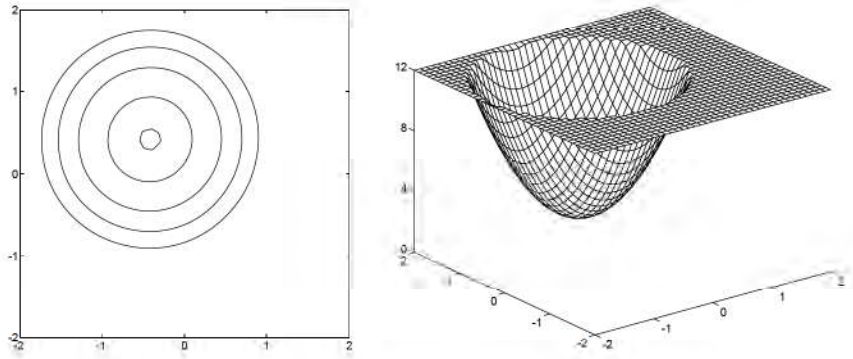


Figure P8.3 Function $F^1(\mathbf{x})$ for Problem P8.2

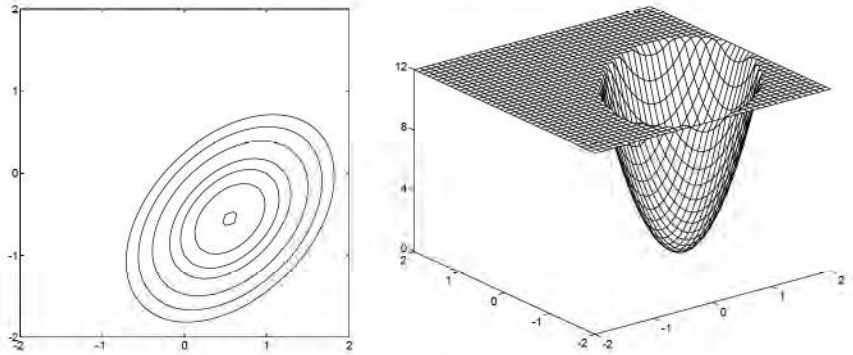


Figure P8.4 Function $F^2(\mathbf{x})$ for Problem P8.2

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.

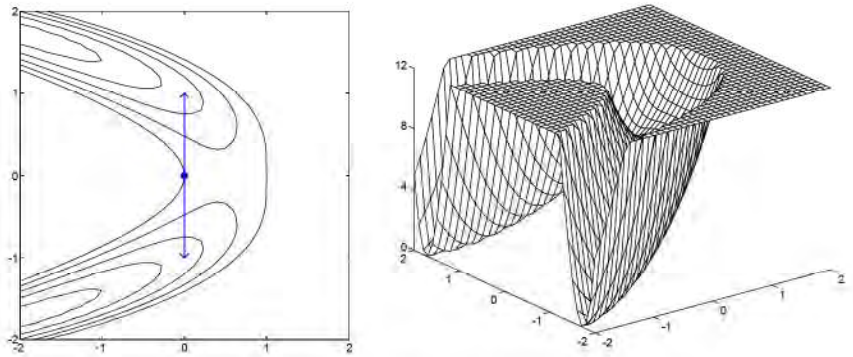


Figure P8.5 Plot of $F(\mathbf{x})$ for Problem P8.3

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.

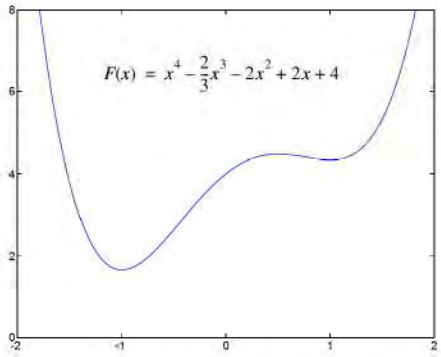


Figure P8.6 Graph of $F(x)$ for Problem P8.4

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:

$$F(\mathbf{x}) = (t_1 - a_1(\mathbf{x}))^2 + (t_2 - a_2(\mathbf{x}))^2.$$

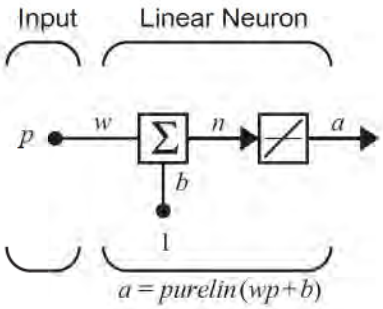


Figure P8.7 Linear Network for Problem P8.6

The parameters of this network are w and b, which make up the parameter vector

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

We want to sketch the performance index $F(\mathbf{x})$. First we will show that the performance index is a quadratic function. Then we will find the eigenvectors and eigenvalues of the Hessian matrix and use them to sketch the contour plot of the function.

Begin by writing $F(\mathbf{x})$ as an explicit function of the parameter vector $\mathbf{x}$:

$$F(\mathbf{x}) = e_1^2 + e_2^2,$$

where

$$(e_1 = t_1 - (wp_1 + b)), (e_2 = t_2 - (wp_2 + b)).$$

This can be written in matrix form:

$$F(\mathbf{x}) = \mathbf{e}^T\mathbf{e},$$

where

$$\mathbf{e} = \mathbf{t} - \begin{bmatrix} p_1 & 1 \\ p_2 & 1 \end{bmatrix}\mathbf{x} = \mathbf{t} - \mathbf{G}\mathbf{x}.$$

The performance index can now be rewritten:

$$F(\mathbf{x}) = [\mathbf{t} - \mathbf{G}\mathbf{x}]^T[\mathbf{t} - \mathbf{G}\mathbf{x}] = \mathbf{t}^T\mathbf{t} - 2\mathbf{t}^T\mathbf{G}\mathbf{x} + \mathbf{x}^T\mathbf{G}^T\mathbf{G}\mathbf{x}.$$

If we compare this with Eq. (8.35):

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{d}^T\mathbf{x} + c\,,$$

we can see that the performance index for this linear network is a quadratic function, with

$$c = \mathbf{t}^T\mathbf{t},\ \mathbf{d} = -2\mathbf{G}^T\mathbf{t}, \text{ and } \mathbf{A} = 2\mathbf{G}^T\mathbf{G}\,.$$

The gradient of the quadratic function is given in Eq. (8.38):

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d} = 2\mathbf{G}^T\mathbf{G}\mathbf{x} - 2\mathbf{G}^T\mathbf{t}\,.$$

The stationary point (also the center of the function contours) will occur where the gradient is equal to zero:

$$\mathbf{x}^* = -\mathbf{A}^{-1}\mathbf{d} = [\mathbf{G}^T\mathbf{G}]^{-1}\mathbf{G}^T\mathbf{t}\,.$$

For

$$\mathbf{G} = \begin{bmatrix} p_1 & 1 \\ p_2 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ -1 & 1 \end{bmatrix} \text{ and } \mathbf{t} = \begin{bmatrix} 0.5 \\ 0 \end{bmatrix}$$

we have

$$\mathbf{x}^* = [\mathbf{G}^T\mathbf{G}]^{-1}\mathbf{G}^T\mathbf{t} = \begin{bmatrix} 5 & 1 \\ 1 & 2 \end{bmatrix}^{-1}\begin{bmatrix} 1 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0.167 \\ 0.167 \end{bmatrix}.$$

(Therefore the optimal network parameters are $w = 0.167$ and $b = 0.167$.)

The Hessian matrix of the quadratic function is given by Eq. (8.39):

$$\nabla^2 F(\mathbf{x}) = \mathbf{A} = 2\mathbf{G}^T\mathbf{G} = \begin{bmatrix} 10 & 2 \\ 2 & 4 \end{bmatrix}.$$

To sketch the contour plot we need the eigenvectors and eigenvalues of the Hessian. For this case we find

$$\left\{(\lambda_1 = 10.6), \left(\mathbf{z}_1 = \begin{bmatrix} 1 \\ 0.3 \end{bmatrix}\right)\right\}, \left\{(\lambda_2 = 3.4), \left(\mathbf{z}_2 = \begin{bmatrix} 0.3 \\ -1 \end{bmatrix}\right)\right\}.$$

Therefore we know that $\mathbf{x}^*$ is a strong minimum. Also, since the first eigenvalue is larger than the second, we know that the contours will be elliptical and that the long axis of the ellipses will be in the direction of the second

eigenvector. The contours will be centered at $\mathbf{x}^*$. This is demonstrated in Figure P8.8.

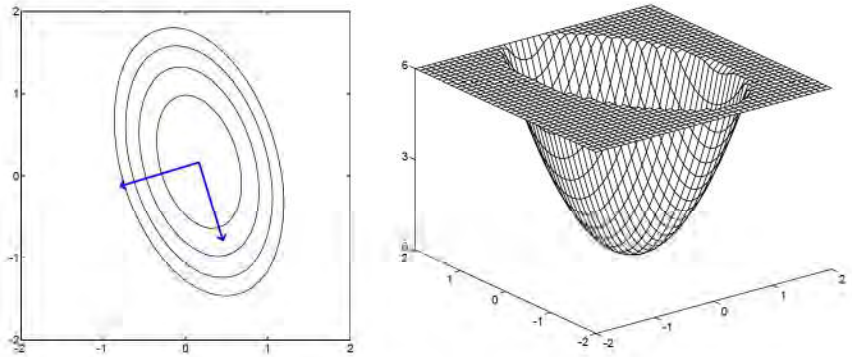


Figure P8.8 Graph of Function for Problem P8.6

P8.7 There are quadratic functions that do not have stationary points. This problem illustrates one such case. Consider the following function:

$$F(\mathbf{x}) = \begin{bmatrix} 1 & -1 \end{bmatrix}\mathbf{x} + \frac{1}{2}\mathbf{x}^T\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\mathbf{x}.$$

Sketch the contour plot of this function.

As with Problem P8.6, we need to find the eigenvalues and eigenvectors of the Hessian matrix. By inspection of the quadratic function we see that the Hessian matrix is

$$\nabla^2 F(\mathbf{x}) = \mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}. \tag{8.63}$$

The eigenvalues and eigenvectors are

$$\left\{(\lambda_1 = 0), \left(\mathbf{z}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}\right)\right\}, \left\{(\lambda_2 = 2), \left(\mathbf{z}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}\right)\right\}.$$

Notice that the first eigenvalue is zero, so there is no curvature along the first eigenvector. The second eigenvalue is positive, so there is positive curvature along the second eigenvector. If we had no linear term in $F(\mathbf{x})$, the plot of the function would show a stationary valley, as in Figure 8.9. In this case we must find out if the linear term creates a slope in the direction of the valley (the direction of the first eigenvector).

The linear term is

$$F_{lin}(\mathbf{x}) = \begin{bmatrix} 1 & -1 \end{bmatrix} \mathbf{x} .$$

From Eq. (8.36) we know that the gradient of this term is

$$\nabla F_{lin}(\mathbf{x}) = \begin{bmatrix} 1 \\ -1 \end{bmatrix},$$

which means that the linear term is increasing most rapidly in the direction of this gradient. Since the quadratic term has no curvature in this direction, the overall function will have a linear slope in this direction. Therefore $F(\mathbf{x})$ will have positive curvature in the direction of the second eigenvector and a linear slope in the direction of the first eigenvector. The contour plot and the 3-D plot for this function are given in Figure P8.9.

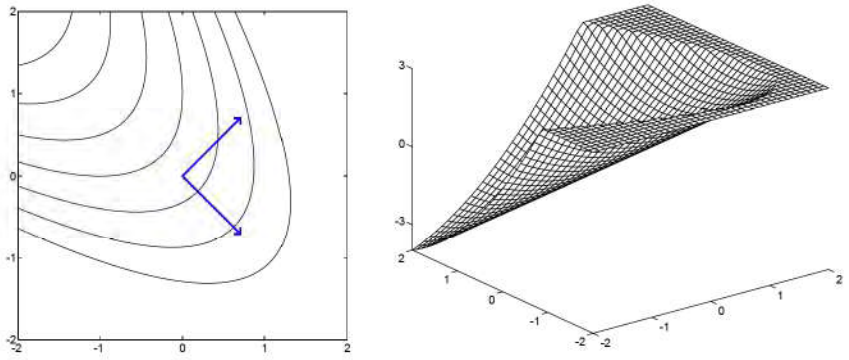


Figure P8.9 Falling Valley Function for Problem P8.7

Whenever any of the eigenvalues of the Hessian matrix are zero it is impossible to solve for the stationary point of the quadratic function using

$$\mathbf{x}^* = -\mathbf{A}^{-1}\mathbf{d},$$

since the Hessian matrix does not have an inverse. This lack of an inverse could mean that we have a weak minimum point, as illustrated in Figure 8.9, or that there is no stationary point, as this example shows.

Epilogue

Performance learning is one of the most important classes of neural network learning rules. With performance learning, network parameters are adjusted to optimize network performance. In this chapter we have introduced tools that we will need to understand performance learning rules. After reading this chapter and solving the exercises, you should be able to:

i. Perform a Taylor series expansion and use it to approximate a function.

ii. Calculate a directional derivative.

iii. Find stationary points and test whether they could be minima.

iv. Sketch contour plots of quadratic functions.

We will be using these concepts in a number of succeeding chapters, including the chapters on performance learning (9–14), the radial basis network chapter (17) and the chapters on stability and Hopfield networks (20–21). In the next chapter we will build on the concepts we have covered here, to design algorithms that will optimize performance functions. Then, in succeeding chapters, we will apply these algorithms to the training of neural networks.

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.

[Gill81] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*, New York: Academic Press, 1981.

As the title implies, this text emphasizes the practical implementation of optimization algorithms. It provides motivation for the optimization methods, as well as details of implementation that affect algorithm performance.

[Himm72] D. M. Himmelblau, *Applied Nonlinear Programming*, New York: McGraw-Hill, 1972.

This is a comprehensive text on nonlinear optimization. It covers both constrained and unconstrained optimization problems. The text is very complete, with many examples worked out in detail.

[Scal85] L. E. Scales, *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

A very readable text describing the major optimization algorithms, this text emphasizes methods of optimization rather than existence theorems and proofs of convergence. Algorithms are presented with intuitive explanations, along with illustrative figures and examples. Pseudo-code is presented for most algorithms.

Exercises

E8.1 Consider the following scalar function:

$$F(x) = \frac{1}{x^3 - \frac{3}{4}x - \frac{1}{2}}.$$

i. Find the second-order Taylor series approximation for $F(x)$ about the point $x = -0.5$.

ii. Find the second-order Taylor series approximation for $F(x)$ about the point $x = 1.1$.

iii. Plot $F(x)$ and the two approximations and discuss their accuracy.

E8.2 Consider the following function of two variables:

$$F(\mathbf{x}) = e^{(2x_1^2 + 2x_2^2 + x_1 - 5x_2 + 10)}.$$

i. Find the second-order Taylor series approximation for $F(\mathbf{x})$ about the point $\mathbf{x} = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$.

ii. Find the stationary point for this approximation.

iii. Find the stationary point for $F(\mathbf{x})$. (Note that the exponent of $F(\mathbf{x})$ is simply a quadratic function.)



iv. Explain the difference between the two stationary points. (Use MATLAB to plot the two functions.)

E8.3 For the following functions find the first and second directional derivatives from the point $\mathbf{x} = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$ in the direction $\mathbf{p} = \begin{bmatrix} -1 & 1 \end{bmatrix}^T$.

i. $F(\mathbf{x}) = \frac{7}{2}x_1^2 - 6x_1x_2 - x_2^2$

ii. $F(\mathbf{x}) = 5x_1^2 - 6x_1x_2 + 5x_2^2 + 4x_1 + 4x_2$

iii. $F(\mathbf{x}) = \frac{9}{2}x_1^2 - 2x_1x_2 + 3x_2^2 + 2x_1 - x_2$

iv. $F(\mathbf{x}) = -\frac{1}{2}(7x_1^2 + 12x_1x_2 - 2x_2^2)$

v. $F(\mathbf{x}) = x_1^2 + x_1x_2 + x_2^2 + 3x_1 + 3x_2$

vi. $F(\mathbf{x}) = \frac{1}{2}x_1^2 - 3x_1x_2 + \frac{1}{2}x_2^2 - 4x_1 + 4x_2$

vii. $F(\mathbf{x}) = \frac{1}{2}x_1^2 - 2x_1x_2 + 2x_2^2 + x_1 - 2x_2$

viii. $F(\mathbf{x}) = \frac{3}{2}x_1^2 + 2x_1x_2 + 4x_1 + 4x_2$

ix. $F(\mathbf{x}) = -\frac{3}{2}x_1^2 + 4x_1x_2 + \frac{3}{2}x_2^2 + 5x_1$

x. $F(\mathbf{x}) = 2x_1^2 - 2x_1x_2 + \frac{1}{2}x_2^2 + x_1 + x_2$

E8.4 For the following function,

$$F(x) = x^4 - \frac{1}{2}x^2 + 1\,,$$

i. find the stationary points,

ii. test the stationary points to find minimum and maximum points, and



iii. plot the function using MATLAB to verify your answers.

E8.5 Consider the following function of two variables:

$$F(\mathbf{x}) = (x_1 + x_2)^4 - 12x_1x_2 + x_1 + x_2 + 1\,.$$

i. Verify that the function has three stationary points at

$$\mathbf{x}^1 = \begin{bmatrix} -0.6504 \\ -0.6504 \end{bmatrix},\ \mathbf{x}^2 = \begin{bmatrix} 0.085 \\ 0.085 \end{bmatrix},\ \mathbf{x}^3 = \begin{bmatrix} 0.5655 \\ 0.5655 \end{bmatrix}.$$

ii. Test the stationary points to find any minima, maxima or saddle points.

iii. Find the second-order Taylor series approximations for the function at each of the stationary points.



iv. Plot the function and the approximations using MATLAB.

E8.6 For the functions of Exercise E8.3:

i. find the stationary points,

ii. test the stationary points to find minima, maxima or saddle points,

iii. provide rough sketches of the contour plots, using the eigenvalues



and eigenvectors of the Hessian matrices, and

iv. plot the functions using MATLAB to verify your answers.

E8.7 Consider the following quadratic function:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix} 1 & -3 \\ -3 & 1 \end{bmatrix}\mathbf{x} + \begin{bmatrix} 4 & -4 \end{bmatrix}\mathbf{x} + 2 .$$

i. Find the gradient and Hessian matrix for $F(\mathbf{x})$.

ii. Sketch the contour plot for $F(\mathbf{x})$.

iii. Find the directional derivative of $F(\mathbf{x})$ at the point $\mathbf{x}_0 = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$ in the direction $\mathbf{p} = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$.

iv. Is your answer to part iii. consistent with your contour plot of part ii.? Explain.

E8.8 Repeat Exercise E8.7 with the following quadratic function:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix} 3 & -2 \\ -2 & 0 \end{bmatrix}\mathbf{x} + \begin{bmatrix} 4 & 4 \end{bmatrix}\mathbf{x} + 2 .$$

E8.9 Consider the following function:

$$F(\mathbf{x}) = (1 + x_1 + x_2)^2 + \frac{1}{4}x_1^4 .$$

i. Find the quadratic approximation to $F(\mathbf{x})$ about the point $\mathbf{x}_0 = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$

ii. Sketch the contour plot of the quadratic approximation in part i.

E8.10 Consider the following function:

$$F(\mathbf{x}) = \frac{3}{2}x_1^2 + 2x_1x_2 + x_2^3 + 4x_1 + 4x_2 .$$

i. Find the quadratic approximation to $F(\mathbf{x})$ about the point $\mathbf{x}_0 = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$.

ii. Locate the stationary point of the quadratic approximation you found in part i.

iii. Is the answer to part ii a minimum of $F(\mathbf{x})$?

E8.11 Consider the following function:

$$F(\mathbf{x}) = x_1x_2 - x_1 + 2x_2 .$$

i. Locate any stationary points.

ii. For each answer to part i., determine, if possible, whether the stationary point is a minimum point, a maximum point, or a saddle point.

iii. Find the directional derivative of the function at the point $\mathbf{x}_0 = \begin{bmatrix} -1 & 1 \end{bmatrix}^T$ in the direction $\mathbf{p} = \begin{bmatrix} -1 & 1 \end{bmatrix}^T$.

E8.12 Consider the following function:

$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + x_2^2 + (x_1 - x_2)^3 .$$

i. Find the quadratic approximation to $F(\mathbf{x})$ about the point $\mathbf{x}_0 = \begin{bmatrix} 2 & 1 \end{bmatrix}^T$.

ii. Sketch the contour plot of the quadratic approximation.

E8.13 Recall the function in Problem P8.7. For that function there was no stationary point. It is possible to modify the function, by changing only the **d** vector, so that a stationary point will exist. Find a new nonzero **d** vector that will create a weak minimum.

9 Performance Optimization

Objectives	9-1
Theory and Examples	9-2
Steepest Descent	9-2
Stable Learning Rates	9-6
Minimizing Along a Line	9-8
Newton's Method	9-10
Conjugate Gradient	9-15
Summary of Results	9-21
Solved Problems	9-23
Epilogue	9-37
Further Reading	9-38
Exercises	9-39

Objectives

We initiated our discussion of performance optimization in Chapter 8. There we introduced the Taylor series expansion as a tool for analyzing the performance surface, and then used it to determine conditions that must be satisfied by optimum points. In this chapter we will again use the Taylor series expansion, in this case to develop algorithms to locate the optimum points. We will discuss three different categories of optimization algorithm: steepest descent, Newton's method and conjugate gradient. In Chapters 10–14 we will apply all of these algorithms to the training of neural networks.

Theory and Examples

In the previous chapter we began our investigation of performance surfaces. Now we are in a position to develop algorithms to search the parameter space and locate minimum points of the surface (find the optimum weights and biases for a given neural network).

It is interesting to note that most of the algorithms presented in this chapter were developed hundreds of years ago. The basic principles of optimization were discovered during the 17th century, by such scientists and mathematicians as Kepler, Fermat, Newton and Leibniz. From 1950 on, these principles were rediscovered to be implemented on "high speed" (in comparison to the pen and paper available to Newton) digital computers. The success of these efforts stimulated significant research on new algorithms, and the field of optimization theory became recognized as a major branch of mathematics. Now neural network researchers have access to a vast storehouse of optimization theory and practice that can be applied to the training of neural networks.

The objective of this chapter, then, is to develop algorithms to optimize a performance index $F(\mathbf{x})$. For our purposes the word "optimize" will mean to find the value of $\mathbf{x}$ that minimizes $F(\mathbf{x})$. All of the optimization algorithms we will discuss are iterative. We begin from some initial guess, $\mathbf{x}_0$, and then update our guess in stages according to an equation of the form

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k, \tag{9.1}$$

or

$$\Delta \mathbf{x}_k = (\mathbf{x}_{k+1} - \mathbf{x}_k) = \alpha_k \mathbf{p}_k, \tag{9.2}$$

where the vector $\mathbf{p}_k$ represents a search direction, and the positive scalar α_k is the learning rate, which determines the length of the step.

The algorithms we will discuss in this chapter are distinguished by the choice of the search direction, $\mathbf{p}_k$. We will discuss three different possibilities. There are also a variety of ways to select the learning rate, α_k, and we will discuss several of these.

Steepest Descent

When we update our guess of the optimum (minimum) point using Eq. (9.1), we would like to have the function decrease at each iteration. In other words,

$$F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k). \tag{9.3}$$

How can we choose a direction, $\mathbf{p}_k$, so that for sufficiently small learning rate, α_k, we will move "downhill" in this way? Consider the first-order Taylor series expansion (see Eq. (8.9)) of $F(\mathbf{x})$ about the old guess $\mathbf{x}_k$:

$$F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k + \Delta\mathbf{x}_k) \approx F(\mathbf{x}_k) + \mathbf{g}_k^T \Delta\mathbf{x}_k, \tag{9.4}$$

where $\mathbf{g}_k$ is the gradient evaluated at the old guess $\mathbf{x}_k$:

$$\mathbf{g}_k \equiv \nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}_k}. \tag{9.5}$$

For $F(\mathbf{x}_{k+1})$ to be less than $F(\mathbf{x}_k)$, the second term on the right-hand side of Eq. (9.4) must be negative:

$$\mathbf{g}_k^T \Delta\mathbf{x}_k = \alpha_k \mathbf{g}_k^T \mathbf{p}_k < 0. \tag{9.6}$$

We will select an α_k that is small, but greater than zero. This implies:

$$\mathbf{g}_k^T \mathbf{p}_k < 0. \tag{9.7}$$

Descent Direction

Any vector $\mathbf{p}_k$ that satisfies this equation is called a *descent direction*. The function must go down if we take a small enough step in this direction. This brings up another question. What is the direction of steepest descent? (In what direction will the function decrease most rapidly?) This will occur when

$$\mathbf{g}_k^T \mathbf{p}_k \tag{9.8}$$

is most negative. (We assume that the length of $\mathbf{p}_k$ does not change, only the direction.) This is an inner product between the gradient and the direction vector. It will be most negative when the direction vector is the negative of the gradient. (Review our discussion of directional derivatives on page 8-6.) Therefore a vector that points in the steepest descent direction is

$$\mathbf{p}_k = -\mathbf{g}_k. \tag{9.9}$$

Steepest Descent

Using this in the iteration of Eq. (9.1) produces the method of *steepest descent*:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k. \tag{9.10}$$

Learning Rate

For steepest descent there are two general methods for determining the *learning rate*, α_k. One approach is to minimize the performance index $F(\mathbf{x})$ with respect to α_k at each iteration. In this case we are minimizing along the line

$$\mathbf{x}_k - \alpha_k \mathbf{g}_k. \tag{9.11}$$

The other method for selecting α_k is to use a fixed value (e.g., $\alpha_k = 0.02$), or to use variable, but predetermined, values (e.g., $\alpha_k = 1/k$). We will discuss the choice of α_k in more detail in the following examples.



Let's apply the steepest descent algorithm to the following function,

$$F(\mathbf{x}) = x_1^2 + 25x_2^2, \tag{9.12}$$

starting from the initial guess

$$\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}. \tag{9.13}$$

The first step is to find the gradient:

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 50x_2 \end{bmatrix}. \tag{9.14}$$

If we evaluate the gradient at the initial guess we find

$$\mathbf{g}_0 = \nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}_0} = \begin{bmatrix} 1 \\ 25 \end{bmatrix}. \tag{9.15}$$

Assume that we use a fixed learning rate of $\alpha = 0.01$. The first iteration of the steepest descent algorithm would be

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha \mathbf{g}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - 0.01 \begin{bmatrix} 1 \\ 25 \end{bmatrix} = \begin{bmatrix} 0.49 \\ 0.25 \end{bmatrix}. \tag{9.16}$$

The second iteration of steepest descent produces

$$\mathbf{x}_2 = \mathbf{x}_1 - \alpha \mathbf{g}_1 = \begin{bmatrix} 0.49 \\ 0.25 \end{bmatrix} - 0.01 \begin{bmatrix} 0.98 \\ 12.5 \end{bmatrix} = \begin{bmatrix} 0.4802 \\ 0.125 \end{bmatrix}. \tag{9.17}$$

If we continue the iterations we obtain the trajectory illustrated in Figure 9.1.

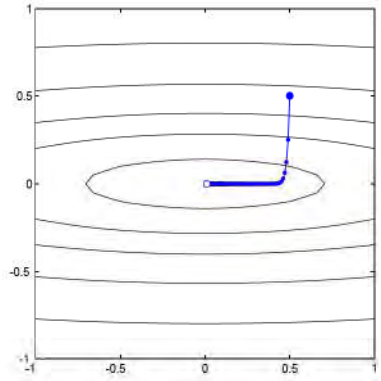


Figure 9.1 Trajectory for Steepest Descent with $\alpha = 0.01$

Note that the steepest descent trajectory, for small learning rate, follows a path that is always orthogonal to the contour lines. This is because the gradient is orthogonal to the contour lines. (See the discussion on page 8-6.)

How would a change in the learning rate change the performance of the algorithm? If we increase the learning rate to $\alpha = 0.035$, we obtain the trajectory illustrated in Figure 9.2. Note that the trajectory now oscillates. If we make the learning rate too large the algorithm will become unstable; the oscillations will increase instead of decaying.

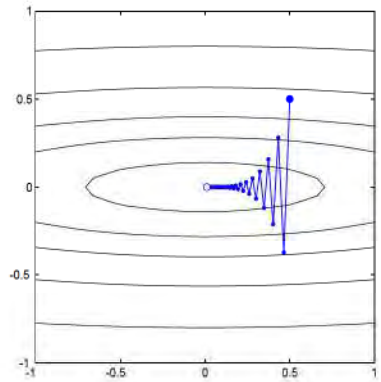


Figure 9.2 Trajectory for Steepest Descent with $\alpha = 0.035$

We would like to make the learning rate large, since then we will be taking large steps and would expect to converge faster. However, as we can see from this example, if we make the learning rate too large the algorithm will become unstable. Is there some way to predict the maximum allowable learning rate? This is not possible for arbitrary functions, but for quadratic functions we can set an upper limit.

Stable Learning Rates

Suppose that the performance index is a quadratic function:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{d}^T\mathbf{x} + c\,. \tag{9.18}$$

From Eq. (8.38) the gradient of the quadratic function is

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d}\,. \tag{9.19}$$

If we now insert this expression into our expression for the steepest descent algorithm (assuming a constant learning rate), we obtain

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha\mathbf{g}_k = \mathbf{x}_k - \alpha(\mathbf{A}\mathbf{x}_k + \mathbf{d}) \tag{9.20}$$

or

$$\mathbf{x}_{k+1} = [\mathbf{I} - \alpha\mathbf{A}]\mathbf{x}_k - \alpha\mathbf{d}\,. \tag{9.21}$$

This is a linear dynamic system, which will be stable if the eigenvalues of the matrix $[\mathbf{I} - \alpha\mathbf{A}]$ are less than one in magnitude (see [Brog91]). We can express the eigenvalues of this matrix in terms of the eigenvalues of the Hessian matrix $\mathbf{A}$. Let $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ and $\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n\}$ be the eigenvalues and eigenvectors of the Hessian matrix. Then

$$[\mathbf{I} - \alpha\mathbf{A}]\mathbf{z}_i = \mathbf{z}_i - \alpha\mathbf{A}\mathbf{z}_i = \mathbf{z}_i - \alpha\lambda_i\mathbf{z}_i = (1 - \alpha\lambda_i)\mathbf{z}_i\,. \tag{9.22}$$

Therefore the eigenvectors of $[\mathbf{I} - \alpha\mathbf{A}]$ are the same as the eigenvectors of $\mathbf{A}$, and the eigenvalues of $[\mathbf{I} - \alpha\mathbf{A}]$ are $(1 - \alpha\lambda_i)$. Our condition for the stability of the steepest descent algorithm is then

$$|(1 - \alpha\lambda_i)| < 1\,. \tag{9.23}$$

If we assume that the quadratic function has a strong minimum point, then its eigenvalues must be positive numbers. Eq. (9.23) then reduces to

$$\alpha < \frac{2}{\lambda_i}\,. \tag{9.24}$$

Since this must be true for all the eigenvalues of the Hessian matrix we have

$$\alpha < \frac{2}{\lambda_{max}}\,. \tag{9.25}$$

The maximum stable learning rate is inversely proportional to the maximum curvature of the quadratic function. The curvature tells us how fast the gradient is changing. If the gradient is changing too fast we may jump

past the minimum point so far that the gradient at the new location will be larger in magnitude (but opposite direction) than the gradient at the old location. This will cause the steps to increase in size at each iteration.



Let's apply this result to our previous example. The Hessian matrix for that quadratic function is

$$\mathbf{A} = \begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix}. \tag{9.26}$$

The eigenvalues and eigenvectors of $\mathbf{A}$ are

$$\left\{(\lambda_1 = 2), \left(\mathbf{z}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right)\right\}, \left\{(\lambda_2 = 50), \left(\mathbf{z}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right)\right\}. \tag{9.27}$$

Therefore the maximum allowable learning rate is

$$\alpha < \frac{2}{\lambda_{max}} = \frac{2}{50} = 0.04. \tag{9.28}$$

This result is illustrated experimentally in Figure 9.3, which shows the steepest descent trajectories when the learning rate is just below ($\alpha = 0.039$) and just above ($\alpha = 0.041$), the maximum stable value.

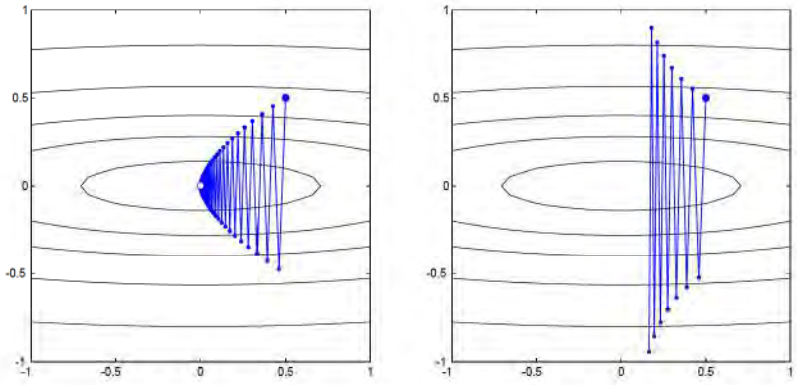


Figure 9.3 Trajectories for $\alpha = 0.039$ (left) and $\alpha = 0.041$ (right).

This example has illustrated several points. The learning rate is limited by the largest eigenvalue (second derivative) of the Hessian matrix. The algorithm tends to converge most quickly in the direction of the eigenvector corresponding to this largest eigenvalue, and we don't want to overshoot the minimum point by too far in that direction. (Note that in our examples the initial step is almost parallel to the x_2 axis, which is $\mathbf{z}_2$.) However, the algorithm will tend to converge most slowly in the direction of the eigenvec-

tor that corresponds to the smallest eigenvalue ($\mathbf{z}_1$ for our example). In the end it is the smallest eigenvalue, in combination with the learning rate, that determines how quickly the algorithm will converge. When there is a great difference in magnitude between the largest and smallest eigenvalues, the steepest descent algorithm will converge slowly.



To experiment with steepest descent on this quadratic function, use the Neural Network Design Demonstration Steepest Descent for a Quadratic (**nnd9sdq**).

Minimizing Along a Line

Another approach for selecting the learning rate is to minimize the performance index with respect to α_k at each iteration. In other words, choose α_k to minimize

$$F(\mathbf{x}_k + \alpha_k \mathbf{p}_k) . \tag{9.29}$$

To do this for arbitrary functions requires a line search, which we will discuss in Chapter 12. For quadratic functions it is possible to perform the linear minimization analytically. The derivative of Eq. (9.29) with respect to α_k, for quadratic $F(\mathbf{x})$, can be shown to be

$$\frac{d}{d\alpha_k} F(\mathbf{x}_k + \alpha_k \mathbf{p}_k) = \nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k + \alpha_k \mathbf{p}_k^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k . \tag{9.30}$$

If we set this derivative equal to zero and solve for α_k, we obtain

$$\alpha_k = - \frac{\nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k}{\mathbf{p}_k^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k} = - \frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{A}_k \mathbf{p}_k} , \tag{9.31}$$

where $\mathbf{A}_k$ is the Hessian matrix evaluated at the old guess $\mathbf{x}_k$:

$$\mathbf{A}_k \equiv \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k} . \tag{9.32}$$

(For quadratic functions the Hessian matrix is not a function of k.)



Let's apply steepest descent with line minimization to the following quadratic function:

$$F(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \mathbf{x} , \tag{9.33}$$

starting from the initial guess

$$\mathbf{x}_0 = \begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix}. \tag{9.34}$$

The gradient of this function is

$$\nabla F(\mathbf{x}) = \begin{bmatrix} 2x_1 + x_2 \\ x_1 + 2x_2 \end{bmatrix}. \tag{9.35}$$

The search direction for steepest descent is the negative of the gradient. For the first iteration this will be

$$\mathbf{p}_0 = -\mathbf{g}_0 = -\nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}_0} = \begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}. \tag{9.36}$$

From Eq. (9.31), the learning rate for the first iteration will be

$$\alpha_0 = -\frac{\begin{bmatrix} 1.35 & 0.3 \end{bmatrix}\begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}}{\begin{bmatrix} -1.35 & -0.3 \end{bmatrix}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}\begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}} = 0.413. \tag{9.37}$$

The first step of steepest descent will then produce

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha_0 \mathbf{g}_0 = \begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix} - 0.413 \begin{bmatrix} 1.35 \\ 0.3 \end{bmatrix} = \begin{bmatrix} 0.24 \\ -0.37 \end{bmatrix}. \tag{9.38}$$

The first five iterations of the algorithm are illustrated in Figure 9.4.

Note that the successive steps of the algorithm are orthogonal. Why does this happen? First, when we minimize along a line we will always stop at a point that is tangent to a contour line. Then, since the gradient is orthogonal to the contour line, the next step, which is along the negative of the gradient, will be orthogonal to the previous step.

We can show this analytically by using the chain rule on Eq. (9.30):

$$\begin{aligned} \frac{d}{d\alpha_k} F(\mathbf{x}_k + \alpha_k \mathbf{p}_k) &= \frac{d}{d\alpha_k} F(\mathbf{x}_{k+1}) = \nabla F(\mathbf{x})^T\big|_{\mathbf{x}=\mathbf{x}_{k+1}} \frac{d}{d\alpha_k}[\mathbf{x}_k + \alpha_k \mathbf{p}_k] \\ &= \nabla F(\mathbf{x})^T\big|_{\mathbf{x}=\mathbf{x}_{k+1}} \mathbf{p}_k = \mathbf{g}_{k+1}^T \mathbf{p}_k . \end{aligned} \tag{9.39}$$

Therefore at the minimum point, where this derivative is zero, the gradient is orthogonal to the previous search direction. Since the next search direction is the negative of this gradient, the consecutive search directions must be orthogonal. (Note that this result implies that when minimizing in any direction, the gradient at the minimum point will be orthogonal to the search direction, even if we are not using steepest descent. We will use this result in our discussion of conjugate directions.)

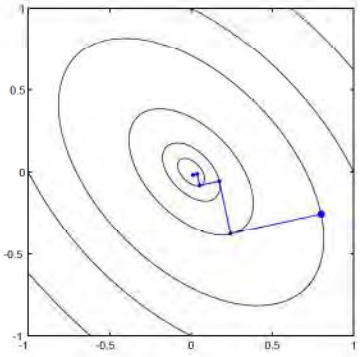


Figure 9.4 Steepest Descent with Minimization Along a Line



To experiment with steepest descent with minimization along a line, use the Neural Network Design Demonstration Method Comparison (`nnd9mc`).

Later in this chapter we will find that we can improve performance if we adjust the search directions, so that instead of being orthogonal they are *conjugate*. (We will define this term later.) If conjugate directions are used the function can be exactly minimized in at most n steps, where n is the dimension of $\mathbf{x}$. (There are certain types of quadratic functions that are minimized in one step by the steepest descent algorithm. Can you think of such a function? How is its Hessian matrix characterized?)

Newton's Method

The derivation of the steepest descent algorithm was based on the first-order Taylor series expansion (Eq. (9.4)). Newton's method is based on the second-order Taylor series:

$$F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k + \Delta\mathbf{x}_k) \approx F(\mathbf{x}_k) + \mathbf{g}_k^T \Delta\mathbf{x}_k + \frac{1}{2}\Delta\mathbf{x}_k^T \mathbf{A}_k \Delta\mathbf{x}_k . \qquad (9.40)$$

The principle behind Newton's method is to locate the stationary point of this quadratic approximation to $F(\mathbf{x})$. If we use Eq. (8.38) to take the gradient of this quadratic function with respect to $\Delta\mathbf{x}_k$ and set it equal to zero, we find

$$\mathbf{g}_k + \mathbf{A}_k \Delta \mathbf{x}_k = \mathbf{0} . \tag{9.41}$$

Solving for $\Delta \mathbf{x}_k$ produces

$$\Delta \mathbf{x}_k = -\mathbf{A}_k^{-1} \mathbf{g}_k . \tag{9.42}$$

Newton's Method

Newton's method is then defined:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1} \mathbf{g}_k . \tag{9.43}$$



To illustrate the operation of Newton's method, let's apply it to our previous example function of Eq. (9.12):

$$F(\mathbf{x}) = x_1^2 + 25x_2^2 . \tag{9.44}$$

The gradient and Hessian matrices are

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 50x_2 \end{bmatrix}, \quad \nabla^2 F(\mathbf{x}) = \begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix} . \tag{9.45}$$

If we start from the same initial guess

$$\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}, \tag{9.46}$$

the first step of Newton's method would be

$$\mathbf{x}_1 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - \begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 25 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} . \tag{9.47}$$

This method will always find the minimum of a quadratic function in one step. This is because Newton's method is designed to approximate a function as quadratic and then locate the stationary point of the quadratic approximation. If the original function is quadratic (with a strong minimum) it will be minimized in one step. The trajectory of Newton's method for this problem is given in Figure 9.5.

If the function $F(\mathbf{x})$ is not quadratic, then Newton's method will not generally converge in one step. In fact, we cannot be sure that it will converge at all, since this will depend on the function and the initial guess.

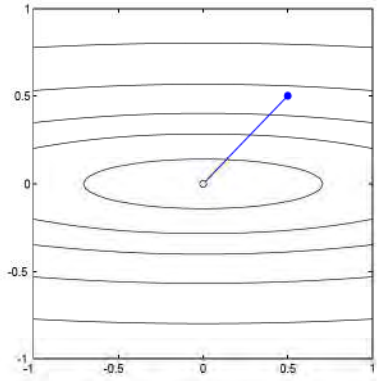


Figure 9.5 Trajectory for Newton's Method



Recall the function given by Eq. (8.18):

$$F(\mathbf{x}) = (x_2 - x_1)^4 + 8x_1x_2 - x_1 + x_2 + 3\,. \tag{9.48}$$

We know from Chapter 8 (see Problem P8.5) that this function has three stationary points:

$$\mathbf{x}^1 = \begin{bmatrix} -0.41878 \\ 0.41878 \end{bmatrix}, \mathbf{x}^2 = \begin{bmatrix} -0.134797 \\ 0.134797 \end{bmatrix}, \mathbf{x}^3 = \begin{bmatrix} 0.55358 \\ -0.55358 \end{bmatrix}. \tag{9.49}$$

The first point is a strong local minimum, the second point is a saddle point, and the third point is a strong global minimum.

If we apply Newton's method to this problem, starting from the initial guess $\mathbf{x}_0 = \begin{bmatrix} 1.5 & 0 \end{bmatrix}^T$, our first iteration will be as shown in Figure 9.6. The graph on the left-hand side of the figure is a contour plot of the original function. On the right we see the quadratic approximation to the function at the initial guess.

The function is not minimized in one step, which is not surprising since the function is not quadratic. However, we do take a step toward the global minimum, and if we continue for two more iterations the algorithm will converge to within 0.01 of the global minimum. Newton's method converges quickly in many applications because analytic functions can be accurately approximated by quadratic functions in a small neighborhood of a strong minimum. So as we move closer to the minimum point, Newton's method will more accurately predict its location. In this case we can see that the contour plot of the quadratic approximation is similar to the contour plot of the original function near the initial guess.

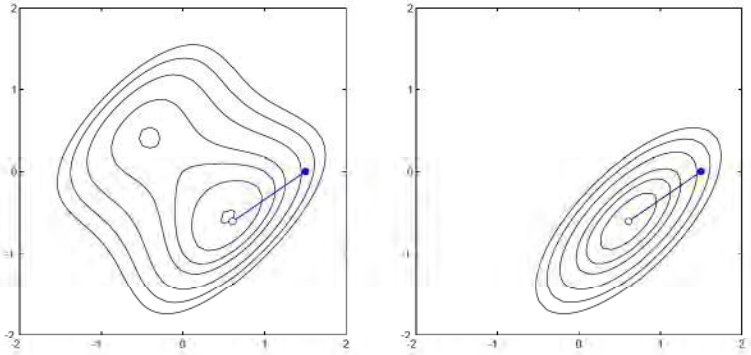


Figure 9.6 One Iteration of Newton's Method from $\mathbf{x}_0 = \begin{bmatrix} 1.5 & 0 \end{bmatrix}^T$

In Figure 9.7 we see one iteration of Newton's method from the initial guess $\mathbf{x}_0 = \begin{bmatrix} -1.5 & 0 \end{bmatrix}^T$. In this case we are converging to the local minimum. Clearly Newton's method cannot distinguish between a local minimum and a global minimum, since it approximates the function as a quadratic, and the quadratic function can have only one minimum. Newton's method, like steepest descent, relies on the local features of the surface (the first and second derivatives). It cannot know the global character of the function.

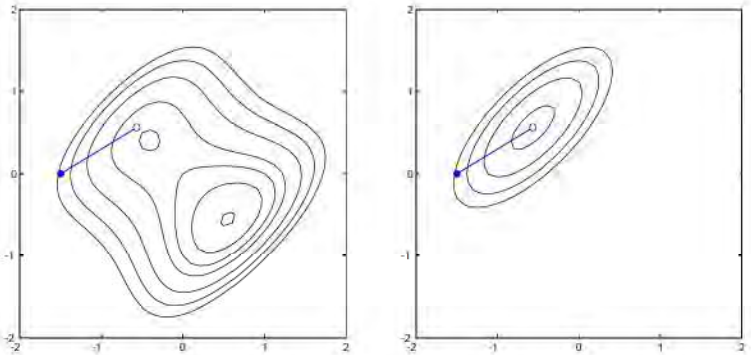


Figure 9.7 One Iteration of Newton's Method from $\mathbf{x}_0 = \begin{bmatrix} -1.5 & 0 \end{bmatrix}^T$

In Figure 9.8 we see one iteration of Newton's method from the initial guess $\mathbf{x}_0 = \begin{bmatrix} 0.75 & 0.75 \end{bmatrix}^T$. Now we are converging toward the saddle point of the function. Note that Newton's method locates the stationary point of the quadratic approximation to the function at the current guess. It does not distinguish between minima, maxima and saddle points. For this problem the quadratic approximation has a saddle point (indefinite Hessian ma-

trix), which is near the saddle point of the original function. If we continue the iterations, the algorithm does converge to the saddle point of $F(\mathbf{x})$.

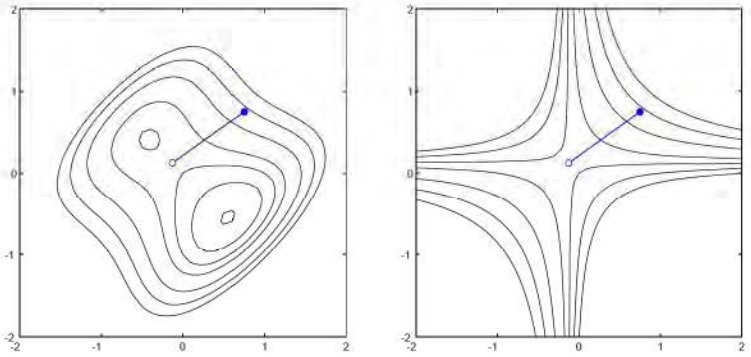


Figure 9.8 One Iteration of Newton's Method from $\mathbf{x}_0 = \begin{bmatrix} 0.75 & 0.75 \end{bmatrix}^T$

In each of the cases we have looked at so far the stationary point of the quadratic approximation has been close to a corresponding stationary point of $F(\mathbf{x})$. This is not always the case. In fact, Newton's method can produce very unpredictable results.

In Figure 9.9 we see one iteration of Newton's method from the initial guess $\mathbf{x}_0 = \begin{bmatrix} 1.15 & 0.75 \end{bmatrix}^T$. In this case the quadratic approximation predicts a saddle point, however, the saddle point is located very close to the local minimum of $F(\mathbf{x})$. If we continue the iterations, the algorithm will converge to the local minimum. Notice that the initial guess was actually farther away from the local minimum than it was for the previous case, in which the algorithm converged to the saddle point.

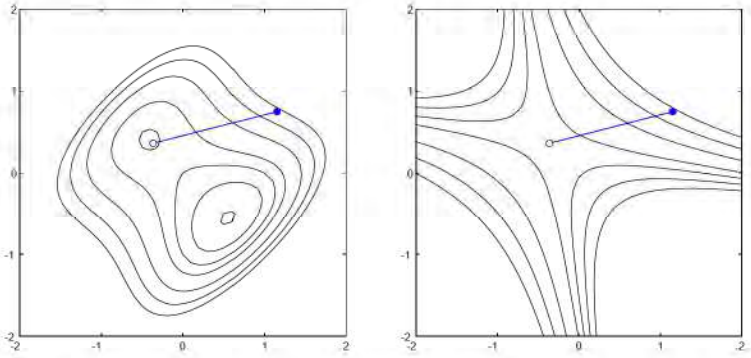


Figure 9.9 One Iteration of Newton's Method from $\mathbf{x}_0 = \begin{bmatrix} 1.15 & 0.75 \end{bmatrix}^T$



To experiment with Newton's method and steepest descent on this function, use the Neural Network Design Demonstrations Newton's Method (**nnd9nm**) *and Steepest Descent* (**nnd9sd**).

This is a good place to summarize some of the properties of Newton's method that we have observed.

While Newton's method usually produces faster convergence than steepest descent, the behavior of Newton's method can be quite complex. In addition to the problem of convergence to saddle points (which is very unlikely with steepest descent), it is possible for the algorithm to oscillate or diverge. Steepest descent is guaranteed to converge, if the learning rate is not too large or if we perform a linear minimization at each stage.

In Chapter 12 we will discuss a variation of Newton's method that is well suited to neural network training. It eliminates the divergence problem by using steepest descent steps whenever divergence begins to occur.

Another problem with Newton's method is that it requires the computation and storage of the Hessian matrix, as well as its inverse. If we compare steepest descent, Eq. (9.10), with Newton's method, Eq. (9.43), we see that their search directions will be identical when

$$\mathbf{A}_k = \mathbf{A}_k^{-1} = \mathbf{I}. \tag{9.50}$$

This observation has lead to a class of optimization algorithms know as quasi-Newton or one-step-secant methods. These methods replace $\mathbf{A}_k^{-1}$ with a positive definite matrix, $\mathbf{H}_k$,which is updated at each iteration without matrix inversion. The algorithms are typically designed so that for quadratic functions $\mathbf{H}_k$ will converge to $\mathbf{A}^{-1}$. (The Hessian is constant for quadratic functions.) See [Gill81], [Scal85] or [Batt92] for a discussion of these methods.

Conjugate Gradient

Quadratic Termination

Newton's method has a property called *quadratic termination*, which means that it minimizes a quadratic function exactly in a finite number of iterations. Unfortunately, it requires calculation and storage of the second derivatives. When the number of parameters, n , is large, it may be impractical to compute all of the second derivatives. (Note that the gradient has n elements, while the Hessian has n^2 elements.) This is especially true with neural networks, where practical applications can require several hundred to many thousand weights. For these cases we would like to have methods that require only first derivatives but still have quadratic termination.

Recall the performance of the steepest descent algorithm, with linear searches at each iteration. The search directions at consecutive iterations were orthogonal (see Figure 9.4). For quadratic functions with elliptical

contours this produces a zig-zag trajectory of short steps. Perhaps quadratic search directions are not the best choice. Is there a set of search directions that will guarantee quadratic termination? One possibility is conjugate directions.

Suppose that we wish to locate the minimum of the following quadratic function:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{d}^T\mathbf{x} + c\,. \tag{9.51}$$

Conjugate

A set of vectors $\{\mathbf{p}_k\}$ is mutually *conjugate* with respect to a positive definite Hessian matrix $\mathbf{A}$ if and only if

$$\mathbf{p}_k^T\mathbf{A}\mathbf{p}_j = 0 \qquad k \neq j\cdot \tag{9.52}$$

As with orthogonal vectors, there are an infinite number of mutually conjugate sets of vectors that span a given n-dimensional space. One set of conjugate vectors consists of the eigenvectors of $\mathbf{A}$. Let $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ and $\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n\}$ be the eigenvalues and eigenvectors of the Hessian matrix. To see that the eigenvectors are conjugate, replace $\mathbf{p}_k$ with $\mathbf{z}_k$ in Eq. (9.52):

$$\mathbf{z}_k^T\mathbf{A}\mathbf{z}_j = \lambda_j\mathbf{z}_k^T\mathbf{z}_j = 0 \qquad k \neq j\,, \tag{9.53}$$

where the last equality holds because the eigenvectors of a symmetric matrix are mutually orthogonal. Therefore the eigenvectors are both conjugate and orthogonal. (Can you find a quadratic function where all orthogonal vectors are also conjugate?)

It is not surprising that we can minimize a quadratic function exactly by searching along the eigenvectors of the Hessian matrix, since they form the principal axes of the function contours. (See the discussion on pages 8-13 through 8-19.) Unfortunately this is not of much practical help, since to find the eigenvectors we must first find the Hessian matrix. We want to find an algorithm that does not require the computation of second derivatives.

It can be shown (see [Scal85] or [Gill81]) that if we make a sequence of exact linear searches along any set of conjugate directions $\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n\}$, then the exact minimum of any quadratic function, with n parameters, will be reached in at most n searches. The question is "How can we construct these conjugate search directions?" First, we want to restate the conjugacy condition, which is given in Eq. (9.52), without use of the Hessian matrix. Recall that for quadratic functions

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d}\,, \tag{9.54}$$

$$\nabla^2 F(\mathbf{x}) = \mathbf{A}\,. \tag{9.55}$$

By combining these equations we find that the change in the gradient at iteration $k+1$ is

$$\Delta \mathbf{g}_k = \mathbf{g}_{k+1} - \mathbf{g}_k = (\mathbf{A}\mathbf{x}_{k+1} + \mathbf{d}) - (\mathbf{A}\mathbf{x}_k + \mathbf{d}) = \mathbf{A}\Delta \mathbf{x}_k , \tag{9.56}$$

where, from Eq. (9.2), we have

$$\Delta \mathbf{x}_k = (\mathbf{x}_{k+1} - \mathbf{x}_k) = \alpha_k \mathbf{p}_k , \tag{9.57}$$

and α_k is chosen to minimize $F(\mathbf{x})$ in the direction $\mathbf{p}_k$.

We can now restate the conjugacy conditions (Eq. (9.52)):

$$\alpha_k \mathbf{p}_k^T \mathbf{A}\mathbf{p}_j = \Delta \mathbf{x}_k^T \mathbf{A}\mathbf{p}_j = \Delta \mathbf{g}_k^T \mathbf{p}_j = 0 \qquad k \neq j . \tag{9.58}$$

Note that we no longer need to know the Hessian matrix. We have restated the conjugacy conditions in terms of the changes in the gradient at successive iterations of the algorithm. The search directions will be conjugate if they are orthogonal to the changes in the gradient.

Note that the first search direction, $\mathbf{p}_0$, is arbitrary, and $\mathbf{p}_1$ can be any vector that is orthogonal to $\Delta \mathbf{g}_0$. Therefore there are an infinite number of sets of conjugate vectors. It is common to begin the search in the steepest descent direction:

$$\mathbf{p}_0 = -\mathbf{g}_0 . \tag{9.59}$$

Then, at each iteration we need to construct a vector $\mathbf{p}_k$ that is orthogonal to $\{\Delta \mathbf{g}_0, \Delta \mathbf{g}_1, \ldots, \Delta \mathbf{g}_{k-1}\}$. It is a procedure similar to Gram-Schmidt orthogonalization, which we discussed in Chapter 5. It can be simplified (see [Scal85]) to iterations of the form

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1} . \tag{9.60}$$

The scalars β_k can be chosen by several different methods, which produce equivalent results for quadratic functions. The most common choices (see [Scal85]) are

$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\Delta \mathbf{g}_{k-1}^T \mathbf{p}_{k-1}} , \tag{9.61}$$

due to Hestenes and Stiefel,

$$\beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}} \tag{9.62}$$

due to Fletcher and Reeves, and

$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}} \tag{9.63}$$

due to Polak and Ribiére.

Conjugate Gradient

To summarize our discussion, the *conjugate gradient* method consists of the following steps:

1. Select the first search direction to be the negative of the gradient, as in Eq. (9.59).
2. Take a step according to Eq. (9.57), selecting the learning rate α_k to minimize the function along the search direction. We will discuss general linear minimization techniques in Chapter 12. For quadratic functions we can use Eq. (9.31).
3. Select the next search direction according to Eq. (9.60), using Eq. (9.61), Eq. (9.62), or Eq. (9.63) to calculate β_k.
4. If the algorithm has not converged, return to step 2.



To illustrate the performance of the algorithm, recall the example we used to demonstrate steepest descent with linear minimization:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \mathbf{x}, \tag{9.64}$$

with initial guess

$$\mathbf{x}_0 = \begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix}. \tag{9.65}$$

The gradient of this function is

$$\nabla F(\mathbf{x}) = \begin{bmatrix} 2x_1 + x_2 \\ x_1 + 2x_2 \end{bmatrix}. \tag{9.66}$$

As with steepest descent, the first search direction is the negative of the gradient:

$$\mathbf{p}_0 = -\mathbf{g}_0 = -\nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}_0} = \begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}. \tag{9.67}$$

From Eq. (9.31), the learning rate for the first iteration will be

$$\alpha_0 = -\frac{\begin{bmatrix}1.35 & 0.3\end{bmatrix}\begin{bmatrix}-1.35\\-0.3\end{bmatrix}}{\begin{bmatrix}-1.35 & -0.3\end{bmatrix}\begin{bmatrix}2 & 1\\1 & 2\end{bmatrix}\begin{bmatrix}-1.35\\-0.3\end{bmatrix}} = 0.413\,. \tag{9.68}$$

The first step of conjugate gradient is therefore:

$$\mathbf{x}_1 = \mathbf{x}_0 + \alpha_0\mathbf{p}_0 = \begin{bmatrix}0.8\\-0.25\end{bmatrix} + 0.413\begin{bmatrix}-1.35\\-0.3\end{bmatrix} = \begin{bmatrix}0.24\\-0.37\end{bmatrix}, \tag{9.69}$$

which is equivalent to the first step of steepest descent with minimization along a line.

Now we need to find the second search direction from Eq. (9.60). This requires the gradient at $\mathbf{x}_1$:

$$\mathbf{g}_1 = \nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}_1} = \begin{bmatrix}2 & 1\\1 & 2\end{bmatrix}\begin{bmatrix}0.24\\-0.37\end{bmatrix} = \begin{bmatrix}0.11\\-0.5\end{bmatrix}. \tag{9.70}$$

We can now find β_1:

$$\beta_1 = \frac{\mathbf{g}_1^T\mathbf{g}_1}{\mathbf{g}_0^T\mathbf{g}_0} = \frac{\begin{bmatrix}0.11 & -0.5\end{bmatrix}\begin{bmatrix}0.11\\-0.5\end{bmatrix}}{\begin{bmatrix}1.35 & 0.3\end{bmatrix}\begin{bmatrix}1.35\\0.3\end{bmatrix}} = \frac{0.2621}{1.9125} = 0.137\,, \tag{9.71}$$

using the method of Fletcher and Reeves (Eq. (9.62)). The second search direction is then computed from Eq. (9.60):

$$\mathbf{p}_1 = -\mathbf{g}_1 + \beta_1\mathbf{p}_0 = \begin{bmatrix}-0.11\\0.5\end{bmatrix} + 0.137\begin{bmatrix}-1.35\\-0.3\end{bmatrix} = \begin{bmatrix}-0.295\\0.459\end{bmatrix}. \tag{9.72}$$

From Eq. (9.31), the learning rate for the second iteration will be

$$\alpha_1 = -\frac{\begin{bmatrix}0.11 & -0.5\end{bmatrix}\begin{bmatrix}-0.295\\0.459\end{bmatrix}}{\begin{bmatrix}-0.295 & 0.459\end{bmatrix}\begin{bmatrix}2 & 1\\1 & 2\end{bmatrix}\begin{bmatrix}-0.295\\0.459\end{bmatrix}} = \frac{0.262}{0.325} = 0.807\,. \tag{9.73}$$

The second step of conjugate gradient is therefore

$$\mathbf{x}_2 = \mathbf{x}_1 + \alpha_1 \mathbf{p}_1 = \begin{bmatrix} 0.24 \\ -0.37 \end{bmatrix} + 0.807 \begin{bmatrix} -0.295 \\ 0.459 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{9.74}$$

As predicted, the algorithm converges exactly to the minimum in two iterations (since this is a two-dimensional quadratic function), as illustrated in Figure 9.10. Compare this result with the steepest descent algorithm, as shown in Figure 9.4. The conjugate gradient algorithm adjusts the second search direction so that it will pass through the minimum of the function (center of the function contours), instead of using an orthogonal search direction, as in steepest descent.

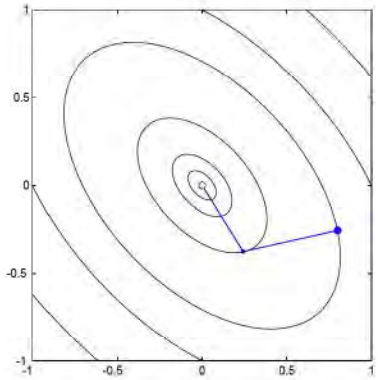


Figure 9.10 Conjugate Gradient Algorithm

We will return to the conjugate gradient algorithm in Chapter 12. In that chapter we will discuss how the algorithm should be adjusted for non-quadratic functions.



To experiment with the conjugate gradient algorithm and compare it with steepest descent, use the Neural Network Design Demonstration Method Comparison (`nnd9mc`).

Summary of Results

General Minimization Algorithm

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

or

$$\Delta \mathbf{x}_k = (\mathbf{x}_{k+1} - \mathbf{x}_k) = \alpha_k \mathbf{p}_k$$

Steepest Descent Algorithm

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$$

Where $\mathbf{g}_k \equiv \nabla F(\mathbf{x})\big|_{\mathbf{x} = \mathbf{x}_k}$

Stable Learning Rate ($\alpha_k = \alpha$, constant)

$$\alpha < \frac{2}{\lambda_{max}}$$

$\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ Eigenvalues of Hessian matrix $\mathbf{A}$

Learning Rate to Minimize Along the Line $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$

$$\alpha_k = -\frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{A} \mathbf{p}_k} \qquad \text{(For quadratic functions)}$$

After Minimizing Along the Line $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$

$$\mathbf{g}_{k+1}^T \mathbf{p}_k = 0$$

Newton's Method

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1} \mathbf{g}_k$$

Where $\mathbf{A}_k \equiv \nabla^2 F(\mathbf{x})\big|_{\mathbf{x} = \mathbf{x}_k}$

Conjugate Gradient Algorithm

$$\Delta \mathbf{x}_k = \alpha_k \mathbf{p}_k$$

Learning rate α_k is chosen to minimize along the line $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$.

$$\mathbf{p}_0 = -\mathbf{g}_0$$

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\Delta \mathbf{g}_{k-1}^T \mathbf{p}_{k-1}} \text{ or } \beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}} \text{ or } \beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

Where $\mathbf{g}_k \equiv \nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}_k}$ and $\Delta \mathbf{g}_k = \mathbf{g}_{k+1} - \mathbf{g}_k$.

Solved Problems

P9.1 We want to find the minimum of the following function:

$$F(\mathbf{x}) = 5x_1^2 - 6x_1x_2 + 5x_2^2 + 4x_1 + 4x_2 .$$

i. Sketch a contour plot of this function.

ii. Sketch the trajectory of the steepest descent algorithm on the contour plot of part (i) if the initial guess is $\mathbf{x}_0 = \begin{bmatrix} -1 & -2.5 \end{bmatrix}^T$. Assume a very small learning rate is used.

iii. What is the maximum stable learning rate?

i. To sketch the contour plot we first need to find the Hessian matrix. For quadratic functions we can do this by putting the function into the standard form (see Eq. (8.35)):

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{d}^T\mathbf{x} + c = \frac{1}{2}\mathbf{x}^T\begin{bmatrix} 10 & -6 \\ -6 & 10 \end{bmatrix}\mathbf{x} + \begin{bmatrix} 4 & 4 \end{bmatrix}\mathbf{x} .$$

From Eq. (8.39) the Hessian matrix is

$$\nabla^2 F(\mathbf{x}) = \mathbf{A} = \begin{bmatrix} 10 & -6 \\ -6 & 10 \end{bmatrix} .$$

The eigenvalues and eigenvectors of this matrix are

$$\lambda_1 = 4 , \mathbf{z}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} , \lambda_2 = 16 , \mathbf{z}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix} .$$

From the discussion on quadratic functions in Chapter 8 (see page 8-15) we know that the function contours are elliptical. The maximum curvature of $F(\mathbf{x})$ is in the direction of $\mathbf{z}_2$, since λ_2 is larger than λ_1, and the minimum curvature is in the direction of $\mathbf{z}_1$ (the long axis of the ellipses).

Next we need to find the center of the contours (the stationary point). This occurs when the gradient is equal to zero. From Eq. (8.38) we find

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d} = \begin{bmatrix} 10 & -6 \\ -6 & 10 \end{bmatrix}\mathbf{x} + \begin{bmatrix} 4 \\ 4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} .$$

Therefore

$$\mathbf{x}^* = -\begin{bmatrix} 10 & -6 \\ -6 & 10 \end{bmatrix}^{-1}\begin{bmatrix} 4 \\ 4 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \end{bmatrix}.$$

The contours will be elliptical, centered at $\mathbf{x}^*$, with long axis in the direction of $\mathbf{z}_1$. The contour plot is shown in Figure P9.1.

ii. We know that the gradient is always orthogonal to the contour line, therefore the steepest descent trajectory, if we take small enough steps, will follow a path that is orthogonal to each contour line it intersects. We can therefore trace the trajectory without performing any computations. The result is shown in Figure P9.1.

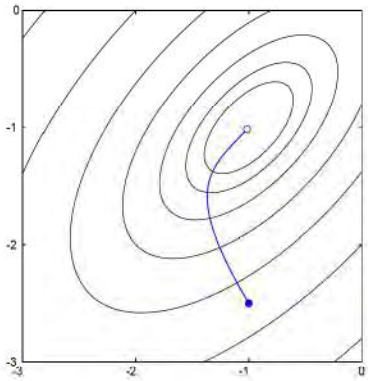


Figure P9.1 Contour Plot and Steep. Desc. Trajectory for Problem P9.1

iii. From Eq. (9.25) we know that the maximum stable learning rate for a quadratic function is determined by the maximum eigenvalue of the Hessian matrix:

$$\alpha < \frac{2}{\lambda_{max}}.$$

The maximum eigenvalue for this problem is $\lambda_2 = 16$, therefore for stability

$$\alpha < \frac{2}{16} = 0.125.$$

This result is verified experimentally in Figure P9.2, which shows the steepest descent trajectories when the learning rate is just below ($\alpha = 0.12$) and just above ($\alpha = 0.13$) the maximum stable value.

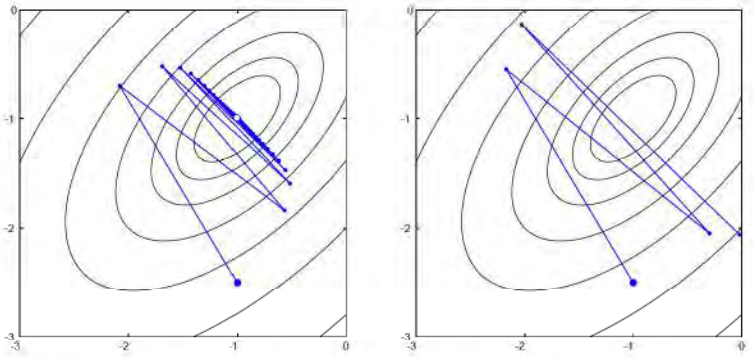


Figure P9.2 Trajectories for $\alpha = 0.12$ (left) and $\alpha = 0.13$ (right)

P9.2 Consider again the quadratic function of Problem P9.1. Take two steps of the steepest descent algorithm, minimizing along a line at each step. Use the following initial condition:

$$\mathbf{x}_0 = \begin{bmatrix} 0 & -2 \end{bmatrix}^T.$$

In Problem P9.1 we found the gradient of the function to be

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d} = \begin{bmatrix} 10 & -6 \\ -6 & 10 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 4 \\ 4 \end{bmatrix}.$$

If we evaluate this at $\mathbf{x}_0$, we find

$$\mathbf{g}_0 = \nabla F(\mathbf{x}_0) = \mathbf{A}\mathbf{x}_0 + \mathbf{d} = \begin{bmatrix} 10 & -6 \\ -6 & 10 \end{bmatrix} \begin{bmatrix} 0 \\ -2 \end{bmatrix} + \begin{bmatrix} 4 \\ 4 \end{bmatrix} = \begin{bmatrix} 16 \\ -16 \end{bmatrix}.$$

Therefore the first search direction is

$$\mathbf{p}_0 = -\mathbf{g}_0 = \begin{bmatrix} -16 \\ 16 \end{bmatrix}.$$

To minimize along a line, for a quadratic function, we can use Eq. (9.31):

$$\alpha_0 = -\frac{\mathbf{g}_0^T \mathbf{p}_0}{\mathbf{p}_0^T \mathbf{A} \mathbf{p}_0} = -\frac{\begin{bmatrix} 16 & -16 \end{bmatrix} \begin{bmatrix} -16 \\ 16 \end{bmatrix}}{\begin{bmatrix} -16 & 16 \end{bmatrix} \begin{bmatrix} 10 & -6 \\ -6 & 10 \end{bmatrix} \begin{bmatrix} -16 \\ 16 \end{bmatrix}} = -\frac{-512}{8192} = 0.0625,$$

Therefore the first iteration of steepest descent will be

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha_0 \mathbf{g}_0 = \begin{bmatrix} 0 \\ -2 \end{bmatrix} - 0.0625 \begin{bmatrix} 16 \\ -16 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \end{bmatrix}.$$

To begin the second iteration we need to find the gradient at $\mathbf{x}_1$:

$$\mathbf{g}_1 = \nabla F(\mathbf{x}_1) = \mathbf{A}\mathbf{x}_1 + \mathbf{d} = \begin{bmatrix} 10 & -6 \\ -6 & 10 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \end{bmatrix} + \begin{bmatrix} 4 \\ 4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Therefore we have reached a stationary point; the algorithm has converged. From Problem P9.1 we know that $\mathbf{x}_1$ is indeed the minimum point of this quadratic function. The trajectory is shown in Figure P9.3.

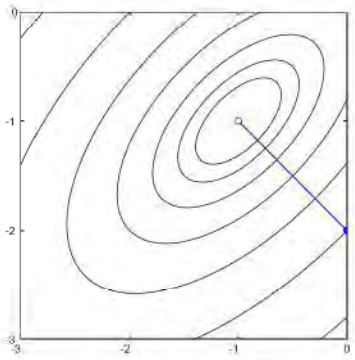


Figure P9.3 Steepest Descent with Linear Minimization for Problem P9.2

This is an unusual case, where the steepest descent algorithm located the minimum in one iteration. Notice that this occurred because the initial guess was located in the direction of one of the eigenvectors of the Hessian matrix, with respect to the minimum point. For those cases where every direction is an eigenvector, the steepest descent algorithm will always locate the minimum in one iteration. What would this imply about the eigenvalues of the Hessian matrix?

P9.3 Recall Problem P8.6, in which we derived a performance index for a linear neural network. The network, which is displayed again in Figure P9.4, was to be trained for the following input/output pairs:

$$\{(p_1 = 2), (t_1 = 0.5)\}, \{(p_2 = -1), (t_2 = 0)\}$$

The performance index for the network was defined to be

$$F(\mathbf{x}) = (t_1 - a_1(\mathbf{x}))^2 + (t_2 - a_2(\mathbf{x}))^2,$$

which was displayed in Figure P8.8.

i. Use the steepest descent algorithm to locate the optimal parameters for this network (recall that $\mathbf{x} = \begin{bmatrix} w & b \end{bmatrix}^T$), starting from the initial guess $\mathbf{x}_0 = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$. Use a learning rate of $\alpha = 0.05$.

ii. What is the maximum stable learning rate?

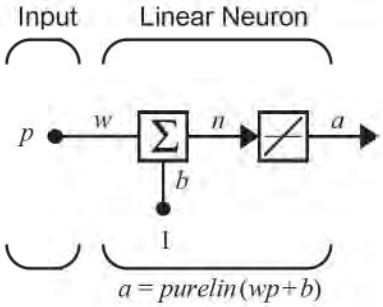


Figure P9.4 Linear Network for Problems P9.3 and P8.6

i. In Problem P8.6 we found that the performance index could be written in quadratic form:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{d}^T\mathbf{x} + c,$$

where

$$c = \mathbf{t}^T\mathbf{t} = \begin{bmatrix} 0.5 & 0 \end{bmatrix}\begin{bmatrix} 0.5 \\ 0 \end{bmatrix} = 0.25,$$

$$\mathbf{d} = -2\mathbf{G}^T\mathbf{t} = -2\begin{bmatrix} 2 & -1 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} 0.5 \\ 0 \end{bmatrix} = \begin{bmatrix} -2 \\ -1 \end{bmatrix},$$

$$\mathbf{A} = 2\mathbf{G}^T\mathbf{G} = \begin{bmatrix} 10 & 2 \\ 2 & 4 \end{bmatrix}.$$

The gradient at $\mathbf{x}_0$ is

$$\mathbf{g}_0 = \nabla F(\mathbf{x}_0) = \mathbf{A}\mathbf{x}_0 + \mathbf{d} = \begin{bmatrix} 10 & 2 \\ 2 & 4 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -2 \\ -1 \end{bmatrix} = \begin{bmatrix} 10 \\ 5 \end{bmatrix}.$$

The first iteration of steepest descent will be

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha \mathbf{g}_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 0.05 \begin{bmatrix} 10 \\ 5 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.75 \end{bmatrix}.$$

The second iteration will be

$$\mathbf{x}_2 = \mathbf{x}_1 - \alpha \mathbf{g}_1 = \begin{bmatrix} 0.5 \\ 0.75 \end{bmatrix} - 0.05 \begin{bmatrix} 4.5 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.275 \\ 0.6 \end{bmatrix}.$$

The remaining iterations are displayed in Figure P9.5. The algorithm converges to the minimum point $\mathbf{x}^* = \begin{bmatrix} 0.167 & 0.167 \end{bmatrix}^T$. Therefore the optimal value for both the weight and the bias of this network is 0.167.

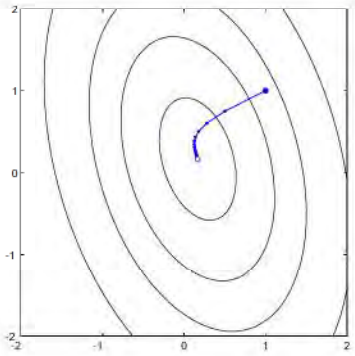


Figure P9.5 Steepest Descent Trajectory for Problem P9.3 with $\alpha = 0.05$

Note that in order to train this network we needed to know all of the input/output pairs. We then performed iterations of the steepest descent algorithm until convergence was achieved. In Chapter 10 we will introduce an adaptive algorithm, based on steepest descent, for training linear networks. With this adaptive algorithm the network parameters are updated after each input/output pair is presented. We will show how this allows the network to adapt to a changing environment.

ii. The maximum eigenvalue of the Hessian matrix for this problem is $\lambda_1 = 10.6$ (see Problem P8.6), therefore for stability

$$\alpha < \frac{2}{10.6} = 0.1887.$$

P9.4 Consider the function

$$F(\mathbf{x}) = e^{(x_1^2 - x_1 + 2x_2^2 + 4)}.$$

Take one iteration of Newton's method from the initial guess $\mathbf{x}_0 = \begin{bmatrix} 1 & -2 \end{bmatrix}^T$. How close is this result to the minimum point of $F(\mathbf{x})$? Explain.

The first step is to find the gradient and the Hessian matrix. The gradient is given by

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = e^{(x_1^2 - x_1 + 2x_2^2 + 4)} \begin{bmatrix} (2x_1 - 1) \\ (4x_2) \end{bmatrix},$$

and the Hessian matrix is given by

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2}{\partial x_1^2} F(\mathbf{x}) & \frac{\partial^2}{\partial x_1 \partial x_2} F(\mathbf{x}) \\ \frac{\partial^2}{\partial x_2 \partial x_1} F(\mathbf{x}) & \frac{\partial^2}{\partial x_2^2} F(\mathbf{x}) \end{bmatrix}$$

$$= e^{(x_1^2 - x_1 + 2x_2^2 + 4)} \begin{bmatrix} 4x_1^2 - 4x_1 + 3 & (2x_1 - 1)(4x_2) \\ (2x_1 - 1)(4x_2) & 16x_2^2 + 4 \end{bmatrix}$$

If we evaluate these at the initial guess we find

$$\mathbf{g}_0 = \nabla F(\mathbf{x})\big|_{\mathbf{x} = \mathbf{x}_0} = \begin{bmatrix} 0.163 \times 10^6 \\ -1.302 \times 10^6 \end{bmatrix},$$

and

$$\mathbf{A}_0 = \nabla^2 F(\mathbf{x})\big|_{\mathbf{x} = \mathbf{x}_0} = \begin{bmatrix} 0.049 \times 10^7 & -0.130 \times 10^7 \\ -0.130 \times 10^7 & 1.107 \times 10^7 \end{bmatrix}.$$

Therefore the first iteration of Newton's method, from Eq. (9.43), will be

$$\mathbf{x}_1 = \mathbf{x}_0 - \mathbf{A}_0^{-1}\mathbf{g}_0 = \begin{bmatrix} 1 \\ -2 \end{bmatrix} - \begin{bmatrix} 0.049\times10^7 & -0.130\times10^7 \\ -0.130\times10^7 & 1.107\times10^7 \end{bmatrix}^{-1} \begin{bmatrix} 0.163\times10^6 \\ -1.302\times10^6 \end{bmatrix} = \begin{bmatrix} 0.971 \\ -1.886 \end{bmatrix}$$

How close is this to the true minimum point of $F(\mathbf{x})$? First, note that the exponent of $F(\mathbf{x})$ is a quadratic function:

$$x_1^2 - x_1 + 2x_2^2 + 4 = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{d}^T\mathbf{x} + c = \frac{1}{2}\mathbf{x}^T\begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}\mathbf{x} + \begin{bmatrix} -1 & 0 \end{bmatrix}\mathbf{x} + 4 .$$

The minimum point of $F(\mathbf{x})$ will be the same as the minimum point of the exponent, which is

$$\mathbf{x}^* = -\mathbf{A}^{-1}\mathbf{d} = -\begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}^{-1}\begin{bmatrix} -1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0 \end{bmatrix} .$$

Therefore Newton's method has taken only a very small step toward the true minimum point. This is because $F(\mathbf{x})$ cannot be accurately approximated by a quadratic function in the neighborhood of $\mathbf{x}_0 = \begin{bmatrix} 1 & -2 \end{bmatrix}^T$.

For this problem Newton's method will converge to the true minimum point, but it will take many iterations. The trajectory for Newton's method is illustrated in Figure P9.6.

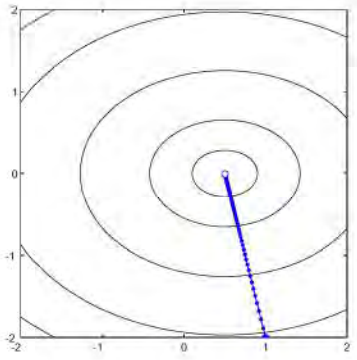


Figure P9.6 Newton's Method Trajectory for Problem P9.4

P9.5 Compare the performance of Newton's method and steepest descent on the following function:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\mathbf{x}.$$

Start from the initial guess

$$\mathbf{x}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Recall that this function is an example of a stationary valley (see Eq. (8.59) and Figure 8.9). The gradient is

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d} = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\mathbf{x}$$

and the Hessian matrix is

$$\nabla^2 F(\mathbf{x}) = \mathbf{A} = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}.$$

Newton's method is given by

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k.$$

Note, however, that we cannot actually perform this algorithm, because the Hessian matrix is singular. We know from our discussion of this function in Chapter 8 that this function does not have a strong minimum, but it does have a weak minimum along the line $x_1 = x_2$.

What about steepest descent? If we start from the initial guess, with learning rate $\alpha = 0.1$, the first two iterations will be

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha\mathbf{g}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} - 0.1\begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.9 \\ 0.1 \end{bmatrix},$$

$$\mathbf{x}_2 = \mathbf{x}_1 - \alpha\mathbf{g}_1 = \begin{bmatrix} 0.9 \\ 0.1 \end{bmatrix} - 0.1\begin{bmatrix} 0.8 \\ -0.8 \end{bmatrix} = \begin{bmatrix} 0.82 \\ 0.18 \end{bmatrix}.$$

The complete trajectory is shown in Figure P9.7. This is a case where the steepest descent algorithm performs better than Newton's method. Steepest descent converges to a minimum point (weak minimum), while Newton's method fails to converge. In Chapter 12 we will discuss a technique that combines steepest descent with Newton's method, to overcome the problem of singular (or almost singular) Hessian matrices.

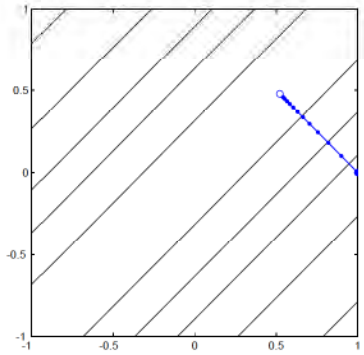


Figure P9.7 Steepest Descent Trajectory for Problem P9.5 with $\alpha = 0.1$

P9.6 Consider the following function:

$$F(\mathbf{x}) = x_1^3 + x_1 x_2 - x_1^2 x_2^2$$

i. Perform one iteration of Newton's method from the initial guess $\mathbf{x}_0 = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$.

ii. Find the second-order Taylor series expansion of $F(\mathbf{x})$ about $\mathbf{x}_0$. Is this quadratic function minimized at the point $\mathbf{x}_1$ found in part (i)? Explain.

i. The gradient of $F(\mathbf{x})$ is

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 3x_1^2 + x_2 - 2x_1 x_2^2 \\ x_1 - 2x_1^2 x_2 \end{bmatrix},$$

and the Hessian matrix is

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} 6x_1 - 2x_2^2 & 1 - 4x_1x_2 \\ 1 - 4x_1x_2 & -2x_1^2 \end{bmatrix}.$$

If we evaluate these at the initial guess we find

$$\mathbf{g}_0 = \nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}_0} = \begin{bmatrix} 2 \\ -1 \end{bmatrix},$$

$$\mathbf{A}_0 = \nabla^2 F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}_0} = \begin{bmatrix} 4 & -3 \\ -3 & -2 \end{bmatrix}.$$

The first iteration of Newton's method is then

$$\mathbf{x}_1 = \mathbf{x}_0 - \mathbf{A}_0^{-1}\mathbf{g}_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 4 & -3 \\ -3 & -2 \end{bmatrix}^{-1} \begin{bmatrix} 2 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.5882 \\ 1.1176 \end{bmatrix}.$$

ii. From Eq. (9.40), the second-order Taylor series expansion of $F(\mathbf{x})$ about $\mathbf{x}_0$ is

$$F(\mathbf{x}) = F(\mathbf{x}_0 + \Delta\mathbf{x}_0) \approx F(\mathbf{x}_0) + \mathbf{g}_0^T \Delta\mathbf{x}_0 + \frac{1}{2}\Delta\mathbf{x}_0^T \mathbf{A}_0 \Delta\mathbf{x}_0 .$$

If we substitute the values for $\mathbf{x}_0$, $\mathbf{g}_0$ and $\mathbf{A}_0$, we find

$$F(\mathbf{x}) \approx 1 + \begin{bmatrix} 2 & -1 \end{bmatrix} \left\{ \mathbf{x} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\} + \frac{1}{2} \left\{ \mathbf{x} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}^T \begin{bmatrix} 4 & -3 \\ -3 & -2 \end{bmatrix} \left\{ \mathbf{x} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}.$$

This can be reduced to

$$F(\mathbf{x}) \approx -2 + \begin{bmatrix} 1 & 4 \end{bmatrix} \mathbf{x} + \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 4 & -3 \\ -3 & -2 \end{bmatrix} \mathbf{x}.$$

This function has a stationary point at $\mathbf{x}_1$. The question is whether or not the stationary point is a strong minimum. This can be determined from the eigenvalues of the Hessian matrix. If both eigenvalues are positive, it is a strong minimum. If both eigenvalues are negative, it is a strong maximum. If the two eigenvalues have opposite signs, it is a saddle point. In this case the eigenvalues of $\mathbf{A}_0$ are

$$\lambda_1 = 5.24 \text{ and } \lambda_2 = -3.24 .$$

Therefore the quadratic approximation to $F(\mathbf{x})$ at $\mathbf{x}_0$ is not minimized at $\mathbf{x}_1$, since it is a saddle point. Figure P9.8 displays the contour plots of $F(\mathbf{x})$ and its quadratic approximation.

This sort of problem was also illustrated in Figure 9.8 and Figure 9.9. Newton's method does locate the stationary point of the quadratic approximation of the function at the current guess. It does not distinguish between minima, maxima and saddle points.

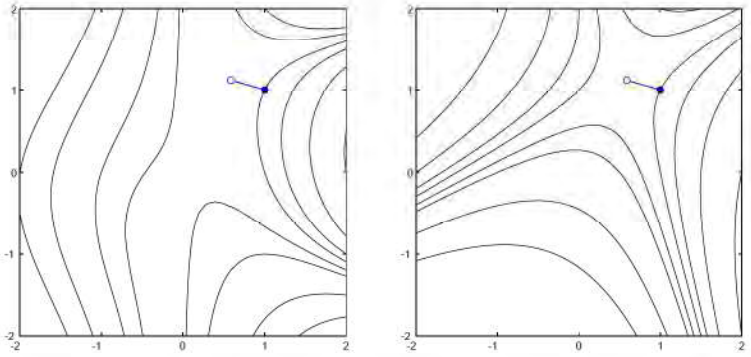


Figure P9.8 One Iteration of Newton's Method from $\mathbf{x}_0 = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$

P9.7 Repeat Problem P9.3 (i) using the conjugate gradient algorithm.

Recall that the function to be minimized was

$$F(\mathbf{x}) = 0.25 + \begin{bmatrix} -2 & -1 \end{bmatrix}\mathbf{x} + \frac{1}{2}\mathbf{x}^T\begin{bmatrix} 10 & 2 \\ 2 & 4 \end{bmatrix}\mathbf{x}.$$

The gradient at $\mathbf{x}_0$ is

$$\mathbf{g}_0 = \nabla F(\mathbf{x}_0) = \mathbf{A}\mathbf{x}_0 + \mathbf{d} = \begin{bmatrix} 10 & 2 \\ 2 & 4 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -2 \\ -1 \end{bmatrix} = \begin{bmatrix} 10 \\ 5 \end{bmatrix}.$$

The first search direction is then

$$\mathbf{p}_0 = -\mathbf{g}_0 = \begin{bmatrix} -10 \\ -5 \end{bmatrix}.$$

To minimize along a line, for a quadratic function, we can use Eq. (9.31):

$$\alpha_0 = -\frac{\mathbf{g}_0^T\mathbf{p}_0}{\mathbf{p}_0^T\mathbf{A}\mathbf{p}_0} = -\frac{\begin{bmatrix}10 & 5\end{bmatrix}\begin{bmatrix}-10\\-5\end{bmatrix}}{\begin{bmatrix}-10 & -5\end{bmatrix}\begin{bmatrix}10 & 2\\2 & 4\end{bmatrix}\begin{bmatrix}-10\\-5\end{bmatrix}} = -\frac{-125}{1300} = 0.0962\,.$$

Therefore the first iteration of conjugate gradient will be

$$\mathbf{x}_1 = \mathbf{x}_0 + \alpha_0\mathbf{p}_0 = \begin{bmatrix}1\\1\end{bmatrix} + 0.0962\begin{bmatrix}-10\\-5\end{bmatrix} = \begin{bmatrix}0.038\\0.519\end{bmatrix}.$$

Now we need to find the second search direction from Eq. (9.60). This requires the gradient at $\mathbf{x}_1$:

$$\mathbf{g}_1 = \nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}_1} = \begin{bmatrix}10 & 2\\2 & 4\end{bmatrix}\begin{bmatrix}0.038\\0.519\end{bmatrix} + \begin{bmatrix}-2\\-1\end{bmatrix} = \begin{bmatrix}-0.577\\1.154\end{bmatrix}.$$

We can now find β_1:

$$\beta_1 = \frac{\Delta\mathbf{g}_0^T\mathbf{g}_1}{\mathbf{g}_0^T\mathbf{g}_0} = \frac{\begin{bmatrix}-10.577 & -3.846\end{bmatrix}\begin{bmatrix}-0.577\\1.154\end{bmatrix}}{\begin{bmatrix}10 & 5\end{bmatrix}\begin{bmatrix}10\\5\end{bmatrix}} = \frac{1.665}{125} = 0.0133\,,$$

using the method of Polak and Ribiére (Eq. (9.63)). (The other two methods for computing β_1 will produce the same results for a quadratic function. You may want to try them.) The second search direction is then computed from Eq. (9.60):

$$\mathbf{p}_1 = -\mathbf{g}_1 + \beta_1\mathbf{p}_0 = \begin{bmatrix}0.577\\-1.154\end{bmatrix} + 0.0133\begin{bmatrix}-10\\-5\end{bmatrix} = \begin{bmatrix}0.444\\-1.220\end{bmatrix}.$$

From Eq. (9.31), the learning rate for the second iteration will be

$$\alpha_1 = -\frac{\begin{bmatrix}-0.577 & 1.154\end{bmatrix}\begin{bmatrix}0.444\\-1.220\end{bmatrix}}{\begin{bmatrix}0.444 & -1.220\end{bmatrix}\begin{bmatrix}10 & 2\\2 & 4\end{bmatrix}\begin{bmatrix}0.444\\-1.220\end{bmatrix}} = -\frac{-1.664}{5.758} = 0.2889\,.$$

The second step of conjugate gradient is therefore

$$\mathbf{x}_2 = \mathbf{x}_1 + \alpha_1 \mathbf{p}_1 = \begin{bmatrix} 0.038 \\ 0.519 \end{bmatrix} + 0.2889 \begin{bmatrix} 0.444 \\ -1.220 \end{bmatrix} = \begin{bmatrix} 0.1667 \\ 0.1667 \end{bmatrix}.$$

As expected, the minimum is reached in two iterations. The trajectory is illustrated in Figure P9.9.

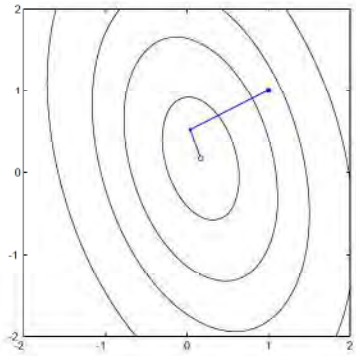


Figure P9.9 Conjugate Gradient Trajectory for Problem P9.7

P9.8 Show that conjugate vectors are independent.

Suppose that we have a set of vectors, $\{\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_{n-1}\}$, which are conjugate with respect to the Hessian matrix $\mathbf{A}$. If these vectors are dependent, then, from Eq. (5.4), it must be true that

$$\sum_{j=0}^{n-1} a_j \mathbf{p}_j = \mathbf{0},$$

for some set of constants $a_0, a_1, \ldots, a_{n-1}$, at least one of which is nonzero. If we multiply both sides of this equation by $\mathbf{p}_k^T \mathbf{A}$, we obtain

$$\mathbf{p}_k^T \mathbf{A} \sum_{j=0}^{n-1} a_j \mathbf{p}_j = \sum_{j=0}^{n-1} a_j \mathbf{p}_k^T \mathbf{A} \mathbf{p}_j = a_k \mathbf{p}_k^T \mathbf{A} \mathbf{p}_k = 0,$$

where the second equality comes from the definition of conjugate vectors in Eq. (9.52). If $\mathbf{A}$ is positive definite (a unique strong minimum exists), then $\mathbf{p}_k^T \mathbf{A} \mathbf{p}_k$ must be strictly positive. This implies that a_k must be zero for all k. Therefore conjugate directions must be independent.

Epilogue

In this chapter we have introduced three different optimization algorithms: steepest descent, Newton's method and conjugate gradient. The basis for these algorithms is the Taylor series expansion. Steepest descent is derived by using a first-order expansion, whereas Newton's method and conjugate gradient are designed for second-order (quadratic) functions.

Steepest descent has the advantage that it is very simple, requiring calculation only of the gradient. It is also guaranteed to converge to a stationary point if the learning rate is small enough. The disadvantage of steepest descent is that training times are generally longer than for other algorithms. This is especially true when the eigenvalues of the Hessian matrix, for quadratic functions, have a wide range of magnitudes.

Newton's method is generally much faster than steepest descent. For quadratic functions it will locate a stationary point in one iteration. One disadvantage is that it requires calculation and storage of the Hessian matrix, as well as its inverse. In addition, the convergence properties of Newton's method are quite complex. In Chapter 12 we will introduce a modification of Newton's method that overcomes some of the disadvantages of the standard algorithm.

The conjugate gradient algorithm is something of a compromise between steepest descent and Newton's method. It will locate the minimum of a quadratic function in a finite number of iterations, but it does not require calculation and storage of the Hessian matrix. It is well suited to problems with large numbers of parameters, where it is impractical to compute and store the Hessian.

In later chapters we will apply each of these optimization algorithms to the training of neural networks. In Chapter 10 we will demonstrate how an approximate steepest descent algorithm, Widrow-Hoff learning, can be used to train linear networks. In Chapter 11 we generalize Widrow-Hoff learning to train multilayer networks. In Chapter 12 the conjugate gradient algorithm, and a variation of Newton's method, are used to speed up the training of multilayer networks.

Further Reading

[Batt92] R. Battiti, "First and Second Order Methods for Learning: Between Steepest Descent and Newton's Method," *Neural Computation*, Vol. 4, No. 2, pp. 141-166, 1992.

This article reviews the latest developments in unconstrained optimization using first and second derivatives. The techniques discussed are those that are most suitable for neural network applications.

[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.

[Gill81] P. E. Gill, W. Murray and M. H. Wright, *Practical Optimization*, New York: Academic Press, 1981.

As the title implies, this text emphasizes the practical implementation of optimization algorithms. It provides motivation for the optimization methods, as well as details of implementation that affect algorithm performance.

[Himm72] D. M. Himmelblau, *Applied Nonlinear Programming*, New York: McGraw-Hill, 1972.

This is a comprehensive text on nonlinear optimization. It covers both constrained and unconstrained optimization problems. The text is very complete, with many examples worked out in detail.

[Scal85] L. E. Scales, *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

A very readable text describing the major optimization algorithms, this text emphasizes methods of optimization rather than existence theorems and proofs of convergence. Algorithms are presented with intuitive explanations, along with illustrative figures and examples. Pseudo-code is presented for most algorithms.

Exercises

E9.1 In Problem P9.1 we found the maximum stable learning rate for the steepest descent algorithm when applied to a particular quadratic function. Will the algorithm always diverge when a larger learning rate is used, or are there any conditions for which the algorithm will still converge?

E9.2 We want to find the minimum of the following function:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix} 6 & -2 \\ -2 & 6 \end{bmatrix}\mathbf{x} + \begin{bmatrix} -1 & -1 \end{bmatrix}\mathbf{x}.$$

i. Sketch a contour plot of this function.

ii. Sketch the trajectory of the steepest descent algorithm on the contour plot of part (i), if the initial guess is $\mathbf{x}_0 = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$. Assume a very small learning rate is used.

iii. Perform two iterations of steepest descent with learning rate $\alpha = 0.1$.

iv. What is the maximum stable learning rate?

v. What is the maximum stable learning rate for the initial guess given in part (ii)? (See Exercise E9.1.)



vi. Write a MATLAB M-file to implement the steepest descent algorithm for this problem, and use it to check your answers to parts (i). through (v).

E9.3 For the quadratic function

$$F(\mathbf{x}) = x_1^2 + 2x_2^2,$$

i. Find the minimum of the function along the line

$$\mathbf{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \alpha\begin{bmatrix} -1 \\ -2 \end{bmatrix}.$$

ii. Verify that the gradient of $F(\mathbf{x})$ at the minimum point from part (i) is orthogonal to the line along which the minimization occurred.



E9.4 For the functions given in Exercise E8.3 perform two iterations of the steepest descent algorithm with linear minimization, starting from the initial guess $\mathbf{x}_0 = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$. Write MATLAB M-files to check your answer.

E9.5 Consider the following function:

$$F(\mathbf{x}) = [1 + (x_1 + x_2 - 5)^2][1 + (3x_1 - 2x_2)^2].$$

i. Perform one iteration of Newton's method, starting from the initial guess $\mathbf{x}_0 = \begin{bmatrix} 10 & 10 \end{bmatrix}^T$.

ii. Repeat part (i), starting from the initial guess $\mathbf{x}_0 = \begin{bmatrix} 2 & 2 \end{bmatrix}^T$.

iii. Find the minimum of the function, and compare with your results from the previous two parts.

E9.6 Consider the following quadratic function

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 3 & 2 \\ 2 & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 4 & 4 \end{bmatrix} \mathbf{x}$$

i. Sketch the contour plot for $F(\mathbf{x})$. Show all work.

ii. Take one iteration of Newton's method from the initial guess $\mathbf{x}_0 = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$.

iii. In part (ii), did you reach the minimum of $F(\mathbf{x})$? Explain.

E9.7 Consider the function

$$F(\mathbf{x}) = (x_1 + x_2)^4 + 2(x_2 - 1)^2$$

i. Find the second-order Taylor series approximation of this function about the point $\mathbf{x}_0 = \begin{bmatrix} -1 & 1 \end{bmatrix}^T$.

ii. Is this point a minimum point? Does it satisfy the first and second order conditions?

iii. Perform one iteration of Newton's method from the initial guess $\mathbf{x}_0 = \begin{bmatrix} 0.5 & 0 \end{bmatrix}^T$.

E9.8 Consider the following quadratic function:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix} 7 & -9 \\ -9 & -17 \end{bmatrix}\mathbf{x} + \begin{bmatrix} 16 & 8 \end{bmatrix}\mathbf{x}$$

i. Sketch the contour plot for this function.

ii. Take one step of Newton's method from the initial guess $\mathbf{x}_0 = \begin{bmatrix} 2 & 2 \end{bmatrix}^T$.

iii. Did you reach the minimum of the function after the Newton step of part (ii)? Explain.

iv. From the initial guess in part ii, trace the path of steepest descent, with very small learning rate, on your contour plot from part (i). Explain how you determined the path. Will steepest descent eventually converge to the same result you found in part (ii)? Explain.

E9.9 Consider the following function:

$$F(\mathbf{x}) = (1 + x_1 + x_2)^2 + \frac{1}{4}x_1^4.$$

i. Find the quadratic approximation to $F(\mathbf{x})$ about the point $\mathbf{x}_0 = \begin{bmatrix} 2 & 2 \end{bmatrix}^T$.

ii. Sketch the contour plot of the quadratic approximation in part i.

iii. Perform one iteration of Newton's method on the function $F(\mathbf{x})$ from the initial condition $\mathbf{x}_0$ given in part (i). Sketch the path from $\mathbf{x}_0$ to $\mathbf{x}_1$ on your contour plot from part (ii).

iv. Is the $\mathbf{x}_1$ in part iii. a strong minimum of the quadratic approximation? Is it a strong minimum of the original function $F(\mathbf{x})$? Explain.

v. Will Newton's method always converge to a strong minimum of $F(\mathbf{x})$, given enough iterations? Will it always converge to a strong minimum of the quadratic approximation of $F(\mathbf{x})$? Explain your answers in detail.



E9.10 Recall the function presented in Exercise E8.5. Write MATLAB M-files to implement the steepest descent algorithm and Newton's method for that function. Test the performance of the algorithms for various initial guesses.

E9.11 Repeat Exercise E9.4 using the conjugate gradient algorithm. Use each of the three methods (Eq. (9.61)–Eq. (9.63)) at least once.

E9.12 Prove or disprove the following statement:

If $\mathbf{p}_1$ is conjugate to $\mathbf{p}_2$ and $\mathbf{p}_2$ is conjugate to $\mathbf{p}_3$, then $\mathbf{p}_1$ is conjugate to $\mathbf{p}_3$.

10 Widrow-Hoff Learning

Objectives	10-1
Theory and Examples	10-2
ADALINE Network	10-2
Single ADALINE	10-3
Mean Square Error	10-4
LMS Algorithm	10-7
Analysis of Convergence	10-9
Adaptive Filtering	10-13
Adaptive Noise Cancellation	10-15
Echo Cancellation	10-21
Summary of Results	10-22
Solved Problems	10-24
Epilogue	10-40
Further Reading	10-41
Exercises	10-42

Objectives

In the previous two chapters we laid the foundation for *performance learning*, in which a network is trained to optimize its performance. In this chapter we apply the principles of performance learning to a single-layer linear neural network.

Widrow-Hoff learning is an approximate steepest descent algorithm, in which the performance index is mean square error. This algorithm is important to our discussion for two reasons. First, it is widely used today in many signal processing applications, several of which we will discuss in this chapter. In addition, it is the precursor to the backpropagation algorithm for multilayer networks, which is presented in Chapter 11

Theory and Examples

Bernard Widrow began working in neural networks in the late 1950s, at about the same time that Frank Rosenblatt developed the perceptron learning rule. In 1960 Widrow, and his graduate student Marcian Hoff, introduced the ADALINE (ADAptive LInear NEuron) network, and a learning rule which they called the LMS (Least Mean Square) algorithm [WiHo60].

Their ADALINE network is very similar to the perceptron, except that its transfer function is linear, instead of hard-limiting. Both the ADALINE and the perceptron suffer from the same inherent limitation: they can only solve linearly separable problems (recall our discussion in Chapters 3 and 4). The LMS algorithm, however, is more powerful than the perceptron learning rule. While the perceptron rule is guaranteed to converge to a solution that correctly categorizes the training patterns, the resulting network can be sensitive to noise, since patterns often lie close to the decision boundaries. The LMS algorithm minimizes mean square error, and therefore tries to move the decision boundaries as far from the training patterns as possible.

The LMS algorithm has found many more practical uses than the perceptron learning rule. This is especially true in the area of digital signal processing. For example, most long distance phone lines use ADALINE networks for echo cancellation. We will discuss these applications in detail later in the chapter.

Because of the great success of the LMS algorithm in signal processing applications, and because of the lack of success in adapting the algorithm to multilayer networks, Widrow stopped work on neural networks in the early 1960s and began to work full time on adaptive signal processing. He returned to the neural network field in the 1980s and began research on the use of neural networks in adaptive control, using temporal backpropagation, a descendant of his original LMS algorithm.

ADALINE Network

The ADALINE network is shown in Figure 10.1. Notice that it has the same basic structure as the perceptron network we discussed in Chapter 4. The only difference is that it has a linear transfer function.

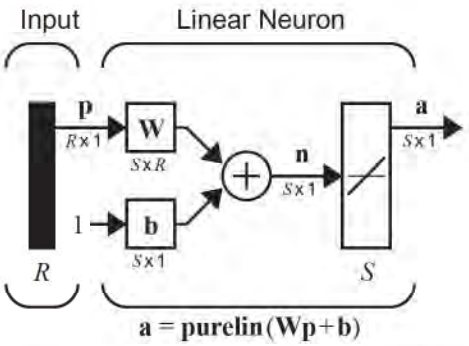


Figure 10.1 ADALINE Network

The output of the network is given by

$$\mathbf{a} = \mathbf{purelin}(\mathbf{Wp} + \mathbf{b}) = \mathbf{Wp} + \mathbf{b}. \tag{10.1}$$

Recall from our discussion of the perceptron network that the ith element of the network output vector can be written

$$a_i = purelin(n_i) = purelin({}_i\mathbf{w}^T\mathbf{p} + b_i) = {}_i\mathbf{w}^T\mathbf{p} + b_i, \tag{10.2}$$

where ${}_i\mathbf{w}$ is made up of the elements of the ith row of $\mathbf{W}$:

$${}_i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix} \tag{10.3}$$

Single ADALINE

To simplify our discussion, let's consider a single ADALINE with two inputs. The diagram for this network is shown in Figure 10.2.

The output of the network is given by

$$\begin{aligned} a &= purelin(n) = purelin({}_1\mathbf{w}^T\mathbf{p} + b) = {}_1\mathbf{w}^T\mathbf{p} + b \\ &= {}_1\mathbf{w}^T\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b. \end{aligned} \tag{10.4}$$

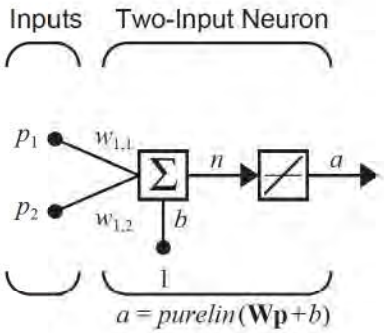


Figure 10.2 Two-Input Linear Neuron

You may recall from Chapter 4 that the perceptron has a *decision boundary*, which is determined by the input vectors for which the net input n is zero. Now, does the ADALINE also have such a boundary? Clearly it does. If we set $n = 0$ then ${}_1\mathbf{w}^T\mathbf{p} + b = 0$ specifies such a line, as shown in Figure 10.3.

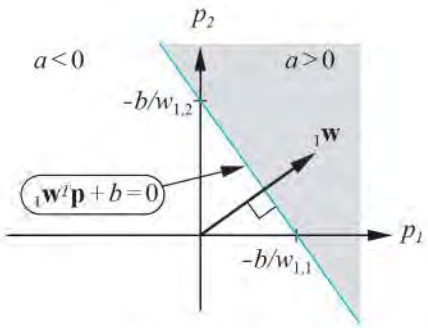


Figure 10.3 Decision Boundary for Two-Input ADALINE

The neuron output is greater than 0 in the gray area. In the white area the output is less than zero. Now, what does this imply about the ADALINE? It says that the ADALINE can be used to classify objects into two categories. However, it can do so only if the objects are linearly separable. Thus, in this respect, the ADALINE has the same limitation as the perceptron.

Mean Square Error

Now that we have examined the characteristics of the ADALINE network, we are ready to begin our development of the LMS algorithm. As with the perceptron rule, the LMS algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of proper network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}, \tag{10.5}$$

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 LMS algorithm will adjust the weights and biases of the ADALINE in order to minimize the mean square error, where the error is the difference between the target output and the network output. In this section we want to discuss this performance index. We will consider first the single-neuron case.

To simply our development, we will lump all of the parameters we are adjusting, including the bias, into one vector:

$$\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w} \\ b \end{bmatrix}. \tag{10.6}$$

Similarly, we include the bias input "1" as a component of the input vector

$$\mathbf{z} = \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix}. \tag{10.7}$$

Now the network output, which we usually write in the form

$$a = {}_1\mathbf{w}^T\mathbf{p} + b, \tag{10.8}$$

can be written as

$$a = \mathbf{x}^T\mathbf{z}. \tag{10.9}$$

Mean Square Error

This allows us to conveniently write out an expression for the ADALINE network *mean square error*:

$$F(\mathbf{x}) = E[e^2] = E[(t-a)^2] = E[(t-\mathbf{x}^T\mathbf{z})^2], \tag{10.10}$$

where the expectation is taken over all sets of input/target pairs. (Here we use $E[\ ]$ to denote expected value. We use a generalized definition of expectation, which becomes a time-average for deterministic signals. See [WiSt85].) We can expand this expression as follows:

$$\begin{aligned} F(\mathbf{x}) &= E[t^2 - 2t\mathbf{x}^T\mathbf{z} + \mathbf{x}^T\mathbf{z}\mathbf{z}^T\mathbf{x}] \\ &= E[t^2] - 2\mathbf{x}^T E[t\mathbf{z}] + \mathbf{x}^T E[\mathbf{z}\mathbf{z}^T]\mathbf{x}. \end{aligned} \tag{10.11}$$

This can be written in the following convenient form:

$$F(\mathbf{x}) = c - 2\mathbf{x}^T\mathbf{h} + \mathbf{x}^T\mathbf{R}\mathbf{x}, \tag{10.12}$$

where

$$c = E[t^2], \ \mathbf{h} = E[t\mathbf{z}] \text{ and } \mathbf{R} = E[\mathbf{z}\mathbf{z}^T]. \tag{10.13}$$

Correlation Matrix

Here the vector $\mathbf{h}$ gives the cross-correlation between the input vector and its associated target, while $\mathbf{R}$ is the input *correlation matrix*. The diagonal elements of this matrix are equal to the mean square values of the elements of the input vectors.

Take a close look at Eq. (10.12), and compare it with the general form of the quadratic function given in Eq. (8.35) and repeated here:

$$F(\mathbf{x}) = c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}. \tag{10.14}$$

We can see that the mean square error performance index for the ADALINE network is a quadratic function, where

$$\mathbf{d} = -2\mathbf{h} \text{ and } \mathbf{A} = 2\mathbf{R}. \tag{10.15}$$

This is a very important result, because we know from Chapter 8 that the characteristics of the quadratic function depend primarily on the Hessian matrix $\mathbf{A}$. For example, if the eigenvalues of the Hessian are all positive, then the function will have one unique global minimum.

In this case the Hessian matrix is twice the correlation matrix $\mathbf{R}$, and it can be shown that all correlation matrices are either positive definite or positive semidefinite, which means that they can never have negative eigenvalues. We are left with two possibilities. If the correlation matrix has only positive eigenvalues, the performance index will have one unique global minimum (see Figure 8.7). If the correlation matrix has some zero eigenvalues, the performance index will either have a weak minimum (see Figure 8.9) or no minimum (see Problem P8.7), depending on the vector $\mathbf{d} = -2\mathbf{h}$.

Now let's locate the stationary point of the performance index. From our previous discussion of quadratic functions we know that the gradient is

$$\nabla F(\mathbf{x}) = \nabla\left(c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}\right) = \mathbf{d} + \mathbf{A}\mathbf{x} = -2\mathbf{h} + 2\mathbf{R}\mathbf{x}. \tag{10.16}$$

The stationary point of $F(\mathbf{x})$ can be found by setting the gradient equal to zero:

$$-2\mathbf{h} + 2\mathbf{R}\mathbf{x} = 0. \tag{10.17}$$

Therefore, if the correlation matrix is positive definite there will be a unique stationary point, which will be a strong minimum:

$$\mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h}. \tag{10.18}$$

It is worth noting here that the existence of a unique solution depends only on the correlation matrix $\mathbf{R}$. Therefore the characteristics of the input vectors determine whether or not a unique solution exists.

LMS Algorithm

Now that we have analyzed our performance index, the next step is to design an algorithm to locate the minimum point. If we could calculate the statistical quantities $\mathbf{h}$ and $\mathbf{R}$, we could find the minimum point directly from Eq. (10.18). If we did not want to calculate the inverse of $\mathbf{R}$, we could use the steepest descent algorithm, with the gradient calculated from Eq. (10.16). In general, however, it is not desirable or convenient to calculate $\mathbf{h}$ and $\mathbf{R}$. For this reason we will use an approximate steepest descent algorithm, in which we use an estimated gradient.

The key insight of Widrow and Hoff was that they could estimate the mean square error $F(\mathbf{x})$ by

$$\hat{F}(\mathbf{x}) = (t(k) - a(k))^2 = e^2(k), \tag{10.19}$$

where the expectation of the squared error has been replaced by the squared error at iteration k. Then, at each iteration we have a gradient estimate of the form:

$$\hat{\nabla} F(\mathbf{x}) = \nabla e^2(k). \tag{10.20}$$

Stochastic Gradient

This is sometimes referred to as the *stochastic gradient*. When this is used in a gradient descent algorithm, it is referred to as "on-line" or incremental learning, since the weights are updated as each input is presented to the network.

The first R elements of $\nabla e^2(k)$ are derivatives with respect to the network weights, while the $(R+1)$st element is the derivative with respect to the bias. Thus we have

$$[\nabla e^2(k)]_j = \frac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k)\frac{\partial e(k)}{\partial w_{1,j}} \text{ for } j = 1, 2, \ldots, R, \tag{10.21}$$

and

$$[\nabla e^2(k)]_{R+1} = \frac{\partial e^2(k)}{\partial b} = 2e(k)\frac{\partial e(k)}{\partial b}. \tag{10.22}$$

Now consider the partial derivative terms at the ends of these equations. First evaluate the partial derivative of $e(k)$ with respect to the weight $w_{1,j}$:

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial [t(k) - a(k)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}[t(k) - ({}_1\mathbf{w}^T\mathbf{p}(k) + b)]$$

$$= \frac{\partial}{\partial w_{1,j}}\left[t(k) - \left(\sum_{i=1}^{R} w_{1,i}p_i(k) + b\right)\right] \tag{10.23}$$

where $p_i(k)$ is the ith element of the input vector at the kth iteration. This simplifies to

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k). \tag{10.24}$$

In a similar way we can obtain the final element of the gradient:

$$\frac{\partial e(k)}{\partial b} = -1. \tag{10.25}$$

Note that $p_j(k)$ and 1 are the elements of the input vector $\mathbf{z}$, so the gradient of the squared error at iteration k can be written

$$\hat{\nabla} F(\mathbf{x}) = \nabla e^2(k) = -2e(k)\mathbf{z}(k). \tag{10.26}$$

Now we can see the beauty of approximating the mean square error by the single error at iteration k, as in Eq. (10.19). To calculate this approximate gradient we need only multiply the error times the input.

This approximation to $\nabla F(\mathbf{x})$ can now be used in the steepest descent algorithm. From Eq. (9.10) the steepest descent algorithm, with constant learning rate, is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}_k}. \tag{10.27}$$

If we substitute $\hat{\nabla} F(\mathbf{x})$, from Eq. (10.26), for $\nabla F(\mathbf{x})$ we find

$$\mathbf{x}_{k+1} = \mathbf{x}_k + 2\alpha e(k)\mathbf{z}(k), \tag{10.28}$$

or

$${}_1\mathbf{w}(k+1) = {}_1\mathbf{w}(k) + 2\alpha e(k)\mathbf{p}(k), \tag{10.29}$$

and

$$b(k+1) = b(k) + 2\alpha e(k). \tag{10.30}$$

These last two equations make up the least mean square (LMS) algorithm. This is also referred to as the delta rule or the Widrow-Hoff learning algorithm.

The preceding results can be modified to handle the case where we have multiple outputs, and therefore multiple neurons, as in Figure 10.1. To update the ith row of the weight matrix use

$$ {}_i\mathbf{w}(k+1) = {}_i\mathbf{w}(k) + 2\alpha e_i(k)\mathbf{p}(k), \tag{10.31} $$

where $e_i(k)$ is the ith element of the error at iteration k. To update the ith element of the bias we use

$$ b_i(k+1) = b_i(k) + 2\alpha e_i(k). \tag{10.32} $$

LMS Algorithm

The *LMS algorithm* can be written conveniently in matrix notation:

$$ \mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha \mathbf{e}(k)\mathbf{p}^T(k), \tag{10.33} $$

and

$$ \mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha \mathbf{e}(k). \tag{10.34} $$

Note that the error $\mathbf{e}$ and the bias $\mathbf{b}$ are now vectors.

Analysis of Convergence

The stability of the steepest descent algorithm was investigated in Chapter 9. There we found that the maximum stable learning rate for quadratic functions is $\alpha < 2/\lambda_{max}$, where λ_{max} is the largest eigenvalue of the Hessian matrix. Now we want to investigate the convergence of the LMS algorithm, which is approximate steepest descent. We will find that the result is the same.

To begin, note that in the LMS algorithm, Eq. (10.28), $\mathbf{x}_k$ is a function only of $\mathbf{z}(k-1), \mathbf{z}(k-2), \ldots, \mathbf{z}(0)$. If we assume that successive input vectors are statistically independent, then $\mathbf{x}_k$ is independent of $\mathbf{z}(k)$. We will show in the following development that for stationary input processes meeting this condition, the expected value of the weight vector will converge to

$$ \mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h}. \tag{10.35} $$

This is the minimum mean square error $\{E[e_k^2]\}$ solution, as we saw in Eq. (10.18).

Recall the LMS algorithm (Eq. (10.28)):

$$ \mathbf{x}_{k+1} = \mathbf{x}_k + 2\alpha e(k)\mathbf{z}(k). \tag{10.36} $$

Now take the expectation of both sides:

$$E[\mathbf{x}_{k+1}] = E[\mathbf{x}_k] + 2\alpha E[e(k)\mathbf{z}(k)]. \tag{10.37}$$

Substitute $t(k) - \mathbf{x}_k^T\mathbf{z}(k)$ for the error to give

$$E[\mathbf{x}_{k+1}] = E[\mathbf{x}_k] + 2\alpha\{E[t(k)\mathbf{z}(k)] - E[(\mathbf{x}_k^T\mathbf{z}(k))\mathbf{z}(k)]\}. \tag{10.38}$$

Finally, substitute $\mathbf{z}^T(k)\mathbf{x}_k$ for $\mathbf{x}_k^T\mathbf{z}(k)$ and rearrange terms to give

$$E[\mathbf{x}_{k+1}] = E[\mathbf{x}_k] + 2\alpha\{E[t_k\mathbf{z}(k)] - E[(\mathbf{z}(k)\mathbf{z}^T(k))\mathbf{x}_k]\}. \tag{10.39}$$

Since $\mathbf{x}_k$ is independent of $\mathbf{z}(k)$:

$$E[\mathbf{x}_{k+1}] = E[\mathbf{x}_k] + 2\alpha\{\mathbf{h} - \mathbf{R}E[\mathbf{x}_k]\}. \tag{10.40}$$

This can be written as

$$E[\mathbf{x}_{k+1}] = [\mathbf{I} - 2\alpha\mathbf{R}]E[\mathbf{x}_k] + 2\alpha\mathbf{h}. \tag{10.41}$$

This dynamic system will be stable if all of the eigenvalues of $[\mathbf{I} - 2\alpha\mathbf{R}]$ fall inside the unit circle (see [Brog91]). Recall from Chapter 9 that the eigenvalues of $[\mathbf{I} - 2\alpha\mathbf{R}]$ will be $1 - 2\alpha\lambda_i$, where the λ_i are the eigenvalues of $\mathbf{R}$. Therefore, the system will be stable if

$$1 - 2\alpha\lambda_i > -1. \tag{10.42}$$

Since $\lambda_i > 0$, $1 - 2\alpha\lambda_i$ is always less than 1. The condition on stability is therefore

$$\alpha < 1/\lambda_i \quad \text{for all } i, \tag{10.43}$$

or

$$0 < \alpha < 1/\lambda_{max}. \tag{10.44}$$

Note that this condition is equivalent to the condition we derived in Chapter 9 for the steepest descent algorithm, although in that case we were using the eigenvalues of the Hessian matrix $\mathbf{A}$. Now we are using the eigenvalues of the input correlation matrix $\mathbf{R}$. (Recall that $\mathbf{A} = 2\mathbf{R}$.)

If this condition on stability is satisfied, the steady state solution is

$$E[\mathbf{x}_{ss}] = [\mathbf{I} - 2\alpha\mathbf{R}]E[\mathbf{x}_{ss}] + 2\alpha\mathbf{h}, \tag{10.45}$$

or

$$E[\mathbf{x}_{ss}] = \mathbf{R}^{-1}\mathbf{h} = \mathbf{x}^*. \tag{10.46}$$

Thus the LMS solution, obtained by applying one input vector at a time, is the same as the minimum mean square error solution of Eq. (10.18).



To test the ADALINE network and the LMS algorithm consider again the apple/orange recognition problem originally discussed in Chapter 3. For simplicity we will assume that the ADALINE network has a zero bias.

The LMS weight update algorithm of Eq. (10.29) will be used to calculate the new weights at each step in the network training:

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha e(k)\mathbf{p}^T(k). \tag{10.47}$$

First let's compute the maximum stable learning rate α. We can get such a value by finding the eigenvalues of the input correlation matrix. Recall that the orange and apple vectors and their associated targets are

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, t_1 = \begin{bmatrix} -1 \end{bmatrix}\right\} \qquad \left\{\mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, t_2 = \begin{bmatrix} 1 \end{bmatrix}\right\}. \tag{10.48}$$

If we assume that the input vectors are generated randomly with equal probability, we can compute the input correlation matrix:

$$\mathbf{R} = E[\mathbf{p}\mathbf{p}^T] = \frac{1}{2}\mathbf{p}_1\mathbf{p}_1^T + \frac{1}{2}\mathbf{p}_2\mathbf{p}_2^T$$

$$= \frac{1}{2}\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}\begin{bmatrix} 1 & -1 & -1 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}\begin{bmatrix} 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}. \tag{10.49}$$

The eigenvalues of $\mathbf{R}$ are

$$\lambda_1 = 1.0, \qquad \lambda_2 = 0.0, \qquad \lambda_3 = 2.0. \tag{10.50}$$

Thus, the maximum stable learning rate is

$$\alpha < \frac{1}{\lambda_{max}} = \frac{1}{2.0} = 0.5. \tag{10.51}$$

To be conservative we will pick $\alpha = 0.2$. (Note that in practical applications it might not be practical to calculate $\mathbf{R}$, and α could be selected by trial and error. Other techniques for choosing α are given in [WiSt85].)

We will start, arbitrarily, with all the weights set to zero, and then will apply inputs $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_1$, $\mathbf{p}_2$, etc., in that order, calculating the new weights after each input is presented. (The presentation of the weights in alternat-

ing order is not necessary. A random sequence would be fine.) Presenting $\mathbf{p}_1$, the orange, and using its target of -1 we get

$$a(0) = \mathbf{W}(0)\mathbf{p}(0) = \mathbf{W}(0)\mathbf{p}_1 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} = 0, \tag{10.52}$$

and

$$e(0) = t(0) - a(0) = t_1 - a(0) = -1 - 0 = -1. \tag{10.53}$$

Now we can calculate the new weight matrix:

$$\mathbf{W}(1) = \mathbf{W}(0) + 2\alpha e(0)\mathbf{p}^T(0)$$

$$= \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} + 2(0.2)(-1)\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}^T = \begin{bmatrix} -0.4 & 0.4 & 0.4 \end{bmatrix}. \tag{10.54}$$

According to plan, we will next present the apple, $\mathbf{p}_2$, and its target of 1:

$$a(1) = \mathbf{W}(1)\mathbf{p}(1) = \mathbf{W}(1)\mathbf{p}_2 = \begin{bmatrix} -0.4 & 0.4 & 0.4 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = -0.4, \tag{10.55}$$

and so the error is

$$e(1) = t(1) - a(1) = t_2 - a(1) = 1 - (-0.4) = 1.4. \tag{10.56}$$

Now we calculate the new weights:

$$\mathbf{W}(2) = \mathbf{W}(1) + 2\alpha e(1)\mathbf{p}^T(1)$$

$$= \begin{bmatrix} -0.4 & 0.4 & 0.4 \end{bmatrix} + 2(0.2)(1.4)\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}^T = \begin{bmatrix} 0.16 & 0.96 & -0.16 \end{bmatrix}. \tag{10.57}$$

Next we present the orange again:

$$a(2) = \mathbf{W}(2)\mathbf{p}(2) = \mathbf{W}(2)\mathbf{p}_1 = \begin{bmatrix} 0.16 & 0.96 & -0.16 \end{bmatrix}\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} = -0.64. \tag{10.58}$$

The error is

$$e(2) = t(2) - a(2) = t_1 - a(2) = -1 - (-0.64) = -0.36 . \tag{10.59}$$

The new weights are

$$\mathbf{W}(3) = \mathbf{W}(2) + 2\alpha e(2)\mathbf{p}^T(2) = \begin{bmatrix} 0.016 & 1.1040 & -0.0160 \end{bmatrix} . \tag{10.60}$$

If we continue this procedure, the algorithm converges to

$$\mathbf{W}(\infty) = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} . \tag{10.61}$$

Compare this result with the result of the perceptron learning rule in Chapter 4. You will notice that the ADALINE has produced the same decision boundary that we designed in Chapter 3 for the apple/orange problem. This boundary falls halfway between the two reference patterns. The perceptron rule did not produce such a boundary. This is because the perceptron rule stops as soon as the patterns are correctly classified, even though some patterns may be close to the boundaries. The LMS algorithm minimizes the mean square error. Therefore it tries to move the decision boundaries as far from the reference patterns as possible.

Adaptive Filtering

As we mentioned at the beginning of this chapter, the ADALINE network has the same major limitation as the perceptron network; it can only solve linearly separable problems. In spite of this, the ADALINE has been much more widely used than the perceptron network. In fact, it is safe to say that it is one of the most widely used neural networks in practical applications. One of the major application areas of the ADALINE has been adaptive filtering, where it is still used extensively. In this section we will demonstrate an adaptive filtering example.

Tapped Delay Line

In order to use the ADALINE network as an adaptive filter, we need to introduce a new building block, the tapped delay line. A *tapped delay line* with R outputs is shown in Figure 10.4.

The input signal enters from the left. At the output of the tapped delay line we have an R-dimensional vector, consisting of the input signal at the current time and at delays of from 1 to $R-1$ time steps.

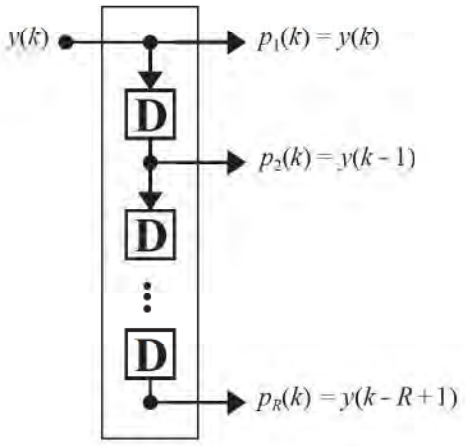


Figure 10.4 Tapped Delay Line

Adaptive Filter

If we combine a tapped delay line with an ADALINE network, we can create an *adaptive filter*, as is shown in Figure 10.5. The output of the filter is given by

$$a(k) = purelin(\mathbf{W}\mathbf{p} + b) = \sum_{i=1}^{R} w_{1,i} y(k - i + 1) + b . \qquad (10.62)$$

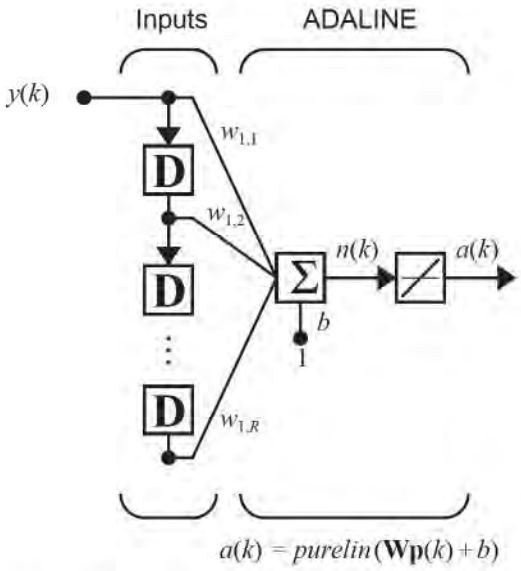


Figure 10.5 Adaptive Filter ADALINE

If you are familiar with digital signal processing, you will recognize the network of Figure 10.5 as a finite impulse response (FIR) filter [WiSt85]. It is beyond the scope of this text to review the field of digital signal processing, but we can demonstrate the usefulness of this adaptive filter through a simple, but practical, example.

Adaptive Noise Cancellation



An adaptive filter can be used in a variety of novel ways. In the following example we will use it for noise cancellation. Take some time to look at this example, for it is a little different from what you might expect. For instance, the output "error" that the network tries to minimize is actually an approximation to the signal we are trying to recover!

Let's suppose that a doctor, in trying to review the electroencephalogram (EEG) of a distracted graduate student, finds that the signal he would like to see has been contaminated by a 60-Hz noise source. He is examining the patient on-line and wants to view the best signal that can be obtained. Figure 10.6 shows how an adaptive filter can be used to remove the contaminating signal.

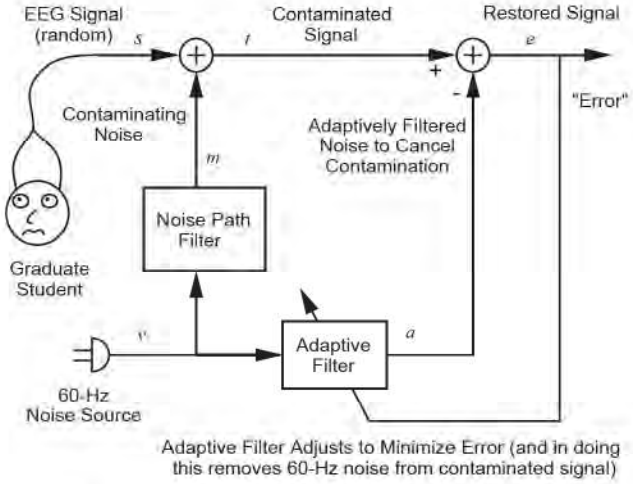


Figure 10.6 Noise Cancellation System

As shown, a sample of the original 60-Hz signal is fed to an adaptive filter, whose elements are adjusted so as to minimize the "error" e. The desired output of the filter is the contaminated EEG signal t. The adaptive filter will do its best to reproduce this contaminated signal, but it only knows about the original noise source, v. Thus, it can only reproduce the part of t that is linearly correlated with v, which is m. In effect, the adaptive filter will attempt to mimic the noise path filter, so that the output of the filter

a will be close to the contaminating noise m. In this way the error e will be close to the original uncontaminated EEG signal s.

In this simple case of a single sine wave noise source, a neuron with two weights and no bias is sufficient to implement the filter. The inputs to the filter are the current and previous values of the noise source. Such a two-input filter can attenuate and phase-shift the noise v in the desired way. The filter is shown in Figure 10.7.

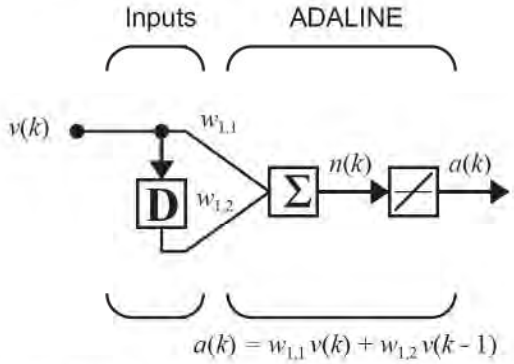


Figure 10.7 Adaptive Filter for Noise Cancellation

We can apply the mathematical relationships developed in the previous sections of this chapter to analyze this system. In order to do so, we will first need to find the input correlation matrix $\mathbf{R}$ and the input/target cross-correlation vector $\mathbf{h}$:

$$\mathbf{R} = [\mathbf{z}\mathbf{z}^T] \text{ and } \mathbf{h} = E[t\mathbf{z}]. \tag{10.63}$$

In our case the input vector is given by the current and previous values of the noise source:

$$\mathbf{z}(k) = \begin{bmatrix} v(k) \\ v(k-1) \end{bmatrix}, \tag{10.64}$$

while the target is the sum of the current signal and filtered noise:

$$t(k) = s(k) + m(k). \tag{10.65}$$

Now expand the expressions for $\mathbf{R}$ and $\mathbf{h}$ to give

$$\mathbf{R} = \begin{bmatrix} E[v^2(k)] & E[v(k)v(k-1)] \\ E[v(k-1)v(k)] & E[v^2(k-1)] \end{bmatrix}, \tag{10.66}$$

and

$$\mathbf{h} = \begin{bmatrix} E[(s(k)+m(k))v(k)] \\ E[(s(k)+m(k))v(k-1)] \end{bmatrix}. \tag{10.67}$$

To obtain specific values for these two quantities we must define the noise signal v, the EEG signal s and the filtered noise m. For this exercise we will assume: the EEG signal is a white (uncorrelated from one time step to the next) random signal uniformly distributed between the values -0.2 and +0.2, the noise source (60-Hz sine wave sampled at 180 Hz) is given by

$$v(k) = 1.2 \sin\left(\frac{2\pi k}{3}\right), \tag{10.68}$$

and the filtered noise that contaminates the EEG is the noise source attenuated by a factor of 10 and shifted in phase by $\pi/2$:

$$m(k) = 0.12 \sin\left(\frac{2\pi k}{3} + \frac{\pi}{2}\right). \tag{10.69}$$

Now calculate the elements of the input correlation matrix $\mathbf{R}$:

$$E[v^2(k)] = (1.2)^2 \frac{1}{3} \sum_{k=1}^{3} \left(\sin\left(\frac{2\pi k}{3}\right)\right)^2 = (1.2)^2 0.5 = 0.72, \tag{10.70}$$

$$E[v^2(k-1)] = E[v^2(k)] = 0.72, \tag{10.71}$$

$$\begin{aligned} E[v(k)v(k-1)] &= \frac{1}{3} \sum_{k=1}^{3} \left(1.2 \sin\frac{2\pi k}{3}\right)\left(1.2 \sin\frac{2\pi(k-1)}{3}\right) \\ &= (1.2)^2 0.5 \cos\left(\frac{2\pi}{3}\right) = -0.36 \end{aligned} \tag{10.72}$$

(where we have used some trigonometric identities).

Thus $\mathbf{R}$ is

$$\mathbf{R} = \begin{bmatrix} 0.72 & -0.36 \\ -0.36 & 0.72 \end{bmatrix}. \tag{10.73}$$

The terms of $\mathbf{h}$ can be found in a similar manner. We will consider the top term in Eq. (10.67) first:

$$E[(s(k)+m(k))v(k)] = E[s(k)v(k)] + E[m(k)v(k)]. \tag{10.74}$$

Here the first term on the right is zero because $s(k)$ and $v(k)$ are independent and zero mean. The second term is also zero:

$$E[m(k)v(k)] = \frac{1}{3}\sum_{k=1}^{3}\left(0.12\ \sin\left(\frac{2\pi k}{3}+\frac{\pi}{2}\right)\right)\left(1.2\sin\frac{2\pi k}{3}\right) = 0 \qquad (10.75)$$

Thus, the first element of $\mathbf{h}$ is zero.

Next consider the second element of $\mathbf{h}$:

$$E[(s(k)+m(k))v(k-1)] = E[s(k)v(k-1)] + E[m(k)v(k-1)]\ . \qquad (10.76)$$

As with the first element of $\mathbf{h}$, the first term on the right is zero because $s(k)$ and $v(k-1)$ are independent and zero mean. The second term is evaluated as follows:

$$E[m(k)v(k-1)] = \frac{1}{3}\sum_{k=1}^{3}\left(0.12\ \sin\left(\frac{2\pi k}{3}+\frac{\pi}{2}\right)\right)\left(1.2\ \sin\frac{2\pi(k-1)}{3}\right)$$

$$= -0.0624\ . \qquad (10.77)$$

Thus, $\mathbf{h}$ is

$$\mathbf{h} = \begin{bmatrix} 0 \\ -0.0624 \end{bmatrix}. \qquad (10.78)$$

The minimum mean square error solution for the weights is given by Eq. (10.18):

$$\mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h} = \begin{bmatrix} 0.72 & -0.36 \\ -0.36 & 0.72 \end{bmatrix}^{-1}\begin{bmatrix} 0 \\ -0.0624 \end{bmatrix} = \begin{bmatrix} -0.0578 \\ -0.1156 \end{bmatrix}. \qquad (10.79)$$

Now, what kind of error will we have at the minimum solution? To find this error recall Eq. (10.12):

$$F(\mathbf{x}) = c - 2\mathbf{x}^T\mathbf{h} + \mathbf{x}^T\mathbf{R}\mathbf{x}\ . \qquad (10.80)$$

We have just found $\mathbf{x}^*$, $\mathbf{h}$ and $\mathbf{R}$, so we only need to find c:

$$c = E[t^2(k)] = E[(s(k)+m(k))^2]$$

$$= E[s^2(k)] + 2E[s(k)m(k)] + E[m^2(k)]\ . \qquad (10.81)$$

The middle term is zero because $s(k)$ and $m(k)$ are independent and zero mean. The first term, the mean squared value of the random signal, can be calculated as follows:

$$E[s^2(k)] = \frac{1}{0.4}\int_{-0.2}^{0.2} s^2 ds = \frac{1}{3(0.4)} s^3\Big|_{-0.2}^{0.2} = 0.0133\,. \tag{10.82}$$

The mean square value of the filtered noise is

$$E[m^2(k)] = \frac{1}{3}\sum_{k=1}^{3}\left\{0.12\,\sin\left(\frac{2\pi}{3}+\frac{\pi}{2}\right)\right\}^2 = 0.0072\,, \tag{10.83}$$

so that

$$c = 0.0133 + 0.0072 = 0.0205\,. \tag{10.84}$$

Substituting $\mathbf{x}^*$, $\mathbf{h}$ and $\mathbf{R}$ into Eq. (10.80), we find that the minimum mean square error is

$$F(\mathbf{x}^*) = 0.0205 - 2(0.0072) + 0.0072 = 0.0133\,. \tag{10.85}$$

The minimum mean square error is the same as the mean square value of the EEG signal. This is what we expected, since the "error" of this adaptive noise canceller is in fact the reconstructed EEG signal.

Figure 10.8 illustrates the trajectory of the LMS algorithm in the weight space with learning rate $\alpha = 0.1$. The system weights $w_{1,1}$ and $w_{1,2}$ in this simulation were initialized arbitrarily to 0 and -2, respectively. You can see from this figure that the LMS trajectory looks like a noisy version of steepest descent.

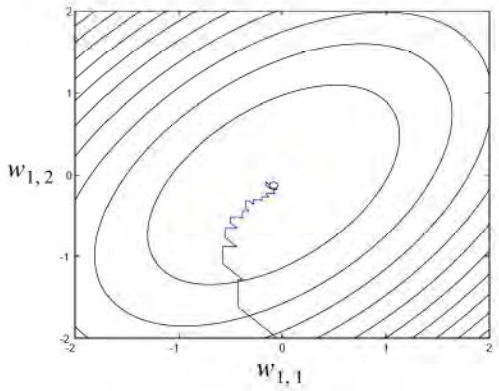


Figure 10.8 LMS Trajectory for $\alpha = 0.1$

Note that the contours in this figure reflect the fact that the eigenvalues and eigenvectors of the Hessian matrix ($\mathbf{A} = 2\mathbf{R}$) are

$$\lambda_1 = 2.16, \mathbf{z}_1 = \begin{bmatrix} -0.7071 \\ 0.7071 \end{bmatrix}, \lambda_2 = 0.72, \mathbf{z}_2 = \begin{bmatrix} -0.7071 \\ -0.7071 \end{bmatrix}. \quad (10.86)$$

(Refer back to our discussion in Chapter 8 on the eigensystem of the Hessian matrix.)

If the learning rate is decreased, the LMS trajectory is smoother than that shown in Figure 10.8, but the learning proceeds more slowly. If the learning rate is increased, the trajectory is more jagged and oscillatory. In fact, as noted earlier in this chapter, if the learning rate is increased too much the system does not converge at all. The maximum stable learning rate is $\alpha < 2/2.16 = 0.926$.

In order to judge the performance of our noise canceller, consider Figure 10.9. This figure illustrates how the filter adapts to cancel the noise. The top graph shows the restored and original EEG signals. At first the restored signal is a poor approximation of the original EEG signal. It takes about 0.2 second (with $\alpha = 0.1$) for the filter to adjust to give a reasonable restored signal. The mean square difference between the original and restored signal over the last half of the experiment was 0.002. This compares favorably with the signal mean square value of 0.0133. The difference between the original and restored signal is shown in the lower graph.

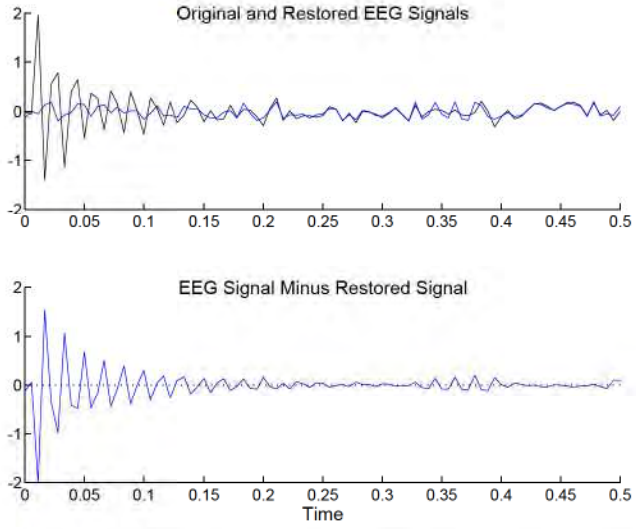


Figure 10.9 Adaptive Filter Cancellation of Contaminating Noise

You might wonder why the error does not go to zero. This is because the LMS algorithm is approximate steepest descent; it uses an estimate of the gradient, not the true gradient, to update the weights. The estimate of the gradient is a noisy version of the true gradient. This will cause the weights to continue to change slightly, even after the mean square error is at the minimum point. You can see this effect in Figure 10.8.



To experiment with the use of this adaptive noise cancellation filter, use the MATLAB® Neural Network Design Demonstration Adaptive Noise Cancellation (**nnd10nc**). *A more complex noise source and actual EEG data are used in the Demonstration Electroencephalogram Noise Cancellation* (**nnd10eeg**).

Echo Cancellation

Another very important practical application of adaptive noise cancellation is echo cancellation. Echoes are common in long distance telephone lines because of impedance mismatch at the "hybrid" device that forms the junction between the long distance line and the customer's local line. You may have experienced this effect on international telephone calls.

Figure 10.10 illustrates how an adaptive noise cancellation filter can be used to reduce these echoes [WiWi85]. At the end of the long distance line the incoming signal is sent to an adaptive filter, as well as to the hybrid device. The target output of the filter is the output of the hybrid. The filter thus tries to cancel the part of the hybrid output that is correlated with the input signal — the echo.

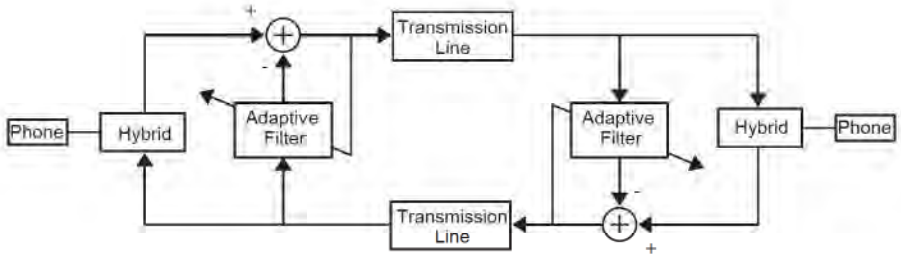
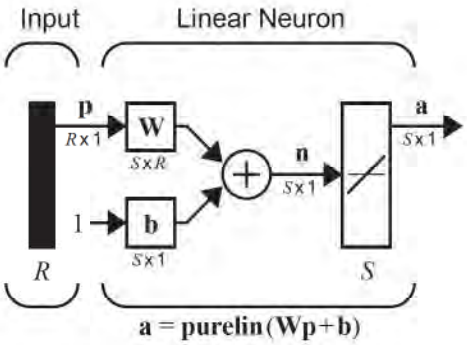


Figure 10.10 Echo Cancellation System

Summary of Results

ADALINE



Mean Square Error

$$F(\mathbf{x}) = E[e^2] = E[(t-a)^2] = E[(t-\mathbf{x}^T\mathbf{z})^2]$$

$$F(\mathbf{x}) = c - 2\mathbf{x}^T\mathbf{h} + \mathbf{x}^T\mathbf{R}\mathbf{x},$$

$$c = E[t^2],\ \mathbf{h} = E[t\mathbf{z}] \text{ and } \mathbf{R} = E[\mathbf{z}\mathbf{z}^T]$$

Unique minimum, if it exists, is $\mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h}$.

Where $\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w} \\ b \end{bmatrix}$ and $\mathbf{z} = \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix}$.

LMS Algorithm

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k)$$

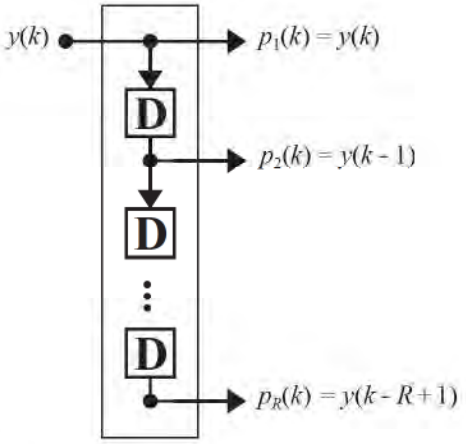
Convergence Point

$$\mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h}$$

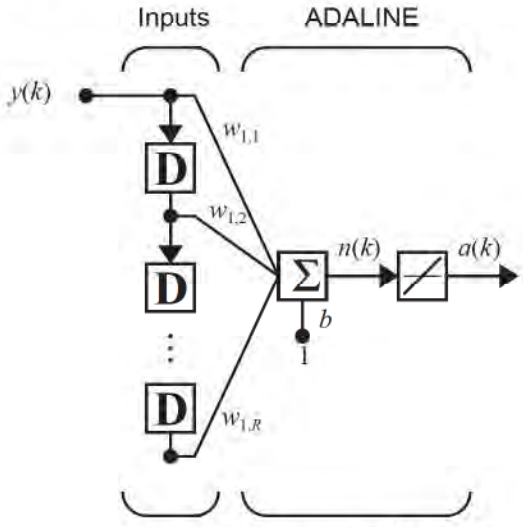
Stable Learning Rate

$0 < \alpha < 1/\lambda_{max}$ where λ_{max} is the maximum eigenvalue of $\mathbf{R}$

Tapped Delay Line



Adaptive Filter ADALINE



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

$$a(k) = purelin(\mathbf{W}\mathbf{p} + b) = \sum_{i=1}^{R} w_{1,i} y(k - i + 1) + b$$

Solved Problems

P10.1 Consider the ADALINE filter in Figure P10.1.

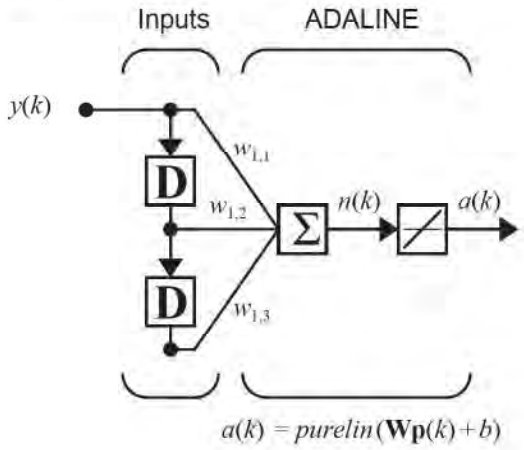


Figure P10.1 ADALINE Filter

Suppose that

$$w_{1,1} = 2, \quad w_{1,2} = -1, \quad w_{1,3} = 3,$$

and the input sequence is

$$\{y(k)\} = \{\ldots, 0, 0, 0, 5, -4, 0, 0, 0, \ldots\}$$

where $y(0) = 5$, $y(1) = -4$, **etc.**

i. **What is the filter output just prior to** $k = 0$**?**

ii. **What is the filter output from** $k = 0$ **to** $k = 5$**?**

iii. **How long does** $y(0)$ **contribute to the output?**

i. Just prior to $k = 0$ three zeros have entered the filter, and the output is zero.

ii. At $k = 0$ the digit "5" has entered the filter, and it will be multiplied by $w_{1,1}$, which has the value 2, so that $a(0) = 10$. This can be viewed as the matrix operation:

$$a(0) = \mathbf{W}\mathbf{p}(0) = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \end{bmatrix} \begin{bmatrix} y(0) \\ y(-1) \\ y(-2) \end{bmatrix} = \begin{bmatrix} 2 & -1 & 3 \end{bmatrix} \begin{bmatrix} 5 \\ 0 \\ 0 \end{bmatrix} = 10 .$$

Similarly, one can calculate the next outputs as

$$a(1) = \mathbf{W}\mathbf{p}(1) = \begin{bmatrix} 2 & -1 & 3 \end{bmatrix} \begin{bmatrix} -4 \\ 5 \\ 0 \end{bmatrix} = -13$$

$$a(2) = \mathbf{W}\mathbf{p}(2) = \begin{bmatrix} 2 & -1 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ -4 \\ 5 \end{bmatrix} = 19$$

$$a(3) = \mathbf{W}\mathbf{p}(3) = \begin{bmatrix} 2 & -1 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -4 \end{bmatrix} = -12 , \; a(4) = \mathbf{W}\mathbf{p}(4) = \begin{bmatrix} 2 & -1 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = 0 .$$

All remaining outputs will be zero.

iii. The effects of $y(0)$ last from $k = 0$ through $k = 2$, so it will have an influence for three time intervals. This corresponds to the length of the impulse response of this filter.

P10.2 Suppose that we want to design an ADALINE network to distinguish between various categories of input vectors. Let us first try the categories listed below:

$$\textbf{Category I:} \quad \mathbf{p}_1 = \begin{bmatrix} 1 & 1 \end{bmatrix}^T \textbf{ and } \mathbf{p}_2 = \begin{bmatrix} -1 & -1 \end{bmatrix}^T$$

$$\textbf{Category II:} \quad \mathbf{p}_3 = \begin{bmatrix} 2 & 2 \end{bmatrix}^T .$$

i. Can an ADALINE network be designed to make such a distinction?

ii. If the answer to part (i) is yes, what set of weights and bias might be used?

Next consider a different set of categories.

$$\textbf{Category III:} \quad \mathbf{p}_1 = \begin{bmatrix} 1 & 1 \end{bmatrix}^T \textbf{ and } \mathbf{p}_2 = \begin{bmatrix} 1 & -1 \end{bmatrix}^T$$

Category IV: $\mathbf{p}_3 = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$.

iii. Can an ADALINE network be designed to make such a distinction?

iv. If the answer to part (iii) is yes, what set of weights and bias might be used?

i. The input vectors are plotted in Figure P10.2.

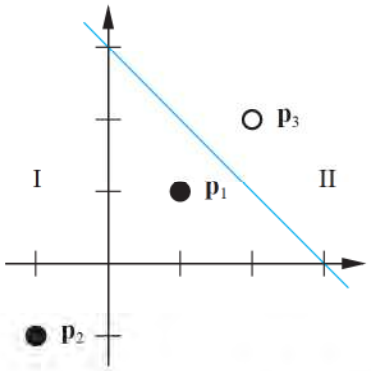


Figure P10.2 Input Vectors for Problem P10.1 (i)

The blue line in this figure is a decision boundary that separates the two categories successfully. Since they are linearly separable, an ADALINE network will do the job.

ii. The decision boundary passes through the points $(3, 0)$ and $(0, 3)$. We know these points to be the intercepts $-b/w_{1,1}$ and $-b/w_{1,2}$. Thus, a solution

$$b = 3,\ w_{1,1} = -1,\ w_{1,2} = -1,$$

is satisfactory. Note that if the output of the ADALINE is positive or zero the input vector is classified as Category I, and if the output is negative the input vector is classified as Category II. This solution also provides for error, since the decision boundary bisects the line between $\mathbf{p}_1$ and $\mathbf{p}_3$.

iii. The input vectors to be distinguished are shown in Figure P10.3. The vectors in the figure are not linearly separable, so an ADALINE network cannot distinguish between them.

iv. As noted in part (iii), an ADALINE cannot do the job, so there are no values for the weights and bias that are satisfactory.

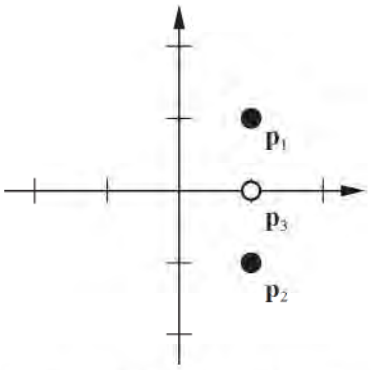


Figure P10.3 Input Vectors for Problem P10.1 (iii)

P10.3 Suppose that we have the following input/target pairs:

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

These patterns occur with equal probability, and they are used to train an ADALINE network with no bias. What does the mean square error performance surface look like?

First we need to calculate the various terms of the quadratic function. Recall from Eq. (10.11) that the performance index can be written as

$$F(\mathbf{x}) = c - 2\mathbf{x}^T\mathbf{h} + \mathbf{x}^T\mathbf{R}\mathbf{x}.$$

Therefore we need to calculate c, $\mathbf{h}$ and $\mathbf{R}$.

The probability of each input occurring is 0.5, so the probability of each target is also 0.5. Thus, the expected value of the square of the targets is

$$c = E[t^2] = (1)^2(0.5) + (-1)^2(0.5) = 1.$$

In a similar way, the cross-correlation between the input and the target can be calculated:

$$\mathbf{h} = E[t\mathbf{z}] = (0.5)(1)\begin{bmatrix}1\\1\end{bmatrix} + (0.5)(-1)\begin{bmatrix}1\\-1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}.$$

Finally, the input correlation matrix $\mathbf{R}$ is

$$\mathbf{R} = E[\mathbf{z}\mathbf{z}^T] = \mathbf{p}_1\mathbf{p}_1^T(0.5) + \mathbf{p}_2\mathbf{p}_2^T(0.5)$$

$$= (0.5)\left[\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}1 & 1\end{bmatrix} + \begin{bmatrix}1\\-1\end{bmatrix}\begin{bmatrix}1 & -1\end{bmatrix}\right] = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$$

Therefore the mean square error performance index is

$$F(\mathbf{x}) = c - 2\mathbf{x}^T\mathbf{h} + \mathbf{x}^T\mathbf{R}\mathbf{x}$$

$$= 1 - 2\begin{bmatrix}w_{1,1} & w_{1,2}\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} + \begin{bmatrix}w_{1,1} & w_{1,2}\end{bmatrix}\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}w_{1,1}\\w_{1,2}\end{bmatrix}$$

$$= 1 - 2w_{1,2} + w_{1,1}^2 + w_{1,2}^2$$

The Hessian matrix of $F(\mathbf{x})$, which is equal to $2\mathbf{R}$, has both eigenvalues at 2. Therefore the contours of the performance surface will be circular. To find the center of the contours (the minimum point), we need to solve Eq. (10.18):

$$\mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h} = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}^{-1}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}.$$

Thus we have a minimum at $w_{1,1} = 0$, $w_{1,2} = 1$. The resulting mean square error performance surface is shown in Figure P10.4.

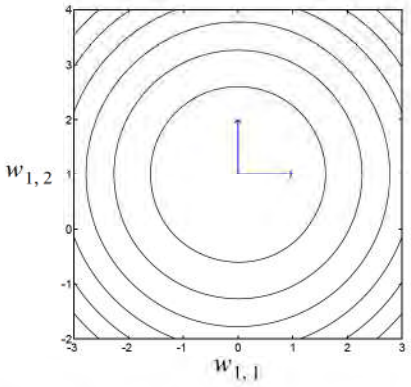


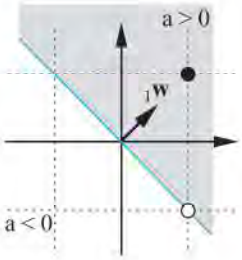
Figure P10.4 Contour Plot of $F(\mathbf{x})$ for Problem P10.3

P10.4 Consider the system of Problem P10.3 again. Train the network using the LMS algorithm, with the initial guess set to zero and a learning rate $\alpha = 0.25$. Apply each reference pattern only once during training. Draw the decision boundary at each stage.

Assume the input vector $\mathbf{p}_1$ is presented first. The output, error and new weights are calculated as follows:

$$a(0) = purelin\left(\begin{bmatrix}0 & 0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\right) = 0,$$

$$e(0) = t(0) - a(0) = 1 - 0 = 1,$$



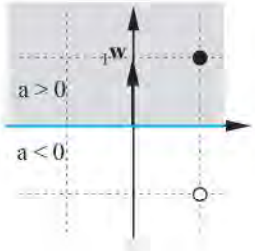
$$\mathbf{W}(1) = \mathbf{W}(0) + 2\alpha e(0)\mathbf{p}(0)^T = \begin{bmatrix}0 & 0\end{bmatrix} + 2\left(\frac{1}{4}\right)(1)\begin{bmatrix}1 & 1\end{bmatrix} = \begin{bmatrix}\frac{1}{2} & \frac{1}{2}\end{bmatrix}.$$

The decision boundary associated with these weights is shown to the left.

Now apply the second input vector:

$$a(1) = purelin\left\{\begin{bmatrix}\frac{1}{2} & \frac{1}{2}\end{bmatrix}\begin{bmatrix}1\\-1\end{bmatrix}\right\} = 0,$$

$$e(1) = t(1) - a(1) = -1 - 0 = -1,$$



$$\mathbf{W}(2) = \mathbf{W}(1) + 2\alpha e(1)\mathbf{p}(1)^T = \begin{bmatrix}\frac{1}{2} & \frac{1}{2}\end{bmatrix} + 2\left(\frac{1}{4}\right)(-1)\begin{bmatrix}1 & -1\end{bmatrix} = \begin{bmatrix}0 & 1\end{bmatrix}.$$

The decision boundary associated with these weights is shown to the left. This boundary shows real promise. It is exactly halfway between the input vectors. You might verify for yourself that each input vector, when applied, yields its correct associated target. (What set of weights would be optimal if the targets associated with the two input vectors were exchanged?)

P10.5 Now consider the convergence of the system of Problems P10.3 and P10.4. What is the maximum stable learning rate for the LMS algorithm?



The LMS convergence is determined by the learning rate α, which should not exceed the reciprocal of the largest eigenvalue of $\mathbf{R}$. We can determine this limit by finding these eigenvalues using MATLAB.

```
[V,D] = eig (R)
V =
        1       0
        0       1

D=
        1       0
        0       1
```

The diagonal terms of matrix D give the eigenvalues, 1 and 1, while the columns of V show the eigenvectors. Note, incidentally, that the eigenvectors have the same direction as those shown in Figure P10.4.

The largest eigenvalue, $\lambda_{max} = 1$, sets the upper limit on the learning rate at

$$\alpha < 1/\lambda_{max} = 1/1 = 1 .$$

The suggested learning rate in the previous problem was 0.25, and you found (perhaps) that the LMS algorithm converged quickly. What do you suppose happens when the learning rate is 1.0 or larger?

P10.6 Consider the adaptive filter ADALINE shown in Figure P10.5. The purpose of this filter is to predict the next value of the input signal from the two previous values. Suppose that the input signal is a stationary random process, with autocorrelation function given by

$$C_y(n) = E[y(k)y(k+n)]$$

$$C_y(0) = 3 , \; C_y(1) = -1 , \; C_y(2) = -1 .$$

i. Sketch the contour plot of the performance index (mean square error).

ii. What is the maximum stable value of the learning rate (α) for the LMS algorithm?

iii. Assume that a very small value is used for α. Sketch the path of the weights for the LMS algorithm, starting with initial guess $\mathbf{W}(0) = \begin{bmatrix} 0.75 & 0 \end{bmatrix}^T$. Explain your procedure for sketching the path.

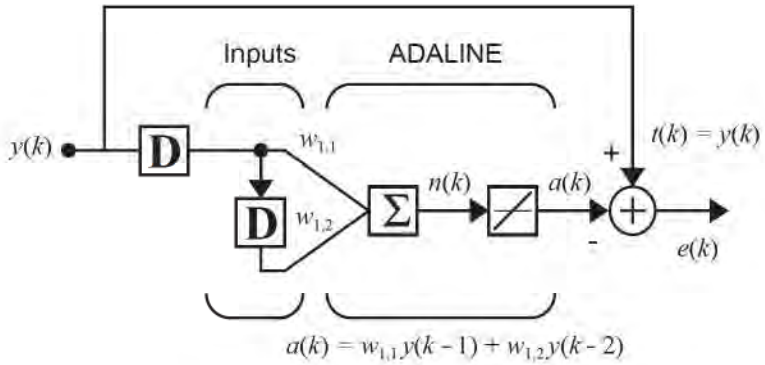


Figure P10.5 Adaptive Predictor

i. To sketch the contour plot we first need to find the performance index and the eigenvalues and eigenvectors of the Hessian matrix. First note that the input vector is given by

$$\mathbf{z}(k) = \mathbf{p}(k) = \begin{bmatrix} y(k-1) \\ y(k-2) \end{bmatrix}.$$

Now consider the performance index. Recall from Eq. (10.12) that

$$F(\mathbf{x}) = c - 2\mathbf{x}^T\mathbf{h} + \mathbf{x}^T\mathbf{R}\mathbf{x}.$$

We can calculate the constants in the performance index as shown below:

$$c = E[t^2(k)] = E[y^2(k)] = C_y(0) = 3,$$

$$\mathbf{R} = E[\mathbf{z}\mathbf{z}^T] = E\begin{bmatrix} y^2(k-1) & y(k-1)y(k-2) \\ y(k-1)y(k-2) & y^2(k-2) \end{bmatrix}$$

$$= \begin{bmatrix} C_y(0) & C_y(1) \\ C_y(1) & C_y(0) \end{bmatrix} = \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix}$$

$$\mathbf{h} = E[t\,\mathbf{z}] = E\begin{bmatrix} y(k)y(k-1) \\ y(k)y(k-2) \end{bmatrix} = \begin{bmatrix} C_y(1) \\ C_y(2) \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \end{bmatrix}.$$

The optimal weights are

$$\mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h} = \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix}^{-1}\begin{bmatrix} -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 3/8 & 1/8 \\ 4/8 & 3/8 \end{bmatrix}\begin{bmatrix} -1 \\ -1 \end{bmatrix} = \begin{bmatrix} -1/2 \\ -1/2 \end{bmatrix}.$$

The Hessian matrix is

$$\nabla^2 F(\mathbf{x}) = \mathbf{A} = 2\mathbf{R} = \begin{bmatrix} 6 & -2 \\ -2 & 6 \end{bmatrix}.$$

Now we can get the eigenvalues:

$$\left| \mathbf{A} - \lambda \mathbf{I} \right| = \begin{vmatrix} 6-\lambda & -2 \\ -2 & 6-\lambda \end{vmatrix} = \lambda^2 - 12\lambda + 32 = (\lambda - 8)(\lambda - 4).$$

Thus,

$$\lambda_1 = 4, \qquad \lambda_2 = 8.$$

To find the eigenvectors we use

$$\left[\mathbf{A} - \lambda \mathbf{I}\right]\mathbf{v} = 0.$$

For $\lambda_1 = 4$,

$$\begin{bmatrix} 2 & -2 \\ -2 & 2 \end{bmatrix}\mathbf{v}_1 = 0 \qquad \mathbf{v}_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix},$$

and for $\lambda_2 = 8$,

$$\begin{bmatrix} -2 & -2 \\ -2 & -2 \end{bmatrix}\mathbf{v}_2 = 0 \qquad \mathbf{v}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

Therefore the contours of $F(\mathbf{x})$ will be elliptical, with the long axis of each ellipse along the first eigenvector, since the first eigenvalue has the smallest magnitude. The ellipses will be centered at $\mathbf{x}^*$. The contour plot is shown in Figure P10.6.

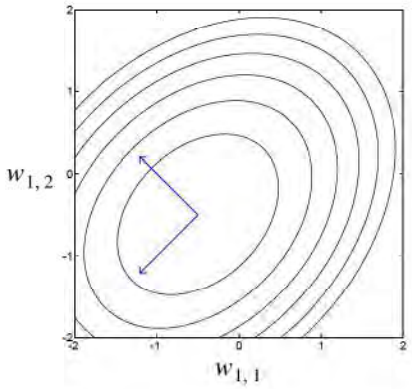


Figure P10.6 Error Contour for Problem P10.6

You might check your sketch by writing a MATLAB M-file to plot the contours.

ii. The maximum stable learning rate is the reciprocal of the maximum eigenvalue of $\mathbf{R}$, which is the same as twice the reciprocal of the largest eigenvalue of the Hessian matrix $\nabla^2 F(\mathbf{x}) = \mathbf{A}$:

$$\alpha < 2/\lambda_{max} = 2/8 = 0.25 .$$

iii. The LMS algorithm is approximate steepest descent, so the trajectory for small learning rates will move perpendicular to the contour lines, as shown in Figure P10.7.

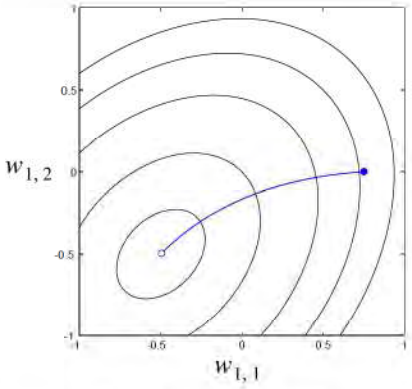


Figure P10.7 LMS Weight Trajectory

P10.7 The pilot of an airplane is talking into a microphone in his cockpit. The sound received by the air traffic controller in the tower is garbled because the pilot's voice signal has been contaminated by engine noise that reaches his microphone. Can you suggest an adaptive ADALINE filter that might help reduce the noise in the signal received by the control tower? Explain your system.

The engine noise that has been inadvertently added to the microphone input can be minimized by using the adaptive filtering system shown in Figure P10.8. A sample of the engine noise is supplied to an adaptive filter through a microphone in the cockpit. The desired output of the filter is the contaminated signal coming from the pilot's microphone. The filter attempts to reduce the "error" signal to a minimum. It can do this only by subtracting the component of the contaminated signal that is linearly correlated with the engine noise (and presumably uncorrelated with the pilot's voice). The result is that a clear voice signal is sent to the control tower, in spite of the fact that the engine noise got into the pilot's microphone along with his voice signal. (See [WiSt85] for discussion of similar noise cancellation systems.)

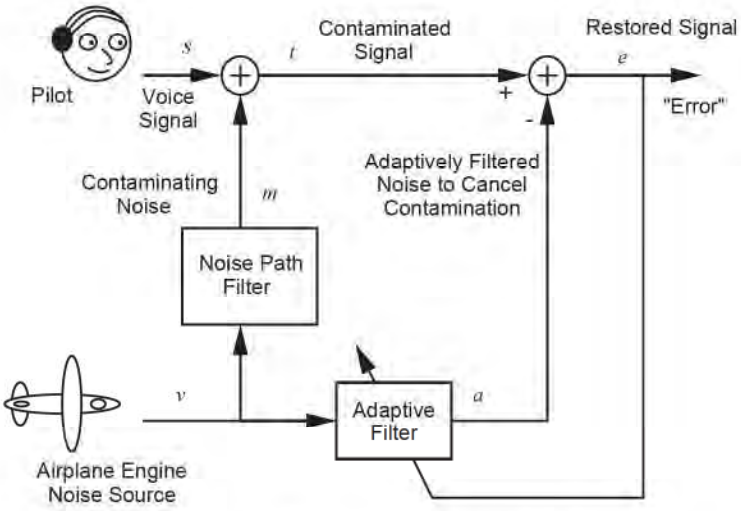


Figure P10.8 Filtering Engine Noise from Pilot's Voice Signal

P10.8 This is a classification problem like that described in Problems P4.3 and P4.5, except that here we will use an ADALINE network and the LMS learning rule rather than the perceptron learning rule. First we will describe the problem.

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

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

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

Train an ADALINE network to solve this problem using the LMS learning rule. Assume that each pattern occurs with probability $1/8$**.**

Let's begin by displaying the input vectors, as in Figure P10.9. 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. These input vectors can be plotted as shown in Figure P10.9.

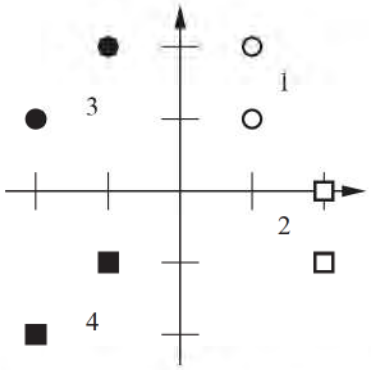


Figure P10.9 Input Vectors for Problem P10.8

We will use target vectors similar to the ones we introduced in Problem P4.3, except that we will replace any targets of 0 by targets of -1. (The perceptron could only output 0 or 1.) Thus, the training set will be:

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

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

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

Also, we will begin as in Problem P4.5 with the following initial weights and biases:

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

Now we are almost ready to train an ADALINE network using the LMS rule. We will use a learning rate of $\alpha = 0.04$, and we will present the input vectors in order according to their subscripts. The first iteration is

$$\mathbf{a}(0) = purelin\,(\mathbf{W}(0)\mathbf{p}(0) + \mathbf{b}(0)) = purelin\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

$$\mathbf{e}(0) = \mathbf{t}(0) - \mathbf{a}(0) = \begin{bmatrix} -1 \\ -1 \end{bmatrix} - \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} -3 \\ -3 \end{bmatrix}$$

$$\mathbf{W}(1) = \mathbf{W}(0) + 2\alpha\mathbf{e}(0)\mathbf{p}^T(0)$$

$$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + 2(0.04)\begin{bmatrix} -3 \\ -3 \end{bmatrix}\begin{bmatrix} 1 & 1 \end{bmatrix} = \begin{bmatrix} 0.76 & -0.24 \\ -0.24 & 0.76 \end{bmatrix}$$

$$\mathbf{b}(1) = \mathbf{b}(0) + 2\alpha\mathbf{e}(0) = \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 2(0.04)\begin{bmatrix} -3 \\ -3 \end{bmatrix} = \begin{bmatrix} 0.76 \\ 0.76 \end{bmatrix}.$$

The second iteration is

$$\mathbf{a}(1) = purelin\,(\mathbf{W}(1)\mathbf{p}(1) + \mathbf{b}(1))$$

$$= purelin\left(\begin{bmatrix} 0.76 & -0.24 \\ -0.24 & 0.76 \end{bmatrix}\begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 0.76 \\ 0.76 \end{bmatrix}\right) = \begin{bmatrix} 1.04 \\ 2.04 \end{bmatrix}$$

$$\mathbf{e}(1) = \mathbf{t}(1) - \mathbf{a}(1) = \begin{bmatrix} -1 \\ -1 \end{bmatrix} - \begin{bmatrix} 1.04 \\ 2.04 \end{bmatrix} = \begin{bmatrix} -2.04 \\ -3.04 \end{bmatrix}$$

$$\mathbf{W}(2) = \mathbf{W}(1) + 2\alpha\mathbf{e}(1)\mathbf{p}^T(1)$$

$$= \begin{bmatrix} 0.76 & -0.24 \\ -0.24 & 0.76 \end{bmatrix} + 2(0.04)\begin{bmatrix} -2.04 \\ -3.04 \end{bmatrix}\begin{bmatrix} 1 & 2 \end{bmatrix} = \begin{bmatrix} 0.5968 & -0.5664 \\ -0.4832 & 0.2736 \end{bmatrix}$$

$$\mathbf{b}(2) = \mathbf{b}(1) + 2\alpha\mathbf{e}(1) = \begin{bmatrix}0.76\\0.76\end{bmatrix} + 2(0.04)\begin{bmatrix}-2.04\\-3.04\end{bmatrix} = \begin{bmatrix}0.5968\\0.5168\end{bmatrix}.$$

If we continue until the weights converge we find

$$\mathbf{W}(\infty) = \begin{bmatrix}-0.5948 & -0.0523\\0.1667 & -0.6667\end{bmatrix}, \quad \mathbf{b}(\infty) = \begin{bmatrix}0.0131\\0.1667\end{bmatrix}.$$

The resulting decision boundaries are shown in Figure P10.10. Compare this result with the final decision boundaries created by the perceptron learning rule in Problem P4.5 (Figure P4.7). The perceptron rule stops training when all the patterns are classified correctly. The LMS algorithm moves the boundaries as far from the patterns as possible.

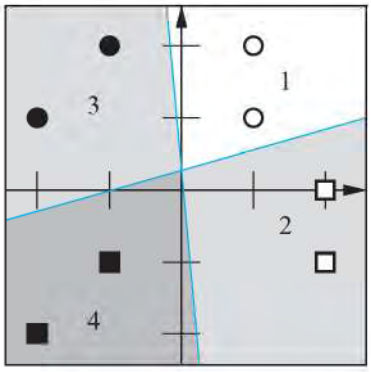


Figure P10.10 Final Decision Boundaries for Problem P10.8

P10.9 Repeat the work of Widrow and Hoff on a pattern recognition problem from their classic 1960 paper [WiHo60]. They wanted to design a recognition system that would classify the six patterns shown in Figure P10.11.

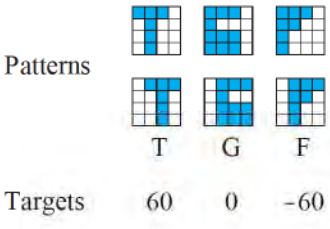


Figure P10.11 Patterns and Their Classification Targets

These patterns represent the letters T, G and F, in an original form on the top and in a shifted form on the bottom. The targets for these letters (in their original and shifted forms) are +60, 0 and -60, respectively. (The values of 60, 0 and -60 were nice for use on the face of a meter that Widrow and Hoff used to display their network output.) The objective is to train a network so that it will classify the six patterns into the appropriate T, G or F groups.

The blue squares in the letters will be assigned the value +1, and the white squares will be assigned the value -1. First we convert each of the letters into a single 16-element vector. We choose to do this by starting at the upper left corner, going down the left column, then going down the second column, etc. For example, the vector corresponding to the unshifted letter T is

$$\mathbf{p}_1 = \begin{bmatrix} 1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}^T$$

We have such an input vector for each of the six letters.

The ADALINE network that we will use is shown in Figure P10.12.

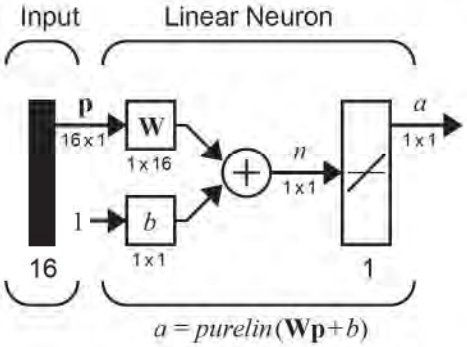


Figure P10.12 Adaptive Pattern Classifier

(Widrow and Hoff built their own machine to realize this ADALINE. According to them, it was "about the size of a lunch pail.")

Now we will present the six vectors to the network in a random sequence and adjust the weights of the network after each presentation using the LMS algorithm with a learning rate of $\alpha = 0.03$. After each adjustment of weights, all six vectors will be presented to the network to generate their outputs and corresponding errors. The sum of the squares of the errors will be examined as a measure of the quality of the network.

Figure P10.13 illustrates the convergence of the network. The network is trained to recognize these six characters in about 60 presentations, or roughly 10 for each of the possible input vectors.

The results shown in Figure P10.13 are quite like those obtained and published by Widrow and Hoff some 35 years ago. Widrow and Hoff did good science. One can indeed duplicate their work, even decades later (without a lunch pail).

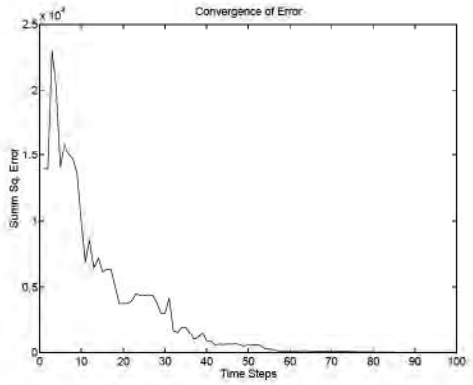


Figure P10.13 Error Convergence with Learning Rate of 0.03



To experiment with this character recognition problem, use the MATLAB® Neural Network Design Demonstration Linear Pattern Classification (`nnd10lc`). Notice the sensitivity of the network to noise in the input pattern.

Epilogue

In this chapter we have presented the ADALINE neural network and the LMS learning rule. The ADALINE network is very similar to the perceptron network of Chapter 4, and it has the same fundamental limitation: it can only classify linearly separable patterns. In spite of this limitation on the network, the LMS algorithm is in fact more powerful than the perceptron learning rule. Because it minimizes mean square error, the algorithm is able to create decision boundaries that are more robust to noise than those of the perceptron learning rule.

The ADALINE network and the LMS algorithm have found many practical applications. Even though they were first presented in the late 1950s, they are still very much in use in adaptive filtering applications. For example, echo cancellers using the LMS algorithm are currently employed on many long distance telephone lines. (Chapter 14 provides more extensive coverage of dynamic networks, which are widely used for filtering, prediction and control.)

In addition to its importance as a practical solution to many adaptive filtering problems, the LMS algorithm is also important because it is the forerunner of the backpropagation algorithm, which we will discuss in Chapters 11 through 14. Like the LMS algorithm, backpropagation is an approximate steepest descent algorithm that minimizes mean square error. The only difference between the two algorithms is in the manner in which the derivatives are calculated. Backpropagation is a generalization of the LMS algorithm that can be used for multilayer networks. These more complex networks are not limited to linearly separable problems. They can solve arbitrary classification problems.

Further Reading

[AnRo89] J. A. Anderson, 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.

[StDo84] W. D. Stanley, G. R. Dougherty, R. Dougherty, *Digital Signal Processing*, Reston VA: Reston, 1984

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

This seminal paper describes an adaptive perceptron-like network that can learn quickly and accurately. The authors assumed that the system had inputs, a desired output classification for each input, and that the system could 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].

[WiSt 85] B. Widrow and S. D. Stearns, *Adaptive Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1985.

This informative book describes the theory and application of adaptive signal processing. The authors include a review of the mathematical background that is needed, give details on their adaptive algorithms, and then discuss practical information about many applications.

[WiWi 88] B. Widrow and R. Winter, "Neural nets for adaptive filtering and adaptive pattern recognition," *IEEE Computer Magazine*, March 1988, pp. 25–39.

This is a particularly readable paper that summarizes applications of adaptive multilayer neural networks. The networks are applied to system modeling, statistical prediction, echo cancellation, inverse modeling and pattern recognition.

Exercises

E10.1 An adaptive filter ADALINE is shown in Figure E10.1. Suppose that the weights of the network are given by

$$w_{1,1} = 1, w_{1,2} = -4, w_{1,3} = 2,$$

and the input to the filter is

$$\{y(k)\} = \{\ldots, 0, 0, 0, 1, 1, 2, 0, 0, \ldots\}.$$

Find the response $\{a(k)\}$ of the filter.

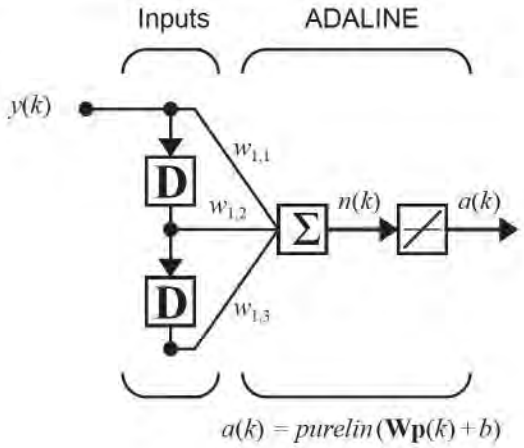


Figure E10.1 Adaptive Filter ADALINE for Exercise E10.1

E10.2 In Figure E10.2 two classes of patterns are given.

i. Use the LMS algorithm to train an ADALINE network to distinguish between class I and class II patterns (we want the network to identify horizontal and vertical lines).

ii. Can you explain why the ADALINE network might have difficulty with this problem?



Figure E10.2 Pattern Classification Problem for Exercise E10.2

E10.3 Suppose that we have the following two reference patterns and their targets:

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

In Problem P10.3 these input vectors to an ADALINE were assumed to occur with equal probability. Now suppose that the probability of vector $\mathbf{p}_1$ is 0.75 and that the probability of vector $\mathbf{p}_2$ is 0.25. Does this change of probabilities change the mean square error surface? If yes, what does the surface look like now? What is the maximum stable learning rate?

E10.4 In this exercise we will modify the reference pattern $\mathbf{p}_2$ from Problem P10.3:

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

i. Assume that the patterns occur with equal probability. Find the mean square error and sketch the contour plot.

ii. Find the maximum stable learning rate.



iii. Write a MATLAB M-file to implement the LMS algorithm for this problem. Take 40 steps of the algorithm for a stable learning rate. Use the zero vector as the initial guess. Sketch the trajectory on the contour plot.

iv. Take 40 steps of the algorithm after setting the initial values of both parameters to 1. Sketch the final decision boundary.

v. Compare the final parameters from parts (iii) and (iv). Explain your results.

E10.5 We again use the reference patterns and targets from Problem P10.3, and assume that they occur with equal probability. This time we want to train an ADALINE network with a bias. We now have three parameters to find: $w_{1,1}$, $w_{1,2}$ and b.

i. Find the mean square error and the maximum stable learning rate.



ii. Write a MATLAB M-file to implement the LMS algorithm for this problem. Take 40 steps of the algorithm for a stable learning rate. Use the zero vector as the initial guess. Sketch the final decision boundary.

iii. Take 40 steps of the algorithm after setting the initial values of all parameters to 1. Sketch the final decision boundary.

iv. Compare the final parameters and the decision boundaries from parts (iii) and (iv). Explain your results.

E10.6 We have two categories of vectors. Category I consists of

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

Category II consists of

$$\left\{\begin{bmatrix}0\\-1\end{bmatrix}, \begin{bmatrix}-4\\1\end{bmatrix}\right\}.$$

We want to train a single-neuron ADALINE network without a bias to recognize these categories ($t = 1$ for Category I and $t = -1$ for Category II). Assume that each pattern occurs with equal probability.

i. Draw the network diagram.

ii. Take four steps of the LMS algorithm, using the zero vector as the initial guess. (one pass through the four vectors above - present each vector once). Use a learning rate of 0.1.

iii. What are the optimal weights?

iv. Sketch the optimal decision boundary.

v. How do you think the boundary would change if the network were allowed to have a bias? If the boundary would change, indicate the approximate new position on your sketch of part iv. You do not need to perform any calculations here - just explain your reasoning.

E10.7 Suppose that we have the following three reference patterns and their targets:

$$\left\{\mathbf{p}_1 = \begin{bmatrix}3\\6\end{bmatrix}, t_1 = \begin{bmatrix}75\end{bmatrix}\right\}, \left\{\mathbf{p}_2 = \begin{bmatrix}6\\3\end{bmatrix}, t_2 = \begin{bmatrix}75\end{bmatrix}\right\}, \left\{\mathbf{p}_3 = \begin{bmatrix}-6\\3\end{bmatrix}, t_3 = \begin{bmatrix}-75\end{bmatrix}\right\}.$$

Each pattern is equally likely.

i. Draw the network diagram for an ADALINE network with no bias that could be trained on these patterns.

ii. We want to train the ADALINE network with no bias using these patterns. Sketch the contour plot of the mean square error performance index.

iii. Find the maximum stable learning rate for the LMS algorithm.

iv. Sketch the trajectory of the LMS algorithm on your contour plot. Assume a very small learning rate, and start with all weights equal to zero. This does not require any calculations.

E10.8 Suppose that we have the following two reference patterns and their targets:

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

The probability of vector $\mathbf{p}_1$ is 0.5 and the probability of vector $\mathbf{p}_2$ is 0.5.We want to train an ADALINE network without a bias on this data set.

i. Sketch the contour plot of the mean square error performance index.

ii. Sketch the optimal decision boundary.

iii. Find the maximum stable learning rate.

iv. Sketch the trajectory of the LMS algorithm on your contour plot. Assume a very small learning rate, and start with initial weights $\mathbf{W}(0) = \begin{bmatrix}0 & 1\end{bmatrix}$.

E10.9 We have the following input/target pairs:

$$\left\{\mathbf{p}_1 = \begin{bmatrix}4\\2\end{bmatrix}, t_1 = 5\right\}, \left\{\mathbf{p}_2 = \begin{bmatrix}2\\-4\end{bmatrix}, t_2 = -2\right\}, \left\{\mathbf{p}_3 = \begin{bmatrix}-4\\4\end{bmatrix}, t_3 = 9\right\}.$$

The first two pair each occurs with probability of 0.25, and the third pair occurs with probability 0.5. We want to train a single-neuron ADALINE network without a bias to perform the desired mapping.

i. Draw the network diagram.

ii. What is the maximum stable learning rate?

iii. Perform one iteration of the LMS algorithm. Apply the input $\mathbf{p}_1$ and use a learning rate of $\alpha = 0.1$. Start from the initial weights $\mathbf{x}_0 = \begin{bmatrix}0 & 0\end{bmatrix}^T$.

E10.10 Repeat E10.9 for the following input/target pairs:

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\-4\end{bmatrix}, t_1 = 1\right\}, \left\{\mathbf{p}_2 = \begin{bmatrix}-4\\4\end{bmatrix}, t_2 = -1\right\}, \left\{\mathbf{p}_3 = \begin{bmatrix}4\\2\end{bmatrix}, t_3 = 1\right\}.$$

The first two pair each occurs with probability of 0.25, and the third pair occurs with probability 0.5. We want to train a single-neuron ADALINE network without a bias to perform the desired mapping.

E10.11 We want to train a single-neuron ADALINE network without a bias, using the following training set, which categorizes vectors into two classes. Each pattern occurs with equal probability.

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

i. Draw the network diagram.

ii. Take one step of the LMS algorithm (present $\mathbf{p}_1$ only) starting from the initial weight $\mathbf{W}(0) = \begin{bmatrix}0 & 0\end{bmatrix}$. Use a learning rate of 0.1.

iii. What are the optimal weights? Show all calculations.

iv. Sketch the optimal decision boundary.

v. How do you think the boundary would change if the network were allowed to have a bias? Indicate the approximate new position on your sketch of part iv.

vi. What is the maximum stable learning rate for the LMS algorithm?

vii. Sketch the contour plot of the mean square error performance surface.

viii. On your contour plot of part vii, sketch the path of the LMS algorithm for a very small learning rate (e.g., 0.001) starting from the initial condition $\mathbf{W}(0) = \begin{bmatrix}2 & 0\end{bmatrix}$. This does not require any calculations, but explain how you obtained your answer.

E10.12 Suppose that we have the following three reference patterns and their targets:

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\4\end{bmatrix}, t_1 = \begin{bmatrix}26\end{bmatrix}\right\}, \left\{\mathbf{p}_2 = \begin{bmatrix}4\\2\end{bmatrix}, t_2 = \begin{bmatrix}26\end{bmatrix}\right\}, \left\{\mathbf{p}_3 = \begin{bmatrix}-2\\-2\end{bmatrix}, t_3 = \begin{bmatrix}-26\end{bmatrix}\right\}.$$

The probability of vector $\mathbf{p}_1$ is 0.25, the probability of vector $\mathbf{p}_2$ is 0.25 and the probability of vector $\mathbf{p}_3$ is 0.5.

i. Draw the network diagram for an ADALINE network with no bias that could be trained on these patterns.

ii. Sketch the contour plot of the mean square error performance index.

iii. Show the optimal decision boundary (for the weights that minimize mean square error) and verify that it separates the patterns into the appropriate categories.

iv. Find the maximum stable learning rate for the LMS algorithm. If the target values are changed from 26 and -26 to 2 and -2, how would this change the maximum stable learning rate?

v. Perform one iteration of the LMS algorithm, starting with all weights equal to zero, and presenting input vector $\mathbf{p}_1$. Use a learning rate of $\alpha = 0.5$.

vi. Sketch the trajectory of the LMS algorithm on your contour plot. Assume a very small learning rate, and start with all weights equal to zero.

E10.13 Consider the adaptive predictor in Figure E10.3.

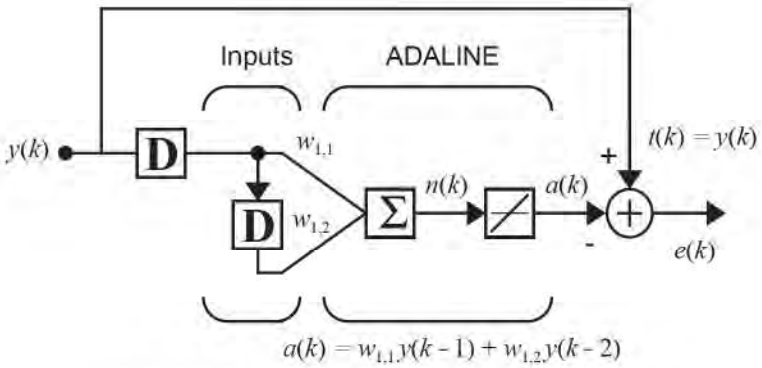


Figure E10.3 Adaptive Predictor for Exercise E10.13

Assume that $y(k)$ is a stationary process with autocorrelation function

$$C_y(n) = E[y(k)(y(k+n))] .$$

i. Write an expression for the mean square error in terms of $C_y(n)$.

ii. Give a specific expression for the mean square error when

$$y(k) = \sin\left(\frac{k\pi}{5}\right).$$

iii. Find the eigenvalues and eigenvectors of the Hessian matrix for the mean square error. Locate the minimum point and sketch a rough contour plot.

iv. Find the maximum stable learning rate for the LMS algorithm.

v. Take three steps of the LMS algorithm by hand, using a stable learning rate. Use the zero vector as the initial guess.



vi. Write a MATLAB M-file to implement the LMS algorithm for this problem. Take 40 steps of the algorithm for a stable learning rate and sketch the trajectory on the contour plot. Use the zero vector as the initial guess. Verify that the algorithm is converging to the optimal point.

vii. Verify experimentally that the algorithm is unstable for learning rates greater than that found in part (iv).

E10.14 Repeat Problem P10.9, but use the numerals "1", "2" and "4", instead of the letters "T", "G" and "F". Test the trained network on each reference pattern and on noisy patterns. Discuss the sensitivity of the network. (*Use the Neural Network Design Demonstration Linear Pattern Classification* (`nnd10lc`).)



11 Backpropagation

Objectives	11-1
Theory and Examples	11-2
Multilayer Perceptrons	11-2
Pattern Classification	11-3
Function Approximation	11-4
The Backpropagation Algorithm	11-7
Performance Index	11-8
Chain Rule	11-9
Backpropagating the Sensitivities	11-11
Summary	11-13
Example	11-14
Batch vs. Incremental Training	11-17
Using Backpropagation	11-18
Choice of Network Architecture	11-18
Convergence	11-20
Generalization	11-22
Summary of Results	11-25
Solved Problems	11-27
Epilogue	11-41
Further Reading	11-42
Exercises	11-44

Objectives

In this chapter we continue our discussion of performance learning, which we began in Chapter 8, by presenting a generalization of the LMS algorithm of Chapter 10. This generalization, called backpropagation, can be used to train multilayer networks. As with the LMS learning law, backpropagation is an approximate steepest descent algorithm, in which the performance index is mean square error. The difference between the LMS algorithm and backpropagation is only in the way in which the derivatives are calculated. For a single-layer linear network the error is an explicit linear function of the network weights, and its derivatives with respect to the weights can be easily computed. In multilayer networks with nonlinear transfer functions, the relationship between the network weights and the error is more complex. In order to calculate the derivatives, we need to use the chain rule of calculus. In fact, this chapter is in large part a demonstration of how to use the chain rule.

Theory and Examples

The perceptron learning rule of Frank Rosenblatt and the LMS algorithm of Bernard Widrow and Marcian Hoff were designed to train single-layer perceptron-like networks. As we have discussed in previous chapters, these single-layer networks suffer from the disadvantage that they are only able to solve linearly separable classification problems. Both Rosenblatt and Widrow were aware of these limitations and proposed multilayer networks that could overcome them, but they were not able to generalize their algorithms to train these more powerful networks.

Apparently the first description of an algorithm to train multilayer networks was contained in the thesis of Paul Werbos in 1974 [Werbo74]. This thesis presented the algorithm in the context of general networks, with neural networks as a special case, and was not disseminated in the neural network community. It was not until the mid 1980s that the backpropagation algorithm was rediscovered and widely publicized. It was rediscovered independently by David Rumelhart, Geoffrey Hinton and Ronald Williams [RuHi86], David Parker [Park85], and Yann Le Cun [LeCu85]. The algorithm was popularized by its inclusion in the book *Parallel Distributed Processing* [RuMc86], which described the work of the Parallel Distributed Processing Group led by psychologists David Rumelhart and James McClelland. The publication of this book spurred a torrent of research in neural networks. The multilayer perceptron, trained by the backpropagation algorithm, is currently the most widely used neural network.

In this chapter we will first investigate the capabilities of multilayer networks and then present the backpropagation algorithm.

Multilayer Perceptrons

We first introduced the notation for multilayer networks in Chapter 2. For ease of reference we have reproduced the diagram of the three-layer perceptron in Figure 11.1. Note that we have simply cascaded three perceptron networks. The output of the first network is the input to the second network, and the output of the second network is the input to the third network. Each layer may have a different number of neurons, and even a different transfer function. Recall from Chapter 2 that we are using superscripts to identify the layer number. Thus, the weight matrix for the first layer is written as $\mathbf{W}^1$ and the weight matrix for the second layer is written $\mathbf{W}^2$.

To identify the structure of a multilayer network, we will sometimes use the following shorthand notation, where the number of inputs is followed by the number of neurons in each layer:

$$R - S^1 - S^2 - S^3 . \tag{11.1}$$

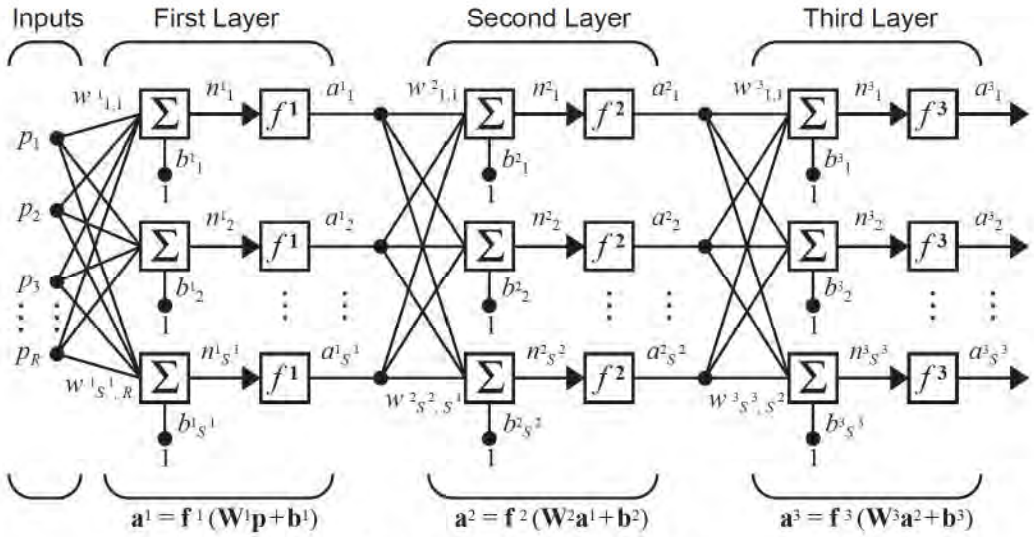


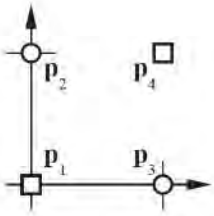
Figure 11.1 Three-Layer Network

Let's now investigate the capabilities of these multilayer perceptron networks. First we will look at the use of multilayer networks for pattern classification, and then we will discuss their application to function approximation.

Pattern Classification

To illustrate the capabilities of the multilayer perceptron for pattern classification, consider the classic exclusive-or (XOR) problem. 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, which is illustrated graphically in the figure to the left, was used by Minsky and Papert in 1969 to demonstrate the limitations of the single-layer perceptron. Because the two categories are not linearly separable, a single-layer perceptron cannot perform the classification.

A two-layer network can solve the XOR problem. In fact, there are many different multilayer solutions. One solution is to use two neurons in the first layer to create two decision boundaries. The first boundary separates $\mathbf{p}_1$ from the other patterns, and the second boundary separates $\mathbf{p}_4$. Then the second layer is used to combine the two boundaries together using an

AND operation. The decision boundaries for each first-layer neuron are shown in Figure 11.2.

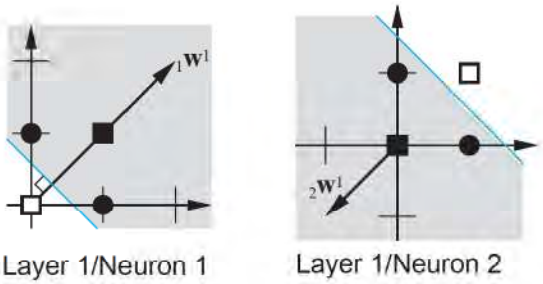
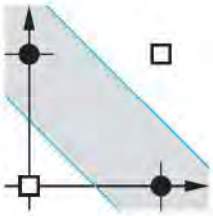


Figure 11.2 Decision Boundaries for XOR Network



The resulting two-layer, 2-2-1 network is shown in Figure 11.3. The overall decision regions for this network are shown in the figure in the left margin. The shaded region indicates those inputs that will produce a network output of 1.

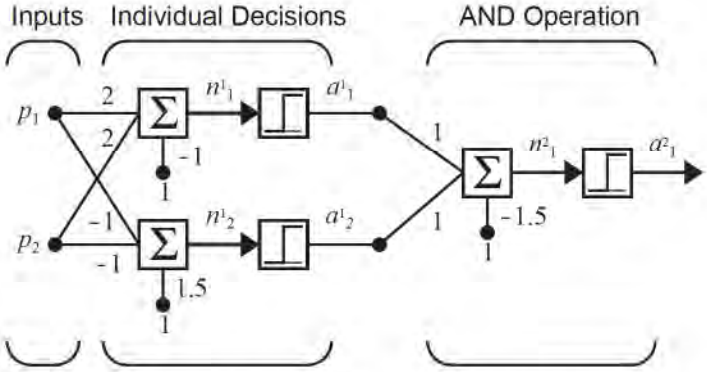


Figure 11.3 Two-Layer XOR Network

See Problems P11.1 and P11.2 for more on the use of multilayer networks for pattern classification.

Function Approximation

Up to this point in the text we have viewed neural networks mainly in the context of pattern classification. It is also instructive to view networks as function approximators. In control systems, for example, the objective is to find an appropriate feedback function that maps from measured outputs to control inputs. In adaptive filtering (Chapter 10) the objective is to find a function that maps from delayed values of an input signal to an appropriate output signal. The following example will illustrate the flexibility of the multilayer perceptron for implementing functions.

Consider the two-layer, 1-2-1 network shown in Figure 11.4. For this example the transfer function for the first layer is log-sigmoid and the transfer function for the second layer is linear. In other words,

$$f^1(n) = \frac{1}{1+e^{-n}} \text{ and } f^2(n) = n. \tag{11.2}$$

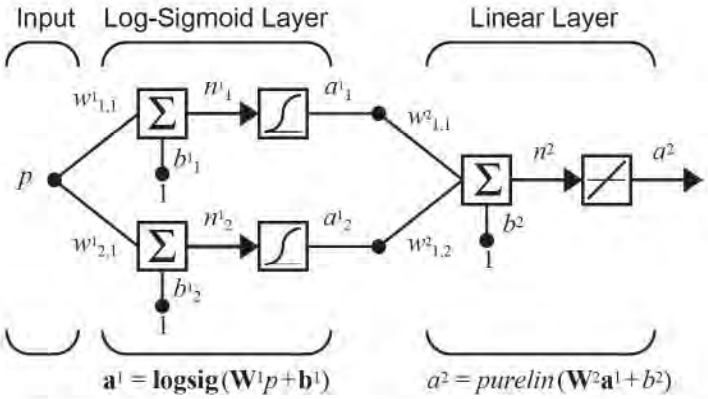


Figure 11.4 Example Function Approximation Network

Suppose that the nominal values of the weights and biases for this network are

$$w^1_{1,1} = 10,\ w^1_{2,1} = 10,\ b^1_1 = -10,\ b^1_2 = 10,$$

$$w^2_{1,1} = 1,\ w^2_{1,2} = 1,\ b^2 = 0.$$

The network response for these parameters is shown in Figure 11.5, which plots the network output a^2 as the input p is varied over the range $[-2, 2]$.

Notice that the response consists of two steps, one for each of the log-sigmoid neurons in the first layer. By adjusting the network parameters we can change the shape and location of each step, as we will see in the following discussion.

The centers of the steps occur where the net input to a neuron in the first layer is zero:

$$n^1_1 = w^1_{1,1} p + b^1_1 = 0 \quad \Rightarrow \quad p = -\frac{b^1_1}{w^1_{1,1}} = -\frac{-10}{10} = 1, \tag{11.3}$$

$$n_2^1 = w_{2,1}^1 p + b_2^1 = 0 \quad \Rightarrow \quad p = -\frac{b_2^1}{w_{2,1}^1} = -\frac{10}{10} = -1. \tag{11.4}$$

The steepness of each step can be adjusted by changing the network weights.

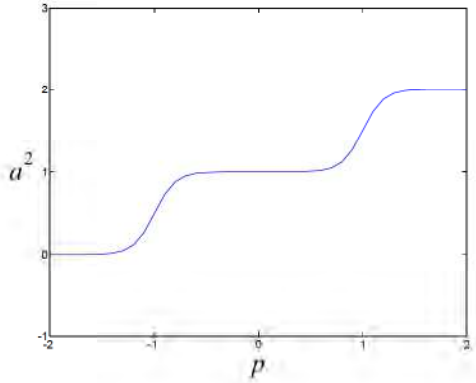


Figure 11.5 Nominal Response of Network of Figure 11.4

Figure 11.6 illustrates the effects of parameter changes on the network response. The blue curve is the nominal response. The other curves correspond to the network response when one parameter at a time is varied over the following ranges:

$$-1 \le w_{1,1}^2 \le 1\,,\ -1 \le w_{1,2}^2 \le 1\,,\ 0 \le b_2^1 \le 20\,,\ -1 \le b^2 \le 1\,. \tag{11.5}$$

Figure 11.6 (a) shows how the network biases in the first (hidden) layer can be used to locate the position of the steps. Figure 11.6 (b) illustrates how the weights determine the slope of the steps. The bias in the second (output) layer shifts the entire network response up or down, as can be seen in Figure 11.6 (d).

From this example we can see how flexible the multilayer network is. It would appear that we could use such networks to approximate almost any function, if we had a sufficient number of neurons in the hidden layer. In fact, it has been shown that two-layer networks, with sigmoid transfer functions in the hidden layer and linear transfer functions in the output layer, can approximate virtually any function of interest to any degree of accuracy, provided sufficiently many hidden units are available (see [HoSt89]).



To experiment with the response of this two-layer network, use the MATLAB® Neural Network Design Demonstration Network Function (`nnd11nf`).

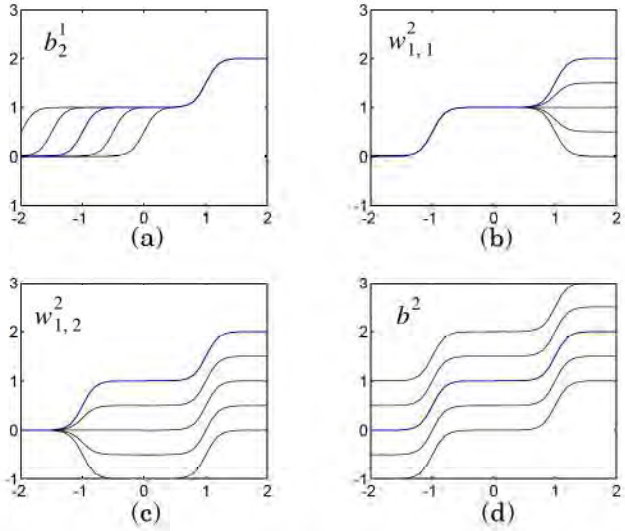


Figure 11.6 Effect of Parameter Changes on Network Response

Now that we have some idea of the power of multilayer perceptron networks for pattern recognition and function approximation, the next step is to develop an algorithm to train such networks.

The Backpropagation Algorithm

It will simplify our development of the backpropagation algorithm if we use the abbreviated notation for the multilayer network, which we introduced in Chapter 2. The three-layer network in abbreviated notation is shown in Figure 11.7.

As we discussed earlier, for multilayer networks the output of one layer becomes the input to the following layer. The equations that describe this operation are

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \text{ for } m = 0, 1, \ldots, M-1, \qquad (11.6)$$

where M is the number of layers in the network. The neurons in the first layer receive external inputs:

$$\mathbf{a}^0 = \mathbf{p}, \qquad (11.7)$$

which provides the starting point for Eq. (11.6). The outputs of the neurons in the last layer are considered the network outputs:

$$\mathbf{a} = \mathbf{a}^M. \qquad (11.8)$$

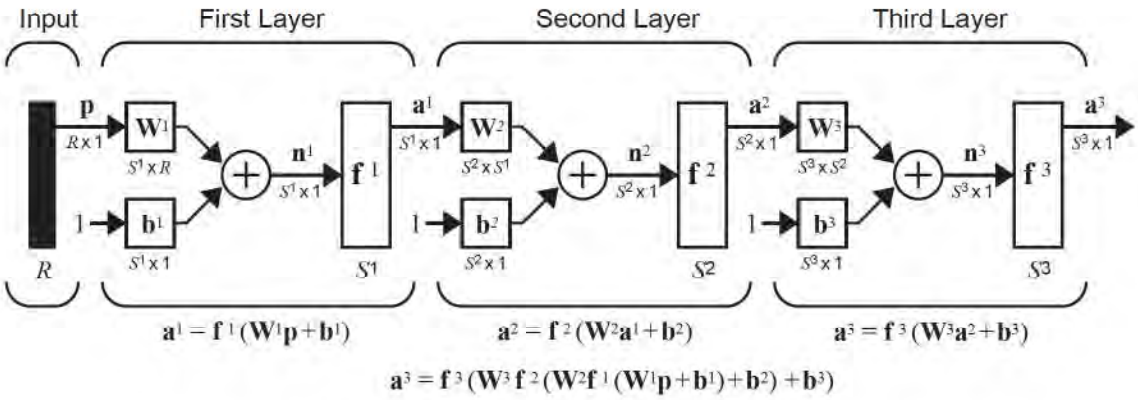


Figure 11.7 Three-Layer Network, Abbreviated Notation

Performance Index

The backpropagation algorithm for multilayer networks is a generalization of the LMS algorithm of Chapter 10, and both algorithms use the same performance index: *mean square error*. The algorithm is provided with a set of examples of proper network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}, \tag{11.9}$$

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 algorithm should adjust the network parameters in order to minimize the mean square error:

$$F(\mathbf{x}) = E[e^2] = E[(t-a)^2]. \tag{11.10}$$

where $\mathbf{x}$ is the vector of network weights and biases (as in Chapter 10). If the network has multiple outputs this generalizes to

$$F(\mathbf{x}) = E[\mathbf{e}^T\mathbf{e}] = E[(\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})]. \tag{11.11}$$

As with the LMS algorithm, we will approximate the mean square error by

$$\hat{F}(\mathbf{x}) = (\mathbf{t}(k)-\mathbf{a}(k))^T(\mathbf{t}(k)-\mathbf{a}(k)) = \mathbf{e}^T(k)\mathbf{e}(k), \tag{11.12}$$

where the expectation of the squared error has been replaced by the squared error at iteration k.

The steepest descent algorithm for the approximate mean square error (stochastic gradient descent) is

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha \frac{\partial \hat{F}}{\partial w_{i,j}^m}, \tag{11.13}$$

$$b_i^m(k+1) = b_i^m(k) - \alpha \frac{\partial \hat{F}}{\partial b_i^m}, \tag{11.14}$$

where α is the learning rate.

So far, this development is identical to that for the LMS algorithm. Now we come to the difficult part – the computation of the partial derivatives.

Chain Rule

For a single-layer linear network (the ADALINE) these partial derivatives are conveniently computed using Eq. (10.33) and Eq. (10.34). For the multilayer network the error is not an explicit function of the weights in the hidden layers, therefore these derivatives are not computed so easily.

Because the error is an indirect function of the weights in the hidden layers, we will use the chain rule of calculus to calculate the derivatives. To review the chain rule, suppose that we have a function f that is an explicit function only of the variable n. We want to take the derivative of f with respect to a third variable w. The chain rule is then:

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw}. \tag{11.15}$$



For example, if

$$f(n) = e^n \text{ and } n = 2w, \text{ so that } f(n(w)) = e^{2w}, \tag{11.16}$$

then

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw} = (e^n)(2). \tag{11.17}$$

We will use this concept to find the derivatives in Eq. (11.13) and Eq. (11.14):

$$\frac{\partial \hat{F}}{\partial w_{i,j}^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m}, \tag{11.18}$$

$$\frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m}. \tag{11.19}$$

The second term in each of these equations can be easily computed, since the net input to layer m is an explicit function of the weights and bias in that layer:

$$n_i^m = \sum_{j=1}^{S^{m-1}} w_{i,j}^m a_j^{m-1} + b_i^m . \tag{11.20}$$

Therefore

$$\frac{\partial n_i^m}{\partial w_{i,j}^m} = a_j^{m-1}, \quad \frac{\partial n_i^m}{\partial b_i^m} = 1 . \tag{11.21}$$

If we now define

$$s_i^m \equiv \frac{\partial \hat{F}}{\partial n_i^m}, \tag{11.22}$$

Sensitivity

(the *sensitivity* of $\hat{F}$ to changes in the ith element of the net input at layer m), then Eq. (11.18) and Eq. (11.19) can be simplified to

$$\frac{\partial \hat{F}}{\partial w_{i,j}^m} = s_i^m a_j^{m-1}, \tag{11.23}$$

$$\frac{\partial \hat{F}}{\partial b_i^m} = s_i^m . \tag{11.24}$$

We can now express the approximate steepest descent algorithm as

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha s_i^m a_j^{m-1}, \tag{11.25}$$

$$b_i^m(k+1) = b_i^m(k) - \alpha s_i^m . \tag{11.26}$$

In matrix form this becomes:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T, \tag{11.27}$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha \mathbf{s}^m, \tag{11.28}$$

where

$$\mathbf{s}^m \equiv \frac{\partial \hat{F}}{\partial \mathbf{n}^m} = \begin{bmatrix} \frac{\partial \hat{F}}{\partial n_1^m} \\ \frac{\partial \hat{F}}{\partial n_2^m} \\ \vdots \\ \frac{\partial \hat{F}}{\partial n_{S^m}^m} \end{bmatrix}. \tag{11.29}$$

(Note the close relationship between this algorithm and the LMS algorithm of Eq. (10.33) and Eq. (10.34)).

Backpropagating the Sensitivities

It now remains for us to compute the sensitivities $\mathbf{s}^m$, which requires another application of the chain rule. It is this process that gives us the term *backpropagation*, because it describes a recurrence relationship in which the sensitivity at layer m is computed from the sensitivity at layer $m+1$.

To derive the recurrence relationship for the sensitivities, we will use the following Jacobian matrix:

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} \equiv \begin{bmatrix} \frac{\partial n_1^{m+1}}{\partial n_1^m} & \frac{\partial n_1^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_1^{m+1}}{\partial n_{S^m}^m} \\ \frac{\partial n_2^{m+1}}{\partial n_1^m} & \frac{\partial n_2^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_2^{m+1}}{\partial n_{S^m}^m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_1^m} & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_{S^m}^m} \end{bmatrix}. \tag{11.30}$$

Next we want to find an expression for this matrix. Consider the i,j element of the matrix:

$$\frac{\partial n_i^{m+1}}{\partial n_j^m} = \frac{\partial\left(\sum_{l=1}^{s^m} w_{i,l}^{m+1} a_l^m + b_i^{m+1}\right)}{\partial n_j^m} = w_{i,j}^{m+1}\frac{\partial a_j^m}{\partial n_j^m}$$

$$= w_{i,j}^{m+1}\frac{\partial f^m(n_j^m)}{\partial n_j^m} = w_{i,j}^{m+1}\dot{f}^m(n_j^m), \tag{11.31}$$

where

$$\dot{f}^m(n_j^m) = \frac{\partial f^m(n_j^m)}{\partial n_j^m}. \tag{11.32}$$

Therefore the Jacobian matrix can be written

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \mathbf{W}^{m+1}\dot{\mathbf{F}}^m(\mathbf{n}^m), \tag{11.33}$$

where

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \dots & 0 \\ 0 & \dot{f}^m(n_2^m) & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \dot{f}^m(n_{S^m}^m) \end{bmatrix}. \tag{11.34}$$

We can now write out the recurrence relation for the sensitivity by using the chain rule in matrix form:

$$\mathbf{s}^m = \frac{\partial \hat{F}}{\partial \mathbf{n}^m} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m}\right)^T \frac{\partial \hat{F}}{\partial \mathbf{n}^{m+1}} = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \frac{\partial \hat{F}}{\partial \mathbf{n}^{m+1}}$$

$$= \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{s}^{m+1}. \tag{11.35}$$

Now we can see where the backpropagation algorithm derives its name. The sensitivities are propagated backward through the network from the last layer to the first layer:

$$\mathbf{s}^M \rightarrow \mathbf{s}^{M-1} \rightarrow \dots \rightarrow \mathbf{s}^2 \rightarrow \mathbf{s}^1. \tag{11.36}$$

At this point it is worth emphasizing that the backpropagation algorithm uses the same approximate steepest descent technique that we used in the LMS algorithm. The only complication is that in order to compute the gradient we need to first backpropagate the sensitivities. The beauty of backpropagation is that we have a very efficient implementation of the chain rule.

We still have one more step to make in order to complete the backpropagation algorithm. We need the starting point, $\mathbf{s}^M$, for the recurrence relation of Eq. (11.35). This is obtained at the final layer:

$$s_i^M = \frac{\partial \hat{F}}{\partial n_i^M} = \frac{\partial (\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})}{\partial n_i^M} = \frac{\partial \sum_{j=1}^{S^M} (t_j - a_j)^2}{\partial n_i^M} = -2(t_i - a_i)\frac{\partial a_i}{\partial n_i^M}. \quad (11.37)$$

Now, since

$$\frac{\partial a_i}{\partial n_i^M} = \frac{\partial a_i^M}{\partial n_i^M} = \frac{\partial f^M(n_i^M)}{\partial n_i^M} = \dot{f}^M(n_i^M), \quad (11.38)$$

we can write

$$s_i^M = -2(t_i - a_i)\dot{f}^M(n_i^M). \quad (11.39)$$

This can be expressed in matrix form as

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t}-\mathbf{a}). \quad (11.40)$$

Summary

Let's summarize the backpropagation algorithm. The first step is to propagate the input forward through the network:

$$\mathbf{a}^0 = \mathbf{p}, \quad (11.41)$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \text{ for } m = 0, 1, \ldots, M-1, \quad (11.42)$$

$$\mathbf{a} = \mathbf{a}^M. \quad (11.43)$$

The next step is to propagate the sensitivities backward through the network:

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t}-\mathbf{a}), \quad (11.44)$$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1}, \text{ for } m = M-1, \ldots, 2, 1. \tag{11.45}$$

Finally, the weights and biases are updated using the approximate steepest descent rule:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\mathbf{s}^m(\mathbf{a}^{m-1})^T, \tag{11.46}$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\mathbf{s}^m. \tag{11.47}$$

Example

To illustrate the backpropagation algorithm, let's choose a network and apply it to a particular problem. To begin, we will use the 1-2-1 network that we discussed earlier in this chapter. For convenience we have reproduced the network in Figure 11.8.

Next we want to define a problem for the network to solve. Suppose that we want to use the network to approximate the function

$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right) \text{ for } -2 \le p \le 2. \tag{11.48}$$

To obtain our training set we will evaluate this function at several values of p.

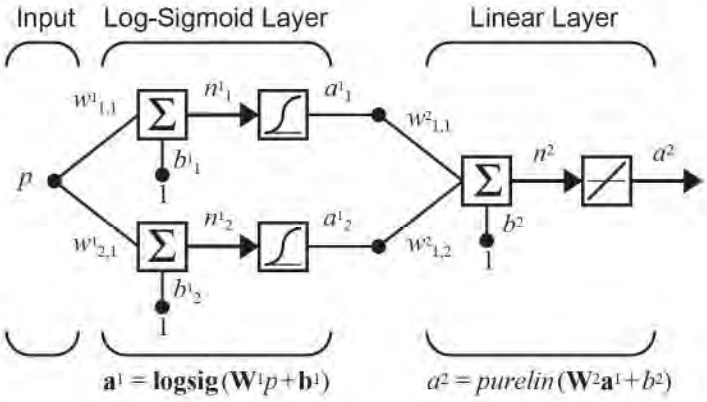


Figure 11.8 Example Function Approximation Network

Before we begin the backpropagation algorithm we need to choose some initial values for the network weights and biases. Generally these are chosen to be small random values. In the next chapter we will discuss some reasons for this. For now let's choose the values

$$\mathbf{W}^1(0) = \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix}, \mathbf{b}^1(0) = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix}, \mathbf{W}^2(0) = \begin{bmatrix} 0.09 & -0.17 \end{bmatrix}, \mathbf{b}^2(0) = \begin{bmatrix} 0.48 \end{bmatrix}.$$

The response of the network for these initial values is illustrated in Figure 11.9, along with the sine function we wish to approximate.

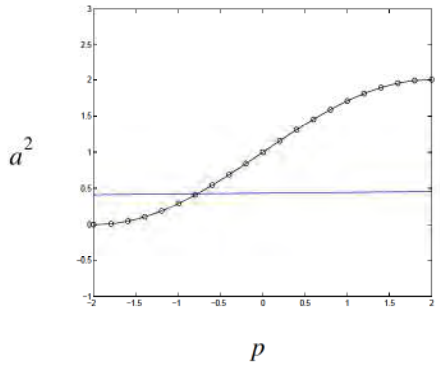


Figure 11.9 Initial Network Response

Next, we need to select a training set $\{p_1, t_1\}, \{p_2, t_2\}, \ldots, \{p_Q, t_Q\}$. In this case, we will sample the function at 21 points in the range [-2,2] at equally spaced intervals of 0.2. The training points are indicated by the circles in Figure 11.9.

Now we are ready to start the algorithm. The training points can be presented in any order, but they are often chosen randomly. For our initial input we will choose $p = 1$, which is the 16th training point:

$$a^0 = p = 1.$$

The output of the first layer is then

$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{a}^0 + \mathbf{b}^1) = \mathbf{logsig}\left(\begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix}[1] + \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix}\right) = \mathbf{logsig}\left(\begin{bmatrix} -0.75 \\ -0.54 \end{bmatrix}\right)$$

$$= \begin{bmatrix} \frac{1}{1+e^{0.75}} \\ \frac{1}{1+e^{0.54}} \end{bmatrix} = \begin{bmatrix} 0.321 \\ 0.368 \end{bmatrix}.$$

The second layer output is

$$a^2 = f^2(\mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2) = purelin\left(\begin{bmatrix}0.09 & -0.17\end{bmatrix}\begin{bmatrix}0.321\\0.368\end{bmatrix} + \begin{bmatrix}0.48\end{bmatrix}\right) = \begin{bmatrix}0.446\end{bmatrix}.$$

The error would then be

$$e = t - a = \left\{1 + \sin\left(\frac{\pi}{4}p\right)\right\} - a^2 = \left\{1 + \sin\left(\frac{\pi}{4}1\right)\right\} - 0.446 = 1.261.$$

The next stage of the algorithm is to backpropagate the sensitivities. Before we begin the backpropagation, recall that we will need the derivatives of the transfer functions, $\dot{f}^1(n)$ and $\dot{f}^2(n)$. For the first layer

$$\dot{f}^1(n) = \frac{d}{dn}\left(\frac{1}{1+e^{-n}}\right) = \frac{e^{-n}}{(1+e^{-n})^2} = \left(1 - \frac{1}{1+e^{-n}}\right)\left(\frac{1}{1+e^{-n}}\right) = (1-a^1)(a^1).$$

For the second layer we have

$$\dot{f}^2(n) = \frac{d}{dn}(n) = 1.$$

We can now perform the backpropagation. The starting point is found at the second layer, using Eq. (11.44):

$$\mathbf{s}^2 = -2\dot{\mathbf{F}}^2(\mathbf{n}^2)(\mathbf{t}-\mathbf{a}) = -2\left[\dot{f}^2(n^2)\right](1.261) = -2\begin{bmatrix}1\end{bmatrix}(1.261) = \begin{bmatrix}-2.522\end{bmatrix}.$$

The first layer sensitivity is then computed by backpropagating the sensitivity from the second layer, using Eq. (11.45):

$$\mathbf{s}^1 = \dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{W}^2)^T\mathbf{s}^2 = \begin{bmatrix}(1-a_1^1)(a_1^1) & 0\\ 0 & (1-a_2^1)(a_2^1)\end{bmatrix}\begin{bmatrix}0.09\\-0.17\end{bmatrix}\begin{bmatrix}-2.522\end{bmatrix}$$

$$= \begin{bmatrix}(1-0.321)(0.321) & 0\\ 0 & (1-0.368)(0.368)\end{bmatrix}\begin{bmatrix}0.09\\-0.17\end{bmatrix}\begin{bmatrix}-2.522\end{bmatrix}$$

$$= \begin{bmatrix}0.218 & 0\\ 0 & 0.233\end{bmatrix}\begin{bmatrix}-0.227\\0.429\end{bmatrix} = \begin{bmatrix}-0.0495\\0.0997\end{bmatrix}.$$

The final stage of the algorithm is to update the weights. For simplicity, we will use a learning rate $\alpha = 0.1$. (In Chapter 12 the choice of learning rate will be discussed in more detail.) From Eq. (11.46) and Eq. (11.47) we have

$$\mathbf{W}^2(1) = \mathbf{W}^2(0) - \alpha \mathbf{s}^2 (\mathbf{a}^1)^T = \begin{bmatrix} 0.09 & -0.17 \end{bmatrix} - 0.1 \begin{bmatrix} -2.522 \end{bmatrix} \begin{bmatrix} 0.321 & 0.368 \end{bmatrix}$$

$$= \begin{bmatrix} 0.171 & -0.0772 \end{bmatrix},$$

$$\mathbf{b}^2(1) = \mathbf{b}^2(0) - \alpha \mathbf{s}^2 = \begin{bmatrix} 0.48 \end{bmatrix} - 0.1 \begin{bmatrix} -2.522 \end{bmatrix} = \begin{bmatrix} 0.732 \end{bmatrix},$$

$$\mathbf{W}^1(1) = \mathbf{W}^1(0) - \alpha \mathbf{s}^1 (\mathbf{a}^0)^T = \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix} - 0.1 \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} = \begin{bmatrix} -0.265 \\ -0.420 \end{bmatrix},$$

$$\mathbf{b}^1(1) = \mathbf{b}^1(0) - \alpha \mathbf{s}^1 = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix} - 0.1 \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix} = \begin{bmatrix} -0.475 \\ -0.140 \end{bmatrix}.$$

This completes the first iteration of the backpropagation algorithm. We next proceed to randomly choose another input from the training set and perform another iteration of the algorithm. We continue to iterate until the difference between the network response and the target function reaches some acceptable level. (Note that this will generally take many passes through the entire training set.) We will discuss convergence criteria in more detail in Chapter 12.



To experiment with the backpropagation calculation for this two-layer network, use the MATLAB® Neural Network Design Demonstration Backpropagation Calculation (**nnd11bc**).

Batch vs. Incremental Training

Incremental Training

Batch Training

The algorithm described above is the stochastic gradient descent algorithm, which involves "on-line" or *incremental training*, in which the network weights and biases are updated after each input is presented (as with the LMS algorithm of Chapter 10). It is also possible to perform *batch training*, in which the complete gradient is computed (after all inputs are applied to the network) before the weights and biases are updated. For example, if each input occurs with equal probability, the mean square error performance index can be written

$$F(\mathbf{x}) = E[\mathbf{e}^T\mathbf{e}] = E[(\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})] = \frac{1}{Q}\sum_{q=1}^{Q}(\mathbf{t}_q-\mathbf{a}_q)^T(\mathbf{t}_q-\mathbf{a}_q). \quad (11.49)$$

The total gradient of this performance index is

$$\nabla F(\mathbf{x}) = \nabla\left\{\frac{1}{Q}\sum_{q=1}^{Q}(\mathbf{t}_q - \mathbf{a}_q)^T(\mathbf{t}_q - \mathbf{a}_q)\right\} = \frac{1}{Q}\sum_{q=1}^{Q}\nabla\{(\mathbf{t}_q - \mathbf{a}_q)^T(\mathbf{t}_q - \mathbf{a}_q)\}. \quad (11.50)$$

Therefore, the total gradient of the mean square error is the mean of the gradients of the individual squared errors. Therefore, to implement a batch version of the backpropagation algorithm, we would step through Eq. (11.41) through Eq. (11.45) for all of the inputs in the training set. Then, the individual gradients would be averaged to get the total gradient. The update equations for the batch steepest descent algorithm would then be

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \frac{\alpha}{Q}\sum_{q=1}^{Q}\mathbf{s}_q^m(\mathbf{a}_q^{m-1})^T, \quad (11.51)$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \frac{\alpha}{Q}\sum_{q=1}^{Q}\mathbf{s}_q^m. \quad (11.52)$$

Using Backpropagation

In this section we will present some issues relating to the practical implementation of backpropagation. We will discuss the choice of network architecture, and problems with network convergence and generalization. (We will discuss implementation issues again in Chapter 12, which investigates procedures for improving the algorithm.)

Choice of Network Architecture

As we discussed earlier in this chapter, multilayer networks can be used to approximate almost any function, if we have enough neurons in the hidden layers. However, we cannot say, in general, how many layers or how many neurons are necessary for adequate performance. In this section we want to use a few examples to provide some insight into this problem.



For our first example let's assume that we want to approximate the following functions:

$$g(p) = 1 + \sin\left(\frac{i\pi}{4}p\right) \text{ for } -2 \le p \le 2, \quad (11.53)$$

where i takes on the values 1, 2, 4 and 8. As i is increased, the function becomes more complex, because we will have more periods of the sine wave over the interval $-2 \le p \le 2$. It will be more difficult for a neural network with a fixed number of neurons in the hidden layers to approximate $g(p)$ as i is increased.

For this first example we will use a 1-3-1 network, where the transfer function for the first layer is log-sigmoid and the transfer function for the second layer is linear. Recall from our example on page 11-5 that this type of two-layer network can produce a response that is a sum of three log-sigmoid functions (or as many log-sigmoids as there are neurons in the hidden layer). Clearly there is a limit to how complex a function this network can implement. Figure 11.10 illustrates the response of the network after it has been trained to approximate $g(p)$ for $i = 1, 2, 4, 8$. The final network responses are shown by the blue lines.

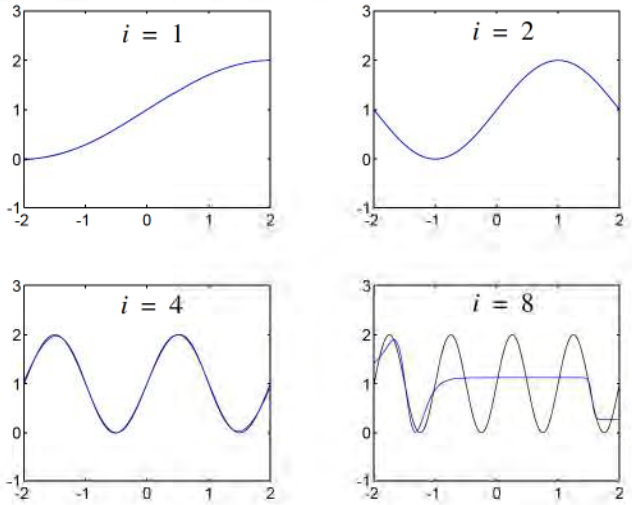


Figure 11.10 Function Approximation Using a 1-3-1 Network

We can see that for $i = 4$ the 1-3-1 network reaches its maximum capability. When $i > 4$ the network is not capable of producing an accurate approximation of $g(p)$. In the bottom right graph of Figure 11.10 we can see how the 1-3-1 network attempts to approximate $g(p)$ for $i = 8$. The mean square error between the network response and $g(p)$ is minimized, but the network response is only able to match a small part of the function.



In the next example we will approach the problem from a slightly different perspective. This time we will pick one function $g(p)$ and then use larger and larger networks until we are able to accurately represent the function. For $g(p)$ we will use

$$g(p) = 1 + \sin\left(\frac{6\pi}{4}p\right) \text{ for } -2 \le p \le 2. \tag{11.54}$$

To approximate this function we will use two-layer networks, where the transfer function for the first layer is log-sigmoid and the transfer function for the second layer is linear (1-S^1-1 networks). As we discussed earlier in

this chapter, the response of this network is a superposition of S^1 sigmoid functions.

Figure 11.11 illustrates the network response as the number of neurons in the first layer (hidden layer) is increased. Unless there are at least five neurons in the hidden layer the network cannot accurately represent $g(p)$.

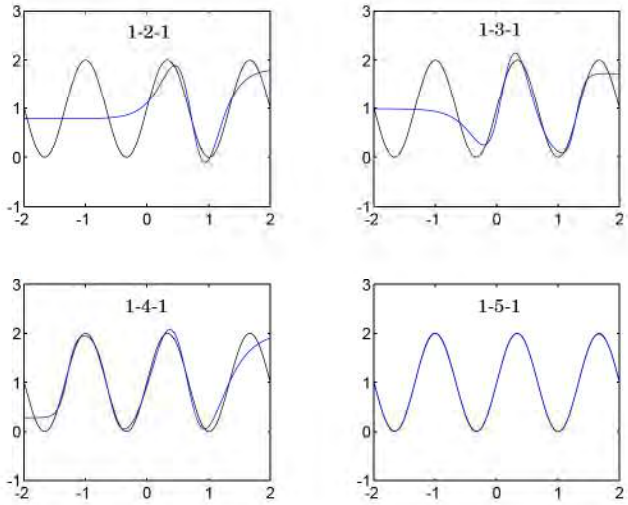


Figure 11.11 Effect of Increasing the Number of Hidden Neurons

To summarize these results, a 1-S^1-1 network, with sigmoid neurons in the hidden layer and linear neurons in the output layer, can produce a response that is a superposition of S^1 sigmoid functions. If we want to approximate a function that has a large number of inflection points, we will need to have a large number of neurons in the hidden layer.



Use the MATLAB® Neural Network Design Demonstration Function Approximation (`nnd11fa`) *to develop more insight into the capability of a two-layer network.*

Convergence

In the previous section we presented some examples in which the network response did not give an accurate approximation to the desired function, even though the backpropagation algorithm produced network parameters that minimized mean square error. This occurred because the capabilities of the network were inherently limited by the number of hidden neurons it contained. In this section we will provide an example in which the network is capable of approximating the function, but the learning algorithm does not produce network parameters that produce an accurate approximation. In the next chapter we will discuss this problem in more detail and explain why it occurs. For now we simply want to illustrate the problem.

The function that we want the network to approximate is

$$g(p) = 1 + \sin(\pi p) \text{ for } -2 \le p \le 2 . \tag{11.55}$$

To approximate this function we will use a 1-3-1 network, where the transfer function for the first layer is log-sigmoid and the transfer function for the second layer is linear.

Figure 11.12 illustrates a case where the learning algorithm converges to a solution that minimizes mean square error. The thin blue lines represent intermediate iterations, and the thick blue line represents the final solution, when the algorithm has converged. (The numbers next to each curve indicate the sequence of iterations, where 0 represents the initial condition and 5 represents the final solution. The numbers do not correspond to the iteration number. There were many iterations for which no curve is represented. The numbers simply indicate an ordering.)

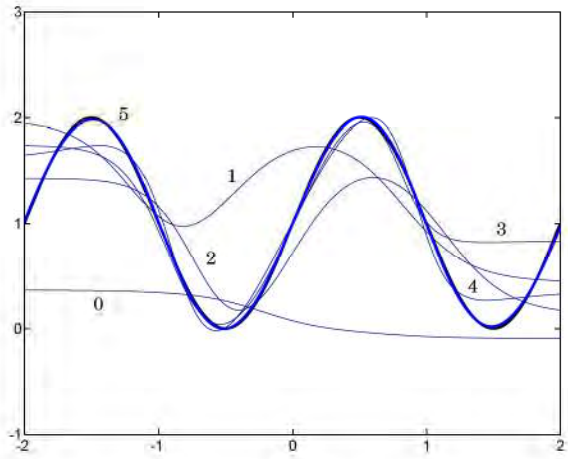


Figure 11.12 Convergence to a Global Minimum

Figure 11.13 illustrates a case where the learning algorithm converges to a solution that does not minimize mean square error. The thick blue line (marked with a 5) represents the network response at the final iteration. The gradient of the mean square error is zero at the final iteration, therefore we have a local minima, but we know that a better solution exists, as evidenced by Figure 11.12. The only difference between this result and the result shown in Figure 11.12 is the initial condition. From one initial condition the algorithm converged to a global minimum point, while from another initial condition the algorithm converged to a local minimum point.

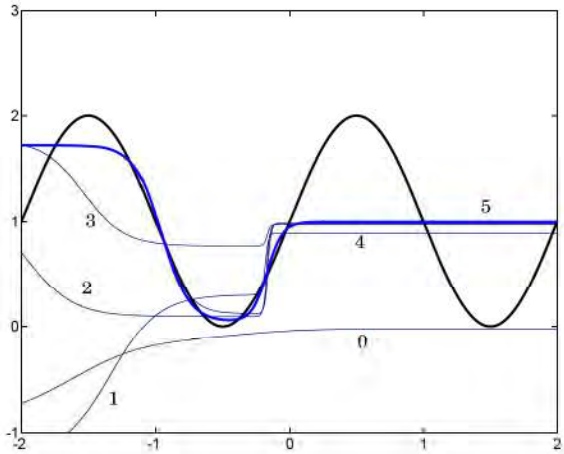


Figure 11.13 Convergence to a Local Minimum

Note that this result could not have occurred with the LMS algorithm. The mean square error performance index for the ADALINE network is a quadratic function with a single minimum point (under most conditions). Therefore the LMS algorithm is guaranteed to converge to the global minimum as long as the learning rate is small enough. The mean square error for the multilayer network is generally much more complex and has many local minima (as we will see in the next chapter). When the backpropagation algorithm converges we cannot be sure that we have an optimum solution. It is best to try several different initial conditions in order to ensure that an optimum solution has been obtained.

Generalization

In most cases the multilayer network is trained with a finite number of examples of proper network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}. \tag{11.56}$$

This training set is normally representative of a much larger class of possible input/output pairs. It is important that the network successfully *generalize* what it has learned to the total population.



For example, suppose that the training set is obtained by sampling the following function:

$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right), \tag{11.57}$$

at the points $p = -2, -1.6, -1.2, \ldots, 1.6, 2$. (There are a total of 11 input/target pairs.) In Figure 11.14 we see the response of a 1-2-1 network that has

been trained on this data. The black line represents $g(p)$, the blue line represents the network response, and the '+' symbols indicate the training set.

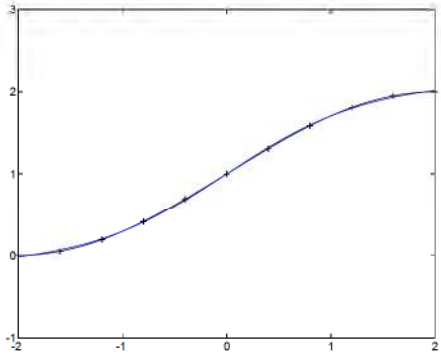


Figure 11.14 1-2-1 Network Approximation of $g(p)$

We can see that the network response is an accurate representation of $g(p)$. If we were to find the response of the network at a value of p that was not contained in the training set (e.g., $p = -0.2$), the network would still produce an output close to $g(p)$. This network generalizes well.

Now consider Figure 11.15, which shows the response of a 1-9-1 network that has been trained on the same data set. Note that the network response accurately models $g(p)$ at all of the training points. However, if we compute the network response at a value of p not contained in the training set (e.g., $p = -0.2$) the network might produce an output far from the true response $g(p)$. This network does not generalize well.

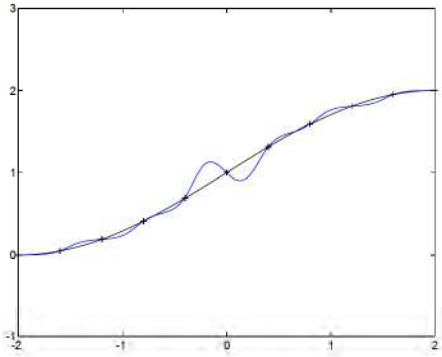


Figure 11.15 1-9-1 Network Approximation of $g(p)$

The 1-9-1 network has too much flexibility for this problem; it has a total of 28 adjustable parameters (18 weights and 10 biases), and yet there are only 11 data points in the training set. The 1-2-1 network has only 7 param-

eters and is therefore much more restricted in the types of functions that it can implement.

For a network to be able to generalize, it should have fewer parameters than there are data points in the training set. In neural networks, as in all modeling problems, we want to use the simplest network that can adequately represent the training set. Don't use a bigger network when a smaller network will work (a concept often referred to as Ockham's Razor).

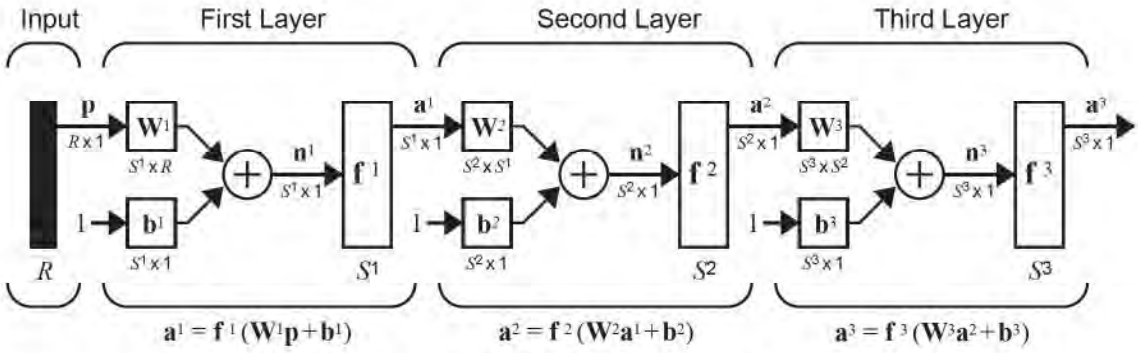
An alternative to using the simplest network is to stop the training before the network overfits. A reference to this procedure and other techniques to improve generalization are given in Chapter 13.



To experiment with generalization in neural networks, use the MATLAB® Neural Network Design Demonstration Generalization (`nnd11gn`).

Summary of Results

Multilayer Network



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

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

Backpropagation Algorithm

Performance Index

$$F(\mathbf{x}) = E[\mathbf{e}^T\mathbf{e}] = E[(\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})]$$

Approximate Performance Index

$$\hat{F}(\mathbf{x}) = \mathbf{e}^T(k)\mathbf{e}(k) = (\mathbf{t}(k)-\mathbf{a}(k))^T(\mathbf{t}(k)-\mathbf{a}(k))$$

Sensitivity

$$\mathbf{s}^m \equiv \frac{\partial \hat{F}}{\partial \mathbf{n}^m} = \begin{bmatrix} \frac{\partial \hat{F}}{\partial n_1^m} \\ \frac{\partial \hat{F}}{\partial n_2^m} \\ \vdots \\ \frac{\partial \hat{F}}{\partial n_{S^m}^m} \end{bmatrix}$$

Forward Propagation

$$\mathbf{a}^0 = \mathbf{p},$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \text{ for } m = 0, 1, \ldots, M-1,$$

$$\mathbf{a} = \mathbf{a}^M.$$

Backward Propagation

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t} - \mathbf{a}),$$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1}, \text{ for } m = M-1, \ldots, 2, 1,$$

where

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \ldots & 0 \\ 0 & \dot{f}^m(n_2^m) & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & & \dot{f}^m(n_{S^m}^m) \end{bmatrix},$$

$$\dot{f}^m(n_j^m) = \frac{\partial f^m(n_j^m)}{\partial n_j^m}.$$

Weight Update (Approximate Steepest Descent)

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\mathbf{s}^m(\mathbf{a}^{m-1})^T,$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\mathbf{s}^m.$$

Solved Problems

P11.1 Consider the two classes of patterns that are shown in Figure P11.1. Class I represents vertical lines and Class II represents horizontal lines.

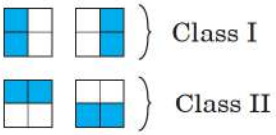


Figure P11.1 Pattern Classes for Problem P11.1

i. Are these categories linearly separable?

ii. Design a multilayer network to distinguish these categories.

i. Let's begin by converting the patterns to vectors by scanning each 2X2 grid one column at a time. Each white square will be represented by a "-1" and each blue square by a "1". The vertical lines (Class I patterns) then become

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} \text{ and } \mathbf{p}_2 = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix},$$

and the horizontal lines (Class II patterns) become

$$\mathbf{p}_3 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \text{ and } \mathbf{p}_4 = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix}.$$

In order for these categories to be linearly separable we must be able to place a hyperplane between the two categories. This means there must be a weight matrix $\mathbf{W}$ and a bias b such that

$$\mathbf{W}\mathbf{p}_1 + b > 0, \ \mathbf{W}\mathbf{p}_2 + b > 0, \ \mathbf{W}\mathbf{p}_3 + b < 0, \ \mathbf{W}\mathbf{p}_4 + b < 0.$$

These conditions can be converted to

$$\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} w_{1,1} + w_{1,2} - w_{1,3} - w_{1,4} \end{bmatrix} > 0,$$

$$\begin{bmatrix} -w_{1,1} - w_{1,2} + w_{1,3} + w_{1,4} \end{bmatrix} > 0,$$

$$\begin{bmatrix} w_{1,1} - w_{1,2} + w_{1,3} - w_{1,4} \end{bmatrix} < 0,$$

$$\begin{bmatrix} -w_{1,1} + w_{1,2} - w_{1,3} + w_{1,4} \end{bmatrix} < 0.$$

The first two conditions reduce to

$$w_{1,1} + w_{1,2} > w_{1,3} + w_{1,4} \text{ and } w_{1,3} + w_{1,4} > w_{1,1} + w_{1,2},$$

which are contradictory. The final two conditions reduce to

$$w_{1,1} + w_{1,3} > w_{1,2} + w_{1,4} \text{ and } w_{1,2} + w_{1,4} > w_{1,1} + w_{1,3},$$

which are also contradictory. Therefore there is no hyperplane that can separate these two categories.

ii. There are many different multilayer networks that could solve this problem. We will design a network by first noting that for the Class I vectors either the first two elements or the last two elements will be "1". The Class II vectors have alternating "1" and "-1" patterns. This leads to the network shown in Figure P11.2.

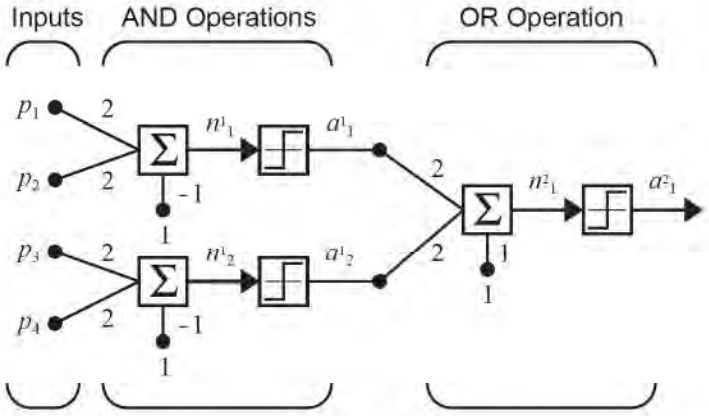


Figure P11.2 Network to Categorize Horizontal and Vertical Lines

The first neuron in the first layer tests the first two elements of the input vector. If they are both "1" it outputs a "1", otherwise it outputs a "-1". The second neuron in the first layer tests the last two elements of the input vector in the same way. Both of the neurons in the first layer perform AND operations. The second layer of the network tests whether either of the outputs of the first layer are "1". It performs an OR operation. In this way, the network will output a "1" if either the first two elements or the last two elements of the input vector are both "1".

P11.2 Figure P11.3 illustrates a classification problem, where Class I vectors are represented by light circles, and Class II vectors are represented by dark circles. These categories are not linearly separable. Design a multilayer network to correctly classify these categories.

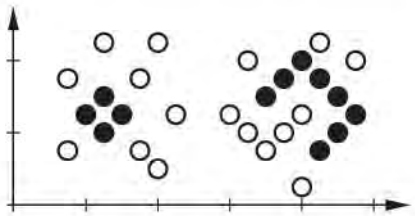


Figure P11.3 Classification Problem

We will solve this problem with a procedure that can be used for arbitrary classification problems. It requires a three-layer network, with hard-limiting neurons in each layer. In the first layer we create a set of linear decision boundaries that separate every Class I vector from every Class II vector. For this problem we used 11 such boundaries. They are shown in Figure P11.4.

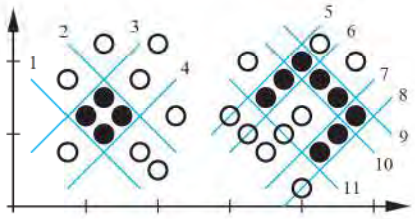


Figure P11.4 First Layer Decision Boundaries

Each row of the weight matrix in the first layer corresponds to one decision boundary. The weight matrix and bias vector for the first layer are

$$(\mathbf{W}^1)^T = \begin{bmatrix} 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & 1 \end{bmatrix},$$

$$(\mathbf{b}^1)^T = \begin{bmatrix} -2 & 3 & 0.5 & 0.5 & -1.75 & 2.25 & -3.25 & 3.75 & 6.25 & -5.75 & -4.75 \end{bmatrix}.$$

(Review Chapters 3, 4 and 10 for procedures for calculating the appropriate weight matrix and bias for a given decision boundary.) Now we can combine the outputs of the 11 first layer neurons into groups with a second layer of AND neurons, such as those we used in the first layer of the network in Problem P11.1. The second layer weight matrix and bias are

$$\mathbf{W}^2 = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{bmatrix}, \quad \mathbf{b}^T = \begin{bmatrix} -3 \\ -3 \\ -3 \\ -3 \end{bmatrix}.$$

The four decision boundaries for the second layer are shown in Figure P11.5. For example, the neuron 2 decision boundary is obtained by combining the boundaries 5, 6, 9 and 11 from layer 1. This can be seen by looking at row 2 of $\mathbf{W}^2$.

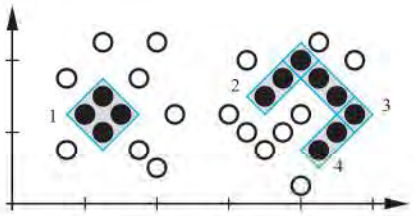


Figure P11.5 Second Layer Decision Regions

In the third layer of the network we will combine together the four decision regions of the second layer into one decision region using an OR operation, just as in the last layer of the network in Problem P11.1. The weight matrix and bias for the third layer are

$$\mathbf{W}^3 = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{b}^3 = \begin{bmatrix} 3 \end{bmatrix}.$$

The complete network is shown in Figure P11.6.

The procedure that we used to develop this network can be used to solve classification problems with arbitrary decision boundaries as long as we have enough neurons in the hidden layers. The idea is to use the first layer to create a number of linear boundaries, which can be combined by using AND neurons in the second layer and OR neurons in the third layer. The decision regions of the second layer are convex, but the final decision boundaries created by the third layer can have arbitrary shapes.

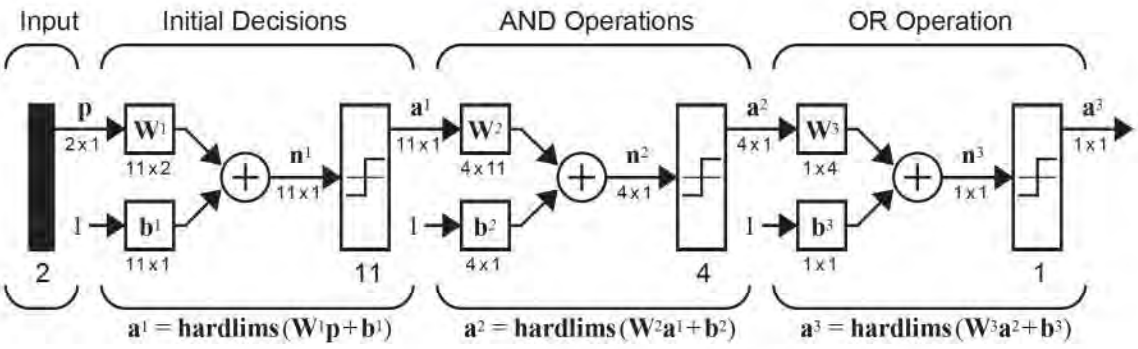


Figure P11.6 Network for Problem P11.2

The final network decision regions are given in Figure P11.7. Any vector in the shaded areas will produce a network output of 1, which corresponds to Class II. All other vectors will produce a network output of -1, which corresponds to Class I.

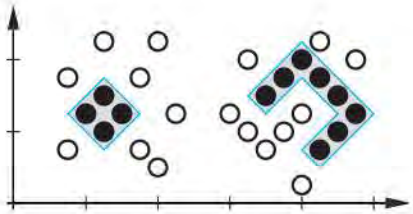


Figure P11.7 Final Decision Regions

P11.3 Show that a multilayer network with linear transfer functions is equivalent to a single-layer linear network.

For a multilayer linear network the forward equations would be

$$\mathbf{a}^1 = \mathbf{W}^1\mathbf{p} + \mathbf{b}^1$$

$$\mathbf{a}^2 = \mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2 = \mathbf{W}^2\mathbf{W}^1\mathbf{p} + [\mathbf{W}^2\mathbf{b}^1 + \mathbf{b}^2],$$

$$\mathbf{a}^3 = \mathbf{W}^3\mathbf{a}^2 + \mathbf{b}^3 = \mathbf{W}^3\mathbf{W}^2\mathbf{W}^1\mathbf{p} + [\mathbf{W}^3\mathbf{W}^2\mathbf{b}^1 + \mathbf{W}^3\mathbf{b}^2 + \mathbf{b}^3].$$

If we continue this process we can see that for an M-layer linear network, the equivalent single-layer linear network would have the following weight matrix and bias vector

$$\mathbf{W} = \mathbf{W}^M\mathbf{W}^{M-1}\ldots\mathbf{W}^2\mathbf{W}^1,$$

$$\mathbf{b} = [\mathbf{W}^M\mathbf{W}^{M-1}\cdots\mathbf{W}^2]\mathbf{b}^1 + [\mathbf{W}^M\mathbf{W}^{M-1}\cdots\mathbf{W}^3]\mathbf{b}^2 + \cdots + \mathbf{b}^M.$$

P11.4 The purpose of this problem is to illustrate the use of the chain rule. Consider the following dynamic system:

$$y(k+1) = f(y(k)).$$

We want to choose the initial condition $y(0)$ so that at some final time $k = K$ the system output $y(K)$ will be as close as possible to some target output t. We will minimize the performance index

$$F(y(0)) = (t - y(K))^2$$

using steepest descent, so we need the gradient

$$\frac{\partial}{\partial y(0)}F(y(0)).$$

Find a procedure for computing this using the chain rule.

The gradient is

$$\frac{\partial}{\partial y(0)}F(y(0)) = \frac{\partial(t-y(K))^2}{\partial y(0)} = 2(t-y(K))\left[-\frac{\partial}{\partial y(0)}y(K)\right].$$

The key term is

$$\left[\frac{\partial}{\partial y(0)}y(K)\right],$$

which cannot be computed directly, since $y(K)$ is not an explicit function of $y(0)$. Let's define an intermediate term

$$r(k) \equiv \frac{\partial}{\partial y(0)}y(k).$$

Then we can use the chain rule:

$$r(k+1) = \frac{\partial}{\partial y(0)}y(k+1) = \frac{\partial y(k+1)}{\partial y(k)} \times \frac{\partial y(k)}{\partial y(0)} = \frac{\partial y(k+1)}{\partial y(k)} \times r(k).$$

From the system dynamics we know

$$\frac{\partial y(k+1)}{\partial y(k)} = \frac{\partial f(y(k))}{\partial y(k)} = \dot{f}(y(k)).$$

Therefore the recursive equation for the computation of $r(k)$ is

$$r(k+1) = \dot{f}(y(k))r(k).$$

This is initialized at $k = 0$:

$$r(0) = \frac{\partial y(0)}{\partial y(0)} = 1.$$

The total procedure for computing the gradient is then

$$r(0) = 1,$$

$$r(k+1) = \dot{f}(y(k))r(k), \text{ for } k = 0, 1, \ldots, K-1,$$

$$\frac{\partial}{\partial y(0)}F(y(0)) = 2(t - y(K))[-r(K)].$$

P11.5 Consider the two-layer network shown in Figure P11.8. The initial weights and biases are set to

$$w^1 = 1,\ b^1 = 1,\ w^2 = -2,\ b^2 = 1.$$

An input/target pair is given to be

$$((p = 1),(t = 1)).$$

i. Find the squared error $(e)^2$ as an explicit function of all weights and biases.

ii. Using part (i) find $\partial(e)^2/\partial w^1$ at the initial weights and biases.

iii. Repeat part (ii) using backpropagation and compare results.

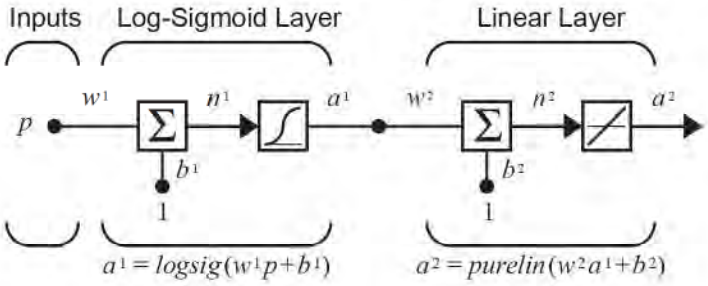


Figure P11.8 Two-Layer Network for Problem P11.5

i. The squared error is given by

$$(e)^2 = (t-a^2)^2 = \left(t - \left\{ w^2 \frac{1}{(1+\exp(-(w^1 p + b^1)))} + b^2 \right\}\right)^2 .$$

ii. The derivative is

$$\frac{\partial (e)^2}{\partial w^1} = 2e\frac{\partial e}{\partial w^1} = 2e\left\{ w^2 \frac{1}{(1+\exp(-(w^1 p + b^1)))^2} \exp(-(w^1 p + b^1))(-p) \right\}$$

To evaluate this at the initial weights and biases we find

$$a^1 = \frac{1}{(1+\exp(-(w^1 p + b^1)))} = \frac{1}{(1+\exp(-(1(1)+1)))} = 0.8808$$

$$a^2 = w^2 a^1 + b^2 = (-2)0.8808 + 1 = -0.7616$$

$$e = (t-a^2) = (1-(-0.7616)) = 1.7616$$

$$\frac{\partial (e)^2}{\partial w^1} = 2e\left\{ w^2 \frac{1}{(1+\exp(-(w^1 p + b^1)))^2} \exp(-(w^1 p + b^1))(-p) \right\}$$

$$= 2(1.7616)\left\{ (-2) \frac{1}{(1+\exp(-(1(1)+1)))^2} \exp(-(1(1)+1))(-1) \right\}$$

$$= 3.5232\left(0.2707 \frac{1}{(1.289)^2} \right) = 0.7398 .$$

iii. To backpropagate the sensitivities we use Eq. (11.44) and Eq. (11.45):

$$\mathbf{s}^2 = -2\dot{\mathbf{F}}^2(\mathbf{n}^2)(\mathbf{t}-\mathbf{a}) = -2(1)(1-(-0.7616)) = -3.5232 ,$$

$$\mathbf{s}^1 = \dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{W}^2)^T \mathbf{s}^2 = [a^1(1-a^1)](-2)\mathbf{s}^2$$

$$= [0.8808(1-0.8808)](-2)(-3.5232) = 0.7398 .$$

From Eq. (11.23) we can compute $\partial (e)^2 / \partial w^1$:

$$\frac{\partial (e)^2}{\partial w^1} = s^1 a^0 = s^1 p = (0.7398)(1) = 0.7398 .$$

This agrees with our result from part (ii).

P11.6 Earlier in this chapter we showed that if the neuron transfer function is log-sigmoid,

$$a = f(n) = \frac{1}{1+e^{-n}},$$

then the derivative can be conveniently computed by

$$\dot{f}(n) = a(1-a).$$

Find a convenient way to compute the derivative for the hyperbolic tangent sigmoid:

$$a = f(n) = tansig(n) = \frac{e^{n}-e^{-n}}{e^{n}+e^{-n}}.$$

Computing the derivative directly we find

$$\dot{f}(n) = \frac{df(n)}{dn} = \frac{d}{dn}\left(\frac{e^{n}-e^{-n}}{e^{n}+e^{-n}}\right) = -\frac{e^{n}-e^{-n}}{(e^{n}+e^{-n})^{2}}(e^{n}-e^{-n}) + \frac{e^{n}+e^{-n}}{e^{n}+e^{-n}}$$

$$= 1 - \frac{(e^{n}-e^{-n})^{2}}{(e^{n}+e^{-n})^{2}} = 1-(a)^{2}.$$

P11.7 For the network shown in Figure P11.9 the initial weights and biases are chosen to be

$$w^{1}(0) = -1,\ b^{1}(0) = 1,\ w^{2}(0) = -2,\ b^{2}(0) = 1.$$

An input/target pair is given to be

$$((p=-1),(t=1)).$$

Perform one iteration of backpropagation with $\alpha = 1$.

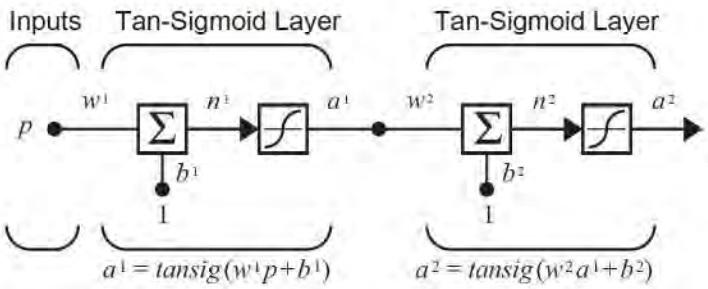


Figure P11.9 Two-Layer Tan-Sigmoid Network

The first step is to propagate the input through the network.

$$n^1 = w^1 p + b^1 = (-1)(-1) + 1 = 2$$

$$a^1 = tansig(n^1) = \frac{\exp(n^1) - \exp(-n^1)}{\exp(n^1) + \exp(-n^1)} = \frac{\exp(2) - \exp(-2)}{\exp(2) + \exp(-2)} = 0.964$$

$$n^2 = w^2 a^1 + b^2 = (-2)(0.964) + 1 = -0.928$$

$$a^2 = tansig(n^2) = \frac{\exp(n^2) - \exp(-n^2)}{\exp(n^2) + \exp(-n^2)} = \frac{\exp(-0.928) - \exp(0.928)}{\exp(-0.928) + \exp(0.928)}$$

$$= -0.7297$$

$$e = (t - a^2) = (1 - (-0.7297)) = 1.7297$$

Now we backpropagate the sensitivities using Eq. (11.44) and Eq. (11.45).

$$\mathbf{s}^2 = -2\dot{\mathbf{F}}^2(\mathbf{n}^2)(\mathbf{t} - \mathbf{a}) = -2[1 - (a^2)^2](e) = -2[1 - (-0.7297)^2]1.7297$$

$$= -1.6175$$

$$\mathbf{s}^1 = \dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{W}^2)^T\mathbf{s}^2 = [1 - (a^1)^2]w^2\mathbf{s}^2 = [1 - (0.964)^2](-2)(-1.6175)$$

$$= 0.2285$$

Finally, the weights and biases are updated using Eq. (11.46) and Eq. (11.47):

$$w^2(1) = w^2(0) - \alpha s^2(a^1)^T = (-2) - 1(-1.6175)(0.964) = -0.4407,$$

$$w^1(1) = w^1(0) - \alpha s^1 (a^0)^T = (-1) - 1(0.2285)(-1) = -0.7715,$$

$$b^2(1) = b^2(0) - \alpha s^2 = 1 - 1(-1.6175) = 2.6175,$$

$$b^1(1) = b^1(0) - \alpha s^1 = 1 - 1(0.2285) = 0.7715.$$

P11.8 In Figure P11.10 we have a network that is a slight modification to the standard two-layer feedforward network. It has a connection from the input directly to the second layer. Derive the backpropagation algorithm for this network.

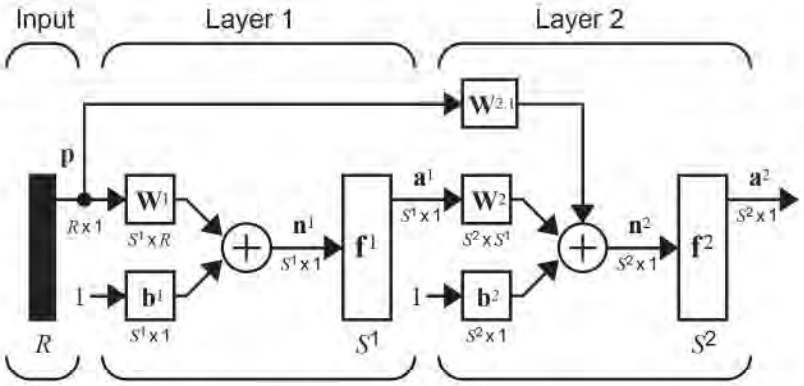


Figure P11.10 Network with Bypass Connection

We begin with the forward equations:

$$\mathbf{n}^1 = \mathbf{W}^1\mathbf{p} + \mathbf{b}^1,$$

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

$$\mathbf{n}^2 = \mathbf{W}^2\mathbf{a}^1 + \mathbf{W}^{2,1}\mathbf{p} + \mathbf{b}^2,$$

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

The backpropagation equations for the sensitivities will not change from those for a standard two-layer network. The sensitivities are the derivatives of the squared error with respect to the net inputs; these derivatives don't change, since we are simply adding a term to the net input.

Next we need the elements of the gradient for the weight update equations. For the standard weights and biases we have

$$\frac{\partial \hat{F}}{\partial w_{i,j}^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m} = s_i^m a_j^{m-1},$$

$$\frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m} = s_i^m.$$

Therefore the update equations for $\mathbf{W}^1$, $\mathbf{b}^1$, $\mathbf{W}^2$ and $\mathbf{b}^2$ do not change. We do need an additional equation for $\mathbf{W}^{2,1}$:

$$\frac{\partial \hat{F}}{\partial w_{i,j}^{2,1}} = \frac{\partial \hat{F}}{\partial n_i^2} \times \frac{\partial n_i^2}{\partial w_{i,j}^{2,1}} = s_i^2 \times \frac{\partial n_i^2}{\partial w_{i,j}^{2,1}}.$$

To find the derivative on the right-hand side of this equation note that

$$n_i^2 = \sum_{j=1}^{S^1} w_{i,j}^2 a_j^1 + \sum_{j=1}^{R} w_{i,j}^{2,1} p_j + b_i^2.$$

Therefore

$$\frac{\partial n_i^2}{\partial w_{i,j}^{2,1}} = p_j \text{ and } \frac{\partial \hat{F}}{\partial w_{i,j}^{2,1}} = s_i^2 p_j.$$

The update equations can thus be written in matrix form as:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T, \; m = 1, 2,$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha \mathbf{s}^m, \; m = 1, 2.$$

$$\mathbf{W}^{2,1}(k+1) = \mathbf{W}^{2,1}(k) - \alpha \mathbf{s}^2 (\mathbf{a}^0)^T = \mathbf{W}^{2,1}(k) - \alpha \mathbf{s}^2 (\mathbf{p})^T.$$

The main point of this problem is that the backpropagation concept can be used on networks more general than the standard multilayer feedforward network.

P11.9 Find an algorithm, based on the backpropagation concept, that can be used to update the weights w_1 and w_2 in the recurrent network shown in Figure P11.11.

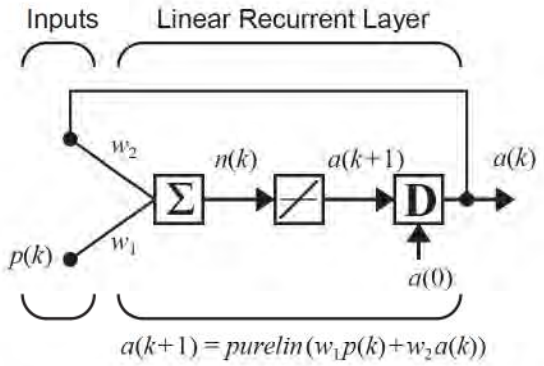


Figure P11.11 Linear Recurrent Network

The first step is to define our performance index. As with the multilayer networks, we will use squared error:

$$\hat{F}(\mathbf{x}) = (t(k) - a(k))^2 = (e(k))^2 .$$

For our weight updates we will use the steepest descent algorithm:

$$\Delta w_i = -\alpha \frac{\partial}{\partial w_i} \hat{F}(\mathbf{x}) .$$

These derivatives can be computed as follows:

$$\frac{\partial}{\partial w_i} \hat{F}(\mathbf{x}) = \frac{\partial}{\partial w_i} (t(k) - a(k))^2 = 2(t(k) - a(k)) \left\{ -\frac{\partial a(k)}{\partial w_i} \right\} .$$

Therefore, the key terms we need to compute are

$$\frac{\partial a(k)}{\partial w_i} .$$

To compute these terms we first need to write out the network equation:

$$a(k+1) = purelin(w_1 p(k) + w_2 a(k)) = w_1 p(k) + w_2 a(k) .$$

Next we take the derivative of both sides of this equation with respect to the network weights:

$$\frac{\partial a(k+1)}{\partial w_1} = p(k) + w_2 \frac{\partial a(k)}{\partial w_1} ,$$

$$\frac{\partial a(k+1)}{\partial w_2} = a(k) + w_2 \frac{\partial a(k)}{\partial w_2} .$$

(Note that we had to take account of the fact that $a(k)$ is itself a function of w_1 and w_2.) These two recursive equations are then used to compute the derivatives needed for the steepest descent weight update. The equations are initialized with

$$\frac{\partial a(0)}{\partial w_1} = 0\,,\ \frac{\partial a(0)}{\partial w_2} = 0\,,$$

since the initial condition is not a function of the weight.

To illustrate the process, let's say that $a(0) = 0$. The first network update would be

$$a(1) = w_1 p(0) + w_2 a(0) = w_1 p(0)\,.$$

The first derivatives would be computed:

$$\frac{\partial a(1)}{\partial w_1} = p(0) + w_2 \frac{\partial a(0)}{\partial w_1} = p(0)\,,\ \frac{\partial a(1)}{\partial w_2} = a(0) + w_2 \frac{\partial a(0)}{\partial w_2} = 0\,.$$

The first weight updates would be

$$\Delta w_i = -\alpha \frac{\partial}{\partial w_i} \hat{F}(\mathbf{x}) = -\alpha \left[2(t(1) - a(1)) \left\{ -\frac{\partial a(1)}{\partial w_i} \right\} \right]$$

$$\Delta w_1 = -2\alpha(t(1) - a(1))\{-p(0)\}$$

$$\Delta w_2 = -2\alpha(t(1) - a(1))\{0\} = 0\,.$$

This algorithm is a type of *dynamic backpropagation*, in which the gradient is computed by means of a difference equation.

P11.10 Show that backpropagation reduces to the LMS algorithm for a single-layer linear network (ADALINE).

The sensitivity calculation for a single-layer linear network would be:

$$\mathbf{s}^1 = -2\dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{t} - \mathbf{a}) = -2\mathbf{I}(\mathbf{t} - \mathbf{a}) = -2\mathbf{e}\,,$$

The weight update (Eq. (11.46) and Eq. (11.47)) would be

$$\mathbf{W}^1(k+1) = \mathbf{W}^1(k) - \alpha \mathbf{s}^1 (\mathbf{a}^0)^T = \mathbf{W}^1(k) - \alpha(-2\mathbf{e})\mathbf{p}^T = \mathbf{W}^1(k) + 2\alpha\mathbf{e}\mathbf{p}^T$$

$$\mathbf{b}^1(k+1) = \mathbf{b}^1(k) - \alpha \mathbf{s}^1 = \mathbf{b}^1(k) - \alpha(-2\mathbf{e}) = \mathbf{b}^1(k) + 2\alpha\mathbf{e}\,.$$

This is identical to the LMS algorithm of Chapter 10.

Epilogue

In this chapter we have presented the multilayer perceptron network and the backpropagation learning rule. The multilayer network is a powerful extension of the single-layer perceptron network. Whereas the single-layer network is only able to classify linearly separable patterns, the multilayer network can be used for arbitrary classification problems. In addition, multilayer networks can be used as universal function approximators. It has been shown that a two-layer network, with sigmoid-type transfer functions in the hidden layer, can approximate any practical function, given enough neurons in the hidden layer.

The backpropagation algorithm is an extension of the LMS algorithm that can be used to train multilayer networks. Both LMS and backpropagation are approximate steepest descent algorithms that minimize squared error. The only difference between them is in the way in which the gradient is calculated. The backpropagation algorithm uses the chain rule in order to compute the derivatives of the squared error with respect to the weights and biases in the hidden layers. It is called backpropagation because the derivatives are computed first at the last layer of the network, and then propagated backward through the network, using the chain rule, to compute the derivatives in the hidden layers.

One of the major problems with backpropagation has been the long training times. It is not feasible to use the basic backpropagation algorithm on practical problems, because it can take weeks to train a network, even on a large computer. Since backpropagation was first popularized, there has been considerable work on methods to accelerate the convergence of the algorithm. In Chapter 12 we will discuss the reasons for the slow convergence of backpropagation and will present several techniques for improving the performance of the algorithm.

Another key problem in training multilayer networks is overfitting. The network may memorize the data in the training set, but fail to generalize to new situations. In Chapter 13 we will describe in detail training procedures that can be used to produce networks with excellent generalization.

This chapter has focused mainly on the theoretical development of the backpropagation learning rule for training multilayer networks. Practical aspects of training networks with this method are discussed in Chapter 22. Real-world case studies that demonstrate how to train and validate multilayer networks are provided in Chapter 23 (function approximation), Chapter 24 (probability estimation) and Chapter 25 (pattern recognition).

Further Reading

[HoSt89] K. M. Hornik, M. Stinchcombe and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

This paper proves that multilayer feedforward networks with arbitrary squashing functions can approximate any Borel integrable function from one finite dimensional space to another finite dimensional space.

[LeCu85] Y. Le Cun, "Une procedure d'apprentissage pour reseau a seuil assymetrique," *Cognitiva*, vol. 85, pp. 599–604, 1985.

Yann Le Cun discovered the backpropagation algorithm at about the same time as Parker and Rumelhart, Hinton and Williams. This paper describes his algorithm.

[Park85] D. B. Parker, "Learning-logic: Casting the cortex of the human brain in silicon," Technical Report TR-47, Center for Computational Research in Economics and Management Science, MIT, Cambridge, MA, 1985.

David Parker independently derived the backpropagation algorithm at about the same time as Le Cun and Rumelhart, Hinton and Williams. This report describes his algorithm.

[RuHi86] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

This paper contains the most widely publicized description of the backpropagation algorithm.

[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.

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

[Werbo74] P. J. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. Thesis, Harvard University, Cambridge, MA, 1974.

This Ph.D. thesis contains what appears to be the first description of the backpropagation algorithm (although that

name is not used). The algorithm is described here in the context of general networks, with neural networks as a special case. Backpropagation did not become widely known until it was rediscovered in the mid 1980s by Rumelhart, Hinton and Williams [RuHi86], David Parker [Park85] and Yann Le Cun [LeCu85].

Exercises

E11.1 Design multilayer networks to perform the classifications illustrated in Figure E11.1. The network should output a 1 whenever the input vector is in the shaded region (or on the boundary) and a -1 otherwise. Draw the network diagram in abbreviated notation and show the weight matrices and bias vectors.

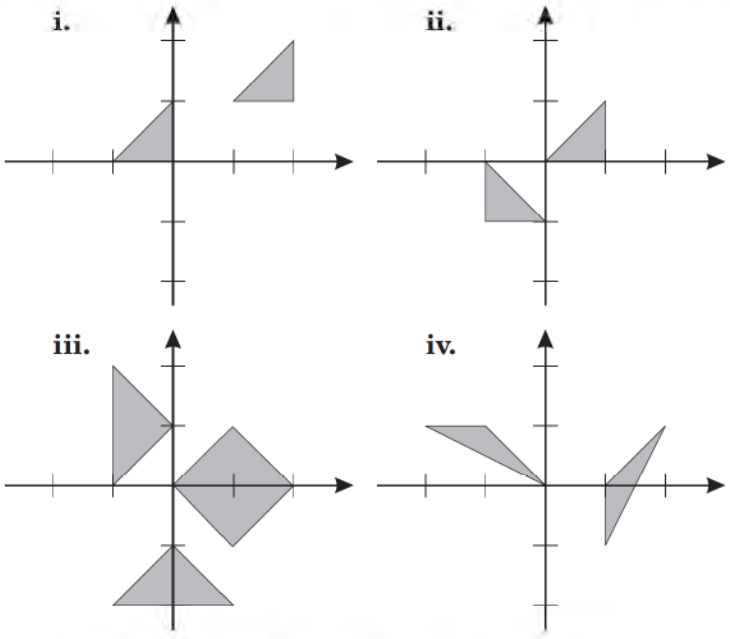


Figure E11.1 Pattern Classification Tasks

E11.2 Choose the weights and biases for the 1-2-1 network shown in Figure 11.4 so that the network response passes through the points indicated by the blue circles in Figure E11.2.



Use the MATLAB® Neural Network Design Demonstration Two-Layer Network Function (`nnd11nf`) *to check your result.*

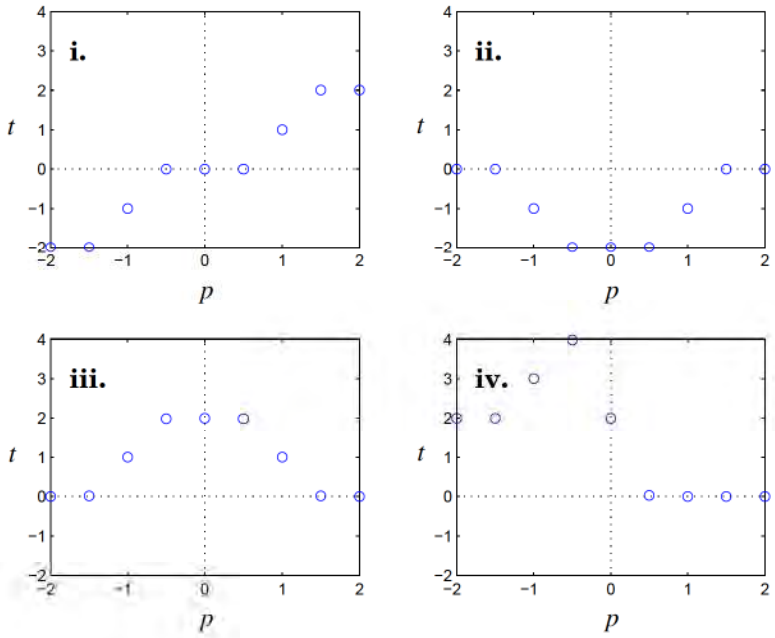


Figure E11.2 Function Approximation Tasks

E11.3 Find a single-layer network that has the same input/output characteristic as the network in Figure E11.3.

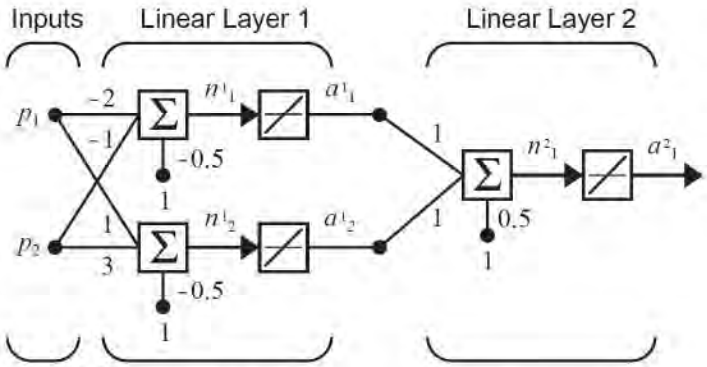


Figure E11.3 Two-Layer Linear Network

E11.4 Use the chain rule to find the derivative $\partial f / \partial w$ in the following cases:

i. $f(n) = \sin(n)$, $n(w) = w^2$.

ii. $f(n) = \tanh(n)$, $n(w) = 5w$.

iii. $f(n) = \exp(n)$, $n(w) = \cos(w)$.

iv. $f(n) = \text{logsig}(n)$, $n(w) = \exp(w)$.

E11.5 Consider again the backpropagation example that begins on page 11-14.

i. Find the squared error $(e)^2$ as an explicit function of all weights and biases.

ii. Using part (i), find $\partial (e)^2 / \partial w^1_{1,1}$ at the initial weights and biases.

iii. Compare the results of part (ii) with the backpropagation results described in the text.

E11.6 For the network shown in Figure E11.4 the initial weights and biases are chosen to be

$$w^1(0) = 1,\ b^1(0) = -2,\ w^2(0) = 1,\ b^2(0) = 1.$$

The network transfer functions are

$$f^1(n) = (n)^2,\ f^2(n) = \frac{1}{n},$$

and an input/target pair is given to be

$$\{p = 1, t = 1\}.$$

Perform one iteration of backpropagation with $\alpha = 1$.

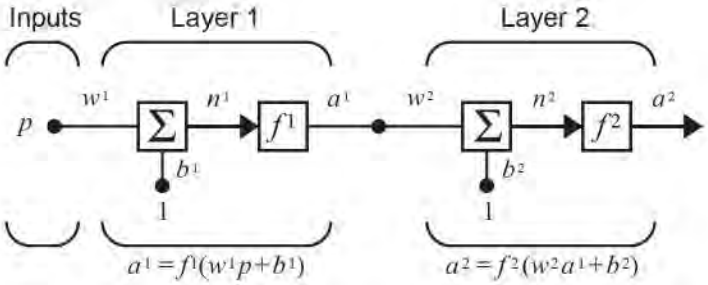


Figure E11.4 Two-Layer Network for Exercise E11.6

E11.7 Consider the two-layer network in Figure E11.5.

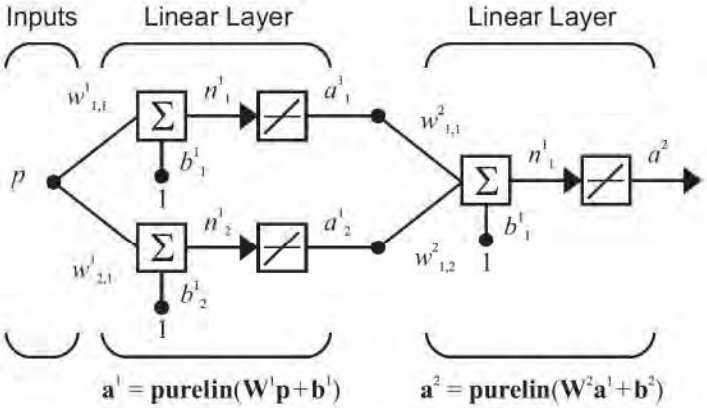


Figure E11.5 Two-Layer Network for Exercise E11.7

with the following input and target: $\{p_1 = 1, t_1 = 2\}$. The initial weights and biases are given by

$$\mathbf{W}^1(0) = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \mathbf{W}^2(0) = \begin{bmatrix} -1 & 1 \end{bmatrix}, \mathbf{b}^1(0) = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \mathbf{b}^2(0) = \begin{bmatrix} 3 \end{bmatrix}$$

i. Apply the input to the network and make one pass forward through the network to compute the output and the error.

ii. Compute the sensitivities by backpropagating through the network.

iii. Compute the derivative $\partial(e)^2/\partial w^1_{1,1}$ using the results of part ii. (Very little calculation is required here.)

E11.8 For the network shown in Figure E11.6 the neuron transfer function is

$$f^1(n) = (n)^2,$$

and an input/target pair is given to be

$$\left\{\mathbf{p} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{t} = \begin{bmatrix} 8 \\ 2 \end{bmatrix}\right\}.$$

Perform one iteration of backpropagation with $\alpha = 1$.

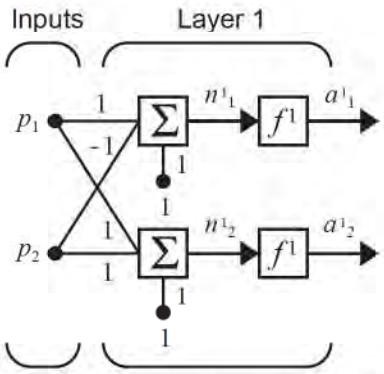


Figure E11.6 Single-Layer Network for Exercise E11.8

E11.9 We want to train the network in Figure E11.7 using the standard backpropagation algorithm (approximate steepest descent).

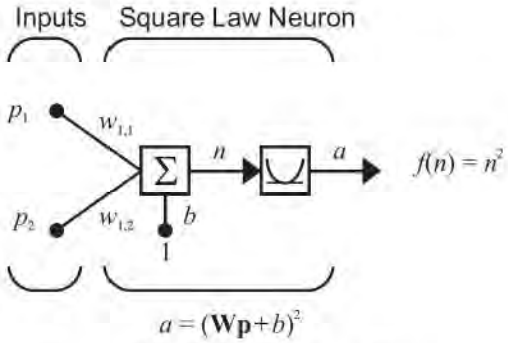


Figure E11.7 Square Law Neuron

The following input and target are given:

$$\left\{\mathbf{p} = \begin{bmatrix}1\\1\end{bmatrix}, t = \begin{bmatrix}0\end{bmatrix}\right\}$$

The initial weights and bias are

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

i. Propagate the input forward through the network.

ii. Compute the error.

iii. Propagate the sensitivities backward through the network.

iv. Compute the gradient of the squared error with respect to the weights and bias.

v. Update the weights and bias (assume a learning rate of $\alpha = 0.1$).

E11.10 Consider the following multilayer perceptron network. (The transfer function of the hidden layer is $f(n) = n^2$.)

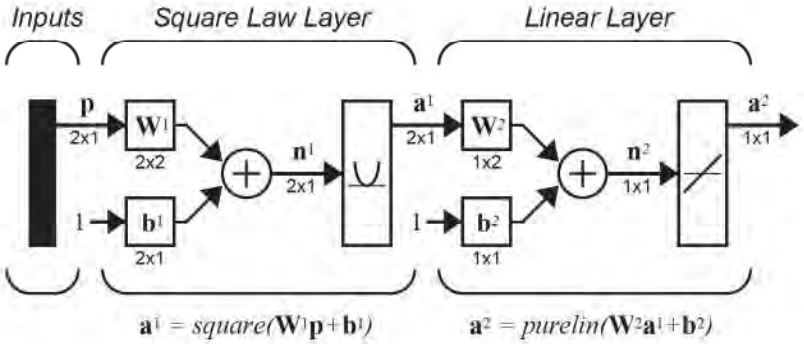


Figure E11.8 Two-Layer Square Law Network

The initial weights and biases are:

$$\mathbf{W}^1(0) = \begin{bmatrix} 1 & -1 \\ 1 & 0 \end{bmatrix}, \mathbf{W}^2(0) = \begin{bmatrix} 2 & 1 \end{bmatrix}, \mathbf{b}^1(0) = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \mathbf{b}^2(0) = \begin{bmatrix} -1 \end{bmatrix}.$$

Perform one iteration of the standard steepest descent backpropagation (use matrix operations) with learning rate $\alpha = 0.5$ for the following input/target pair:

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

E11.11 Consider the network shown in Figure E11.9.

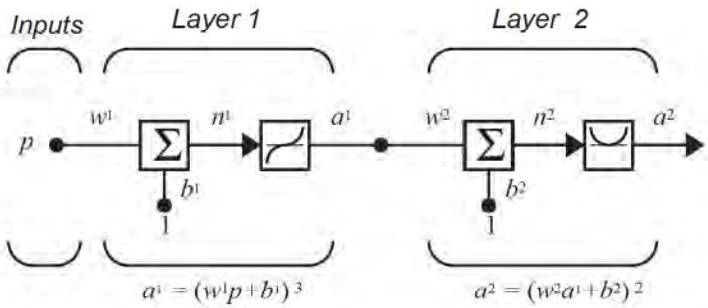


Figure E11.9 Two-Layer Network for Exercise E11.11

The initial weights and biases are chosen to be

$$w^1(0) = -2, b^1(0) = 1, w^2(0) = 1, b^2(0) = -2 .$$

An input/target pair is given to be

$$\{p_1 = 1, t_1 = 0\},$$

Perform one iteration of backpropagation (steepest descent) with $\alpha = 1$.

E11.12 Consider the multilayer perceptron network in Figure E11.10. (The transfer function of the hidden layer is $f(n) = n^3$.)

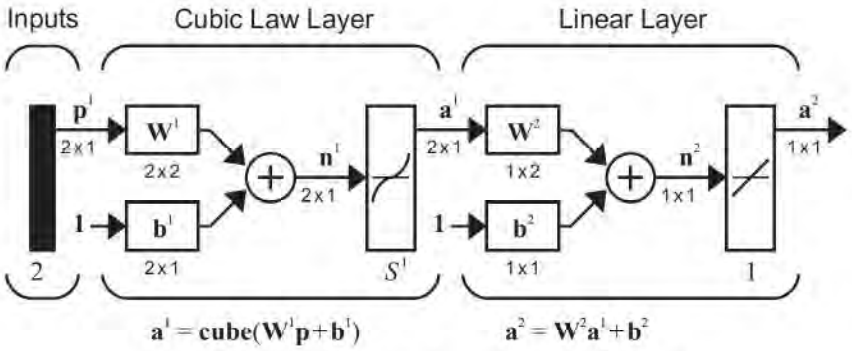


Figure E11.10 Cubic Law Neural Network

The initial weights and biases are:

$$\mathbf{W}^1(0) = \begin{bmatrix} 1 & -1 \\ 1 & 0 \end{bmatrix}, \mathbf{b}^1(0) = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \mathbf{W}^2(0) = \begin{bmatrix} 1 & 1 \end{bmatrix}, \mathbf{b}^2(0) = \begin{bmatrix} 1 \end{bmatrix}.$$

Perform one iteration of the standard steepest descent backpropagation (use matrix operations) with learning rate $\alpha = 0.5$ for the following input/target pair:

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

E11.13 Someone has proposed that the standard multilayer network should be modified to include a scalar gain at each layer. This means that the net input at layer m would be computed as

$$\mathbf{n}^m = \beta^m [\mathbf{W}^m \mathbf{a}^{m-1} + \mathbf{b}^m],$$

where β^m is the scalar gain at layer m. This gain would be trained like the weights and biases of the network. Modify the backpropagation algorithm (Eq. (11.41) to Eq. (11.47)) for this new network. (There will need to be a new equation added to update β^m, but some of the other equations may have to be modified as well.)

E11.14 Consider the two-layer network shown in Figure E11.11.

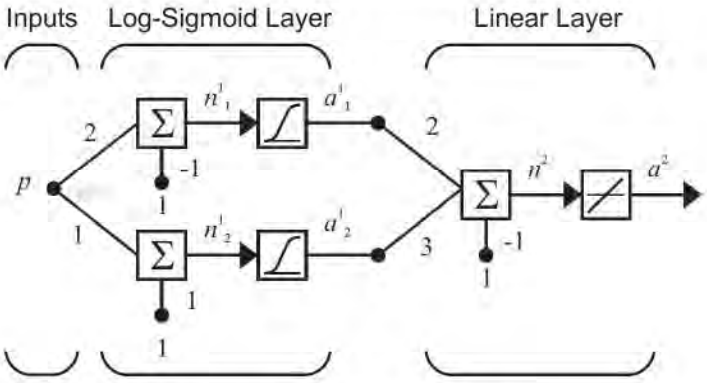


Figure E11.11 Two-Layer Network for Exercise E11.14

i. If $p = 1$, use a (slightly) modified form of backpropagation (as developed in Eq. (11.41) through Eq. (11.47)) to find

$$\frac{\partial a^2}{\partial n^2}, \frac{\partial a^2}{\partial n_1^1}, \frac{\partial a^2}{\partial n_2^1}.$$

ii. Use the results of i. and the chain rule to find $\frac{\partial a^2}{\partial p}$.

Your answers to both parts should be numerical.

E11.15 Consider the network shown in Figure E11.12, where the inputs to the neuron involve both the original inputs and their product. This is a type of higher-order network.

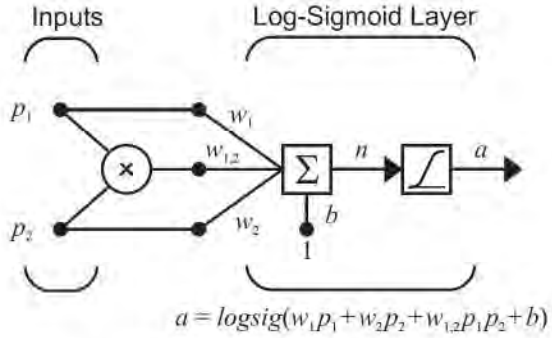


Figure E11.12 Higher-Order Network

i. Find a learning rule for the network parameters, using the approximate steepest descent algorithm (as was done for backpropagation).

ii. For the following initial parameter values, inputs and target, perform one iteration of your learning rule with learning rate $\alpha = 1$:

$$w_1 = 1, w_2 = -1, w_{1,2} = 0.5, b_1 = 1, p_1 = 0, p_2 = 1, t = 0.75$$

E11.16 In Figure E11.13 we have a two-layer network that has an additional connection from the input directly to the second layer. Derive the backpropagation algorithm for this network.

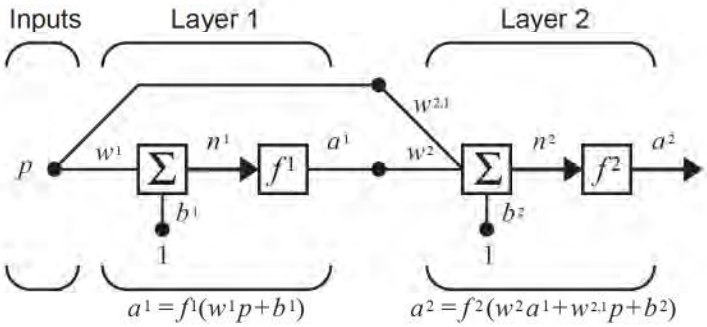


Figure E11.13 Two-Layer Network with Bypass Connection

E11.17 In the multilayer network, the net input is computed as follows

$$\mathbf{n}^{m+1} = \mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1} \quad \text{or} \quad n_i^{m+1} = \sum_{j=1}^{S^m} w_{i,j}^{m+1} a_j^m + b_i^{m+1}.$$

If the net input calculation is changed to the following equation (squared distance calculation), how will the sensitivity backpropagation (Eq. (11.35)) change?

$$n_i^{m+1} = \sum_{j=1}^{S^m} (w_{i,j}^{m+1} - a_j^m)^2$$

E11.18 Consider again the net input calculation, as described in Exercise E11.17. If the net input calculation is changed to the following equation (multiply by the bias, instead of add), how will the sensitivity backpropagation (Eq. (11.35)) change?

$$n_i^{m+1} = \left(\sum_{j=1}^{S^m} w_{i,j}^{m+1} a_j^m \right) \times b_i^{m+1}.$$

E11.19 Consider the system shown in Figure E11.14. There are a series of stages, with different transfer functions in each stage. (There are no weights or biases.) We want to take the derivative of the output of this system (a^M) with respect to the input of the system (p). Derive a recursive algorithm that you can use to compute this derivative. Use the concepts that we used to derive the backpropagation algorithm, and use the following intermediate variable in your algorithm:

$$q^i = \frac{\partial a^M}{\partial a^i}.$$

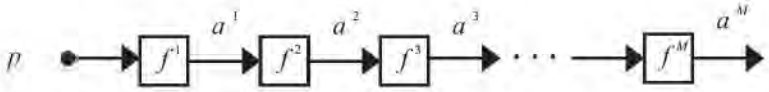


Figure E11.14 Cascade System

E11.20 The backpropagation algorithm is used to compute the gradient of the squared error with respect to the weights and biases of a multilayer network. How would the algorithm be changed if you wanted to compute the gradient with respect to the inputs of the network (i.e., with respect to the elements of the input vector **p**)? Carefully explain all of your steps, and write out the final algorithm.

E11.21 With the standard backpropagation algorithm, we want to compute the derivative

$$\frac{\partial F}{\partial w}.$$

To calculate this derivative, we use the chain rule in the form

$$\frac{\partial F}{\partial w} = \frac{\partial F}{\partial n} \cdot \frac{\partial n}{\partial w}.$$

Suppose that we want to use Newton's method. We would need to find the second derivative

$$\frac{\partial^2 F}{\partial w^2}.$$

What form will the chain rule take in this case?

E11.22 The standard steepest descent backpropagation algorithm, which is summarized in Eq. (11.41) through Eq. (11.47), was designed to minimize the performance function that was the sum of squares of the network errors, as given in Eq. (11.12). Suppose that we want to change the performance function to the sum of the fourth powers of the errors (e^4) plus the sum of the squares of the weights and biases in the network. Show how Eq. (11.41)

through Eq. (11.47) will change for this new performance function. (You don't need to rederive any steps which are already given in this chapter and do not change.)

E11.23 Repeat Problem P11.4 using the "backward" method described below.

In Problem P11.4. we had the dynamic system

$$y(k+1) = f(y(k)).$$

We had to choose the initial condition $y(0)$ so that at some final time $k = K$ the system output $y(K)$ would be as close as possible to some target output t. We minimized the performance index

$$F(y(0)) = (t - y(K))^2 = e^2(K)$$

using steepest descent, so we needed the gradient

$$\frac{\partial}{\partial y(0)} F(y(0)).$$

We developed a procedure for computing this gradient using the chain rule. The procedure involved a recursive equation for the term

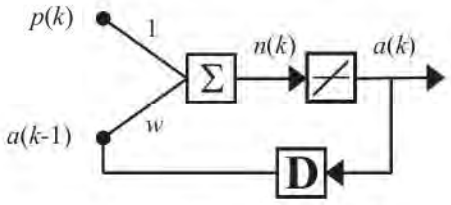
$$r(k) \equiv \frac{\partial}{\partial y(0)} y(k),$$

which evolved forward in time. The gradient can also be computed in a different way by evolving the term

$$q(k) \equiv \frac{\partial}{\partial y(k)} e^2(K)$$

backward through time.

E11.24 Consider the recurrent neural network in Figure E11.15.



$$a(k) = p(k) + w\, a(k\text{-}1)$$

Figure E11.15 Recurrent Network

We want to find the weight value w so that at some final time $k = K$ the system output $a(K)$ will be as close as possible to some target output t. We will minimize the performance index $F(w) = (t - a(K))^2$ using steepest descent, so we need the gradient $\partial F(w)/\partial w$.

i. Find a general procedure to compute this gradient using the chain rule. Develop an equation to evolve the following term forward through time:

$$s(k) \equiv \frac{\partial}{\partial w} a(k).$$

Show each step of your entire procedure carefully. This will involve updating $s(k)$ and also computing the gradient $\partial F(w)/\partial w$.

ii. Assume that $K = 3$. Write out the complete expression for $a(3)$ as a function of $p(1)$, $p(2)$, $p(3)$ and w (assuming $a(0) = 0$). Take the derivative of this expression with respect to w, and show that it equals $s(3)$.

E11.25 Write a MATLAB program to implement the backpropagation algorithm for a $1 - S^1 - 1$ network. Write the program using matrix operations, as in Eq. (11.41) to Eq. (11.47). Choose the initial weights and biases to be random numbers uniformly distributed between -0.5 and 0.5 (using the MATLAB function `rand`), and train the network to approximate the function



$$g(p) = 1 + \sin\left(\frac{\pi}{2}p\right) \text{ for } -2 \le p \le 2.$$

Use $S^1 = 2$ and $S^1 = 10$. Experiment with several different values for the learning rate α, and use several different initial conditions. Discuss the convergence properties of the algorithm as the learning rate changes.

12 Variations on Backpropagation

Objectives	12-1
Theory and Examples	12-2
Drawbacks of Backpropagation	12-3
Performance Surface Example	12-3
Convergence Example	12-7
Heuristic Modifications to Backpropagation	12-9
Momentum	12-9
Variable Learning Rate	12-12
Numerical Optimization Techniques	12-14
Conjugate Gradient	12-14
Levenberg-Marquardt Algorithm	12-19
Summary of Results	12-28
Solved Problems	12-32
Epilogue	12-46
Further Reading	12-47
Exercises	12-50

Objectives

The backpropagation algorithm introduced in Chapter 11 was a major breakthrough in neural network research. However, the basic algorithm is too slow for most practical applications. In this chapter we present several variations of backpropagation that provide significant speedup and make the algorithm more practical.

We will begin by using a function approximation example to illustrate why the backpropagation algorithm is slow in converging. Then we will present several modifications to the algorithm. Recall that backpropagation is an approximate steepest descent algorithm. In Chapter 9 we saw that steepest descent is the simplest, and often the slowest, minimization method. The conjugate gradient algorithm and Newton's method generally provide faster convergence. In this chapter we will explain how these faster procedures can be used to speed up the convergence of backpropagation.

Theory and Examples

When the basic backpropagation algorithm is applied to a practical problem the training may take days or weeks of computer time. This has encouraged considerable research on methods to accelerate the convergence of the algorithm.

The research on faster algorithms falls roughly into two categories. The first category involves the development of heuristic techniques, which arise out of a study of the distinctive performance of the standard backpropagation algorithm. These heuristic techniques include such ideas as varying the learning rate, using momentum and rescaling variables (e.g., [VoMa88], [Jacob88], [Toll90] and [RiIr90]). In this chapter we will discuss the use of momentum and variable learning rates.

Another category of research has focused on standard numerical optimization techniques (e.g., [Shan90], [Barn92], [Batt92] and [Char92]). As we have discussed in Chapters 10 and 11, training feedforward neural networks to minimize squared error is simply a numerical optimization problem. Because numerical optimization has been an important research subject for 30 or 40 years (see Chapter 9), it seems reasonable to look for fast training algorithms in the large number of existing numerical optimization techniques. There is no need to "reinvent the wheel" unless absolutely necessary. In this chapter we will present two existing numerical optimization techniques that have been very successfully applied to the training of multilayer perceptrons: the conjugate gradient algorithm and the Levenberg-Marquardt algorithm (a variation of Newton's method).

We should emphasize that all of the algorithms that we will describe in this chapter use the backpropagation procedure, in which derivatives are processed from the last layer of the network to the first. For this reason they could all be called "backpropagation" algorithms. The differences between the algorithms occur in the way in which the resulting derivatives are used to update the weights. In some ways it is unfortunate that the algorithm we usually refer to as backpropagation is in fact a steepest descent algorithm. In order to clarify our discussion, for the remainder of this chapter we will refer to the basic backpropagation algorithm as steepest descent backpropagation (*SDBP*).

SDBP

In the next section we will use a simple example to explain why SDBP has problems with convergence. Then, in the following sections, we will present various procedures to improve the convergence of the algorithm.

Drawbacks of Backpropagation

Recall from Chapter 10 that the LMS algorithm is guaranteed to converge to a solution that minimizes the mean squared error, so long as the learning rate is not too large. This is true because the mean squared error for a single-layer linear network is a quadratic function. The quadratic function has only a single stationary point. In addition, the Hessian matrix of a quadratic function is constant, therefore the curvature of the function in a given direction does not change, and the function contours are elliptical.

SDBP is a generalization of the LMS algorithm. Like LMS, it is also an approximate steepest descent algorithm for minimizing the mean squared error. In fact, SDBP is equivalent to the LMS algorithm when used on a single-layer linear network. (See Problem P11.10.) When applied to multilayer networks, however, the characteristics of SDBP are quite different. This has to do with the differences between the mean squared error performance surfaces of single-layer linear networks and multilayer nonlinear networks. While the performance surface for a single-layer linear network has a single minimum point and constant curvature, the performance surface for a multilayer network may have many local minimum points, and the curvature can vary widely in different regions of the parameter space. This will become clear in the example that follows.

Performance Surface Example

To investigate the mean squared error performance surface for multilayer networks we will employ a simple function approximation example. We will use the 1-2-1 network shown in Figure 12.1, with log-sigmoid transfer functions in both layers.

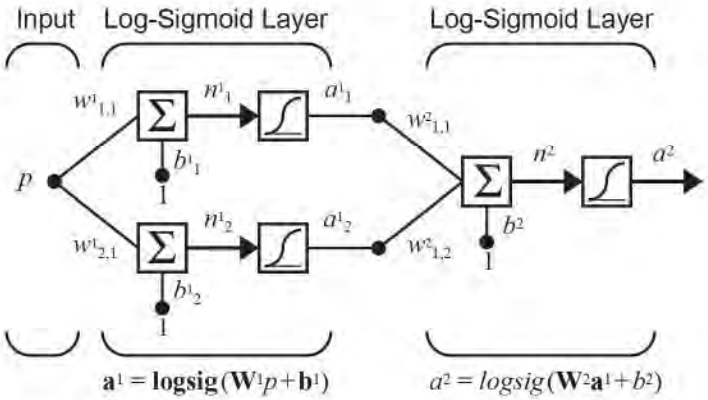


Figure 12.1 1-2-1 Function Approximation Network

In order to simplify our analysis, we will give the network a problem for which we know the optimal solution. The function we will approximate is

the response of the same 1-2-1 network, with the following values for the weights and biases:

$$w^1_{1,1} = 10,\ w^1_{2,1} = 10,\ b^1_1 = -5,\ b^1_2 = 5, \tag{12.1}$$

$$w^2_{1,1} = 1,\ w^2_{1,2} = 1,\ b^2 = -1. \tag{12.2}$$

The network response for these parameters is shown in Figure 12.2, which plots the network output a^2 as the input p is varied over the range $[-2, 2]$.

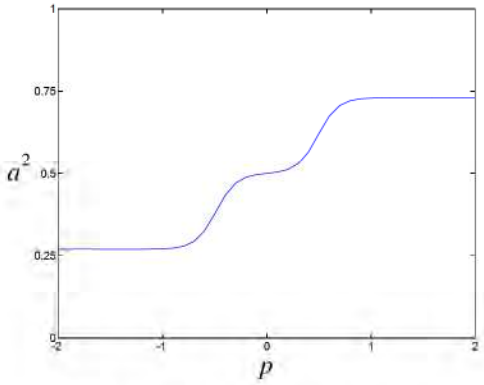


Figure 12.2 Nominal Function

We want to train the network of Figure 12.1 to approximate the function displayed in Figure 12.2. The approximation will be exact when the network parameters are set to the values given in Eq. (12.1) and Eq. (12.2). This is, of course, a very contrived problem, but it is simple and it illustrates some important concepts.

Let's now consider the performance index for our problem. We will assume that the function is sampled at the values

$$p = -2, -1.9, -1.8, \ldots, 1.9, 2, \tag{12.3}$$

and that each occurs with equal probability. The performance index will be the sum of the squared errors at these 41 points. (We won't bother to find the mean squared error, which just requires dividing by 41.)

In order to be able to graph the performance index, we will vary only two parameters at a time. Figure 12.3 illustrates the squared error when only $w^1_{1,1}$ and $w^2_{1,1}$ are being adjusted, while the other parameters are set to their optimal values given in Eq. (12.1) and Eq. (12.2). Note that the minimum error will be zero, and it will occur when $w^1_{1,1} = 10$ and $w^2_{1,1} = 1$, as indicated by the open blue circle in the figure.

There are several features to notice about this error surface. First, it is clearly not a quadratic function. The curvature varies drastically over the parameter space. For this reason it will be difficult to choose an appropriate learning rate for the steepest descent algorithm. In some regions the surface is very flat, which would allow a large learning rate, while in other regions the curvature is high, which would require a small learning rate. (Refer to discussions in Chapters 9 and 10 on the choice of learning rate for the steepest descent algorithm.)

It should be noted that the flat regions of the performance surface should not be unexpected, given the sigmoid transfer functions used by the network. The sigmoid is very flat for large inputs.

A second feature of this error surface is the existence of more than one local minimum point. The global minimum point is located at $w^1_{1,1} = 10$ and $w^2_{1,1} = 1$, along the valley that runs parallel to the $w^1_{1,1}$ axis. However, there is also a local minimum, which is located in the valley that runs parallel to the $w^2_{1,1}$ axis. (This local minimum is actually off the graph at $w^1_{1,1} = 0.88$, $w^2_{1,1} = 38.6$.) In the next section we will investigate the performance of backpropagation on this surface.

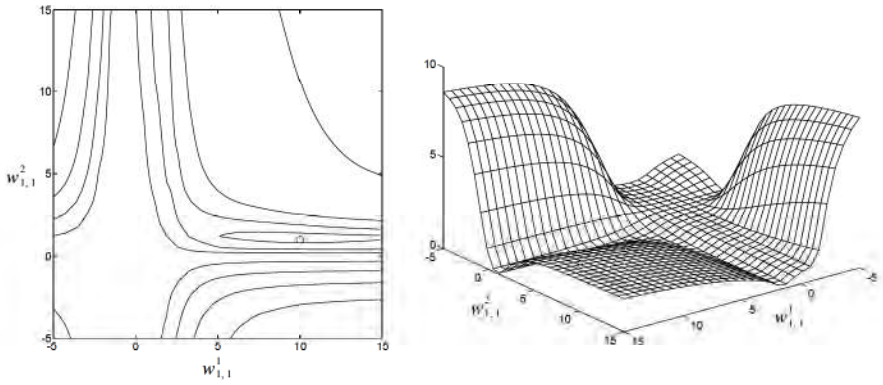


Figure 12.3 Squared Error Surface Versus $w^1_{1,1}$ and $w^2_{1,1}$

Figure 12.4 illustrates the squared error when $w^1_{1,1}$ and b^1_1 are being adjusted, while the other parameters are set to their optimal values. Note that the minimum error will be zero, and it will occur when $w^1_{1,1} = 10$ and $b^1_1 = -5$, as indicated by the open blue circle in the figure.

Again we find that the surface has a very contorted shape, steep in some regions and very flat in others. Surely the standard steepest descent algorithm will have some trouble with this surface. For example, if we have an initial guess of $w^1_{1,1} = 0$, $b^1_1 = -10$, the gradient will be very close to zero,

and the steepest descent algorithm would effectively stop, even though it is not close to a local minimum point.

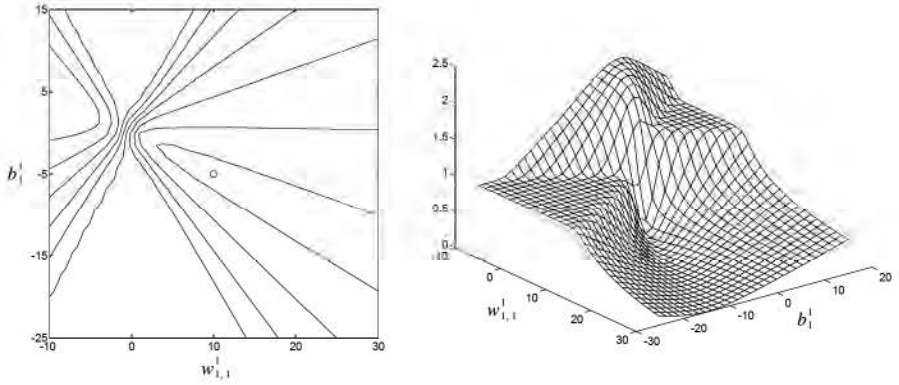


Figure 12.4 Squared Error Surface Versus $w_{1,1}^1$ and b_1^1

Figure 12.5 illustrates the squared error when b_1^1 and b_2^1 are being adjusted, while the other parameters are set to their optimal values. The minimum error is located at $b_1^1 = -5$ and $b_2^1 = 5$, as indicated by the open blue circle in the figure.

This surface illustrates an important property of multilayer networks: they have a symmetry to them. Here we see that there are two local minimum points and they both have the same value of squared error. The second solution corresponds to the same network being turned upside down (i.e., the top neuron in the first layer is exchanged with the bottom neuron). It is because of this characteristic of neural networks that we do not set the initial weights and biases to zero. The symmetry causes zero to be a saddle point of the performance surface.

This brief study of the performance surfaces for multilayer networks gives us some hints as to how to set the initial guess for the SDBP algorithm. First, we do not want to set the initial parameters to zero. This is because the origin of the parameter space tends to be a saddle point for the performance surface. Second, we do not want to set the initial parameters to large values. This is because the performance surface tends to have very flat regions as we move far away from the optimum point.

Typically we choose the initial weights and biases to be small random values. In this way we stay away from a possible saddle point at the origin without moving out to the very flat regions of the performance surface. (Another procedure for choosing the initial parameters is described in [NgWi90].) As we will see in the next section, it is also useful to try several different initial guesses, in order to be sure that the algorithm converges to a global minimum point.

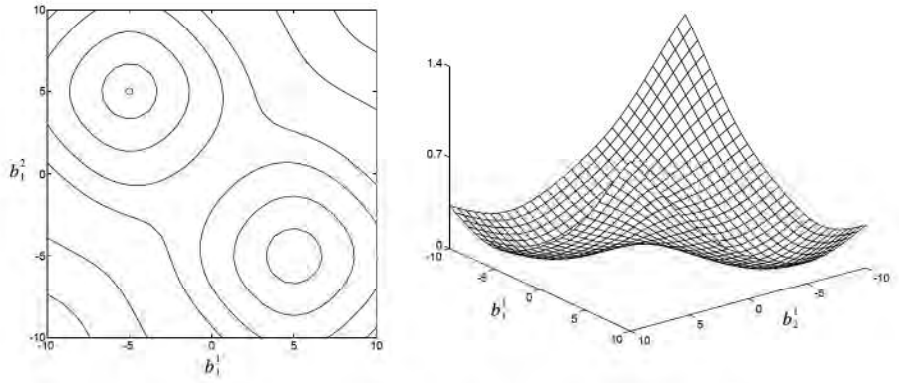


Figure 12.5 Squared Error Surface Versus b_1^1 and b_2^1

Convergence Example

Batching

Now that we have examined the performance surface, let's investigate the performance of SDBP. For this section we will use a variation of the standard algorithm, called *batching*, in which the parameters are updated only after the entire training set has been presented. The gradients calculated at each training example are averaged together to produce a more accurate estimate of the gradient. (If the training set is complete, i.e., covers all possible input/output pairs, then the gradient estimate will be exact.)

In Figure 12.6 we see two trajectories of SDBP (batch mode) when only two parameters, $w_{1,1}^1$ and $w_{1,1}^2$ are adjusted. For the initial condition labeled "a" the algorithm does eventually converge to the optimal solution, but the convergence is slow. The reason for the slow convergence is the change in curvature of the surface over the path of the trajectory. After an initial moderate slope, the trajectory passes over a very flat surface, until it falls into a very gently sloping valley. If we were to increase the learning rate, the algorithm would converge faster while passing over the initial flat surface, but would become unstable when falling into the valley, as we will see in a moment.

Trajectory "b" illustrates how the algorithm can converge to a local minimum point. The trajectory is trapped in a valley and diverges from the optimal solution. If allowed to continue the trajectory converges to $w_{1,1}^1 = 0.88$, $w_{1,1}^2 = 38.6$. The existence of multiple local minimum points is typical of the performance surface of multilayer networks. For this reason it is best to try several different initial guesses in order to ensure that a global minimum has been obtained. (Some of the local minimum points may have the same value of squared error, as we saw in Figure 12.5, so we would not expect the algorithm to converge to the same parameter values for each initial guess. We just want to be sure that the same minimum error is obtained.)

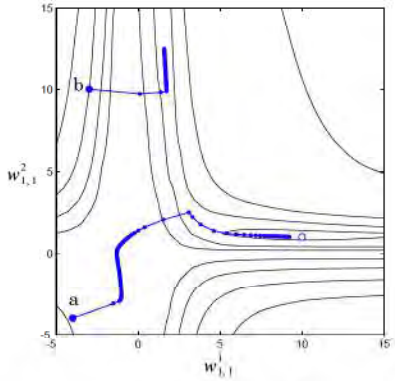


Figure 12.6 Two SDBP (Batch Mode) Trajectories

The progress of the algorithm can also be seen in Figure 12.7, which shows the squared error versus the iteration number. The curve on the left corresponds to trajectory "a" and the curve on the right corresponds to trajectory "b." These curves are typical of SDBP, with long periods of little progress and then short periods of rapid advance.

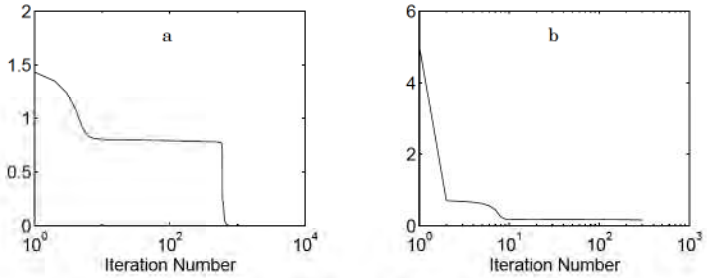


Figure 12.7 Squared Error Convergence Patterns

We can see that the flat sections in Figure 12.7 correspond to times when the algorithm is traversing a flat section of the performance surface, as shown in Figure 12.6. During these periods we would like to increase the learning rate, in order to speed up convergence. However, if we increase the learning rate the algorithm will become unstable when it reaches steeper portions of the performance surface.

This effect is illustrated in Figure 12.8. The trajectory shown here corresponds to trajectory "a" in Figure 12.6, except that a larger learning rate was used. The algorithm converges faster at first, but when the trajectory reaches the narrow valley that contains the minimum point the algorithm begins to diverge. This suggests that it would be useful to vary the learning rate. We could increase the learning rate on flat surfaces and then decrease the learning rate as the slope increased. The question is: "How will the al-

gorithm know when it is on a flat surface?" We will discuss this in a later section.

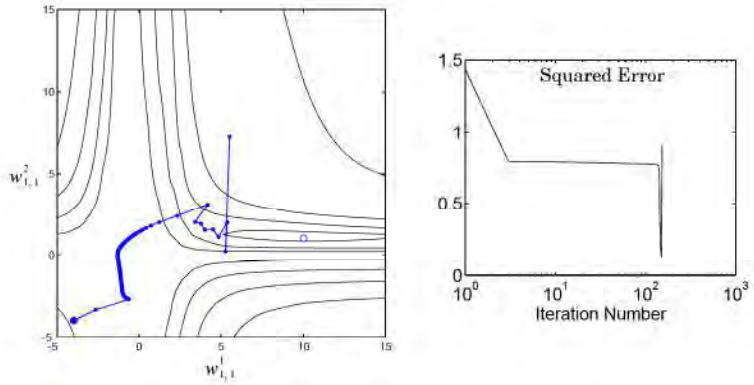


Figure 12.8 Trajectory with Learning Rate Too Large

Another way to improve convergence would be to smooth out the trajectory. Note in Figure 12.8 that when the algorithm begins to diverge it is oscillating back and forth across a narrow valley. If we could filter the trajectory, by averaging the updates to the parameters, this might smooth out the oscillations and produce a stable trajectory. We will discuss this procedure in the next section.



To experiment with this backpropagation example, use the MATLAB® Neural Network Design Demonstration Steepest Descent Backpropagation (**nnd12sd**).

Heuristic Modifications of Backpropagation

Now that we have investigated some of the drawbacks of backpropagation (steepest descent), let's consider some procedures for improving the algorithm. In this section we will discuss two heuristic methods. In a later section we will present two methods based on standard numerical optimization algorithms.

Momentum

The first method we will discuss is the use of momentum. This is a modification based on our observation in the last section that convergence might be improved if we could smooth out the oscillations in the trajectory. We can do this with a low-pass filter.

Before we apply momentum to a neural network application, let's investigate a simple example to illustrate the smoothing effect. Consider the following first-order filter:

$$y(k) = \gamma y(k-1) + (1-\gamma)w(k), \tag{12.4}$$

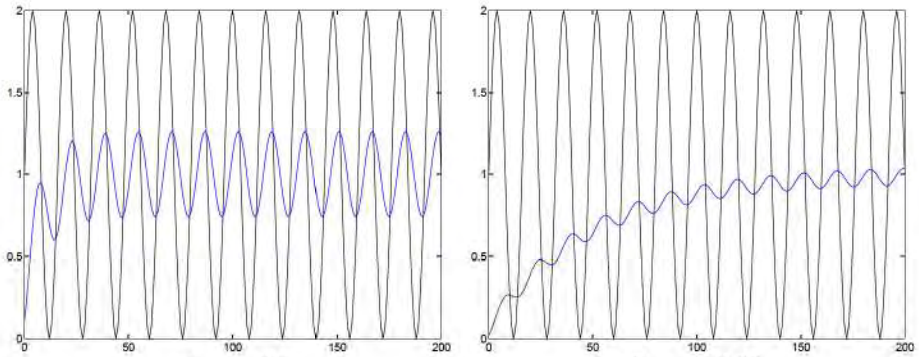
where $w(k)$ is the input to the filter, $y(k)$ is the output of the filter and γ is the momentum coefficient that must satisfy

$$0 \le \gamma < 1. \tag{12.5}$$

The effect of this filter is shown in Figure 12.9. For these examples the input to the filter was taken to be the sine wave:

$$w(k) = 1 + \sin\left(\frac{2\pi k}{16}\right), \tag{12.6}$$

and the momentum coefficient was set to $\gamma = 0.9$ (left graph) and $\gamma = 0.98$ (right graph). Here we can see that the oscillation of the filter output is less than the oscillation in the filter input (as we would expect for a low-pass filter). In addition, as γ is increased the oscillation in the filter output is reduced. Notice also that the average filter output is the same as the average filter input, although as γ is increased the filter output is slower to respond.



a) $\gamma = 0.9$ b) $\gamma = 0.98$

Figure 12.9 Smoothing Effect of Momentum

To summarize, the filter tends to reduce the amount of oscillation, while still tracking the average value. Now let's see how this works on the neural network problem. First, recall that the parameter updates for SDBP (Eq. (11.46) and Eq. (11.47)) are

$$\Delta \mathbf{W}^m(k) = -\alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T, \tag{12.7}$$

$$\Delta \mathbf{b}^m(k) = -\alpha \mathbf{s}^m. \tag{12.8}$$

Momentum

MOBP

When the *momentum* filter is added to the parameter changes, we obtain the following equations for the momentum modification to backpropagation (*MOBP*):

$$\Delta \mathbf{W}^m(k) = \gamma \Delta \mathbf{W}^m(k-1) - (1-\gamma)\alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T, \tag{12.9}$$

$$\Delta \mathbf{b}^m(k) = \gamma \Delta \mathbf{b}^m(k-1) - (1-\gamma)\alpha \mathbf{s}^m. \tag{12.10}$$

If we now apply these modified equations to the example in the preceding section, we obtain the results shown in Figure 12.10. (For this example we have used a batching form of MOBP, in which the parameters are updated only after the entire training set has been presented. The gradients calculated at each training example are averaged together to produce a more accurate estimate of the gradient.) This trajectory corresponds to the same initial condition and learning rate shown in Figure 12.8, but with a momentum coefficient of $\gamma = 0.8$. We can see that the algorithm is now stable. By the use of momentum we have been able to use a larger learning rate, while maintaining the stability of the algorithm. Another feature of momentum is that it tends to accelerate convergence when the trajectory is moving in a consistent direction.

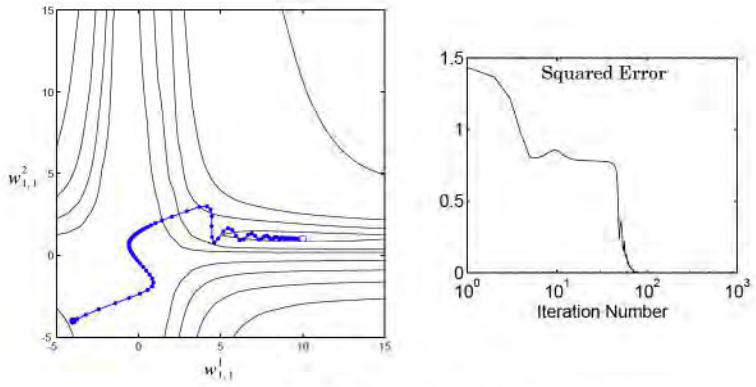


Figure 12.10 Trajectory with Momentum

If you look carefully at the trajectory in Figure 12.10, you can see why the procedure is given the name *momentum*. It tends to make the trajectory continue in the same direction. The larger the value of γ, the more "momentum" the trajectory has.



To experiment with momentum, use the MATLAB® Neural Network Design Demonstration Momentum Backpropagation (`nnd12mo`).

Variable Learning Rate

We suggested earlier in this chapter that we might be able to speed up convergence if we increase the learning rate on flat surfaces and then decrease the learning rate when the slope increases. In this section we want to explore this concept.

Recall that the mean squared error performance surface for single-layer linear networks is always a quadratic function, and the Hessian matrix is therefore constant. The maximum stable learning rate for the steepest descent algorithm is two divided by the maximum eigenvalue of the Hessian matrix. (See Eq. (9.25).)

As we have seen, the error surface for the multilayer network is not a quadratic function. The shape of the surface can be very different in different regions of the parameter space. Perhaps we can speed up convergence by adjusting the learning rate during the course of training. The trick will be to determine when to change the learning rate and by how much.

Variable Learning Rate
VLBP

There are many different approaches for varying the learning rate. We will describe a very straightforward batching procedure [VoMa88], where the learning rate is varied according to the performance of the algorithm. The rules of the *variable learning rate* backpropagation algorithm (*VLBP*) are:

1. If the squared error (over the entire training set) increases by more than some set percentage ζ (typically one to five percent) after a weight update, then the weight update is discarded, the learning rate is multiplied by some factor $0 < \rho < 1$, and the momentum coefficient γ (if it is used) is set to zero.

2. If the squared error decreases after a weight update, then the weight update is accepted and the learning rate is multiplied by some factor $\eta > 1$. If γ has been previously set to zero, it is reset to its original value.

3. If the squared error increases by less than ζ, then the weight update is accepted but the learning rate is unchanged. If γ has been previously set to zero, it is reset to its original value.

(See Problem P12.3 for a numerical example of VLBP.)

To illustrate VLBP, let's apply it to the function approximation problem of the previous section. Figure 12.11 displays the trajectory for the algorithm using the same initial guess, initial learning rate and momentum coefficient as was used in Figure 12.10. The new parameters were assigned the values

$$\eta = 1.05, \ \rho = 0.7 \text{ and } \zeta = 4\%. \tag{12.11}$$

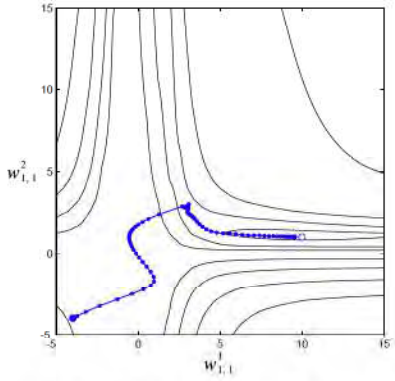


Figure 12.11 Variable Learning Rate Trajectory

Notice how the learning rate, and therefore the step size, tends to increase when the trajectory is traveling in a straight line with constantly decreasing error. This effect can also be seen in Figure 12.12, which shows the squared error and the learning rate versus iteration number.

When the trajectory reaches a narrow valley, the learning rate is rapidly decreased. Otherwise the trajectory would have become oscillatory, and the error would have increased dramatically. For each potential step where the error would have increased by more than 4% the learning rate is reduced and the momentum is eliminated, which allows the trajectory to make the quick turn to follow the valley toward the minimum point. The learning rate then increases again, which accelerates the convergence. The learning rate is reduced again when the trajectory overshoots the minimum point when the algorithm has almost converged. This process is typical of a VLBP trajectory.

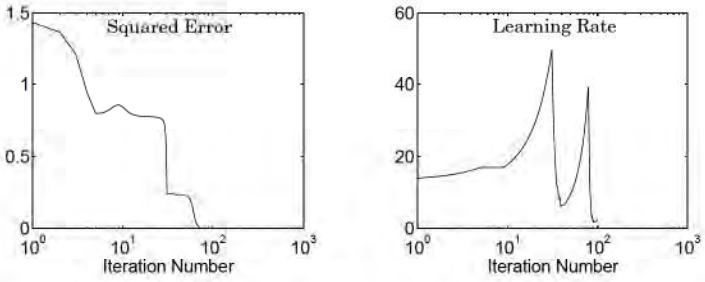


Figure 12.12 Convergence Characteristics of Variable Learning Rate

There are many variations on this variable learning rate algorithm. Jacobs [Jaco88] proposed the *delta-bar-delta* learning rule, in which each network parameter (weight or bias) has its own learning rate. The algorithm increases the learning rate for a network parameter if the parameter change

has been in the same direction for several iterations. If the direction of the parameter change alternates, then the learning rate is reduced. The *SuperSAB* algorithm of Tollenaere [Toll90] is similar to the delta-bar-delta rule, but it has more complex rules for adjusting the learning rates.

Another heuristic modification to SDBP is the Quickprop algorithm of Fahlman [Fahl88]. It assumes that the error surface is parabolic and concave upward around the minimum point and that the effect of each weight can be considered independently.

The heuristic modifications to SDBP can often provide much faster convergence for some problems. However, there are two main drawbacks to these methods. The first is that the modifications require that several parameters be set (e.g., ζ, ρ and γ), while the only parameter required for SDBP is the learning rate. Some of the more complex heuristic modifications can have five or six parameters to be selected. Often the performance of the algorithm is sensitive to changes in these parameters. The choice of parameters is also problem dependent. The second drawback to these modifications to SDBP is that they can sometimes fail to converge on problems for which SDBP will eventually find a solution. Both of these drawbacks tend to occur more often when using the more complex algorithms.



To experiment with VLBP, use the MATLAB® Neural Network Design Demonstration Variable Learning Rate Backpropagation (`nnd12vl`).

Numerical Optimization Techniques

Now that we have investigated some of the heuristic modifications to SDBP, let's consider those methods that are based on standard numerical optimization techniques. We will investigate two techniques: conjugate gradient and Levenberg-Marquardt. The conjugate gradient algorithm for quadratic functions was presented in Chapter 9. We need to add two procedures to this algorithm in order to apply it to more general functions.

The second numerical optimization method we will discuss in this chapter is the Levenberg-Marquardt algorithm, which is a modification to Newton's method that is well-suited to neural network training.

Conjugate Gradient

In Chapter 9 we presented three numerical optimization techniques: steepest descent, conjugate gradient and Newton's method. Steepest descent is the simplest algorithm, but is often slow in converging. Newton's method is much faster, but requires that the Hessian matrix and its inverse be calculated. The conjugate gradient algorithm is something of a compromise; it does not require the calculation of second derivatives, and yet it still has the quadratic convergence property. (It converges to the minimum of a quadratic function in a finite number of iterations.) In this section we will describe how the conjugate gradient algorithm can be used to train

multilayer networks. We will call this algorithm *conjugate gradient backpropagation* (*CGBP*).

CGBP

Let's begin by reviewing the conjugate gradient algorithm. For ease of reference, we will repeat the algorithm steps from Chapter 9 (page 9-18):

1. Select the first search direction $\mathbf{p}_0$ to be the negative of the gradient, as in Eq. (9.59):

$$\mathbf{p}_0 = -\mathbf{g}_0, \tag{12.12}$$

where

$$\mathbf{g}_k \equiv \nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}_k}. \tag{12.13}$$

2. Take a step according to Eq. (9.57), selecting the learning rate α_k to minimize the function along the search direction:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k. \tag{12.14}$$

3. Select the next search direction according to Eq. (9.60), using Eq. (9.61), Eq. (9.62), or Eq. (9.63) to calculate β_k:

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}, \tag{12.15}$$

with

$$\beta_k = \frac{\Delta\mathbf{g}_{k-1}^T \mathbf{g}_k}{\Delta\mathbf{g}_{k-1}^T \mathbf{p}_{k-1}} \text{ or } \beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}} \text{ or } \beta_k = \frac{\Delta\mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}. \tag{12.16}$$

4. If the algorithm has not converged, continue from step 2.

This conjugate gradient algorithm cannot be applied directly to the neural network training task, because the performance index is not quadratic. This affects the algorithm in two ways. First, we will not be able to use Eq. (9.31) to minimize the function along a line, as required in step 2. Second, the exact minimum will not normally be reached in a finite number of steps, and therefore the algorithm will need to be reset after some set number of iterations.

Let's address the linear search first. We need to have a general procedure for locating the minimum of a function in a specified direction. This will involve two steps: interval location and interval reduction. The purpose of the interval location step is to find some initial interval that contains a local minimum. The interval reduction step then reduces the size of the initial interval until the minimum is located to the desired accuracy.

Interval Location

We will use a function comparison method [Scal85] to perform the *interval location* step. This procedure is illustrated in Figure 12.13. We begin by evaluating the performance index at an initial point, represented by a_1 in the figure. This point corresponds to the current values of the network weights and biases. In other words, we are evaluating

$$F(\mathbf{x}_0)\,. \tag{12.17}$$

The next step is to evaluate the function at a second point, represented by b_1 in the figure, which is a distance ε from the initial point, along the first search direction $\mathbf{p}_0$. In other words, we are evaluating

$$F(\mathbf{x}_0 + \varepsilon\mathbf{p}_0)\,. \tag{12.18}$$

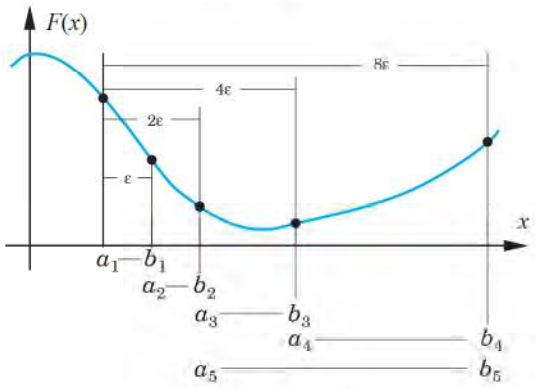


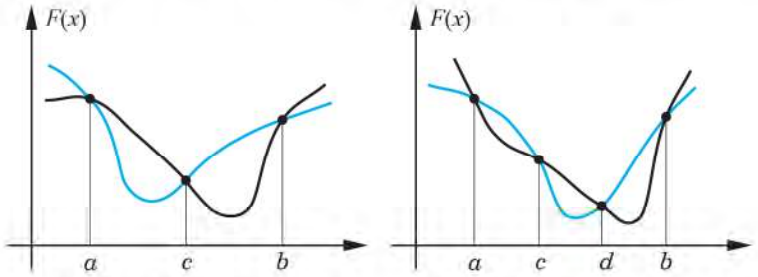
Figure 12.13 Interval Location

We then continue to evaluate the performance index at new points b_i, successively doubling the distance between points. This process stops when the function increases between two consecutive evaluations. In Figure 12.13 this is represented by b_3 to b_4. At this point we know that the minimum is bracketed by the two points a_5 and b_5. We cannot narrow the interval any further, because the minimum may occur either in the interval $[a_4, b_4]$ or in the interval $[a_3, b_3]$. These two possibilities are illustrated in Figure 12.14 (a).

Interval Reduction

Now that we have located an interval containing the minimum, the next step in the linear search is *interval reduction*. This will involve evaluating the function at points inside the interval $[a_5, b_5]$, which was selected in the interval location step. From Figure 12.14 we can see that we will need to evaluate the function at two internal points (at least) in order to reduce the size of the interval of uncertainty. Figure 12.14 (a) shows that one internal function evaluation does not provide us with any information on the location of the minimum. However, if we evaluate the function at two points c and d, as in Figure 12.14 (b), we can reduce the interval of uncertainty. If

$F(c) > F(d)$, as shown in Figure 12.14 (b), then the minimum must occur in the interval $[c, b]$. Conversely, if $F(c) < F(d)$, then the minimum must occur in the interval $[a, d]$. (Note that we are assuming that there is a single minimum located in the initial interval. More about that later.)



(a) Interval is not reduced.

(b) Minimum must occur between c and b.

Figure 12.14 Reducing the Size of the Interval of Uncertainty

Golden Section Search

The procedure described above suggests a method for reducing the size of the interval of uncertainty. We now need to decide how to determine the locations of the internal points c and d. There are several ways to do this (see [Scal85]). We will use a method called the *Golden Section search*, which is designed to reduce the number of function evaluations required. At each iteration one new function evaluation is required. For example, in the case illustrated in Figure 12.14 (b), point a would be discarded and point c would become the new a. Then point d would become the new point c, and a new d would be placed between the original points d and b. The trick is to place the new point so that the interval of uncertainty will be reduced as quickly as possible.

The algorithm for the Golden Section search is as follows [Scal85]:

$\tau = 0.618$

Set $c_1 = a_1 + (1-\tau)(b_1 - a_1)$, $F_c = F(c_1)$.

$d_1 = b_1 - (1-\tau)(b_1 - a_1)$, $F_d = F(d_1)$.

For $k = 1, 2, \ldots$ repeat

If $F_c < F_d$ then

Set $a_{k+1} = a_k$; $b_{k+1} = d_k$; $d_{k+1} = c_k$

$c_{k+1} = a_{k+1} + (1-\tau)(b_{k+1} - a_{k+1})$

$F_d = F_c$; $F_c = F(c_{k+1})$

else

Set $a_{k+1} = c_k$; $b_{k+1} = b_k$; $c_{k+1} = d_k$

$d_{k+1} = b_{k+1} - (1-\tau)(b_{k+1} - a_{k+1})$

$F_c = F_d$; $F_d = F(d_{k+1})$

end

end until $b_{k+1} - a_{k+1} < tol$

Where tol is the accuracy tolerance set by the user.

(See Problem P12.4 for a numerical example of the interval location and interval reduction procedures.)

There is one more modification to the conjugate gradient algorithm that needs to be made before we apply it to neural network training. For quadratic functions the algorithm will converge to the minimum in at most n iterations, where n is the number of parameters being optimized. The mean squared error performance index for multilayer networks is not quadratic, therefore the algorithm would not normally converge in n iterations. The development of the conjugate gradient algorithm does not indicate what search direction to use once a cycle of n iterations has been completed. There have been many procedures suggested, but the simplest method is to reset the search direction to the steepest descent direction (negative of the gradient) after n iterations [Scal85]. We will use this method.

Let's now apply the conjugate gradient algorithm to the function approximation example that we have been using to demonstrate the other neural network training algorithms. We will use the backpropagation algorithm to compute the gradient (using Eq. (11.23) and Eq. (11.24)) and the conjugate gradient algorithm to determine the weight updates. This is a batch mode algorithm, as the gradient is computed after the entire training set has been presented to the network.

Figure 12.15 shows the intermediate steps of the CGBP algorithm for the first three iterations. The interval location process is illustrated by the open blue circles; each one represents one evaluation of the function. The final interval is indicated by the larger open black circles. The black dots in Figure 12.15 indicate the location of the new interior points during the Golden Section search, one for each iteration of the procedure. The final point is indicated by a blue dot.

Figure 12.16 shows the total trajectory to convergence. Notice that the CGBP algorithm converges in many fewer iterations than the other algorithms that we have tested. This is a little deceiving, since each iteration of CGBP requires more computations than the other methods; there are many function evaluations involved in each iteration of CGBP. Even so, CGBP has been shown to be one of the fastest batch training algorithms for multilayer networks [Char92].

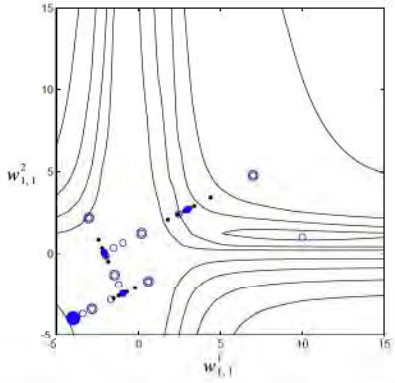


Figure 12.15 Intermediate Steps of CGBP

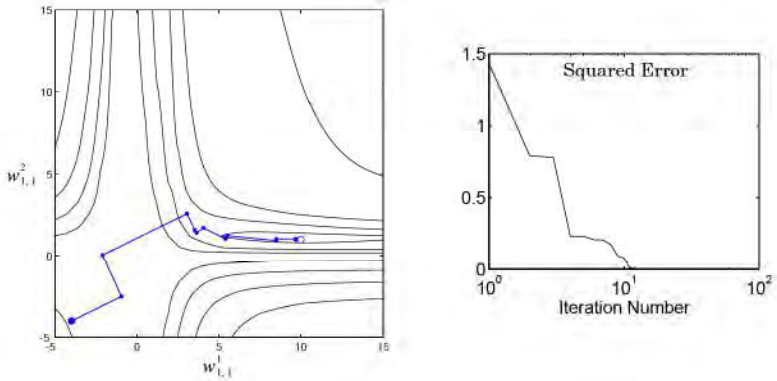


Figure 12.16 Conjugate Gradient Trajectory



To experiment with CGBP, use the MATLAB® Neural Network Design Demonstrations Conjugate Gradient Line Search (`nnd12ls`) *and Conjugate Gradient Backpropagation* (`nnd12cg`).

Levenberg-Marquardt Algorithm

The Levenberg-Marquardt algorithm is a variation of Newton's method that was designed for minimizing functions that are sums of squares of other nonlinear functions. This is very well suited to neural network training where the performance index is the mean squared error.

Basic Algorithm

Let's begin by considering the form of Newton's method where the performance index is a sum of squares. Recall from Chapter 9 that Newton's method for optimizing a performance index $F(\mathbf{x})$ is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k, \tag{12.19}$$

$$\text{where } \mathbf{A}_k \equiv \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_k} \text{ and } \mathbf{g}_k \equiv \nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_k}.$$

If we assume that $F(\mathbf{x})$ is a sum of squares function:

$$F(\mathbf{x}) = \sum_{i=1}^{N} v_i^2(\mathbf{x}) = \mathbf{v}^T(\mathbf{x})\mathbf{v}(\mathbf{x}), \tag{12.20}$$

then the jth element of the gradient would be

$$[\nabla F(\mathbf{x})]_j = \frac{\partial F(\mathbf{x})}{\partial x_j} = 2\sum_{i=1}^{N} v_i(\mathbf{x})\frac{\partial v_i(\mathbf{x})}{\partial x_j}. \tag{12.21}$$

The gradient can therefore be written in matrix form:

$$\nabla F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{v}(\mathbf{x}), \tag{12.22}$$

where

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial v_1(\mathbf{x})}{\partial x_1} & \frac{\partial v_1(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_1(\mathbf{x})}{\partial x_n} \\ \frac{\partial v_2(\mathbf{x})}{\partial x_1} & \frac{\partial v_2(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_2(\mathbf{x})}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial v_N(\mathbf{x})}{\partial x_1} & \frac{\partial v_N(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_N(\mathbf{x})}{\partial x_n} \end{bmatrix}. \tag{12.23}$$

Jacobian Matrix

is the *Jacobian matrix*.

Next we want to find the Hessian matrix. The k, j element of the Hessian matrix would be

$$[\nabla^2 F(\mathbf{x})]_{k,j} = \frac{\partial^2 F(\mathbf{x})}{\partial x_k \partial x_j} = 2\sum_{i=1}^{N}\left\{\frac{\partial v_i(\mathbf{x})}{\partial x_k}\frac{\partial v_i(\mathbf{x})}{\partial x_j} + v_i(\mathbf{x})\frac{\partial^2 v_i(\mathbf{x})}{\partial x_k \partial x_j}\right\}. \tag{12.24}$$

The Hessian matrix can then be expressed in matrix form:

$$\nabla^2 F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) + 2\mathbf{S}(\mathbf{x}), \tag{12.25}$$

where

$$\mathbf{S}(\mathbf{x}) = \sum_{i=1}^{N} v_i(\mathbf{x})\nabla^2 v_i(\mathbf{x}) . \tag{12.26}$$

If we assume that $\mathbf{S}(\mathbf{x})$ is small, we can approximate the Hessian matrix as

$$\nabla^2 F(\mathbf{x}) \cong 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) . \tag{12.27}$$

Gauss-Newton

If we then substitute Eq. (12.27) and Eq. (12.22) into Eq. (12.19), we obtain the *Gauss-Newton* method:

$$\begin{aligned}\mathbf{x}_{k+1} &= \mathbf{x}_k - [2\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1} 2\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) \\ &= \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) .\end{aligned} \tag{12.28}$$

Note that the advantage of Gauss-Newton over the standard Newton's method is that it does not require calculation of second derivatives.

One problem with the Gauss-Newton method is that the matrix $\mathbf{H} = \mathbf{J}^T\mathbf{J}$ may not be invertible. This can be overcome by using the following modification to the approximate Hessian matrix:

$$\mathbf{G} = \mathbf{H} + \mu\mathbf{I} . \tag{12.29}$$

To see how this matrix can be made invertible, suppose that the eigenvalues and eigenvectors of $\mathbf{H}$ are $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ and $\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n\}$. Then

$$\mathbf{G}\mathbf{z}_i = [\mathbf{H} + \mu\mathbf{I}]\mathbf{z}_i = \mathbf{H}\mathbf{z}_i + \mu\mathbf{z}_i = \lambda_i\mathbf{z}_i + \mu\mathbf{z}_i = (\lambda_i + \mu)\mathbf{z}_i . \tag{12.30}$$

Therefore the eigenvectors of $\mathbf{G}$ are the same as the eigenvectors of $\mathbf{H}$, and the eigenvalues of $\mathbf{G}$ are $(\lambda_i + \mu)$. $\mathbf{G}$ can be made positive definite by increasing μ until $(\lambda_i + \mu) > 0$ for all i, and therefore the matrix will be invertible.

Levenberg-Marquardt

This leads to the *Levenberg-Marquardt* algorithm [Scal85]:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k\mathbf{I}]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) . \tag{12.31}$$

or

$$\Delta\mathbf{x}_k = -[\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k\mathbf{I}]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) . \tag{12.32}$$

This algorithm has the very useful feature that as μ_k is increased it approaches the steepest descent algorithm with small learning rate:

$$\mathbf{x}_{k+1} \cong \mathbf{x}_k - \frac{1}{\mu_k}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) = \mathbf{x}_k - \frac{1}{2\mu_k}\nabla F(\mathbf{x}), \text{ for large } \mu_k, \qquad (12.33)$$

while as μ_k is decreased to zero the algorithm becomes Gauss-Newton.

The algorithm begins with μ_k set to some small value (e.g., $\mu_k = 0.01$). If a step does not yield a smaller value for $F(\mathbf{x})$, then the step is repeated with μ_k multiplied by some factor $\vartheta > 1$ (e.g., $\vartheta = 10$). Eventually $F(\mathbf{x})$ should decrease, since we would be taking a small step in the direction of steepest descent. If a step does produce a smaller value for $F(\mathbf{x})$, then μ_k is divided by ϑ for the next step, so that the algorithm will approach Gauss-Newton, which should provide faster convergence. The algorithm provides a nice compromise between the speed of Newton's method and the guaranteed convergence of steepest descent.

Now let's see how we can apply the Levenberg-Marquardt algorithm to the multilayer network training problem. The performance index for multilayer network training is the mean squared error (see Eq. (11.11)). If each target occurs with equal probability, the mean squared error is proportional to the sum of squared errors over the Q targets in the training set:

$$\begin{aligned} F(\mathbf{x}) &= \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{a}_q)^T(\mathbf{t}_q - \mathbf{a}_q) \\ &= \sum_{q=1}^{Q} \mathbf{e}_q^T\mathbf{e}_q = \sum_{q=1}^{Q}\sum_{j=1}^{S^M} (e_{j,q})^2 = \sum_{i=1}^{N} (v_i)^2, \end{aligned} \qquad (12.34)$$

where $e_{j,q}$ is the jth element of the error for the qth input/target pair.

Eq. (12.34) is equivalent to the performance index, Eq. (12.20), for which Levenberg-Marquardt was designed. Therefore it should be a straightforward matter to adapt the algorithm for network training. It turns out that this is true in concept, but it does require some care in working out the details.

Jacobian Calculation

The key step in the Levenberg-Marquardt algorithm is the computation of the Jacobian matrix. To perform this computation we will use a variation of the backpropagation algorithm. Recall that in the standard backpropagation procedure we compute the derivatives of the squared errors, with respect to the weights and biases of the network. To create the Jacobian matrix we need to compute the derivatives of the errors, instead of the derivatives of the squared errors.

It is a simple matter conceptually to modify the backpropagation algorithm to compute the elements of the Jacobian matrix. Unfortunately, although

the basic concept is simple, the details of the implementation can be a little tricky. For that reason you may want to skim through the rest of this section on your first reading, in order to obtain an overview of the general flow of the presentation, and return later to pick up the details. It may also be helpful to review the development of the backpropagation algorithm in Chapter 11 before proceeding.

Before we present the procedure for computing the Jacobian, let's take a closer look at its form (Eq. (12.23)). Note that the error vector is

$$\mathbf{v}^T = \begin{bmatrix} v_1 & v_2 & \cdots & v_N \end{bmatrix} = \begin{bmatrix} e_{1,1} & e_{2,1} & \cdots & e_{S^M,1} & e_{1,2} & \cdots & e_{S^M,Q} \end{bmatrix}, \tag{12.35}$$

the parameter vector is

$$\mathbf{x}^T = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} = \begin{bmatrix} w^1_{1,1} & w^1_{1,2} & \cdots & w^1_{S^1,R} & b^1_1 & \cdots & b^1_{S^1} & w^2_{1,1} & \cdots & b^M_{S^M} \end{bmatrix}, \tag{12.36}$$

$N = Q \times S^M$ and $n = S^1(R+1) + S^2(S^1+1) + \cdots + S^M(S^{M-1}+1)$.

Therefore, if we make these substitutions into Eq. (12.23), the Jacobian matrix for multilayer network training can be written

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w^1_{1,1}} & \frac{\partial e_{1,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{1,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{1,1}}{\partial b^1_1} & \cdots \\ \frac{\partial e_{2,1}}{\partial w^1_{1,1}} & \frac{\partial e_{2,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{2,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{2,1}}{\partial b^1_1} & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \\ \frac{\partial e_{S^M,1}}{\partial w^1_{1,1}} & \frac{\partial e_{S^M,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{S^M,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{S^M,1}}{\partial b^1_1} & \cdots \\ \frac{\partial e_{1,2}}{\partial w^1_{1,1}} & \frac{\partial e_{1,2}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{1,2}}{\partial w^1_{S^1,R}} & \frac{\partial e_{1,2}}{\partial b^1_1} & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \end{bmatrix}. \tag{12.37}$$

The terms in this Jacobian matrix can be computed by a simple modification to the backpropagation algorithm.

Standard backpropagation calculates terms like

$$\frac{\partial \hat{F}(\mathbf{x})}{\partial x_l} = \frac{\partial \mathbf{e}_q^T \mathbf{e}_q}{\partial x_l}. \tag{12.38}$$

For the elements of the Jacobian matrix that are needed for the Levenberg-Marquardt algorithm we need to calculate terms like

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial x_l}. \tag{12.39}$$

Recall from Eq. (11.18) in our derivation of backpropagation that

$$\frac{\partial \hat{F}}{\partial w^m_{i,j}} = \frac{\partial \hat{F}}{\partial n^m_i} \times \frac{\partial n^m_i}{\partial w^m_{i,j}}, \tag{12.40}$$

where the first term on the right-hand side was defined as the sensitivity:

$$s^m_i \equiv \frac{\partial \hat{F}}{\partial n^m_i}. \tag{12.41}$$

The backpropagation process computed the sensitivities through a recurrence relationship from the last layer backward to the first layer. We can use the same concept to compute the terms needed for the Jacobian matrix (Eq. (12.37)) if we define a new *Marquardt sensitivity*:

Marquardt Sensitivity

$$\tilde{s}^m_{i,h} \equiv \frac{\partial v_h}{\partial n^m_{i,q}} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}}, \tag{12.42}$$

where, from Eq. (12.35), $h = (q-1)S^M + k$.

Now we can compute elements of the Jacobian by

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial w^m_{i,j}} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \times \frac{\partial n^m_{i,q}}{\partial w^m_{i,j}} = \tilde{s}^m_{i,h} \times \frac{\partial n^m_{i,q}}{\partial w^m_{i,j}} = \tilde{s}^m_{i,h} \times a^{m-1}_{j,q}, \tag{12.43}$$

or if x_l is a bias,

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial b^m_i} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \times \frac{\partial n^m_{i,q}}{\partial b^m_i} = \tilde{s}^m_{i,h} \times \frac{\partial n^m_{i,q}}{\partial b^m_i} = \tilde{s}^m_{i,h}. \tag{12.44}$$

The Marquardt sensitivities can be computed through the same recurrence relations as the standard sensitivities (Eq. (11.35)) with one modification at the final layer, which for standard backpropagation is computed with Eq. (11.40). For the Marquardt sensitivities at the final layer we have

$$\tilde{s}_{i,h}^{M} = \frac{\partial v_h}{\partial n_{i,q}^{M}} = \frac{\partial e_{k,q}}{\partial n_{i,q}^{M}} = \frac{\partial (t_{k,q} - a_{k,q}^{M})}{\partial n_{i,q}^{M}} = -\frac{\partial a_{k,q}^{M}}{\partial n_{i,q}^{M}}$$

$$= \begin{cases} -\dot{f}^{M}(n_{i,q}^{M}) & \text{for } i = k \\ 0 & \text{for } i \neq k \end{cases} . \tag{12.45}$$

Therefore when the input $\mathbf{p}_q$ has been applied to the network and the corresponding network output $\mathbf{a}_q^M$ has been computed, the Levenberg-Marquardt backpropagation is initialized with

$$\tilde{\mathbf{S}}_q^M = -\dot{\mathbf{F}}^M(\mathbf{n}_q^M), \tag{12.46}$$

where $\dot{\mathbf{F}}^M(\mathbf{n}^M)$ is defined in Eq. (11.34). Each column of the matrix $\tilde{\mathbf{S}}_q^M$ must be backpropagated through the network using Eq. (11.35) to produce one row of the Jacobian matrix. The columns can also be backpropagated together using

$$\tilde{\mathbf{S}}_q^m = \dot{\mathbf{F}}^m(\mathbf{n}_q^m)(\mathbf{W}^{m+1})^T \tilde{\mathbf{S}}_q^{m+1} . \tag{12.47}$$

The total Marquardt sensitivity matrices for each layer are then created by augmenting the matrices computed for each input:

$$\tilde{\mathbf{S}}^m = \left[\tilde{\mathbf{S}}_1^m \middle| \tilde{\mathbf{S}}_2^m \middle| \cdots \middle| \tilde{\mathbf{S}}_Q^m\right] . \tag{12.48}$$

Note that for each input that is presented to the network we will backpropagate S^M sensitivity vectors. This is because we are computing the derivatives of each individual error, rather than the derivative of the sum of squares of the errors. For every input applied to the network there will be S^M errors (one for each element of the network output). For each error there will be one row of the Jacobian matrix.

After the sensitivities have been backpropagated, the Jacobian matrix is computed using Eq. (12.43) and Eq. (12.44). See Problem P12.5 for a numerical illustration of the Jacobian computation.

LMBP

The iterations of the Levenberg-Marquardt backpropagation algorithm (*LMBP*) can be summarized as follows:

1. Present all inputs to the network and compute the corresponding network outputs (using Eq. (11.41) and Eq. (11.42)) and the errors $\mathbf{e}_q = \mathbf{t}_q - \mathbf{a}_q^M$. Compute the sum of squared errors over all inputs, $F(\mathbf{x})$,

using Eq. (12.34).

2. Compute the Jacobian matrix, Eq. (12.37). Calculate the sensitivities with the recurrence relations Eq. (12.47), after initializing with Eq. (12.46). Augment the individual matrices into the Marquardt sensitivities using Eq. (12.48). Compute the elements of the Jacobian matrix with Eq. (12.43) and Eq. (12.44).

3. Solve Eq. (12.32) to obtain $\Delta \mathbf{x}_k$.

4. Recompute the sum of squared errors using $\mathbf{x}_k + \Delta \mathbf{x}_k$. If this new sum of squares is smaller than that computed in step 1, then divide μ by ϑ, let $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k$ and go back to step 1. If the sum of squares is not reduced, then multiply μ by ϑ and go back to step 3.

The algorithm is assumed to have converged when the norm of the gradient, Eq. (12.22), is less than some predetermined value, or when the sum of squares has been reduced to some error goal.

To illustrate LMBP, let's apply it to the function approximation problem introduced at the beginning of this chapter. We will begin by looking at the basic Levenberg-Marquardt step. Figure 12.17 illustrates the possible steps the LMBP algorithm could take on the first iteration.

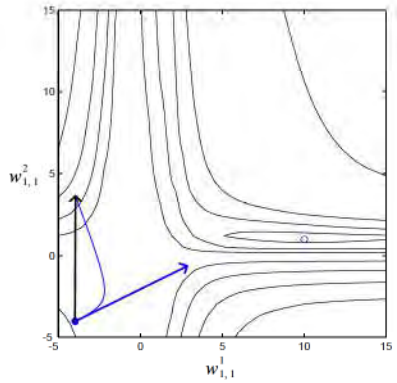


Figure 12.17 Levenberg-Marquardt Step

The black arrow represents the direction taken for small μ_k, which corresponds to the Gauss-Newton direction. The blue arrow represents the direction taken for large μ_k, which corresponds to the steepest descent direction. (This was the initial direction taken by all of the previous algorithms discussed.) The blue curve represents the Levenberg-Marquardt step for all intermediate values of μ_k. Note that as μ_k is increased the algorithm moves toward a small step in the direction of steepest descent. This guarantees that the algorithm will always be able to reduce the sum of squares at each iteration.

Figure 12.18 shows the path of the LMBP trajectory to convergence, with $\mu_0 = 0.01$ and $\vartheta = 5$. Note that the algorithm converges in fewer iterations than any of the methods we have discussed so far. Of course this algorithm also requires more computation per iteration than any of the other algorithms, since it involves a matrix inversion. Even given the large number of computations, however, the LMBP algorithm appears to be the fastest neural network training algorithm for moderate numbers of network parameters [HaMe94].

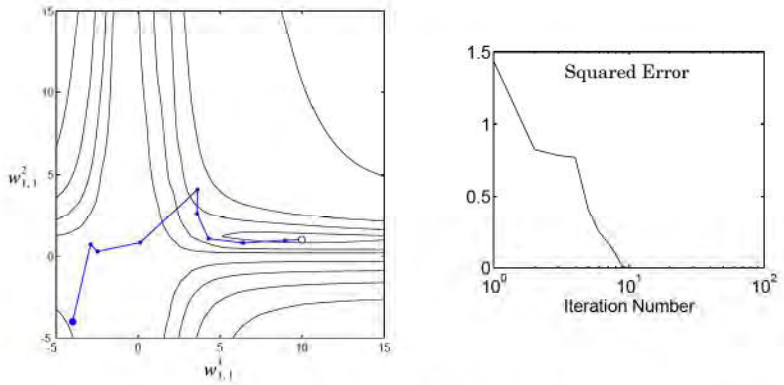


Figure 12.18 LMBP Trajectory



To experiment with the LMBP algorithm, use the MATLAB® Neural Network Design Demonstrations Marquardt Step (`nnd12ms`) *and Marquardt Backpropagation* (`nnd12m`).

The key drawback of the LMBP algorithm is the storage requirement. The algorithm must store the approximate Hessian matrix $\mathbf{J}^T\mathbf{J}$. This is an $n \times n$ matrix, where n is the number of parameters (weights and biases) in the network. Recall that the other methods discussed need only store the gradient, which is an n-dimensional vector. When the number of parameters is very large, it may be impractical to use the Levenberg-Marquardt algorithm. (What constitutes "very large" depends on the available memory on your computer, but typically a few thousand parameters is an upper limit.)

Summary of Results

Heuristic Variations of Backpropagation

Batching

The parameters are updated only after the entire training set has been presented. The gradients calculated for each training example are averaged together to produce a more accurate estimate of the gradient. (If the training set is complete, i.e., covers all possible input/output pairs, then the gradient estimate will be exact.)

Backpropagation with Momentum (MOBP)

$$\Delta \mathbf{W}^m(k) = \gamma \Delta \mathbf{W}^m(k-1) - (1-\gamma)\alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T$$

$$\Delta \mathbf{b}^m(k) = \gamma \Delta \mathbf{b}^m(k-1) - (1-\gamma)\alpha \mathbf{s}^m$$

Variable Learning Rate Backpropagation (VLBP)

1. If the squared error (over the entire training set) increases by more than some set percentage ζ (typically one to five percent) after a weight update, then the weight update is discarded, the learning rate is multiplied by some factor $\rho < 1$, and the momentum coefficient γ (if it is used) is set to zero.

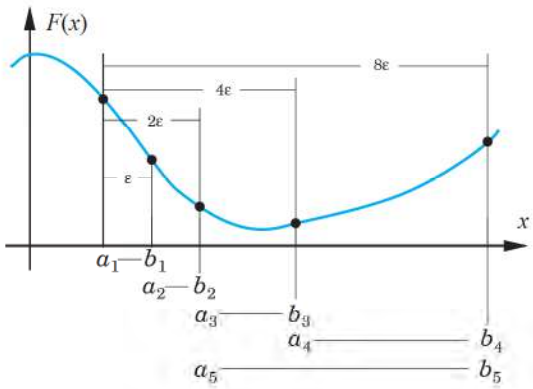
2. If the squared error decreases after a weight update, then the weight update is accepted and the learning rate is multiplied by some factor $\eta > 1$. If γ has been previously set to zero, it is reset to its original value.

3. If the squared error increases by less than ζ, then the weight update is accepted but the learning rate and the momentum coefficient are unchanged.

Numerical Optimization Techniques

Conjugate Gradient

Interval Location



Interval Reduction (Golden Section Search)

$\tau = 0.618$

Set $c_1 = a_1 + (1-\tau)(b_1 - a_1)$, $F_c = F(c_1)$.

$d_1 = b_1 - (1-\tau)(b_1 - a_1)$, $F_d = F(d_1)$.

For $k = 1, 2, \ldots$ repeat

If $F_c < F_d$ then

Set $a_{k+1} = a_k$; $b_{k+1} = d_k$; $d_{k+1} = c_k$

$c_{k+1} = a_{k+1} + (1-\tau)(b_{k+1} - a_{k+1})$

$F_d = F_c$; $F_c = F(c_{k+1})$

else

Set $a_{k+1} = c_k$; $b_{k+1} = b_k$; $c_{k+1} = d_k$

$d_{k+1} = b_{k+1} - (1-\tau)(b_{k+1} - a_{k+1})$

$F_c = F_d$; $F_d = F(d_{k+1})$

end

end until $b_{k+1} - a_{k+1} < tol$

Levenberg-Marquardt Backpropagation (LMBP)

$$\Delta \mathbf{x}_k = -[\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k \mathbf{I}]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)$$

$$\mathbf{v}^T = \begin{bmatrix} v_1 & v_2 & \dots & v_N \end{bmatrix} = \begin{bmatrix} e_{1,1} & e_{2,1} & \dots & e_{S^M,1} & e_{1,2} & \dots & e_{S^M,Q} \end{bmatrix}$$

$$\mathbf{x}^T = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix} = \begin{bmatrix} w^1_{1,1} & w^1_{1,2} & \dots & w^1_{S^1,R} & b^1_1 & \dots & b^1_{S^1} & w^2_{1,1} & \dots & b^M_{S^M} \end{bmatrix}$$

$$N = Q \times S^M \text{ and } n = S^1(R+1) + S^2(S^1+1) + \dots + S^M(S^{M-1}+1)$$

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w^1_{1,1}} & \frac{\partial e_{1,1}}{\partial w^1_{1,2}} & \dots & \frac{\partial e_{1,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{1,1}}{\partial b^1_1} & \dots \\ \frac{\partial e_{2,1}}{\partial w^1_{1,1}} & \frac{\partial e_{2,1}}{\partial w^1_{1,2}} & \dots & \frac{\partial e_{2,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{2,1}}{\partial b^1_1} & \dots \\ \vdots & \vdots & & \vdots & \vdots & \\ \frac{\partial e_{S^M,1}}{\partial w^1_{1,1}} & \frac{\partial e_{S^M,1}}{\partial w^1_{1,2}} & \dots & \frac{\partial e_{S^M,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{S^M,1}}{\partial b^1_1} & \dots \\ \frac{\partial e_{1,2}}{\partial w^1_{1,1}} & \frac{\partial e_{1,2}}{\partial w^1_{1,2}} & \dots & \frac{\partial e_{1,2}}{\partial w^1_{S^1,R}} & \frac{\partial e_{1,2}}{\partial b^1_1} & \dots \\ \vdots & \vdots & & \vdots & \vdots & \end{bmatrix}$$

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial w^m_{i,j}} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \times \frac{\partial n^m_{i,q}}{\partial w^m_{i,j}} = \tilde{s}^m_{i,h} \times \frac{\partial n^m_{i,q}}{\partial w^m_{i,j}} = \tilde{s}^m_{i,h} \times a^{m-1}_{j,q} \text{ for weight } x_l$$

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial b^m_i} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \times \frac{\partial n^m_{i,q}}{\partial b^m_i} = \tilde{s}^m_{i,h} \times \frac{\partial n^m_{i,q}}{\partial b^m_i} = \tilde{s}^m_{i,h} \text{ for bias } x_l$$

$$\tilde{s}^m_{i,h} \equiv \frac{\partial v_h}{\partial n^m_{i,q}} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \text{ (Marquardt Sensitivity) where } h = (q-1)S^M + k$$

$$\tilde{\mathbf{S}}^M_q = -\dot{\mathbf{F}}^M(\mathbf{n}^M_q)$$

$$\tilde{\mathbf{S}}^m_q = \dot{\mathbf{F}}^m(\mathbf{n}^m_q)(\mathbf{W}^{m+1})^T\tilde{\mathbf{S}}^{m+1}_q$$

$$\tilde{\mathbf{S}}^m = \left[\tilde{\mathbf{S}}^m_1 \middle| \tilde{\mathbf{S}}^m_2 \middle| \dots \middle| \tilde{\mathbf{S}}^m_Q\right]$$

Levenberg-Marquardt Iterations

1. Present all inputs to the network and compute the corresponding network outputs (using Eq. (11.41) and Eq. (11.42)) and the errors $\mathbf{e}_q = \mathbf{t}_q - \mathbf{a}_q^M$. Compute the sum of squared errors over all inputs, $F(\mathbf{x})$, using Eq. (12.34).

2. Compute the Jacobian matrix, Eq. (12.37). Calculate the sensitivities with the recurrence relations Eq. (12.47), after initializing with Eq. (12.46). Augment the individual matrices into the Marquardt sensitivities using Eq. (12.48). Compute the elements of the Jacobian matrix with Eq. (12.43) and Eq. (12.44).

3. Solve Eq. (12.32) to obtain $\Delta\mathbf{x}_k$.

4. Recompute the sum of squared errors using $\mathbf{x}_k + \Delta\mathbf{x}_k$. If this new sum of squares is smaller than that computed in step 1, then divide μ by ϑ, let $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$ and go back to step 1. If the sum of squares is not reduced, then multiply μ by ϑ and go back to step 3.

Solved Problems

P12.1 We want to train the network shown in Figure P12.1 on the training set

$$\left\{(\mathbf{p}_1 = \begin{bmatrix}-3\end{bmatrix}), (\mathbf{t}_1 = \begin{bmatrix}0.5\end{bmatrix})\right\}, \left\{(\mathbf{p}_2 = \begin{bmatrix}2\end{bmatrix}), (\mathbf{t}_2 = \begin{bmatrix}1\end{bmatrix})\right\},$$

starting from the initial guess

$$w(0) = 0.4\,,\; b(0) = 0.15\,.$$

Demonstrate the effect of batching by computing the direction of the initial step for SDBP with and without batching.

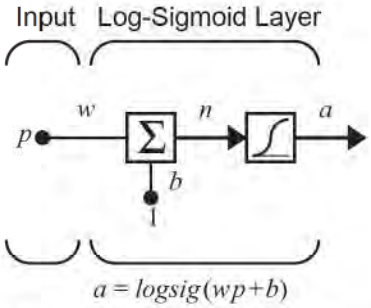


Figure P12.1 Network for Problem P12.1

Let's begin by computing the direction of the initial step if batching is not used. In this case the first step is computed from the first input/target pair. The forward and backpropagation steps are

$$a = logsig(wp + b) = \frac{1}{1 + \exp(-(0.4(-3) + 0.15))} = 0.2592$$

$$e = t - a = 0.5 - 0.2592 = 0.2408$$

$$s = -2\dot{f}(n)e = -2a(1-a)e = -2(0.2592)(1-0.2592)0.2408 = -0.0925\,.$$

The direction of the initial step is the negative of the gradient. For the weight this will be

$$-sp = -(-0.0925)(-3) = -0.2774\,.$$

For the bias we have

$$-s = -(-0.0925) = 0.0925\,.$$

Therefore the direction of the initial step in the (w, b) plane would be

$$\begin{bmatrix} -0.2774 \\ 0.0925 \end{bmatrix}.$$

Now let's consider the initial direction for the batch mode algorithm. In this case the gradient is found by adding together the individual gradients found from the two sets of input/target pairs. For this we need to apply the second input to the network and perform the forward and backpropagation steps:

$$a = logsig(wp + b) = \frac{1}{1 + \exp(-(0.4(2) + 0.15))} = 0.7211$$

$$e = t - a = 1 - 0.7211 = 0.2789$$

$$s = -2\dot{f}(n)e = -2a(1-a)e = -2(0.7211)(1-0.7211)0.2789 = -0.1122.$$

The direction of the step is the negative of the gradient. For the weight this will be

$$-sp = -(-0.1122)(2) = 0.2243.$$

For the bias we have

$$-s = -(-0.1122) = 0.1122.$$

The partial gradient for the second input/target pair is therefore

$$\begin{bmatrix} 0.2243 \\ 0.1122 \end{bmatrix}.$$

If we now add the results from the two input/target pairs we find the direction of the first step of the batch mode SDBP to be

$$\frac{1}{2}\left(\begin{bmatrix} -0.2774 \\ 0.0925 \end{bmatrix} + \begin{bmatrix} 0.2243 \\ 0.1122 \end{bmatrix}\right) = \frac{1}{2}\begin{bmatrix} -0.0531 \\ 0.2047 \end{bmatrix} = \begin{bmatrix} -0.0265 \\ 0.1023 \end{bmatrix}.$$

The results are illustrated in Figure P12.2. The blue circle indicates the initial guess. The two blue arrows represent the directions of the partial gradients for each of the two input/target pairs, and the black arrow represents the direction of the total gradient. The function that is plotted is the sum of squared errors for the entire training set. Note that the individual partial gradients can point in quite different directions than the true gradient. However, on the average, over several iterations, the path will generally follow the steepest descent trajectory.

The relative effectiveness of the batch mode over the incremental approach depends very much on the particular problem. The incremental approach requires less storage, and, if the inputs are presented randomly to the network, the trajectory is stochastic, which makes the algorithm somewhat less likely to be trapped in a local minimum. It may also take longer to converge than the batch mode algorithm.

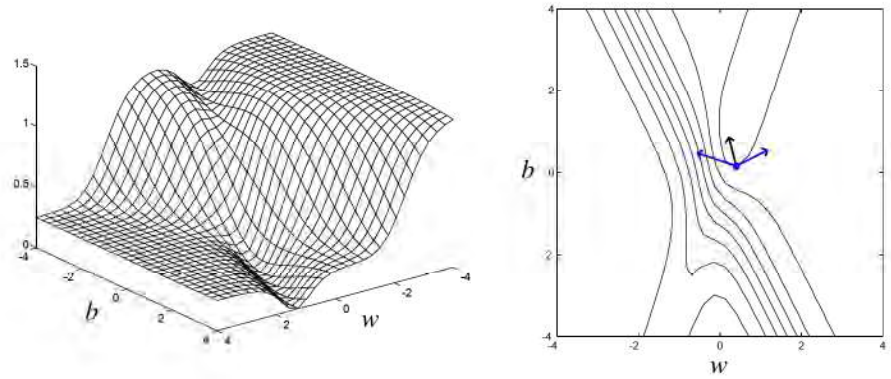


Figure P12.2 Effect of Batching in Problem P12.1

P12.2 In Chapter 9 we proved that the steepest descent algorithm, when applied to a quadratic function, would be stable if the learning rate was less than 2 divided by the maximum eigenvalue of the Hessian matrix. Show that if a momentum term is added to the steepest descent algorithm there will always be a momentum coefficient that will make the algorithm stable, regardless of the learning rate. Follow the format of the proof on page 9-6.

The standard steepest descent algorithm is

$$\Delta \mathbf{x}_k = -\alpha \nabla F(\mathbf{x}_k) = -\alpha \mathbf{g}_k ,$$

If we add momentum this becomes

$$\Delta \mathbf{x}_k = \gamma \Delta \mathbf{x}_{k-1} - (1-\gamma)\alpha \mathbf{g}_k .$$

Recall from Chapter 8 that the quadratic function has the form

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathbf{A}\mathbf{x} + \mathbf{d}^T \mathbf{x} + c ,$$

and the gradient of the quadratic function is

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d} .$$

If we now insert this expression into our expression for the steepest descent algorithm with momentum we obtain

$$\Delta \mathbf{x}_k = \gamma \Delta \mathbf{x}_{k-1} - (1-\gamma)\alpha(\mathbf{A}\mathbf{x}_k + \mathbf{d}).$$

Using the definition $\Delta \mathbf{x}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ this can be rewritten

$$\mathbf{x}_{k+1} - \mathbf{x}_k = \gamma(\mathbf{x}_k - \mathbf{x}_{k-1}) - (1-\gamma)\alpha(\mathbf{A}\mathbf{x}_k + \mathbf{d})$$

or

$$\mathbf{x}_{k+1} = [(1+\gamma)\mathbf{I} - (1-\gamma)\alpha\mathbf{A}]\mathbf{x}_k - \gamma\mathbf{x}_{k-1} - (1-\gamma)\alpha\mathbf{d}.$$

Now define a new vector

$$\mathbf{x}_k = \begin{bmatrix} \mathbf{x}_{k-1} \\ \mathbf{x}_k \end{bmatrix}.$$

The momentum variation of steepest descent can then be written

$$\mathbf{x}_{k+1} = \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ -\gamma\mathbf{I} & [(1+\gamma)\mathbf{I} - (1-\gamma)\alpha\mathbf{A}] \end{bmatrix} \mathbf{x}_k + \begin{bmatrix} \mathbf{0} \\ -(1-\gamma)\alpha\mathbf{d} \end{bmatrix} = \mathbf{W}\mathbf{x}_k + \mathbf{v}.$$

This is a linear dynamic system that will be stable if the eigenvalues of $\mathbf{W}$ are less than one in magnitude. We will find the eigenvalues of $\mathbf{W}$ in stages. First, rewrite $\mathbf{W}$ as

$$\mathbf{W} = \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ -\gamma\mathbf{I} & \mathbf{T} \end{bmatrix} \text{ where } \mathbf{T} = [(1+\gamma)\mathbf{I} - (1-\gamma)\alpha\mathbf{A}].$$

The eigenvalues and eigenvectors of $\mathbf{W}$ should satisfy

$$\mathbf{W}\mathbf{z}^w = \lambda^w \mathbf{z}^w, \text{ or } \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ -\gamma\mathbf{I} & \mathbf{T} \end{bmatrix} \begin{bmatrix} \mathbf{z}_1^w \\ \mathbf{z}_2^w \end{bmatrix} = \lambda^w \begin{bmatrix} \mathbf{z}_1^w \\ \mathbf{z}_2^w \end{bmatrix}.$$

This means that

$$\mathbf{z}_2^w = \lambda^w \mathbf{z}_1^w \text{ and } -\gamma\mathbf{z}_1^w + \mathbf{T}\mathbf{z}_2^w = \lambda^w \mathbf{z}_2^w.$$

At this point we will choose $\mathbf{z}_2^w$ to be an eigenvector of the matrix $\mathbf{T}$, with corresponding eigenvalue λ^t. (If this choice is not appropriate it will lead to a contradiction.) Therefore the previous equations become

$$\mathbf{z}_2^w = \lambda^w \mathbf{z}_1^w \text{ and } -\gamma \mathbf{z}_1^w + \lambda^t \mathbf{z}_2^w = \lambda^w \mathbf{z}_2^w .$$

If we substitute the first equation into the second equation we find

$$-\frac{\gamma}{\lambda^w}\mathbf{z}_2^w + \lambda^t \mathbf{z}_2^w = \lambda^w \mathbf{z}_2^w \text{ or } [(\lambda^w)^2 - \lambda^t(\lambda^w) + \gamma]\mathbf{z}_2^w = 0 .$$

Therefore for each eigenvalue λ^t of $\mathbf{T}$ there will be two eigenvalues λ^w of $\mathbf{W}$ that are roots of the quadratic equation

$$(\lambda^w)^2 - \lambda^t(\lambda^w) + \gamma = 0 .$$

From the quadratic formula we have

$$\lambda^w = \frac{\lambda^t \pm \sqrt{(\lambda^t)^2 - 4\gamma}}{2} .$$

For the algorithm to be stable the magnitude of each eigenvalue must be less than 1. We will show that there always exists some range of γ for which this is true.

Note that if the eigenvalues λ^w are complex then their magnitude will be $\sqrt{\gamma}$:

$$|\lambda^w| = \sqrt{\frac{(\lambda^t)^2}{4} + \frac{4\gamma - (\lambda^t)^2}{4}} = \sqrt{\gamma} .$$

(This is true only for real λ^t. We will show later that λ^t is real.) Since γ is between 0 and 1, the magnitude of the eigenvalue must be less than 1. It remains to show that there exists some range of γ for which all of the eigenvalues are complex.

In order for λ^w to be complex we must have

$$(\lambda^t)^2 - 4\gamma < 0 \text{ or } |\lambda^t| < 2\sqrt{\gamma} .$$

Let's now consider the eigenvalues λ^t of $\mathbf{T}$. These eigenvalues can be expressed in terms of the eigenvalues of $\mathbf{A}$. Let $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ and $\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n\}$ be the eigenvalues and eigenvectors of the Hessian matrix. Then

$$\mathbf{T}\mathbf{z}_i = [(1+\gamma)\mathbf{I} - (1-\gamma)\alpha\mathbf{A}]\mathbf{z}_i = (1+\gamma)\mathbf{z}_i - (1-\gamma)\alpha\mathbf{A}\mathbf{z}_i$$

$$= (1+\gamma)\mathbf{z}_i - (1-\gamma)\alpha\lambda_i\mathbf{z}_i = \{(1+\gamma) - (1-\gamma)\alpha\lambda_i\}\mathbf{z}_i = \lambda_i^t \mathbf{z}_i .$$

Therefore the eigenvectors of $\mathbf{T}$ are the same as the eigenvectors of $\mathbf{A}$, and the eigenvalues of $\mathbf{T}$ are

$$\lambda_i^t = \{(1+\gamma) - (1-\gamma)\alpha\lambda_i\}.$$

(Note that λ_i^t is real, since γ, α and λ_i for symmetric $\mathbf{A}$ are real.) Therefore, in order for λ^w to be complex we must have

$$|\lambda^t| < 2\sqrt{\gamma} \text{ or } |(1+\gamma) - (1-\gamma)\alpha\lambda_i| < 2\sqrt{\gamma}.$$

For $\gamma = 1$ both sides of the inequality will equal 2. The function on the right of the inequality, as a function of γ, has a slope of 1 at $\gamma = 1$. The function on the left of the inequality has a slope of $1 + \alpha\lambda_i$. Since the eigenvalues of the Hessian will be positive real numbers if the function has a strong minimum, and the learning rate is a positive number, this slope must be greater than 1. This shows that the inequality will always hold for γ close enough to 1.

To summarize the results, we have shown that if a momentum term is added to the steepest descent algorithm on a quadratic function, then there will always be a momentum coefficient that will make the algorithm stable, regardless of the learning rate. In addition we have shown that if γ is close enough to 1, then the magnitudes of the eigenvalues of $\mathbf{W}$ will be $\sqrt{\gamma}$. It can be shown (see [Brog91]) that the magnitudes of the eigenvalues determine how fast the algorithm will converge. The smaller the magnitude, the faster the convergence. As the magnitude approaches 1, the convergence time increases.

We can demonstrate these results using the example on page 9-7. There we showed that the steepest descent algorithm, when applied to the function $F(\mathbf{x}) = x_1^2 + 25x_2^2$, was unstable for a learning rate $\alpha \geq 0.4$. In Figure P12.3 we see the steepest descent trajectory (with momentum) with $\alpha = 0.041$ and $\gamma = 0.2$. Compare this trajectory with Figure 9.3, which uses the same learning rate but no momentum.

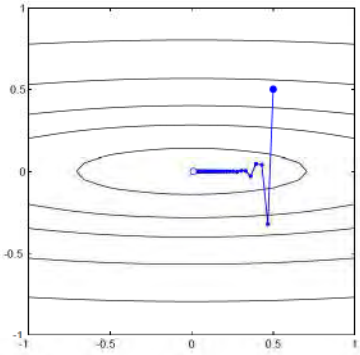


Figure P12.3 Trajectory for $\alpha = 0.041$ and $\gamma = 0.2$

P12.3 Execute three iterations of the variable learning rate algorithm on the following function (from the Chapter 9 example on page 9-7):

$$F(\mathbf{x}) = x_1^2 + 25x_2^2,$$

starting from the initial guess

$$\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix},$$

and use the following values for the algorithm parameters:

$$\alpha = 0.05,\ \gamma = 0.2,\ \eta = 1.5,\ \rho = 0.5,\ \zeta = 5\%.$$

The first step is to evaluate the function at the initial guess:

$$F(\mathbf{x}_0) = \frac{1}{2}\mathbf{x}_0^T \begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix} \mathbf{x}_0 = \frac{1}{2}\begin{bmatrix} 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} = 6.5.$$

The next step is to find the gradient:

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 50x_2 \end{bmatrix}.$$

If we evaluate the gradient at the initial guess we find:

$$\mathbf{g}_0 = \nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}_0} = \begin{bmatrix} 1 \\ 25 \end{bmatrix}.$$

With the initial learning rate of $\alpha = 0.05$, the tentative first step of the algorithm is

$$\Delta\mathbf{x}_0 = \gamma\Delta\mathbf{x}_{-1} - (1-\gamma)\alpha\mathbf{g}_0 = 0.2\begin{bmatrix} 0 \\ 0 \end{bmatrix} - 0.8(0.05)\begin{bmatrix} 1 \\ 25 \end{bmatrix} = \begin{bmatrix} -0.04 \\ -1 \end{bmatrix}$$

$$\mathbf{x}_1^t = \mathbf{x}_0 + \Delta\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} + \begin{bmatrix} -0.04 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.46 \\ -0.5 \end{bmatrix}.$$

To verify that this is a valid step we must test the value of the function at this new point:

$$F(\mathbf{x}_1^t) = \frac{1}{2}(\mathbf{x}_1^t)^T\begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix}\mathbf{x}_1^t = \frac{1}{2}\begin{bmatrix} 0.46 & -0.5 \end{bmatrix}\begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix}\begin{bmatrix} 0.46 \\ -0.5 \end{bmatrix} = 6.4616.$$

This is less than $F(\mathbf{x}_0)$. Therefore this tentative step is accepted and the learning rate is increased:

$$\mathbf{x}_1 = \mathbf{x}_1^t = \begin{bmatrix} 0.46 \\ -0.5 \end{bmatrix},\ F(\mathbf{x}_1) = 6.4616 \text{ and } \alpha = \eta\alpha = 1.5(0.05) = 0.075.$$

The tentative second step of the algorithm is

$$\Delta\mathbf{x}_1 = \gamma\Delta\mathbf{x}_0 - (1-\gamma)\alpha\mathbf{g}_1 = 0.2\begin{bmatrix} -0.04 \\ -1 \end{bmatrix} - 0.8(0.075)\begin{bmatrix} 0.92 \\ -25 \end{bmatrix} = \begin{bmatrix} -0.0632 \\ 1.3 \end{bmatrix}$$

$$\mathbf{x}_2^t = \mathbf{x}_1 + \Delta\mathbf{x}_1 = \begin{bmatrix} 0.46 \\ -0.5 \end{bmatrix} + \begin{bmatrix} -0.0632 \\ 1.3 \end{bmatrix} = \begin{bmatrix} 0.3968 \\ 0.8 \end{bmatrix}.$$

We evaluate the function at this point:

$$F(\mathbf{x}_2^t) = \frac{1}{2}(\mathbf{x}_2^t)^T\begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix}\mathbf{x}_2^t = \frac{1}{2}\begin{bmatrix} 0.3968 & 0.8 \end{bmatrix}\begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix}\begin{bmatrix} 0.3968 \\ 0.8 \end{bmatrix} = 16.157.$$

Since this is more than 5% larger than $F(\mathbf{x}_1)$, we reject this step, reduce the learning rate and set the momentum coefficient to zero.

$$\mathbf{x}_2 = \mathbf{x}_1,\ F(\mathbf{x}_2) = F(\mathbf{x}_1) = 6.4616,\ \alpha = \rho\alpha = 0.5(0.075) = 0.0375,\ \gamma = 0$$

Now a new tentative step is computed (momentum is zero).

$$\Delta \mathbf{x}_2 = -\alpha \mathbf{g}_2 = -(0.0375)\begin{bmatrix} 0.92 \\ -25 \end{bmatrix} = \begin{bmatrix} -0.0345 \\ 0.9375 \end{bmatrix}$$

$$\mathbf{x}_3^t = \mathbf{x}_2 + \Delta \mathbf{x}_2 = \begin{bmatrix} 0.46 \\ -0.5 \end{bmatrix} + \begin{bmatrix} -0.0345 \\ 0.9375 \end{bmatrix} = \begin{bmatrix} 0.4255 \\ 0.4375 \end{bmatrix}$$

$$F(\mathbf{x}_3^t) = \frac{1}{2}(\mathbf{x}_3^t)^T \begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix} \mathbf{x}_3^t = \frac{1}{2}\begin{bmatrix} 0.4255 & 0.4375 \end{bmatrix}\begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix}\begin{bmatrix} 0.4255 \\ 0.4375 \end{bmatrix} = 4.966$$

This is less than $F(\mathbf{x}_2)$. Therefore this step is accepted, the momentum is reset to its original value, and the learning rate is increased.

$$\mathbf{x}_3 = \mathbf{x}_3^t, \gamma = 0.2, \alpha = \eta\alpha = 1.5(0.0375) = 0.05625$$

This completes the third iteration.

P12.4 Recall the example from Chapter 9 that we used to demonstrate the conjugate gradient algorithm (page 9-18):

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \mathbf{x},$$

with initial guess

$$\mathbf{x}_0 = \begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix}.$$

Perform one iteration of the conjugate gradient algorithm. For the linear minimization use interval location by function evaluation and interval reduction by the Golden Section search.

The gradient of this function is

$$\nabla F(\mathbf{x}) = \begin{bmatrix} 2x_1 + x_2 \\ x_1 + 2x_2 \end{bmatrix}.$$

As with steepest descent, the first search direction for the conjugate gradient algorithm is the negative of the gradient:

$$\mathbf{p}_0 = -\mathbf{g}_0 = -\nabla F(\mathbf{x})^T\Big|_{\mathbf{x}=\mathbf{x}_0} = \begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}.$$

For the first iteration we need to minimize $F(\mathbf{x})$ along the line

$$\mathbf{x}_1 = \mathbf{x}_0 + \alpha_0 \mathbf{p}_0 = \begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix} + \alpha_0 \begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}.$$

The first step is interval location. Assume that the initial step size is $\varepsilon = 0.075$. Then the interval location would proceed as follows:

$$F(a_1) = F\left(\begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix}\right) = 0.5025,$$

$$b_1 = \varepsilon = 0.075,\ F(b_1) = F\left(\begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix} + 0.075\begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}\right) = 0.3721$$

$$b_2 = 2\varepsilon = 0.15,\ F(b_2) = F\left(\begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix} + 0.15\begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}\right) = 0.2678$$

$$b_3 = 4\varepsilon = 0.3,\ F(b_3) = F\left(\begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix} + 0.3\begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}\right) = 0.1373$$

$$b_4 = 8\varepsilon = 0.6,\ F(b_4) = F\left(\begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix} + 0.6\begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}\right) = 0.1893.$$

Since the function increases between two consecutive evaluations we know that the minimum must occur in the interval $[0.15, 0.6]$. This process is illustrated by the open blue circles in Figure P12.4, and the final interval is indicated by the large open black circles.

The next step in the linear minimization is interval reduction using the Golden Section search. This proceeds as follows:

$$c_1 = a_1 + (1-\tau)(b_1 - a_1) = 0.15 + (0.382)(0.6 - 0.15) = 0.3219,$$

$$d_1 = b_1 - (1-\tau)(b_1 - a_1) = 0.6 - (0.382)(0.6 - 0.15) = 0.4281,$$

$$F_a = 0.2678,\ F_b = 0.1893,\ F_c = 0.1270,\ F_d = 0.1085.$$

Since $F_c > F_d$, we have

$$a_2 = c_1 = 0.3219,\ b_2 = b_1 = 0.6,\ c_2 = d_1 = 0.4281$$

$$d_2 = b_2 - (1-\tau)(b_2 - a_2) = 0.6 - (0.382)(0.6 - 0.3219) = 0.4938,$$

$$F_a = F_c = 0.1270,\ F_c = F_d = 0.1085,\ F_d = F(d_2) = 0.1232.$$

This time $F_c < F_d$, therefore

$$a_3 = a_2 = 0.3219,\ b_3 = d_2 = 0.4938,\ d_3 = c_2 = 0.4281,$$

$$c_3 = a_3 + (1-\tau)(b_3 - a_3) = 0.3219 + (0.382)(0.4938 - 0.3219) = 0.3876,$$

$$F_b = F_d = 0.1232,\ F_d = F_c = 0.1085,\ F_c = F(c_3) = 0.1094.$$

This routine continues until $b_{k+1} - a_{k+1} < tol$. The black dots in Figure P12.4 indicate the location of the new interior points, one for each iteration of the procedure. The final point is indicated by a blue dot. Compare this result with the first iteration shown in Figure 9.10.

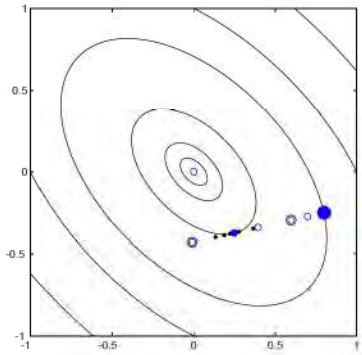


Figure P12.4 Linear Minimization Example

P12.5 To illustrate the computation of the Jacobian matrix for the Levenberg-Marquardt method, consider using the network of Figure P12.5 for function approximation. The network transfer functions are chosen to be

$$f^1(n) = (n)^2,\ f^2(n) = n.$$

Therefore their derivatives are

$$\dot{f}^1(n) = 2n,\ \dot{f}^2(n) = 1.$$

Assume that the training set consists of

$$\{(\mathbf{p}_1 = \begin{bmatrix}1\end{bmatrix}, (\mathbf{t}_1 = \begin{bmatrix}1\end{bmatrix})\} \ , \ \{(\mathbf{p}_2 = \begin{bmatrix}2\end{bmatrix}, (\mathbf{t}_2 = \begin{bmatrix}2\end{bmatrix})\} \ ,$$

and that the parameters are initialized to

$$\mathbf{W}^1 = \begin{bmatrix}1\end{bmatrix} , \ \mathbf{b}^1 = \begin{bmatrix}0\end{bmatrix} , \ \mathbf{W}^2 = \begin{bmatrix}2\end{bmatrix} , \ \mathbf{b}^1 = \begin{bmatrix}1\end{bmatrix} .$$

Find the Jacobian matrix for the first step of the Levenberg-Marquardt method.

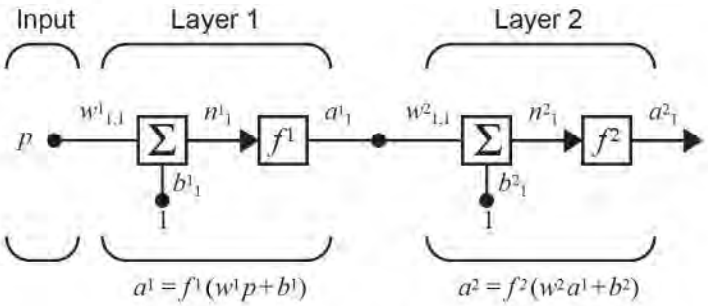


Figure P12.5 Two-Layer Network for LMBP Demonstration

The first step is to propagate the inputs through the network and compute the errors.

$$\mathbf{a}_1^0 = \mathbf{p}_1 = \begin{bmatrix}1\end{bmatrix}$$

$$\mathbf{n}_1^1 = \mathbf{W}^1\mathbf{a}_1^0 + \mathbf{b}^1 = \begin{bmatrix}1\end{bmatrix}\begin{bmatrix}1\end{bmatrix} + \begin{bmatrix}0\end{bmatrix} = \begin{bmatrix}1\end{bmatrix}, \ \mathbf{a}_1^1 = \mathbf{f}^1(\mathbf{n}_1^1) = (\begin{bmatrix}1\end{bmatrix})^2 = \begin{bmatrix}1\end{bmatrix}$$

$$\mathbf{n}_1^2 = \mathbf{W}^2\mathbf{a}_1^1 + \mathbf{b}^2 = (\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}1\end{bmatrix} + \begin{bmatrix}1\end{bmatrix}) = \begin{bmatrix}3\end{bmatrix}, \ \mathbf{a}_1^2 = \mathbf{f}^2(\mathbf{n}_1^2) = (\begin{bmatrix}3\end{bmatrix}) = \begin{bmatrix}3\end{bmatrix}$$

$$\mathbf{e}_1 = (\mathbf{t}_1 - \mathbf{a}_1^2) = (\begin{bmatrix}1\end{bmatrix} - \begin{bmatrix}3\end{bmatrix}) = \begin{bmatrix}-2\end{bmatrix}$$

$$\mathbf{a}_2^0 = \mathbf{p}_2 = \begin{bmatrix}2\end{bmatrix}$$

$$\mathbf{n}_2^1 = \mathbf{W}^1\mathbf{a}_2^0 + \mathbf{b}^1 = \begin{bmatrix}1\end{bmatrix}\begin{bmatrix}2\end{bmatrix} + \begin{bmatrix}0\end{bmatrix} = \begin{bmatrix}2\end{bmatrix}, \ \mathbf{a}_2^1 = \mathbf{f}^1(\mathbf{n}_2^1) = (\begin{bmatrix}2\end{bmatrix})^2 = \begin{bmatrix}4\end{bmatrix}$$

$$\mathbf{n}_2^2 = \mathbf{W}^2\mathbf{a}_2^1 + \mathbf{b}^2 = (\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}4\end{bmatrix} + \begin{bmatrix}1\end{bmatrix}) = \begin{bmatrix}9\end{bmatrix}, \ \mathbf{a}_2^2 = \mathbf{f}^2(\mathbf{n}_2^2) = (\begin{bmatrix}9\end{bmatrix}) = \begin{bmatrix}9\end{bmatrix}$$

$$\mathbf{e}_2 = (\mathbf{t}_2 - \mathbf{a}_2^2) = (\begin{bmatrix}2\end{bmatrix} - \begin{bmatrix}9\end{bmatrix}) = \begin{bmatrix}-7\end{bmatrix}$$

The next step is to initialize and backpropagate the Marquardt sensitivities using Eq. (12.46) and Eq. (12.47).

$$\mathbf{S}_1^2 = -\dot{\mathbf{F}}^2(\mathbf{n}_1^2) = -\begin{bmatrix}1\end{bmatrix}$$

$$\mathbf{S}_1^1 = \dot{\mathbf{F}}^1(\mathbf{n}_1^1)(\mathbf{W}^2)^T\mathbf{S}_1^2 = \begin{bmatrix}2n_{1,1}^1\end{bmatrix}\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}-1\end{bmatrix} = \begin{bmatrix}2(1)\end{bmatrix}\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}-1\end{bmatrix} = \begin{bmatrix}-4\end{bmatrix}$$

$$\mathbf{S}_2^2 = -\dot{\mathbf{F}}^2(\mathbf{n}_2^2) = -\begin{bmatrix}1\end{bmatrix}$$

$$\mathbf{S}_2^1 = \dot{\mathbf{F}}^1(\mathbf{n}_2^1)(\mathbf{W}^2)^T\mathbf{S}_2^2 = \begin{bmatrix}2n_{1,2}^2\end{bmatrix}\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}-1\end{bmatrix} = \begin{bmatrix}2(2)\end{bmatrix}\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}-1\end{bmatrix} = \begin{bmatrix}-8\end{bmatrix}$$

$$\mathbf{S}^1 = \begin{bmatrix}\mathbf{S}_1^1 \middle| \mathbf{S}_2^1\end{bmatrix} = \begin{bmatrix}-4 & -8\end{bmatrix}, \quad \mathbf{S}^2 = \begin{bmatrix}\mathbf{S}_1^2 \middle| \mathbf{S}_2^2\end{bmatrix} = \begin{bmatrix}-1 & -1\end{bmatrix}$$

We can now compute the Jacobian matrix using Eq. (12.43), Eq. (12.44) and Eq. (12.37).

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix}\frac{\partial v_1}{\partial x_1} & \frac{\partial v_1}{\partial x_2} & \frac{\partial v_1}{\partial x_3} & \frac{\partial v_1}{\partial x_4} \\ \frac{\partial v_2}{\partial x_1} & \frac{\partial v_2}{\partial x_2} & \frac{\partial v_2}{\partial x_3} & \frac{\partial v_2}{\partial x_4}\end{bmatrix} = \begin{bmatrix}\frac{\partial e_{1,1}}{\partial w_{1,1}^1} & \frac{\partial e_{1,1}}{\partial b_1^1} & \frac{\partial e_{1,1}}{\partial w_{1,1}^2} & \frac{\partial e_{1,1}}{\partial b_1^2} \\ \frac{\partial e_{1,2}}{\partial w_{1,1}^1} & \frac{\partial e_{1,2}}{\partial b_1^1} & \frac{\partial e_{1,2}}{\partial w_{1,1}^2} & \frac{\partial e_{1,2}}{\partial b_1^2}\end{bmatrix}$$

$$[\mathbf{J}]_{1,1} = \frac{\partial v_1}{\partial x_1} = \frac{\partial e_{1,1}}{\partial w_{1,1}^1} = \frac{\partial e_{1,1}}{\partial n_{1,1}^1} \times \frac{\partial n_{1,1}^1}{\partial w_{1,1}^1} = \tilde{s}_{1,1}^1 \times \frac{\partial n_{1,1}^1}{\partial w_{1,1}^1} = \tilde{s}_{1,1}^1 \times a_{1,1}^0$$

$$= (-4)(1) = -4$$

$$[\mathbf{J}]_{1,2} = \frac{\partial v_1}{\partial x_2} = \frac{\partial e_{1,1}}{\partial b_1^1} = \frac{\partial e_{1,1}}{\partial n_{1,1}^1} \times \frac{\partial n_{1,1}^1}{\partial b_1^1} = \tilde{s}_{1,1}^1 \times \frac{\partial n_{1,1}^1}{\partial b_1^1} = \tilde{s}_{1,1}^1 = -4$$

$$[\mathbf{J}]_{1,3} = \frac{\partial v_1}{\partial x_3} = \frac{\partial e_{1,1}}{\partial n_{1,1}^2} \times \frac{\partial n_{1,1}^2}{\partial w_{1,1}^2} = \tilde{s}_{1,1}^2 \times \frac{\partial n_{1,1}^2}{\partial w_{1,1}^2} = \tilde{s}_{1,1}^2 \times a_{1,1}^1 = (-1)(1) = -1$$

$$[\mathbf{J}]_{1,4} = \frac{\partial v_1}{\partial x_4} = \frac{\partial e_{1,1}}{\partial n^2_{1,1}} \times \frac{\partial n^1_{1,1}}{\partial b^2_1} = \tilde{s}^2_{1,1} \times \frac{\partial n^2_{1,1}}{\partial b^2_1} = \tilde{s}^2_{1,1} = -1$$

$$[\mathbf{J}]_{2,1} = \frac{\partial v_2}{\partial x_1} = \frac{\partial e_{1,2}}{\partial n^1_{1,2}} \times \frac{\partial n^1_{1,2}}{\partial w^1_{1,1}} = \tilde{s}^1_{1,2} \times \frac{\partial n^1_{1,2}}{\partial w^1_{1,1}} = \tilde{s}^1_{1,2} \times a^0_{1,2} = (-8)(2) = -16$$

$$[\mathbf{J}]_{2,2} = \frac{\partial v_2}{\partial x_2} = \frac{\partial e_{1,2}}{\partial b^1_1} = \frac{\partial e_{1,2}}{\partial n^1_{1,2}} \times \frac{\partial n^1_{1,2}}{\partial b^1_1} = \tilde{s}^1_{1,2} \times \frac{\partial n^1_{1,2}}{\partial b^1_1} = \tilde{s}^1_{1,2} = -8$$

$$[\mathbf{J}]_{2,3} = \frac{\partial v_2}{\partial x_3} = \frac{\partial e_{1,2}}{\partial n^2_{1,2}} \times \frac{\partial n^2_{1,2}}{\partial w^2_{1,1}} = \tilde{s}^2_{1,2} \times \frac{\partial n^2_{1,2}}{\partial w^2_{1,1}} = \tilde{s}^2_{1,2} \times a^1_{1,2} = (-1)(4) = -4$$

$$[\mathbf{J}]_{2,4} = \frac{\partial v_2}{\partial x_4} = \frac{\partial e_{1,2}}{\partial b^2_1} = \frac{\partial e_{1,2}}{\partial n^2_{1,2}} \times \frac{\partial n^2_{1,2}}{\partial b^2_1} = \tilde{s}^2_{1,2} \times \frac{\partial n^2_{1,2}}{\partial b^2_1} = \tilde{s}^2_{1,2} = -1$$

Therefore the Jacobian matrix is

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} -4 & -4 & -1 & -1 \\ -16 & -8 & -4 & -1 \end{bmatrix}.$$

Epilogue

One of the major problems with the basic backpropagation algorithm (steepest descent backpropagation — SDBP) has been the long training times. It is not feasible to use SDBP on practical problems, because it can take weeks to train a network, even on a large computer. Since backpropagation was first popularized, there has been considerable work on methods to accelerate the convergence of the algorithm. In this chapter we have discussed the reasons for the slow convergence of SDBP and have presented several techniques for improving the performance of the algorithm.

The techniques for speeding up convergence have fallen into two main categories: heuristic methods and standard numerical optimization methods. We have discussed two heuristic methods: momentum (MOBP) and variable learning rate (VLBP). MOBP is simple to implement, can be used in batch mode or incremental mode and is significantly faster than SDBP. It does require the selection of the momentum coefficient, but γ is limited to the range $[0, 1]$ and the algorithm is not extremely sensitive to this choice.

The VLBP algorithm is faster than MOBP but must be used in batch mode. For this reason it requires more storage. VLBP also requires the selection of a total of five parameters. The algorithm is reasonably robust, but the choice of the parameters can affect the convergence speed and is problem dependent.

We also presented two standard numerical optimization techniques: conjugate gradient (CGBP) and Levenberg-Marquardt (LMBP). CGBP is generally faster than VLBP. It is a batch mode algorithm, which requires a linear search at each iteration, but its storage requirements are not significantly different than VLBP. There are many variations of the conjugate gradient algorithm proposed for neural network applications. We have presented only one.

The LMBP algorithm is the fastest algorithm that we have tested for training multilayer networks of moderate size, even though it requires a matrix inversion at each iteration. It requires that two parameters be selected, but the algorithm does not appear to be sensitive to this selection. The main drawback of LMBP is the storage requirement. The $\mathbf{J}^T\mathbf{J}$ matrix, which must be inverted, is $n \times n$, where n is the total number of weights and biases in the network. If the network has more than a few thousand parameters, the LMBP algorithm becomes impractical on current machines.

There are many other variations on backpropagation that have not been discussed in this chapter. Some references to other techniques are given in Chapter 19.

Further Reading

[Barn92] E. Barnard, "Optimization for training neural nets," *IEEE Trans. on Neural Networks*, vol. 3, no. 2, pp. 232–240, 1992.

A number of optimization algorithms that have promise for neural network training are discussed in this paper.

[Batt92] R. Battiti, "First- and second-order methods for learning: Between steepest descent and Newton's method," *Neural Computation*, vol. 4, no. 2, pp. 141–166, 1992.

This paper is an excellent survey of the current optimization algorithms that are suitable for neural network training.

[Char92] C. Charalambous, "Conjugate gradient algorithm for efficient training of artificial neural networks," *IEE Proceedings*, vol. 139, no. 3, pp. 301–310, 1992.

This paper explains how the conjugate gradient algorithm can be used to train multilayer networks. Comparisons are made to other training algorithms.

[Fahl88] S. E. Fahlman, "Faster-learning variations on back-propagation: An empirical study," In D. Touretsky, G. Hinton & T. Sejnowski, eds., *Proceedings of the 1988 Connectionist Models Summer School*, San Mateo, CA: Morgan Kaufmann, pp. 38–51, 1988.

The QuickProp algorithm, which is described in this paper, is one of the more popular heuristic modifications to backpropagation. It assumes that the error curve can be approximated by a parabola, and that the effect of each weight can be considered independently. QuickProp provides significant speedup over standard backpropagation on many problems.

[HaMe94] M. T. Hagan and M. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, vol. 5, no. 6, 1994.

This paper describes the use of the Levenberg-Marquardt algorithm for training multilayer networks and compares the performance of the algorithm with variable learning rate backpropagation and conjugate gradient. The Levenberg-Marquardt algorithm is faster, but requires more storage.

[Jaco88] R. A. Jacobs, "Increased rates of convergence through learning rate adaptation," *Neural Networks*, vol. 1, no. 4, pp. 295–308, 1988.

This is another early paper discussing the use of variable learning rate backpropagation. The procedure described here is called the delta-bar-delta learning rule, in which each network parameter has its own learning rate that varies at each iteration.

[NgWi90] D. Nguyen and B. Widrow, "Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights," *Proceedings of the IJCNN*, vol. 3, pp. 21–26, July 1990.

This paper describes a procedure for setting the initial weights and biases for the backpropagation algorithm. It uses the shape of the sigmoid transfer function and the range of the input variables to determine how large the weights should be, and then uses the biases to center the sigmoids in the operating region. The convergence of backpropagation is improved significantly by this procedure.

[RiIr90] A. K. Rigler, J. M. Irvine and T. P. Vogl, "Rescaling of variables in back propagation learning," *Neural Networks*, vol. 4, no. 2, pp. 225–230, 1991.

This paper notes that the derivative of a sigmoid function is very small on the tails. This means that the elements of the gradient associated with the first few layers will generally be smaller that those associated with the last layer. The terms in the gradient are then scaled to equalize them.

[Scal85] L. E. Scales, *Introduction to Non-Linear Optimization*. New York: Springer-Verlag, 1985.

Scales has written a very readable text describing the major optimization algorithms. The book emphasizes methods of optimization rather than existence theorems and proofs of convergence. Algorithms are presented with intuitive explanations, along with illustrative figures and examples. Pseudocode is presented for most algorithms.

[Shan90] D. F. Shanno, "Recent advances in numerical techniques for large-scale optimization," *Neural Networks for Control*, Miller, Sutton and Werbos, eds., Cambridge MA: MIT Press, 1990.

This paper discusses some conjugate gradient and quasi-Newton optimization algorithms that could be used for neural network training.

[Toll90] T. Tollenaere, "SuperSAB: Fast adaptive back propagation with good scaling properties," *Neural Networks*, vol. 3, no. 5, pp. 561–573, 1990.

This paper presents a variable learning rate backpropagation algorithm in which different learning rates are used for each weight.

[VoMa88] T. P. Vogl, J. K. Mangis, A. K. Zigler, W. T. Zink and D. L. Alkon, "Accelerating the convergence of the backpropagation method," *Biological Cybernetics.*, vol. 59, pp. 256–264, Sept. 1988.

This was one of the first papers to introduce several heuristic techniques for accelerating the convergence of backpropagation. It included batching, momentum and variable learning rate.

Exercises

E12.1 We want to train the network shown in Figure E12.1 on the training set

$$\left\{(\mathbf{p}_1 = \begin{bmatrix}-2\end{bmatrix}), (\mathbf{t}_1 = \begin{bmatrix}0.8\end{bmatrix})\right\}, \left\{(\mathbf{p}_2 = \begin{bmatrix}2\end{bmatrix}), (\mathbf{t}_2 = \begin{bmatrix}1\end{bmatrix})\right\},$$

where each pair is equally likely to occur.



Write a MATLAB M-file to create a contour plot for the mean squared error performance index.

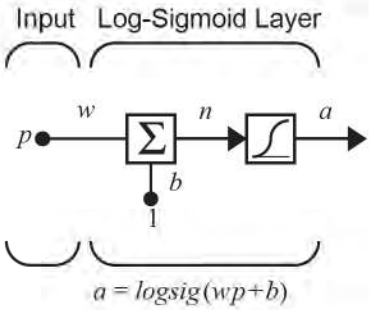


Figure E12.1 Network for Exercise E12.1

E12.2 Demonstrate the effect of batching by computing the direction of the initial step for SDBP with and without batching for the problem described in Exercise E12.1, starting from the initial guess

$$w(0) = 0, \; b(0) = 0.5.$$

E12.3 Recall the quadratic function used in Problem P9.1:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix}10 & -6\\ -6 & 10\end{bmatrix}\mathbf{x} + \begin{bmatrix}4 & 4\end{bmatrix}\mathbf{x}.$$

We want to use the steepest descent algorithm with momentum to minimize this function.

i. Suppose that the learning rate is $\alpha = 0.2$. Find a value for the momentum coefficient γ for which the algorithm will be stable. Use the ideas presented in Problem P12.2.

ii. Suppose that the learning rate is $\alpha = 20$. Find a value for the momentum coefficient γ for which the algorithm will be stable.



iii. Write a MATLAB program to plot the trajectories of the algorithm for the α and γ values of both part (i) and part (ii) on the contour plot of $F(\mathbf{x})$, starting from the initial guess

$$\mathbf{x}_0 = \begin{bmatrix} -1 \\ -2.5 \end{bmatrix}.$$

E12.4 Consider the following quadratic function.

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 4 & -4 \end{bmatrix} \mathbf{x}.$$

We want to use the steepest descent algorithm with momentum to minimize this function.

i. Perform two iterations (finding $\mathbf{x}_1$ and $\mathbf{x}_2$) of steepest descent with momentum, starting from the initial condition $\mathbf{x}_0 = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$. Use a learning rate of $\alpha = 1$ and a momentum coefficient of $\gamma = 0.75$.

ii. Is the algorithm stable with this learning rate and this momentum? Use the ideas presented in Problem P12.2.

iii. Would the algorithm be stable with this learning rate, if the momentum were zero?

E12.5 Consider the following quadratic function.

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 1 & 2 \end{bmatrix} \mathbf{x} + 2.$$

We want to use the steepest descent algorithm with momentum to minimize this function.

i. Suppose the learning rate is $\alpha = 1$. Is the algorithm stable, if the momentum coefficient is $\gamma = 0$? Use the ideas presented in Problem P12.2.

ii. Suppose the learning rate is $\alpha = 1$. Is the algorithm stable, if the momentum coefficient is $\gamma = 0.6$?

E12.6 Consider the following quadratic function.

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix}2 & 1\\ 1 & 2\end{bmatrix}\mathbf{x} + \begin{bmatrix}1 & 2\end{bmatrix}\mathbf{x} + 2 .$$

We want to use the steepest descent algorithm with momentum to minimize this function. Suppose the learning rate is $\alpha = 1$. Find a value for the momentum coefficient γ so that the algorithm will be stable. Use the ideas presented in Problem Eq. P12.2.

E12.7 For the function of Exercise E12.3, perform three iterations of the variable learning rate algorithm, with initial guess

$$\mathbf{x}_0 = \begin{bmatrix}-1\\ -2.5\end{bmatrix}.$$

Plot the algorithm trajectory on a contour plot of $F(\mathbf{x})$. Use the algorithm parameters

$$\alpha = 0.4,\ \gamma = 0.1,\ \eta = 1.5,\ \rho = 0.5,\ \zeta = 5\%.$$

E12.8 Consider the following quadratic function:

$$F(\mathbf{x}) = x_1^2 + 2x_2^2.$$

Perform three iterations of the variable learning rate algorithm, with initial guess

$$\mathbf{x}_0 = \begin{bmatrix}0\\ -1\end{bmatrix}.$$

Use the algorithm parameters

$$\alpha = 1,\ \gamma = 0.2,\ \eta = 1.5,\ \rho = 0.5,\ \zeta = 5\%.$$

(Count an iteration each time the function is evaluated after the initial guess.)

E12.9 For the function of Exercise E12.3, perform one iteration of the conjugate gradient algorithm, with initial guess

$$\mathbf{x}_0 = \begin{bmatrix}-1\\ -2.5\end{bmatrix}.$$

For the linear minimization use interval location by function evaluation and interval reduction by the Golden Section search. Plot the path of the search on a contour plot of $F(\mathbf{x})$.

E12.10 Consider the following quadratic function.

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix}4 & 0\\ 0 & 2\end{bmatrix}\mathbf{x} + \begin{bmatrix}-2 & -1\end{bmatrix}\mathbf{x}.$$

We want to minimize this function along the line

$$\mathbf{x} = \begin{bmatrix}0\\ 0\end{bmatrix} + \alpha\begin{bmatrix}1\\ 1\end{bmatrix}.$$

i. Sketch this line in the x_1, x_2 plane.

ii. The learning rate α must fall somewhere between 0 and 3. Perform one iteration of the golden section search. You should find a_2, b_2, c_2 and d_2, and indicate these points along the line that you drew in part i.

E12.11 Consider the following quadratic function.

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}\mathbf{x} + \begin{bmatrix}1 & 1\end{bmatrix}\mathbf{x}.$$

We want to minimize this function along the line

$$\mathbf{x} = \begin{bmatrix}0\\ 0\end{bmatrix} + \alpha\begin{bmatrix}-1\\ 0\end{bmatrix}.$$

i. Use the method described on page 12-16 to determine an initial interval containing the minimum. Use $\varepsilon = 0.5$.

ii. Take one iteration of the golden section search to reduce the interval you obtained in part i.

E12.12 Consider the following quadratic function.

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix}1 & 0\\ 0 & 2\end{bmatrix}\mathbf{x},$$

We want to minimize this function along the line

$$\mathbf{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \alpha \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Perform two iterations of the Golden Section search ($k = 1, 2$) to find the interval $[a_3, b_3]$. Assume that the initial interval is defined by $a_1 = 0$ and $b_1 = 1$. Make a rough sketch of the contour plot of $F(\mathbf{x})$, draw the search line in the same figure and indicate your search points (points where you evaluated $F(\mathbf{x})$) on the line.

E12.13 Consider the following quadratic function.

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{x}.$$

We want to minimize this function along the line

$$\mathbf{x} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \alpha \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Perform two iterations of the Golden Section search ($k = 1, 2$) to find the interval $[a_3, b_3]$. Assume that the initial interval is defined by $a_1 = 0$ and $b_1 = 1$. Make a rough sketch of the contour plot of $F(\mathbf{x})$, draw the search line in the same figure and indicate your search points (points where you evaluated $F(\mathbf{x})$) on the line.

E12.14 We want to use the network of Figure E12.2 to approximate the function

$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right) \text{ for } -2 \le p \le 2.$$

The initial network parameters are chosen to be

$$\mathbf{w}^1(0) = \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix}, \mathbf{b}^1(0) = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix}, \mathbf{w}^2(0) = \begin{bmatrix} 0.09 & -0.17 \end{bmatrix}, \mathbf{b}^2(0) = \begin{bmatrix} 0.48 \end{bmatrix}.$$

To create the training set we sample the function $g(p)$ at the points $p = 1$ and $p = 0$. Find the Jacobian matrix for the first step of the LMBP algorithm. (Some of the information you will need has been computed in the example starting on page 11-14.)

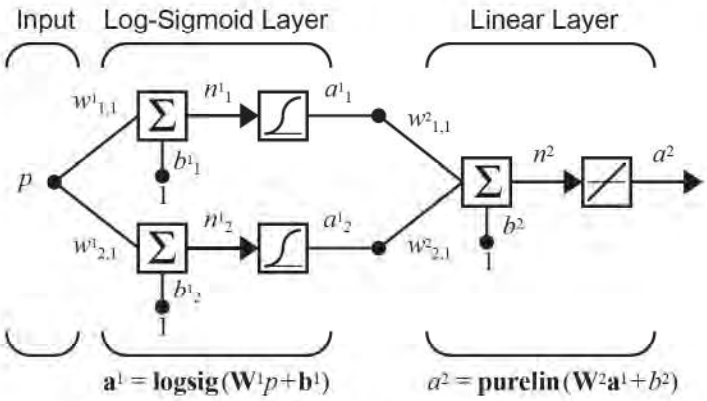


Figure E12.2 Network for Exercise E12.14

E12.15 Show that for a linear network the LMBP algorithm will converge to an optimum solution in one iteration if $\mu = 0$.

E12.16 In Exercise E11.25 you wrote a MATLAB program to implement the SDBP algorithm for a $1 - S^1 - 1$ network, and trained the network to approximate the function

$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right) \text{ for } -2 \le p \le 2 .$$



Repeat this exercise, modifying your program to use the training procedures discussed in this chapter: batch mode SDBP, MOBP, VLBP, CGBP and LMBP. Compare the convergence results of the various methods.

13 Generalization

Objectives	13-1
Theory and Examples	13-2
Problem Statement	13-2
Methods for Improving Generalization	13-5
Estimating Generalization Error - The Test Set	13-6
Early Stopping	13-6
Regularization	13-8
Bayesian Analysis	13-10
Bayesian Regularization	13-12
Relationship Between Early Stopping and Regularization	13-19
Summary of Results	13-29
Solved Problems	13-32
Epilogue	13-44
Further Reading	13-45
Exercises	13-47

Objectives

One of the key issues in designing a multilayer network is determining the number of neurons to use. In effect, that is the objective of this chapter.

In Chapter 11 we showed that if the number of neurons is too large, the network will overfit the training data. This means that the error on the training data will be very small, but the network will fail to perform as well when presented with new data. A network that generalizes well will perform as well on new data as it does on the training data.

The complexity of a neural network is determined by the number of free parameters that it has (weights and biases), which in turn is determined by the number of neurons. If a network is too complex for a given data set, then it is likely to overfit and to have poor generalization.

In this chapter we will see that we can adjust the complexity of a network to fit the complexity of the data. In addition, this can be done without changing the number of neurons. We can adjust the effective number of free parameters without changing the actual number of free parameters.

Theory and Examples

Mark Twain once said "We should be careful to get out of an experience only the wisdom that is in it-and stop there; lest we be like the cat that sits down on a hot stove-lid. She will never sit down on a hot stove-lid again-and that is well; but also she will never sit down on a cold one any more." (From *Following the Equator*, 1897.)

Generalization

That is the objective of this chapter. We want to train neural networks to get out of the data only the wisdom that is in it. This concept is called *generalization*. A network trained to generalize will perform as well in new situations as it does on the data on which it was trained.

Ockham's Razor

The key strategy we will use for obtaining good generalization is to find the simplest model that explains the data. This is a variation of a principle called *Ockham's razor*, which is named after the English logician William of Ockham, who worked in the 14th Century. The idea is that the more complexity you have in your model, the greater the possibility for errors.

In terms of neural networks, the simplest model is the one that contains the smallest number of free parameters (weights and biases), or, equivalently, the smallest number of neurons. To find a network that generalizes well, we need to find the simplest network that fits the data.

There are at least five different approaches that people have used to produce simple networks: growing, pruning, global searches, regularization, and early stopping. Growing methods start with no neurons in the network and then add neurons until the performance is adequate. Pruning methods start with large networks, which likely overfit, and then remove neurons (or weights) one at a time until the performance degrades significantly. Global searches, such as genetic algorithms, search the space of all possible network architectures to locate the simplest model that explains the data.

The final two approaches, regularization and early stopping, keep the network small by constraining the *magnitude* of the network weights, rather than by constraining the *number* of network weights. In this chapter we will concentrate on these two approaches. We will begin by defining the problem of generalization and by showing examples of both good and poor generalization. We will then describe the regularization and early stopping methods for training neural networks. Finally, we will demonstrate how these two methods are, in effect, performing the same operation.

Problem Statement

Let's begin our discussion of generalization by defining the problem. We start with a training set of example network inputs and corresponding target outputs:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}. \tag{13.1}$$

For our development of the concept of generalization, we will assume that the target outputs are generated by

$$\mathbf{t}_q = \mathbf{g}(\mathbf{p}_q) + \varepsilon_q, \tag{13.2}$$

where $\mathbf{g}(.)$ is some unknown function, and ε_q is a random, independent and zero mean noise source. Our training objective will be to produce a neural network that approximates $\mathbf{g}(.)$, while ignoring the noise.

The standard performance index for neural network training is the sum squared error on the training set:

$$F(\mathbf{x}) = E_D = \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q), \tag{13.3}$$

where $\mathbf{a}_q$ is the network output for input $\mathbf{p}_q$. We are using the variable E_D to represent the sum squared error on the training data, because later we will modify the performance index to include an additional term.

Overfitting

The problem of *overfitting* is illustrated in Figure 13.1. The blue curve represents the function $\mathbf{g}(.)$. The large open circles represent the noisy target points. The black curve represents the trained network response, and the smaller circles filled with crosses represent the network response at the training points. In this figure we can see that the network response exactly matches the training points. However, it does a very poor job of matching the underlying function. It overfits.

There are actually two kinds of errors that occur in Figure 13.1. The first type of error, which is caused by overfitting, occurs for input values between -3 and 0. This is the region where all of the training data points occur. The network response in this region overfits the training data and will fail to perform well for input values that are not in the training set. The network does a poor job of *interpolation*; it fails to accurately approximate the function near the training points.

Interpolation

The second type of error occurs for inputs in the region between 0 and 3. The network fails to perform well in this region, not because it is overfitting, but because there is no training data there. The network is *extrapolating* beyond the range of the input data.

Extrapolation

In this chapter we will discuss methods for preventing errors of interpolation (overfitting). There is no way to prevent errors of extrapolation, unless the data that is used to train the network covers all regions of the input space where the network will be used. The network has no way of knowing what the true function looks like in regions where there is no data.

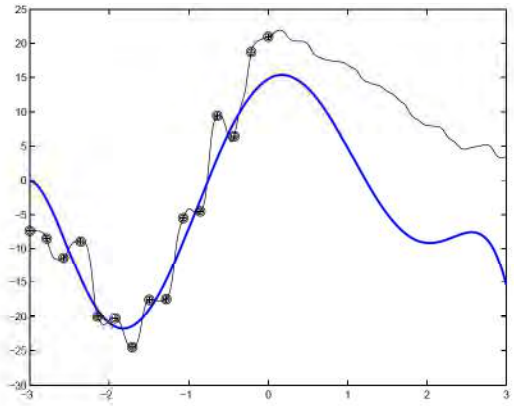


Figure 13.1 Example of Overfitting and Poor Extrapolation

In Figure 13.2 we have an example of a network that has been trained to generalize well. The network has the same number of weights as the network of Figure 13.1, and it was trained using the same data set, but it has been trained in such a way that it does not fully use all of the weights that are available. It only uses as many weights as necessary to fit the data. The network response does not fit the function perfectly, but it does the best job it can, based on limited and noisy data.

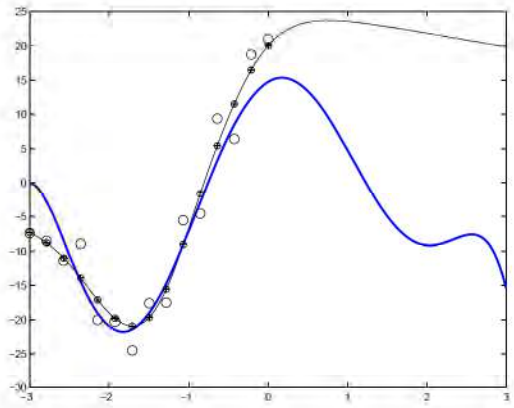


Figure 13.2 Example of Good Interpolation and Poor Extrapolation

In both Figure 13.1 and Figure 13.2 we can see that the network fails to extrapolate accurately. This is understandable, since the network has been provided with no information about the characteristics of the function out-

side of the range $-3 \le p \le 0$. The network response outside this range will be unpredictable. This is why it is important to have training data for all regions of the input space where the network will be used. It is usually not difficult to determine the required input range when the network has a single input, as in this example. However, when the network has many inputs, it becomes more difficult to determine when the network is interpolating and when it is extrapolating.

This problem is illustrated in a simple way in Figure 13.3. On the left side of this figure we see the function that is to be approximated. The range for the input variables is $-3 \le p_1 \le 3$ and $-3 \le p_2 \le 3$. The neural network was trained over these ranges of the two variables, but only for $p_1 \le p_2$. Therefore, both p_1 and p_2 cover their individual ranges, but only half of the total input space is covered. When $p_1 \ge p_2$, the network is extrapolating, and we can see on the right side of Figure 13.3 that the network performs poorly in this region. (See Problem P13.4 for another example of extrapolation.) If there are many input variables, it will be quite difficult to determine when the network is interpolating and when it is extrapolating. We will discuss some practical ways of dealing with this problem in Chapter 22.

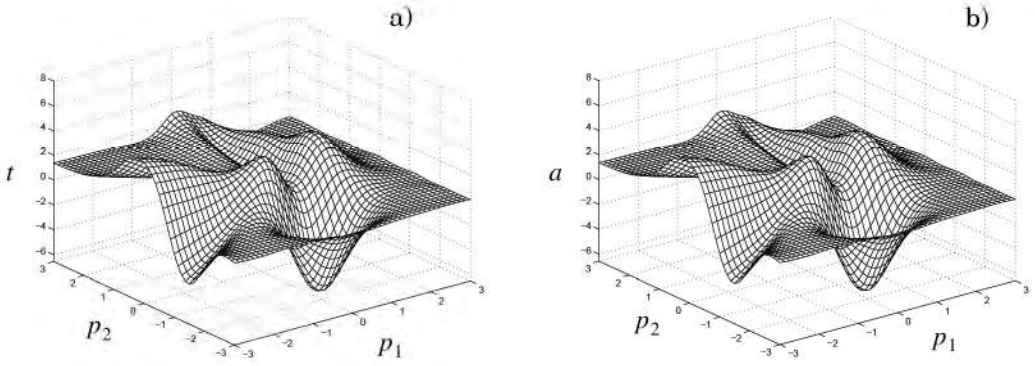


Figure 13.3 Function (a) and Neural Network Approximation (b)

Methods for Improving Generalization

The remainder of this chapter will discuss methods for improving the generalization capability of neural networks. As we discussed earlier, there are a number of approaches to this problem - all of which try to find the simplest network that will fit the data. These approaches fit into two general categories: restricting the number of weights (or, equivalently, the number of neurons) in the network, or restricting the magnitude of the weights. We will concentrate on two methods that we have found to be particularly useful: early stopping and regularization. Both of these approaches attempt to restrict the magnitude of the weights, although they do so in very different

ways. At the end of this chapter, we will demonstrate the approximate equivalence of the two methods.

We should note that in this chapter we are assuming that there is a limited amount of data with which to train the network. If the amount of data is unlimited, which in practical terms means that the number of data points is significantly larger than the number of network parameters, then there will not be a problem of overfitting.

Estimating Generalization Error - The Test Set

Before we discuss methods for improving the generalization capability of neural networks, we should first discuss how we can estimate this error for a specific neural network. Given a limited amount of available data, it is important to hold aside a certain subset during the training process. After the network has been trained, we will compute the errors that the trained network makes on this *test set*. The test set errors will then give us an indication of how the network will perform in the future; they are a measure of the generalization capability of the network.

Test Set

In order for the test set to be a valid indicator of generalization capability, there are two important things to keep in mind. First, the test set must never be used in any way to train the neural network, or even to select one network from a group of candidate networks. The test set should only be used after all training and selection is complete. Second, the test set must be representative of all situations for which the network will be used. This can sometimes be difficult to guarantee, especially when the input space is high-dimensional or has a complex shape. We will discuss this problem in more detail in Chapter 22, Practical Training Issues.

In the remaining sections of this chapter, we will assume that a test set has been removed from the data set before training begins, and that this set will be used at the completion of training to measure generalization capability.

Early Stopping

The first method we will discuss for improving generalization is also the simplest method. It is called early stopping [WaVe94]. The idea behind this method is that as training progresses the network uses more and more of its weights, until all weights are fully used when training reaches a minimum of the error surface. By increasing the number of iterations of training, we are increasing the complexity of the resulting network. If training is stopped before the minimum is reached, then the network will effectively be using fewer parameters and will be less likely to overfit. In a later section of this chapter we will demonstrate how the number of parameters changes as the number of iterations increases.

In order to use early stopping effectively, we need to know when to stop the training. We will describe a method, called *cross-validation*, that uses a

Cross-Validation

Validation Set

validation set to decide when to stop [Sarl95]. The available data (after removing the test set, as described above) is divided into two parts: a training set and a validation set. The training set is used to compute gradients or Jacobians and to determine the weight update at each iteration. The validation set is an indicator of what is happening to the network function "in between" the training points, and its error is monitored during the training process. When the error on the validation set goes up for several iterations, the training is stopped, and the weights that produced the minimum error on the validation set are used as the final trained network weights.

This process is illustrated in Figure 13.4. The graph at the bottom of this figure shows the progress of the training and validation performance indices, F (the sum squared errors), during training. Although the training error continues to go down throughout the training process, a minimum of the validation error occurs at the point labeled "a," which corresponds to training iteration 14. The graph at the upper left shows the network response at this *early stopping* point. The resulting network provides a good fit to the true function. The graph at the upper right demonstrates the network response if we continue to train to point "b," where the validation error has increased and the network is overfitting.

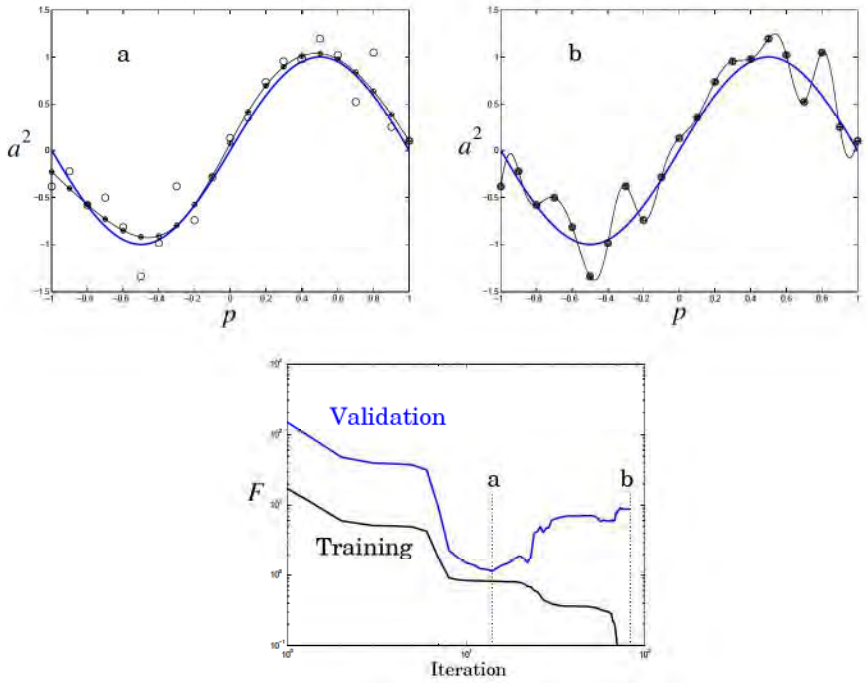


Figure 13.4 Illustration of Early Stopping

The basic concept for early stopping is simple, but there are several practical issues to be addressed. First, the validation set must be chosen so that it is representative of all situations for which the network will be used. This is also true for the test and training sets, as we mentioned earlier. Each set must be roughly equivalent in its coverage of the input space, although the size of each set may be different.

When we divide the data, approximately 70% is typically used for training, with 15% for validation and 15% for testing. These are only approximate numbers. A complete discussion of how to select the amount of data for the validation set is given in [AmMu97].

Another practical point to be made about early stopping is that we should use a relatively slow training method. During training, the network will use more and more of the available network parameters (as we will explain in the last section of this chapter). If the training method is too fast, it will likely jump past the point at which the validation error is minimized.



To experiment with the effect of early stopping, use the MATLAB® Neural Network Design Demonstration Early Stopping (`nnd13es`).

Regularization

The second method we will discuss for improving generalization is called regularization. For this method, we modify the sum squared error performance index of Eq. (13.3) to include a term that penalizes network complexity. This concept was introduced by Tikhonov [Tikh63]. He added a penalty, or regularization, term that involved the derivatives of the approximating function (neural network in our case), which forced the resulting function to be smooth. Under certain conditions, this regularization term can be written as the sum of squares of the network weights, as in

$$F(\mathbf{x}) = \beta E_D + \alpha E_W = \beta \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q) + \alpha \sum_{i=1}^{n} x_i^2, \qquad (13.4)$$

where the ratio α/β controls the effective complexity of the network solution. The larger this ratio is, the smoother the network response. (Note that we could have used a single parameter here, but developments in later sections will require two parameters.)

Why do we want to penalize the sum squared weights, and how is this similar to reducing the number of neurons? Consider again the example multilayer network shown in Figure 11.4. Recall how increasing a weight increased the slope of the network function. You can see this effect again in Figure 13.5, where we have changed the weight $w_{1,1}^2$ from 0 to 2. When the weights are large, the function created by the network can have large slopes, and is therefore more likely to overfit the training data. If we restrict the weights to be small, then the network function will create a

smooth interpolation through the training data - just as if the network had a small number of neurons.

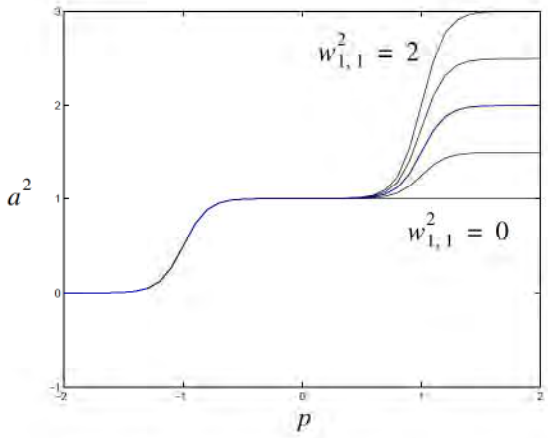


Figure 13.5 Effect of Weight on Network Response



To experiment with the effect of weight changes on the network function, use the MATLAB® Neural Network Design Demonstration Network Function (`nnd11nf`).

The key to the success of the regularization method in producing a network that generalizes well is the correct choice of the regularization ratio α/β. Figure 13.6 illustrates the effect of changing this ratio. Here we have trained a 1-20-1 network on 21 noisy samples of a sine wave.

In the figure, the blue line represents the true function, and the large open circles represent the noisy data. The black curve represents the trained network response, and the smaller circles filled with crosses represent the network response at the training points. From the figure, we can see that the ratio $\alpha/\beta = 0.01$ produces the best fit to the true function. For ratios larger than this, the network response is too smooth, and for ratios smaller than this, the network overfits.

There are several techniques for setting the regularization parameter. One approach is to use a validation set, such as we described in the section on early stopping; the regularization parameter is set to minimize the squared error on the validation set [GoLa98]. In the next two sections we will describe a different technique for automatically setting the regularization parameter. It is called Bayesian regularization.

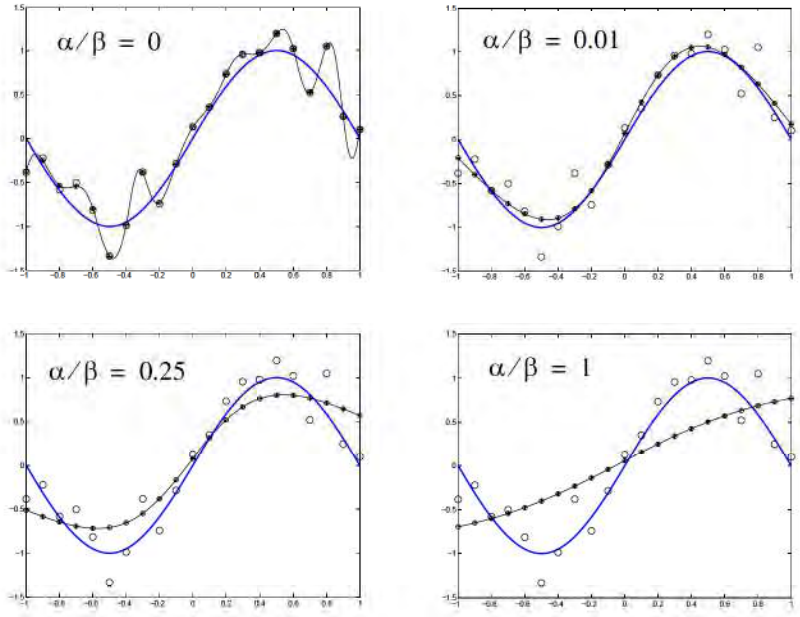


Figure 13.6 Effect of Regularization Ratio



To experiment with the effect of regularization, use the MATLAB® Neural Network Design Demonstration Regularization (`nnd13reg`).

Bayesian Analysis

Thomas Bayes was a Presbyterian minister who lived in England during the 1700's. He was also an amateur mathematician. His most important work was published after his death. In it, he presented what is now known as Bayes' Theorem. The theorem states that if you have two random events, A and B, then the conditional probability of the occurrence of A, given the occurrence of B can be computed as

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \tag{13.5}$$

Eq. (13.5) is called Bayes' rule. Each of the terms in this expression has a name by which it is commonly referred. $P(A)$ is called the *prior* probability. It tells us what we know about A before we know the outcome of B. $P(A|B)$ is called the *posterior* probability. This tells us what we know about A after we learn about B. $P(B|A)$ is the conditional probability of B given A. Normally this term is given by our knowledge of the system that describes the relationship between B and A. $P(B)$ is the marginal probability of the event B, and it acts as a normalization factor in Bayes' rule.



To illustrate how Bayes' rule can be used, consider the following medical situation. Assume that 1% of the population have a certain disease. There is a test that can be performed to detect the presence of this disease. The test is 80% accurate in detecting the disease in people who have it. However, 10% of the time, someone without the disease will register a positive test. If you take the test and register positive, your question would be: What is the probability that I actually have the disease? Most of us (including most physicians, as has been shown in many studies), would guess that the probability is very high, considering that the test is 80% accurate in detecting the disease in a sick person. However, this turns out not to be the case, and Bayes' rule can help us overcome this lack of intuition, when it comes to probability.

Let A represent the event that you have the disease. Let B represent the event that you have a positive test result. We can then use Bayes' rule to find $P(A|B)$, which is the probability that you have the disease, given that you have a positive test. We know that the prior probability $P(A)$ would be 0.01, because 1% of the population have the disease. $P(B|A)$ is 0.8, because the test is 80% accurate in detecting the disease in people who have it. (Notice that this conditional probability is based on our knowledge of the test procedure and its accuracy.) In order to use Bayes' rule, we need one more term, which is $P(B)$. This is the probability of getting a positive test, whether or not you have the disease. This can be obtained by adding the probability of having a positive test when you have the disease to the probability of having a positive test when you don't have the disease:

$$P(B) = P(A \cap B) + P(\bar{A} \cap B) = P(B|A)P(A) + P(B|\bar{A})P(\bar{A}), \qquad (13.6)$$

where we have used the definition of conditional probability:

$$P(B|A) = \frac{P(A \cap B)}{P(A)}, \text{ or } P(A \cap B) = P(B|A)P(A). \qquad (13.7)$$

If we plug in our known probabilities into Eq. (13.6), we find

$$P(B) = 0.8 \times 0.01 + 0.1 \times 0.99 = 0.107, \qquad (13.8)$$

where $P(B|\bar{A})$ is 0.1, because 10% of healthy people register a positive test. We can now use Bayes' rule to find the posterior probability $P(A|B)$:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} = \frac{0.8 \times 0.01}{0.107} = 0.0748. \qquad (13.9)$$

This tells us that even if you get a positive test, you only have a 7.5% chance of having the disease. For most of us, this result is not intuitive.

The key to Bayes' rule is the prior probability $P(A)$. In this case, the prior odds of having the disease were only 1 in 100. If this number had been much higher, then our posterior probability $P(A|B)$ would have also in-

creased significantly. It is important when using Bayes' rule to have the prior probability $P(A)$ accurately reflect our prior knowledge.

For another example of using Bayes' rule and the effect of the prior density, see Solved Problem P13.2 and its associated demonstration.

In the next section, we will apply Bayesian analysis to the training of multilayer networks. The advantage of Bayesian methods is that we can insert prior knowledge through the selection of the prior probability. For neural network training, we will make the prior assumption that the function we are approximating is smooth. This means that the weights cannot be too large, as was demonstrated in Figure 13.5. The trick will be to incorporate this prior knowledge into an appropriate choice for the prior probability.

Bayesian Regularization

Although there have been many approaches to the automatic selection of the regularization parameter, we will concentrate on one developed by David MacKay [MacK92]. This approach puts the training of neural networks into a Bayesian statistical framework. This framework is useful for many aspects of training, in addition to the selection of the regularization parameter, so it is an important concept to become familiar with. There are two levels to this Bayesian analysis. We will begin with Level I.

Level I Bayesian Framework

The Bayesian framework begins with the assumption that the network weights are random variables. We then choose the weights that maximize the conditional probability of the weights given the data. Bayes' rule is used to find this probability function:

$$P(\mathbf{x}|D, \alpha, \beta, M) = \frac{P(D|\mathbf{x}, \beta, M)P(\mathbf{x}|\alpha, M)}{P(D|\alpha, \beta, M)}, \tag{13.10}$$

where $\mathbf{x}$ is the vector containing all of the weights and biases in the network, D represents the training data set, α and β are parameters associated with the density functions $P(D|\mathbf{x}, \beta, M)$ and $P(\mathbf{x}|\alpha, M)$, and M is the selected model - the architecture of the network we have chosen (i.e., how many layers and how may neurons in each layer).

It is worth taking some time to investigate each of the terms in Eq. (13.10). First, $P(D|\mathbf{x}, \beta, M)$ is the probability density for the data, given a certain set of weights $\mathbf{x}$, the parameter β (which we will explain shortly), and the choice of model M. If we assume that the noise terms in Eq. (13.2) are independent and have a Gaussian distribution, then

$$P(D|\mathbf{x}, \beta, M) = \frac{1}{Z_D(\beta)} \exp(-\beta E_D), \tag{13.11}$$

where $\beta = 1/(2\sigma_\varepsilon^2)$, σ_ε^2 is the variance of each element of ε_q, E_D is the squared error (as defined in Eq. (13.3)), and

$$Z_D(\beta) = (2\pi\sigma_\varepsilon^2)^{N/2} = (\pi/\beta)^{N/2}, \tag{13.12}$$

where N is $Q \times S^M$, as in Eq. (12.34).

Likelihood Function

Eq. (13.11) is called the *likelihood function*. It is a function of the network weights $\mathbf{x}$, and it describes how likely a given data set is to occur, given a specific set of weights. The *maximum likelihood* method selects the weights so as to maximize the likelihood function, which in this Gaussian case is the same as minimizing the squared error E_D. Therefore, our standard sum squared error performance index can be derived statistically with the assumption of Gaussian noise in the training set, and our standard choice for the weights is the maximum likelihood estimate.

Maximum Likelihood

Prior Density

Now consider the second term on the right side of Eq. (13.10): $P(\mathbf{x}|\alpha, M)$. This is called the *prior density*. It embodies our knowledge about the network weights before we collect any data. Bayesian statistics allows us to incorporate prior knowledge through the prior density. For example, if we assume that the weights are small values centered around zero, we might select a zero-mean Gaussian prior density:

$$P(\mathbf{x}|\alpha, M) = \frac{1}{Z_W(\alpha)}\exp(-\alpha E_W) \tag{13.13}$$

where $\alpha = 1/(2\sigma_w^2)$, σ_w^2 is the variance of each of the weights, E_W is the sum squared weights (as defined in Eq. (13.4)), and

$$Z_W(\alpha) = (2\pi\sigma_w^2)^{n/2} = (\pi/\alpha)^{n/2}, \tag{13.14}$$

where n is the number of weights and biases in the network, as in Eq. (12.35).

Evidence

Posterior Density

The final term on the right side of Eq. (13.10) is $P(D|\alpha, \beta, M)$. This is called the *evidence*, and it is a normalizing term that is not a function of $\mathbf{x}$. If our objective is to find the weights $\mathbf{x}$ that maximize the *posterior density* $P(\mathbf{x}|D, \alpha, \beta, M)$, then we do not need to be concerned with $P(D|\alpha, \beta, M)$. (However, it will be important later for estimating α and β.)

With the Gaussian assumptions that we made earlier, we can rewrite the posterior density, using Eq. (13.10), in the following form:

$$P(\mathbf{x} \mid D, \alpha, \beta, M) = \frac{\frac{1}{Z_W(\alpha)}\frac{1}{Z_D(\beta)}\exp(-(\beta E_D + \alpha E_W))}{\text{Normalization Factor}} = \frac{1}{Z_F(\alpha, \beta)}\exp(-F(\mathbf{x})) \tag{13.15}$$

where $Z_F(\alpha, \beta)$ is a function of α and β (but not a function of $\mathbf{x}$), and $F(\mathbf{x})$ is our regularized performance index, which we defined in Eq. (13.4). To find the most probable value for the weights, we should maximize the posterior density $P(\mathbf{x} \mid D, \alpha, \beta, M)$. This is equivalent to minimizing the regularized performance index $F(\mathbf{x}) = \beta E_D + \alpha E_W$.

Therefore, our regularized performance index can be derived using Bayesian statistics, with the assumption of Gaussian noise in the training set and a Gaussian prior density for the network weights. We will identify the weights that maximize the posterior density as $\mathbf{x}^{MP}$, or *most probable*. This is to be contrasted with the weights that maximize the likelihood function: $\mathbf{x}^{ML}$.

Most Probable

Note how this statistical framework provides a physical meaning for the parameters α and β. The parameter β is inversely proportional to the variance in the measurement noise ε_q. Therefore, if the noise variance is large, β will be small, and the regularization ratio α/β will be large. This will force the resulting weights to be small and the network function to be smooth (as seen in Figure 13.6). The larger the measurement noise, the more we will smooth the network function, in order to average out the affects of the noise.

The parameter α is inversely proportional to the variance in the prior distribution for the network weights. If this variance is large, it means that we have very little certainty about the values of the network weights, and, therefore, they might be very large. The parameter α will then be small, and the regularization ratio α/β will also be small. This will allow the network weights to be large, and the network function will be allowed to have more variation (as seen in Figure 13.6). The larger the variance in the prior density for the network weights, the more variation the network function will be allowed to have.

Level II Bayesian Framework

So far we have an interesting statistical derivation of the regularized performance index and some new insight into the meanings of the parameters α and β, but what we really want to find is a way to estimate these parameters from the data. In order to do this, we need to take the Bayesian analysis to another level. If we want to estimate α and β using Bayesian analysis, we need the probability density $P(\alpha, \beta | D, M)$. Using Bayes' rule this can written

$$P(\alpha, \beta|D, M) = \frac{P(D|\alpha, \beta, M)P(\alpha, \beta|M)}{P(D|M)}. \tag{13.16}$$

This has the same format as Eq. (13.10), with the likelihood function and the prior density in the numerator of the right hand side. If we assume a uniform (constant) prior density $P(\alpha, \beta|M)$ for the regularization parameters α and β, then maximizing the posterior is achieved by maximizing the likelihood function $P(D|\alpha, \beta, M)$. However, note that this likelihood function is the normalization factor (evidence) from Eq. (13.10). Since we have assumed that all probabilities have a Gaussian form, we know the form for the posterior density of Eq. (13.10). It is shown in Eq. (13.15). Now we can solve Eq. (13.10) for the normalization factor (evidence).

$$\begin{aligned} P(D|\alpha, \beta, M) &= \frac{P(D|\mathbf{x}, \beta, M)P(\mathbf{x}|\alpha, M)}{P(\mathbf{x}|D, \alpha, \beta, M)} \\ &= \frac{\left[\frac{1}{Z_D(\beta)}\exp(-\beta E_D)\right]\left[\frac{1}{Z_W(\alpha)}\exp(-\alpha E_W)\right]}{\frac{1}{Z_F(\alpha, \beta)}\exp(-F(\mathbf{x}))} \\ &= \frac{Z_F(\alpha, \beta)}{Z_D(\beta)Z_W(\alpha)} \cdot \frac{\exp(-\beta E_D - \alpha E_W)}{\exp(-F(\mathbf{x}))} = \frac{Z_F(\alpha, \beta)}{Z_D(\beta)Z_W(\alpha)} \end{aligned} \tag{13.17}$$

Note that we know the constants $Z_D(\beta)$ and $Z_W(\alpha)$ from Eq. (13.12) and Eq. (13.14). The only part we do not know is $Z_F(\alpha, \beta)$. However, we can estimate it by using a Taylor series expansion.

Since the objective function has the shape of a quadratic in a small area surrounding a minimum point, we can expand $F(\mathbf{x})$ in a second order Taylor series (see Eq. (8.9)) around its minimum point, $\mathbf{x}^{MP}$, where the gradient is zero:

$$F(\mathbf{x}) \approx F(\mathbf{x}^{MP}) + \frac{1}{2}(\mathbf{x} - \mathbf{x}^{MP})^T \mathbf{H}^{MP}(\mathbf{x} - \mathbf{x}^{MP}), \tag{13.18}$$

where $\mathbf{H} = \beta\nabla^2 E_D + \alpha\nabla^2 E_W$ is the Hessian matrix of $F(\mathbf{x})$, and $\mathbf{H}^{MP}$ is the Hessian evaluated at $\mathbf{x}^{MP}$. We can now substitute this approximation into the expression for the posterior density, Eq. (13.15):

$$P(\mathbf{x}|D, \alpha, \beta, M) \approx \frac{1}{Z_F}\exp\left[-F(\mathbf{x}^{MP}) - \frac{1}{2}(\mathbf{x} - \mathbf{x}^{MP})^T \mathbf{H}^{MP}(\mathbf{x} - \mathbf{x}^{MP})\right], \tag{13.19}$$

which can be rewritten as

$$P(\mathbf{x}|D, \alpha, \beta, M) \approx \left\{\frac{1}{Z_F}\exp(-F(\mathbf{x}^{MP}))\right\}\exp\left[-\frac{1}{2}(\mathbf{x}-\mathbf{x}^{MP})^T\mathbf{H}^{MP}(\mathbf{x}-\mathbf{x}^{MP})\right]. \quad (13.20)$$

The standard form of the Gaussian density is

$$P(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^n\left|(\mathbf{H}^{MP})^{-1}\right|}}\exp\left(-\frac{1}{2}(\mathbf{x}-\mathbf{x}^{MP})^T\mathbf{H}^{MP}(\mathbf{x}-\mathbf{x}^{MP})\right). \quad (13.21)$$

Therefore, equating Eq. (13.21) with Eq. (13.20), we can solve for $Z_F(\alpha, \beta)$:

$$Z_F(\alpha, \beta) \approx (2\pi)^{n/2}(\det((\mathbf{H}^{MP})^{-1}))^{1/2}\exp(-F(\mathbf{x}^{MP})). \quad (13.22)$$

Placing this result into Eq. (13.17), we can solve for the optimal values for α and β at the minimum point. We do this by taking the derivative with respect to each of the log of Eq. (13.17) and set them equal to zero. This yields (see Solved Problem P13.3):

$$\alpha^{MP} = \frac{\gamma}{2E_W(\mathbf{x}^{MP})} \text{ and } \beta^{MP} = \frac{N-\gamma}{2E_D(\mathbf{x}^{MP})}, \quad (13.23)$$

Effective # of Parameters

where $\gamma = n - 2\alpha^{MP}\mathrm{tr}(\mathbf{H}^{MP})^{-1}$ is called the *effective number of parameters*, and n is the total number of parameters in the network. The term γ is a measure of how many parameters (weights and biases) in the neural network are effectively used in reducing the error function. It can range from zero to n. (See the example on page 13-23 for more analysis of γ.)

Bayesian Regularization Algorithm

The Bayesian optimization of the regularization parameters requires the computation of the Hessian matrix of $F(\mathbf{x})$ at the minimum point $\mathbf{x}^{MP}$. We propose using the Gauss-Newton approximation to the Hessian matrix [FoHa97], which is readily available if the Levenberg-Marquardt optimization algorithm is used to locate the minimum point (see Eq. (12.31)). The additional computation required for optimization of the regularization is minimal.

Here are the steps required for Bayesian optimization of the regularization parameters, with the Gauss-Newton approximation to the Hessian matrix:

0. Initialize α, β and the weights. The weights are initialized randomly, and then E_D and E_W are computed. Set $\gamma = n$, and compute α and β using Eq. (13.23).

1. Take one step of the Levenberg-Marquardt algorithm toward minimizing the objective function $F(\mathbf{x}) = \beta E_D + \alpha E_W$.

2. Compute the effective number of parameters $\gamma = n - 2\alpha\mathrm{tr}(\mathbf{H})^{-1}$, mak-

ing use of the Gauss-Newton approximation to the Hessian available in the Levenberg-Marquardt training algorithm:
$\mathbf{H} = \nabla^2 F(\mathbf{x}) \approx 2\beta \mathbf{J}^T \mathbf{J} + 2\alpha \mathbf{I}_n$, where $\mathbf{J}$ is the Jacobian matrix of the training set errors (see Eq. (12.37)).

3. Compute new estimates for the regularization parameters $\alpha = \frac{\gamma}{2E_W(\mathbf{x})}$ and $\beta = \frac{N-\gamma}{2E_D(\mathbf{x})}$.
4. Now iterate steps 1 through 3 until convergence.

Bear in mind that with each reestimate of the regularization parameters α and β the objective function $F(\mathbf{x})$ changes; therefore, the minimum point is moving. If traversing the performance surface generally moves toward the next minimum point, then the new estimates for the regularization parameters will be more precise. Eventually, the precision will be good enough that the objective function will not significantly change in subsequent iterations. Thus, we will obtain convergence.

GNBR

When this Gauss-Newton approximation to Bayesian regularization (*GNBR*) algorithm is used, the best results are obtained if the training data is first mapped into the range [-1,1] (or some similar region). We will discuss this preprocessing of the training data in Chapter 22.

In Figure 13.7 you can see the results of training a 1-20-1 network with GNBR on the same data set represented in Figure 13.4 and Figure 13.6. The network has fit the underlying function, without overfitting to the noise. The fit looks similar to that obtained in Figure 13.6, with the regularization ratio set to $\alpha/\beta = 0.01$. In fact, at the completion of training with GNBR, the final regularization ratio for this example was $\alpha/\beta = 0.0137$.

The training process for this example is illustrated in Figure 13.8. In the upper left of this figure, you see the squared error on the training set. Notice that it does not necessarily go down at each iteration. In the upper right of the figure, you see the squared testing error. This was obtained by comparing the network function to the true function at a number of points between -1 and 1. It is a measure of the generalization capability of the network. (This would not be possible in a practical case, where the true function was unknown.) Note that the testing error is at its minimum at the completion of training.

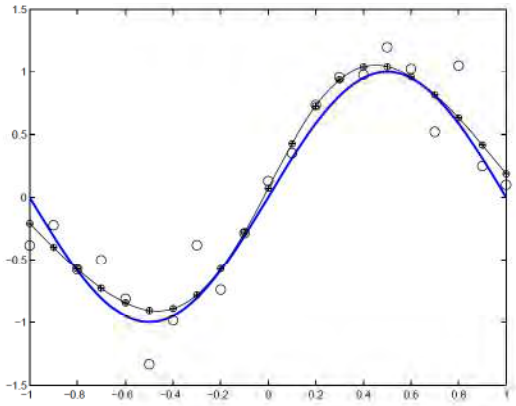


Figure 13.7 Bayesian Regularization Fit

Figure 13.8 also shows the regularization ratio α/β and the effective number of parameters γ during training. These parameters have no particular meaning during the training process, but at the completion of training they are significant. As we mentioned earlier, the final regularization ratio was $\alpha/\beta = 0.0137$, which is consistent with our earlier investigation of regularization—illustrated in Figure 13.6. The final effective number of parameters was $\gamma = 5.2$. This is out of a total of 61 total weights and biases in the network.

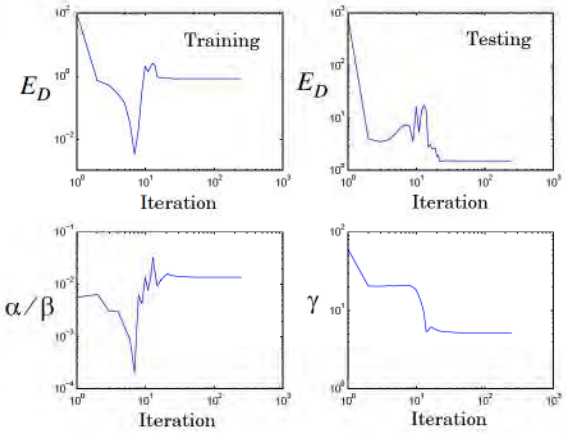


Figure 13.8 Bayesian Regularization Training Process

The fact that in this example the effective number of parameters is much less than the total number of parameters (6 versus 61) means that we

might well have been able to use a smaller network to fit this data. There are two disadvantages of a large network: 1) it may overfit the data, and 2) it requires more computation to calculate the network output. We have overcome the first disadvantage by training with GNBR; although the network has 61 parameters, it is equivalent to a network with only 6 parameters. The second disadvantage is only important if the calculation time for the network response is critical to the application. This is not usually the case, since the time to calculate a network response to a particular input is measured in milliseconds. In those cases where the calculation time is significant, you can train a smaller network on the data.

On the other hand, when the effective number of parameters is close to the total number of parameters, this can mean that the network is not large enough to fit the data. In this case, you should increase the size of the network and retrain on the data set.



To experiment with Bayesian Regularization, use the MATLAB® Neural Network Design Demonstration Bayesian Regularization (`nnd17breg`).

Relationship Between Early Stopping and Regularization

We have discussed two techniques for improving network generalization: early stopping and regularization. These two methods were developed in very different ways, but they both improve generalization by restricting the network weights and, therefore, producing a network with fewer effective parameters. Early stopping restricts the network weights by stopping the training before the weights have converged to the minimum of the squared error. Regularization restricts the weights by adding a term to the squared error that penalizes large weights. In this section we want to demonstrate, using a linear example, an approximate equivalence of these two methods. During the process, we will also shed some light on the meaning of the effective number of parameters, γ. This development is based on the more general procedures described in [SjLj94].

Early Stopping Analysis

Consider the single layer linear network shown in Figure 10.1. We have shown in Eq. (10.12) and Eq. (10.14) that the mean square error performance function for this linear network is quadratic, of the form

$$F(\mathbf{x}) = c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}, \tag{13.24}$$

where $\mathbf{A}$ is the Hessian matrix. In order to study the performance of early stopping, we will analyze the evolution of the steepest descent algorithm on this linear network. From Eq. (10.16), we know that the gradient of the performance index is

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d}. \tag{13.25}$$

Therefore, the steepest descent algorithm (see Eq. (9.10)) will be

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \mathbf{g}_k = \mathbf{x}_k - \alpha(\mathbf{A}\mathbf{x}_k + \mathbf{d}). \tag{13.26}$$

We want to know how close we come to the minimum of the squared error at each iteration. For quadratic performance indices, we know that the minimum will occur at the following point (see Eq. (8.62)):

$$\mathbf{x}^{ML} = -\mathbf{A}^{-1}\mathbf{d}, \tag{13.27}$$

where the superscript ML indicates that this result maximizes the likelihood function, in addition to minimizing the squared error, as we saw in Eq. (13.11).

We can now rewrite Eq. (13.26) as

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha\mathbf{A}(\mathbf{x}_k + \mathbf{A}^{-1}\mathbf{d}) = \mathbf{x}_k - \alpha\mathbf{A}(\mathbf{x}_k - \mathbf{x}^{ML}). \tag{13.28}$$

With some additional algebra we can find

$$\mathbf{x}_{k+1} = [\mathbf{I} - \alpha\mathbf{A}]\mathbf{x}_k + \alpha\mathbf{A}\mathbf{x}^{ML} = \mathbf{M}\mathbf{x}_k + [\mathbf{I} - \mathbf{M}]\mathbf{x}^{ML}, \tag{13.29}$$

where $\mathbf{M} = [\mathbf{I} - \alpha\mathbf{A}]$. The next step is to relate $\mathbf{x}_{k+1}$ to the initial guess $\mathbf{x}_0$. Starting at the first iteration, using Eq. (13.29), we have

$$\mathbf{x}_1 = \mathbf{M}\mathbf{x}_0 + [\mathbf{I} - \mathbf{M}]\mathbf{x}^{ML}, \tag{13.30}$$

where the initial guess $\mathbf{x}_0$ usually consists of random values near zero. Continuing to the second iteration:

$$\begin{aligned}
\mathbf{x}_2 &= \mathbf{M}\mathbf{x}_1 + [\mathbf{I} - \mathbf{M}]\mathbf{x}^{ML} \\
&= \mathbf{M}^2\mathbf{x}_0 + \mathbf{M}[\mathbf{I} - \mathbf{M}]\mathbf{x}^{ML} + [\mathbf{I} - \mathbf{M}]\mathbf{x}^{ML} \\
&= \mathbf{M}^2\mathbf{x}_0 + \mathbf{M}\mathbf{x}^{ML} - \mathbf{M}^2\mathbf{x}^{ML} + \mathbf{x}^{ML} - \mathbf{M}\mathbf{x}^{ML} \\
&= \mathbf{M}^2\mathbf{x}_0 + \mathbf{x}^{ML} - \mathbf{M}^2\mathbf{x}^{ML} = \mathbf{M}^2\mathbf{x}_0 + [\mathbf{I} - \mathbf{M}^2]\mathbf{x}^{ML}
\end{aligned} \tag{13.31}$$

Following similar steps, at the k^{th} iteration we have

$$\mathbf{x}_k = \mathbf{M}^k\mathbf{x}_0 + [\mathbf{I} - \mathbf{M}^k]\mathbf{x}^{ML}, \tag{13.32}$$

This key result shows how far we progress from the initial guess to the maximum likelihood weights in k iterations. We will use this result later to compare with regularization.

Regularization Analysis

Recall from Eq. (13.4) that the regularized performance index adds a penalty term to the sum squared error, as in

$$F(\mathbf{x}) = \beta E_D + \alpha E_W . \tag{13.33}$$

For the following analysis, it will more convenient to consider the following equivalent (because the minimum occurs at the same place) performance index

$$F^*(\mathbf{x}) = \frac{F(\mathbf{x})}{\beta} = E_D + \frac{\alpha}{\beta} E_W = E_D + \rho E_W , \tag{13.34}$$

which has only one regularization parameter.

The sum squared weight penalty term E_W can be written

$$E_W = (\mathbf{x} - \mathbf{x}_0)^T (\mathbf{x} - \mathbf{x}_0) , \tag{13.35}$$

where the nominal value $\mathbf{x}_0$ is normally taken to be the zero vector.

In order to locate the minimum of the regularized performance index, which is also the most probable value $\mathbf{x}^{MP}$, we will set the gradient equal to zero:

$$\nabla F^*(\mathbf{x}) = \nabla E_D + \rho \nabla E_W = \mathbf{0} . \tag{13.36}$$

The gradient of the penalty term, Eq. (13.35), is

$$\nabla E_W = 2(\mathbf{x} - \mathbf{x}_0) . \tag{13.37}$$

From Eq. (13.25) and Eq. (13.28), the gradient of the sum squared error is

$$\nabla E_D = \mathbf{A}\mathbf{x} + \mathbf{d} = \mathbf{A}(\mathbf{x} + \mathbf{A}^{-1}\mathbf{d}) = \mathbf{A}(\mathbf{x} - \mathbf{x}^{ML}) . \tag{13.38}$$

We can now set the total gradient to zero:

$$\nabla F^*(\mathbf{x}) = \mathbf{A}(\mathbf{x} - \mathbf{x}^{ML}) + 2\rho(\mathbf{x} - \mathbf{x}_0) = \mathbf{0} . \tag{13.39}$$

The solution of Eq. (13.39) is the most probable value for the weights, $\mathbf{x}^{MP}$. We can make that substitution and perform some algebra to obtain

$$\begin{aligned} \mathbf{A}(\mathbf{x}^{MP} - \mathbf{x}^{ML}) &= -2\rho(\mathbf{x}^{MP} - \mathbf{x}_0) = -2\rho(\mathbf{x}^{MP} - \mathbf{x}^{ML} + \mathbf{x}^{ML} - \mathbf{x}_0) \\ &= -2\rho(\mathbf{x}^{MP} - \mathbf{x}^{ML}) - 2\rho(\mathbf{x}^{ML} - \mathbf{x}_0) \end{aligned} \tag{13.40}$$

Now combine the terms multiplying $(\mathbf{x}^{MP} - \mathbf{x}^{ML})$:

$$(\mathbf{A} + 2\rho\mathbf{I})(\mathbf{x}^{MP} - \mathbf{x}^{ML}) = 2\rho(\mathbf{x}_0 - \mathbf{x}^{ML}). \tag{13.41}$$

Solving for $(\mathbf{x}^{MP} - \mathbf{x}^{ML})$, we find

$$(\mathbf{x}^{MP} - \mathbf{x}^{ML}) = 2\rho(\mathbf{A} + 2\rho\mathbf{I})^{-1}(\mathbf{x}_0 - \mathbf{x}^{ML}) = \mathbf{M}_\rho(\mathbf{x}_0 - \mathbf{x}^{ML}), \tag{13.42}$$

where $\mathbf{M}_\rho = 2\rho(\mathbf{A} + 2\rho\mathbf{I})^{-1}$.

We want to know the relationship between the regularized solution $\mathbf{x}^{MP}$ and the minimum of the squared error $\mathbf{x}^{ML}$, so we can solve Eq. (13.42) for $\mathbf{x}^{MP}$:

$$\mathbf{x}^{MP} = \mathbf{M}_\rho\mathbf{x}_0 + [\mathbf{I} - \mathbf{M}_\rho]\mathbf{x}^{ML}. \tag{13.43}$$

This is the key result that describes the relationship between the regularized solution and the minimum of the squared error. By comparing Eq. (13.43) with Eq. (13.32), we can investigate the relationship between early stopping and regularization. We will do that in the next section.

Connection Between Early Stopping and Regularization

To compare early stopping and regularization, we need to compare Eq. (13.43) and Eq. (13.32). They are summarized in Figure 13.9. We would like to find out when these two solutions are equal. In other words, when do early stopping and regularization produce the same weights?

Early Stopping	*Regularization*
$\mathbf{x}_k = \mathbf{M}^k\mathbf{x}_0 + [\mathbf{I} - \mathbf{M}^k]\mathbf{x}^{ML}$	$\mathbf{x}^{MP} = \mathbf{M}_\rho\mathbf{x}_0 + [\mathbf{I} - \mathbf{M}_\rho]\mathbf{x}^{ML}$
$\mathbf{M} = [\mathbf{I} - \alpha\mathbf{A}]$	$\mathbf{M}_\rho = 2\rho(\mathbf{A} + 2\rho\mathbf{I})^{-1}$

Figure 13.9 Early Stopping and Regularization Solutions

The key matrix for early stopping is $\mathbf{M}^k = [\mathbf{I} - \alpha\mathbf{A}]^k$. The key matrix for regularization is $\mathbf{M}_\rho = 2\rho(\mathbf{A} + 2\rho\mathbf{I})^{-1}$. If these two matrices are equal, then the weights for early stopping will be the same as the weights for regularization. In Eq. (9.22) we showed that the eigenvectors of $\mathbf{M}$ are the same as the eigenvectors of $\mathbf{A}$ and that the eigenvalues of $\mathbf{M}$ are $(1 - \alpha\lambda_i)$, where the eigenvalues of $\mathbf{A}$ are λ_i. The eigenvalues of $\mathbf{M}^k$ are then

$$eig(\mathbf{M}^k) = (1 - \alpha\lambda_i)^k. \tag{13.44}$$

Now let's consider the matrix $\mathbf{M}_\rho$. First, using the same procedures that led to Eq. (9.22), we can show that the eigenvectors of $(\mathbf{A} + 2\rho\mathbf{I})$ are the same as the eigenvectors of $\mathbf{A}$, and the eigenvalues of $(\mathbf{A} + 2\rho\mathbf{I})$ are

$(2\rho + \lambda_i)$. Also, the eigenvectors of the inverse of a matrix are the same as the eigenvectors of the original matrix, and the eigenvalues of the inverse are the reciprocals of the original eigenvalues. Therefore, the eigenvectors of $\mathbf{M}_\rho$ are the same as the eigenvectors of $\mathbf{A}$, and the eigenvalues of $\mathbf{M}_\rho$ are

$$eig(\mathbf{M}_\rho) = \frac{2\rho}{(\lambda_i + 2\rho)}. \tag{13.45}$$

Therefore, in order for $\mathbf{M}^k$ to equal $\mathbf{M}_\rho$, they just need to have equal eigenvalues:

$$\frac{2\rho}{(\lambda_i + 2\rho)} = (1 - \alpha\lambda_i)^k. \tag{13.46}$$

Take the logarithm of both sides:

$$-\log\left(1 + \frac{\lambda_i}{2\rho}\right) = k\log(1 - \alpha\lambda_i). \tag{13.47}$$

These expressions are equal at $\lambda_i = 0$, so they will always be equal if their derivatives are equal. Taking derivatives of both sides, we have

$$-\frac{1}{\left(1 + \frac{\lambda_i}{2\rho}\right)}\frac{1}{\rho} = \frac{k}{1 - \alpha\lambda_i}(-\alpha), \tag{13.48}$$

or

$$\alpha k = \frac{1}{2\rho}\,\frac{(1 - \alpha\lambda_i)}{(1 + \lambda_i/(2\rho))}. \tag{13.49}$$

If $\alpha\lambda_i$ is small (slow, stable learning) and $\lambda_i/(2\rho)$ is small, then we have the approximate result

$$\alpha k \cong \frac{1}{2\rho}. \tag{13.50}$$

Therefore, early stopping is approximately equivalent to regularization. Increasing the number of iterations k is approximately the same as decreasing the regularization parameter ρ. This makes intuitive sense, because increasing the number of iterations, or decreasing the regularization parameter, can lead to overfitting.

Example, Interpretation of Effective Number of Parameters



We will illustrate this result with a simple example. Suppose that we have a single layer, linear network with no bias. The input/target pairs are given by

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

where the probability of the first pair is 0.75, and the probability of the second pair is 0.25. Following Eq. (10.13) and Eq. (10.15), we can find the quadratic mean square error performance index as

$$c = E[t^2] = (1)^2(0.75) + (-1)^2(0.25) = 1,$$

$$\mathbf{h} = E[t\mathbf{z}] = (0.75)(1)\begin{bmatrix}1\\1\end{bmatrix} + (0.25)(-1)\begin{bmatrix}-1\\1\end{bmatrix} = \begin{bmatrix}1\\0.5\end{bmatrix},$$

$$\mathbf{d} = -2\mathbf{h} = (-2)\begin{bmatrix}1\\0.5\end{bmatrix} = \begin{bmatrix}-2\\-1\end{bmatrix},$$

$$\mathbf{A} = 2\mathbf{R} = 2(E[\mathbf{z}\mathbf{z}^T]) = 2\left((0.75)\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}1 & 1\end{bmatrix} + 0.25\begin{bmatrix}-1\\1\end{bmatrix}\begin{bmatrix}-1 & 1\end{bmatrix}\right) = \begin{bmatrix}2 & 1\\1 & 2\end{bmatrix},$$

$$E_D = c + \mathbf{x}^T\mathbf{d} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}.$$

The minimum of the mean squared error occurs at

$$\mathbf{x}^{ML} = -\mathbf{A}^{-1}\mathbf{d} = \mathbf{R}^{-1}\mathbf{h} = \begin{bmatrix}1 & 0.5\\0.5 & 1\end{bmatrix}^{-1}\begin{bmatrix}1\\0.5\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}.$$

Now let's investigate the eigensystem of the Hessian matrix of E_D:

$$\nabla^2 E_D(\mathbf{x}) = \mathbf{A} = 2\mathbf{R} = \begin{bmatrix}2 & 1\\1 & 2\end{bmatrix}.$$

To find the eigenvalues:

$$\left|\mathbf{A} - \lambda\mathbf{I}\right| = \begin{vmatrix}2-\lambda & 1\\1 & 2-\lambda\end{vmatrix} = \lambda^2 - 4\lambda + 3 = (\lambda - 1)(\lambda - 3),$$

$$\lambda_1 = 1, \qquad \lambda_2 = 3.$$

To find the eigenvectors:

$$\left[\mathbf{A} - \lambda\mathbf{I}\right]\mathbf{v} = 0.$$

For $\lambda_1 = 1$,

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\mathbf{v}_1 = 0 \qquad \mathbf{v}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix},$$

and for $\lambda_2 = 3$,

$$\begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}\mathbf{v}_2 = 0 \qquad \mathbf{v}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

The contour plot for E_D is shown in Figure 13.10

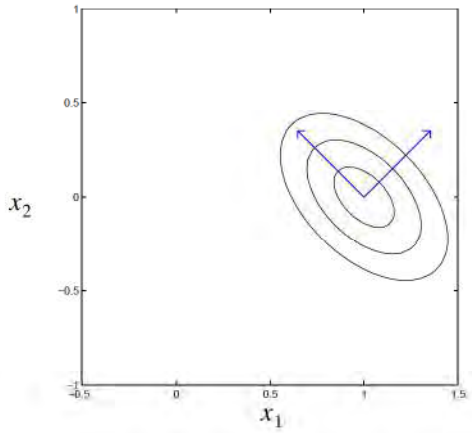


Figure 13.10 Contour Plot for E_D

Now consider the regularized performance index of Eq. (13.34). Its Hessian matrix will be

$$\nabla^2 F^*(\mathbf{x}) = \nabla^2 E_D + \rho\nabla^2 E_W = \nabla^2 E_D + 2\rho\mathbf{I} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} + \rho\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 2+2\rho & 1 \\ 1 & 2+2\rho \end{bmatrix}.$$

In Figure 13.11 we have contour plots for F as ρ is equal to 0, 1 and ∞.

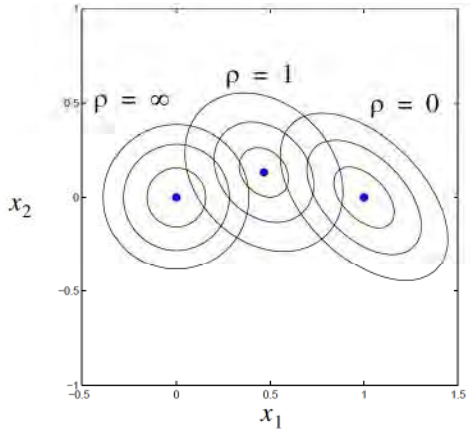


Figure 13.11 Contour Plot for F

In Figure 13.12 the blue curve represents the movement of $\mathbf{x}^{MP}$ as ρ is varied.

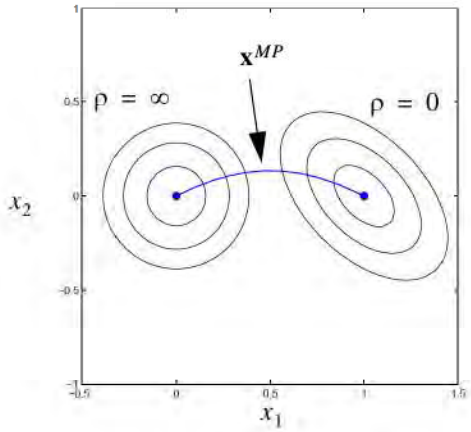


Figure 13.12 $\mathbf{x}^{MP}$ as ρ is Varied

Now let's compare this regularization result with early stopping. Figure 13.13 shows the steepest descent trajectory for minimizing E_D, starting from very small values for the weights. If we stop early, the result will fall along the blue curve. Notice that this curve is very close to the regularization curve in Figure 13.12. If the number of iterations is very small, this is equivalent to a very large value for ρ. As the number of iterations increases, it is equivalent to reducing ρ.

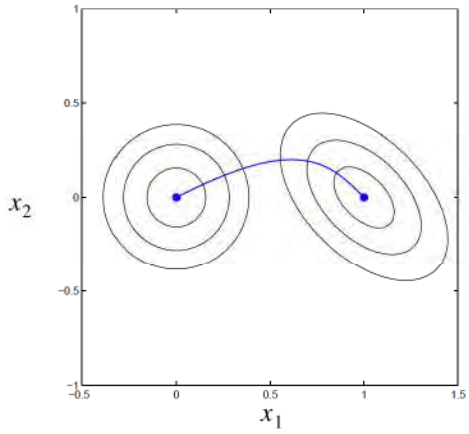


Figure 13.13 Steepest Descent Trajectory



To experiment with the relationship between Early Stopping and Regularization, use the MATLAB® Neural Network Design Demonstration Early Stopping/Regularization (`nnd17esr`).

It is useful to consider the relationship between the eigenvalues and eigenvectors of the Hessian matrix $\nabla^2 E_D(\mathbf{x})$ and the results of regularization and early stopping. In this example, λ_2 is larger than λ_1, so E_D has higher curvature in the $\mathbf{v}_2$ direction. This means that we will get a quicker reduction in the squared error if we move in that direction first. This is shown in Figure 13.13, as the initial steepest descent movement is almost in the direction of $\mathbf{v}_2$. Note also that in regularization, as shown in Figure 13.12, as ρ decreases from a large value, the weights move first in the $\mathbf{v}_2$ direction. For a given change in the weights, this direction provides the largest reduction in the squared error.

Since the eigenvalue λ_1 is smaller than λ_2, we only move in the $\mathbf{v}_1$ direction after achieving significant reduction in E_D in the $\mathbf{v}_2$ direction. This would be even more pronounced if the difference between λ_1 and λ_2 were greater. In the limiting case, where $\lambda_1 = 0$, we would not have to move in the $\mathbf{v}_1$ direction at all. We would only need to move in the $\mathbf{v}_2$ direction to get the complete reduction in the squared error. (This would be the case of the stationary valley, as in Figure 8.9.) Note that in this case we would only be effectively using one parameter, even though the network has two weights. (Of course, this one effective parameter is some combination of the two weights.) Therefore, the effective number of parameters is related to the number of eigenvalues of $\nabla^2 E_D(\mathbf{x})$ that are significantly different than zero. We will analyze this in detail in the next section.

Effective Number of Parameters

Recall the previous definition for the effective number of parameters:

$$\gamma = n - 2\alpha^{MP}\mathrm{tr}\{(\mathbf{H}^{MP})^{-1}\} \tag{13.51}$$

We can express this in terms of the eigenvalues of $\nabla^2 E_D(\mathbf{x})$. First, we can write the Hessian matrix as

$$\mathbf{H}(\mathbf{x}) = \nabla^2 F(\mathbf{x}) = \beta\nabla^2 E_D + \alpha\nabla^2 E_W = \beta\nabla^2 E_D + 2\alpha\mathbf{I}. \tag{13.52}$$

Using arguments similar to those leading to Eq. (13.44), we can show that the eigenvalues of $\mathbf{H}(\mathbf{x})$ are $(\beta\lambda_i + 2\alpha)$. We can then use two properties of eigenvalues to compute $\mathrm{tr}\{\mathbf{H}^{-1}\}$. First, the eigenvalues of $\mathbf{H}^{-1}$ are the reciprocals of the eigenvalues of $\mathbf{H}$, and, second, the trace of a matrix is equal to the sum of its eigenvalues. Using these two properties, we can write

$$\mathrm{tr}\{\mathbf{H}^{-1}\} = \sum_{i=1}^{n} \frac{1}{\beta\lambda_i + 2\alpha}. \tag{13.53}$$

We can now write the effective number of parameters as

$$\gamma = n - 2\alpha^{MP}\mathrm{tr}\{(\mathbf{H}^{MP})^{-1}\} = n - \sum_{i=1}^{n} \frac{2\alpha}{\beta\lambda_i + 2\alpha} = \sum_{i=1}^{n} \frac{\beta\lambda_i}{\beta\lambda_i + 2\alpha}, \tag{13.54}$$

or

$$\gamma = \sum_{i=1}^{n} \frac{\beta\lambda_i}{\beta\lambda_i + 2\alpha} = \sum_{i=1}^{n} \gamma_i, \tag{13.55}$$

where

$$\gamma_i = \frac{\beta\lambda_i}{\beta\lambda_i + 2\alpha}. \tag{13.56}$$

Note that $0 \le \gamma_i \le 1$, so the effective number of parameters γ must fall between zero and n. If all of the eigenvalues of $\nabla^2 E_D(\mathbf{x})$ are large, then the effective number of parameters will equal the total number of parameters. If some of the eigenvalues are very small, then the effective number of parameters will equal the number of large eigenvalues, as was also demonstrated by our example in the previous section. Large eigenvalues mean large curvature, which means that the performance index changes rapidly along those eigenvectors. Every large eigenvector represents a productive direction for optimizing performance.

Summary of Results

Problem Statement

A network trained to generalize will perform as well in new situations as it does on the data on which it was trained.

$$E_D = \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q)$$

Methods for Improving Generalization

Estimating Generalization Error - The Test Set

Given a limited amount of available data, it is important to hold aside a certain subset during the training process. After the network has been trained, we will compute the errors that the trained network makes on this *test set*. The test set errors will then give us an indication of how the network will perform in the future; they are a measure of the generalization capability of the network.

Early Stopping

The available data (after removing the test set) is divided into two parts: a training set and a validation set. The training set is used to compute gradients or Jacobians and to determine the weight update at each iteration. When the error on the validation set goes up for several iterations, the training is stopped, and the weights that produced the minimum error on the validation set are used as the final trained network weights.

Regularization

$$F(\mathbf{x}) = \beta E_D + \alpha E_W = \beta \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q) + \alpha \sum_{i=1}^{n} x_i^2$$

Bayesian Regularization

Level I Bayesian Framework

$$P(\mathbf{x} \mid D, \alpha, \beta, M) = \frac{P(D \mid \mathbf{x}, \beta, M) P(\mathbf{x} \mid \alpha, M)}{P(D \mid \alpha, \beta, M)}$$

$$P(D|\mathbf{x}, \beta, M) = \frac{1}{Z_D(\beta)} \exp(-\beta E_D), \ \beta = 1/(2\sigma_\varepsilon^2)$$

$$Z_D(\beta) = (2\pi\sigma_\varepsilon^2)^{N/2} = (\pi/\beta)^{N/2}$$

$$P(\mathbf{x}|\alpha, M) = \frac{1}{Z_W(\alpha)} \exp(-\alpha E_W), \ \alpha = 1/(2\sigma_w^2)$$

$$Z_W(\alpha) = (2\pi\sigma_w^2)^{n/2} = (\pi/\alpha)^{n/2}$$

$$P(\mathbf{x}|D, \alpha, \beta, M) = \frac{1}{Z_F(\alpha, \beta)} \exp(-F(\mathbf{x}))$$

Level II Bayesian Framework

$$P(\alpha, \beta|D, M) = \frac{P(D|\alpha, \beta, M)P(\alpha, \beta|M)}{P(D|M)}$$

$$\alpha^{MP} = \frac{\gamma}{2E_W(\mathbf{x}^{MP})} \text{ and } \beta^{MP} = \frac{N-\gamma}{2E_D(\mathbf{x}^{MP})}$$

$$\gamma = n - 2\alpha^{MP} \text{tr}(\mathbf{H}^{MP})^{-1}$$

Bayesian Regularization Algorithm

0. Initialize α, β and the weights. The weights are initialized randomly, and then E_D and E_W are computed. Set $\gamma = n$, and compute α and β using Eq. (13.23).
1. Take one step of the Levenberg-Marquardt algorithm toward minimizing the objective function $F(\mathbf{x}) = \beta E_D + \alpha E_W$.
2. Compute the effective number of parameters $\gamma = N - 2\alpha \text{tr}(\mathbf{H})^{-1}$, making use of the Gauss-Newton approximation to the Hessian available in the Levenberg-Marquardt training algorithm:
$\mathbf{H} = \nabla^2 F(\mathbf{x}) \approx 2\beta \mathbf{J}^T\mathbf{J} + 2\alpha \mathbf{I}_n$, where $\mathbf{J}$ is the Jacobian matrix of the training set errors (see Eq. (12.37)).
3. Compute new estimates for the regularization parameters $\alpha = \frac{\gamma}{2E_W(\mathbf{x})}$ and $\beta = \frac{N-\gamma}{2E_D(\mathbf{x})}$.
4. Now iterate steps 1 through 3 until convergence.

Relationship Between Early Stopping and Regularization

Early Stopping	*Regularization*
$\mathbf{x}_k = \mathbf{M}^k\mathbf{x}_0 + [\mathbf{I} - \mathbf{M}^k]\mathbf{x}^{ML}$	$\mathbf{x}^{MP} = \mathbf{M}_\rho\mathbf{x}_0 + [\mathbf{I} - \mathbf{M}_\rho]\mathbf{x}^{ML}$
$\mathbf{M} = [\mathbf{I} - \alpha\mathbf{A}]$	$\mathbf{M}_\rho = 2\rho(\mathbf{A} + 2\rho\mathbf{I})^{-1}$

$$eig(\mathbf{M}^k) = (1 - \alpha\lambda_i)^k$$

$$eig(\mathbf{M}_\rho) = \frac{2\rho}{(\lambda_i + 2\rho)}$$

$$\alpha k \cong \frac{1}{2\rho}$$

Effective Number of Parameters

$$\gamma = \sum_{i=1}^{n} \frac{\beta\lambda_i}{\beta\lambda_i + 2\alpha}$$

$$0 \le \gamma \le n$$

Solved Problems

P13.1 In this problem and in the following one we want to investigate the relationship between maximum likelihood methods and Bayesian methods. Suppose that we have a random variable that is uniformly distributed between 0 and x. We take a series of Q independent samples of the random variable. Find the maximum likelihood estimate of x.

Before we begin this problem, let's review the Level I Bayesian formulation of Eq. (13.10). We will not need the Level II formulation for this simple problem, so we do not need the regularization parameters. Also, we only have a single parameter to estimate, so x is a scalar. Eq. (13.10) can then be simplified to

$$P(x|D) = \frac{P(D|x)P(x)}{P(D)}.$$

We are interested in the maximum likelihood estimate for this problem, so we need to find the value of x that maximizes the likelihood term $P(D|x)$. The data is the Q independent samples from the uniformly distributed random variable. A graph of the uniform density function is given in Figure P13.1.

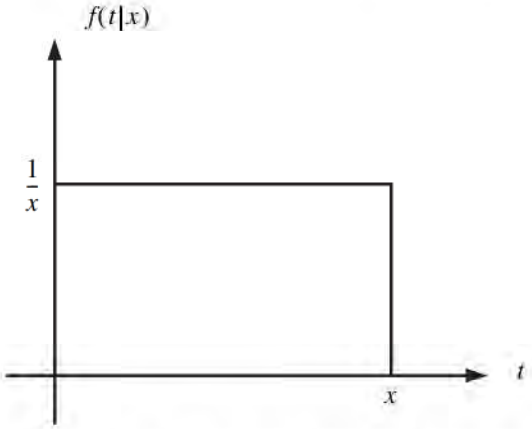


Figure P13.1 Uniform Density Function

The definition can be written

$$f(t|x) = \begin{cases} \frac{1}{x}, 0 \le t \le x \\ 0, \text{elsewhere} \end{cases}.$$

If we have Q independent samples of the random variable, then we can multiply each of the individual probabilities to get the joint probability of all samples:

$$P(D|x) = \prod_{i=1}^{Q} f(t_i|x) = \begin{cases} \frac{1}{x^Q}, 0 \le t_i \le x, \text{ for all } i \\ 0, \text{elsewhere} \end{cases} = \begin{cases} \frac{1}{x^Q}, x \ge max(t_i) \\ 0, x < max(t_i) \end{cases}$$

The plot of the resulting likelihood function is shown in Figure P13.1.

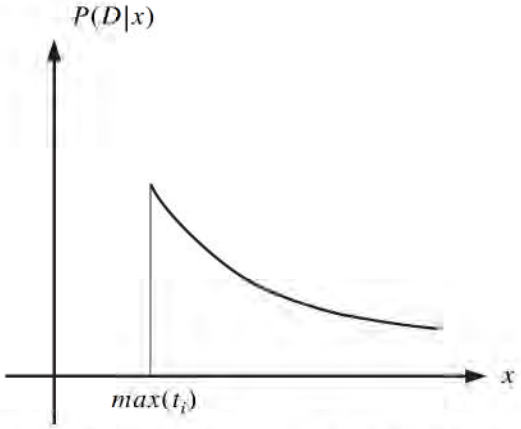


Figure P13.2 Likelihood Function for Solved Problem P13.1

From this plot, we can see that the value of x that maximizes the likelihood function is

$$x^{ML} = max(t_i).$$

Therefore, the maximum likelihood estimate of x is the maximum value obtained from the Q independent samples of the random variable. This seems like a reasonable estimate of x, which is the upper limit of the random variable.

P13.2 In this problem we will compare the maximum likelihood and Bayesian estimators. Assume that we have a series of measurements of a random signal in noise:

$$t_i = x + \varepsilon_i.$$

Assume that the noise has a Gaussian density, with zero mean:

$$f(\varepsilon_i) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{\varepsilon_i^2}{2\sigma^2}\right)$$

i. Find the maximum likelihood estimate of x.

ii. Find the most probable estimate of x. Assume that x is a zero-mean random variable, with Gaussian prior density:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma_x}\exp\left(-\frac{x^2}{2\sigma_x^2}\right) = \frac{1}{Z_W(\alpha)}\exp(-\alpha E_W)$$

i. To find the maximum likelihood estimate, we need to find the likelihood function $P(D|x)$. This represents the density of the data, given x. The first step is to use the noise density to find the density of the measurement. Since, with x given, the density for the measurement would be the same as the density for the noise, but with a mean of x, we have

$$f(t_i|x) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(t_i-x)^2}{2\sigma^2}\right).$$

Assuming that the measurement noises are independent, we can multiply the probability densities:

$$P(D|x) = f(t_1, t_2, \ldots, t_Q|x) = f(t_1|x)f(t_2|x)\ldots f(t_Q|x) = P(D|x)$$

$$= \frac{1}{(2\pi)^{Q/2}\sigma^Q}\exp\left(-\frac{\sum_{i=1}^{Q}(t_i-x)^2}{2\sigma^2}\right) = \frac{1}{Z(\beta)}\exp(-\beta E_D)$$

where

$$\beta = \frac{1}{2\sigma^2},\ E_D = \sum_{i=1}^{Q}(t_i-x)^2 = \sum_{i=1}^{Q} e_i^2,\ Z(\beta) = (\pi/\beta)^{Q/2}.$$

To maximize the likelihood, we should minimize E_D. Setting the derivative to zero, we find

$$\frac{dE_D}{dx} = \frac{d}{dx}\sum_{i=1}^{Q}(t_i-x)^2 = -2\sum_{i=1}^{Q}(t_i-x) = -2\left[\left(\sum_{i=1}^{Q} t_i\right) - Qx\right] = 0.$$

Solving for x, we find the maximum likelihood estimate:

$$x^{ML} = \frac{1}{Q}\sum_{i=1}^{Q} t_i$$

ii. To find the most probable estimate, we need to use Bayes' rule (Eq. (13.10)) to find the posterior density:

$$P(x|D) = \frac{P(D|x)P(x)}{P(D)}.$$

The likelihood function $P(D|x)$ was found above to be

$$P(D|x) = \frac{1}{Z(\beta)}\exp(-\beta E_D)$$

The prior density is

$$P(x) = f(x) = \frac{1}{\sqrt{2\pi}\sigma_x}\exp\left(-\frac{x^2}{2\sigma_x^2}\right) = \frac{1}{Z_W(\alpha)}\exp(-\alpha E_W),$$

where

$$\alpha = \frac{1}{2\sigma_x^2},\ Z_W(\alpha) = (\pi/\alpha)^{1/2},\ E_W = x^2.$$

The posterior density can then be computed as

$$\begin{aligned} P(x|D) &= f(x|t_1, t_2, \ldots, t_Q) \\ &= \frac{f(t_1, t_2, \ldots, t_Q|x)f(x)}{f(t_1, t_2, \ldots, t_Q)} \\ &= \frac{\frac{1}{Z_D(\beta)}\frac{1}{Z_W(\alpha)}\exp(-(\beta E_D + \alpha E_W))}{\text{Normalization Factor}} \end{aligned}$$

To find the most probable value for x, we maximize the posterior density. This is equivalent to minimizing

$$\beta E_D + \alpha E_W = \beta\sum_{i=1}^{Q}(t_i - x)^2 + \alpha x^2.$$

To find the minimum, we take the derivative with respect to x and set it equal to zero:

$$\frac{d}{dx}(\beta E_D + \alpha E_W) = \frac{d}{dx}\left(\beta \sum_{i=1}^{Q} (t_i - x)^2 + \alpha x^2\right) = -2\beta \sum_{i=1}^{Q} (t_i - x) + 2\alpha x$$

$$= -2\beta\left[\left(\sum_{i=1}^{Q} t_i\right) - Qx\right] + 2\alpha x$$

$$= -2\left[\beta\left(\sum_{i=1}^{Q} t_i\right) - (\alpha + Q\beta)x\right] = 0$$

Solving for μ, we obtain

$$x^{MP} = \frac{\beta\left(\sum_{i=1}^{Q} t_i\right)}{\alpha + Q\beta}$$

Notice that as α goes to zero (variance σ_x^2 goes to infinity), x^{MP} approaches x^{ML}. Increasing the variance of the prior density represents increased uncertainty in our prior knowledge about x. With large prior uncertainty, we rely on the data for our estimate of x, which leads to the maximum likelihood estimate.

Figure P13.3 illustrates $P(D|x)$, $P(x)$ and $P(x|D)$ for the case where $\sigma_x^2 = 2$, $\sigma^2 = 1$, $Q = 1$ and $t_1 = 1$. Here the variance associated with the measurement is smaller than the variance associated with our prior density for x, so x^{MP} is closer to $x^{ML} = t_1 = 1$ than it is to the maximum of the prior density, which occurs at 0.



To experiment with this signal in noise example, use the MATLAB® Neural Network Design Demonstration Signal Plus Noise (**nnd13spn**).

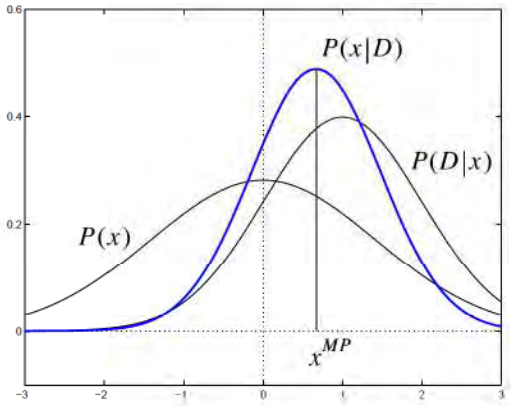


Figure P13.3 Prior and Posterior Density Functions

P13.3 Derive Eq. (13.23).

To solve for α^{MP} and β^{MP}, we will take the derivatives of the log of $P(D|\alpha, \beta, M)$, given in Eq. (13.17), with respect to α and β, and set the derivatives to zero. Taking the log of Eq. (13.17), and substituting Eq. (13.12), Eq. (13.14) and Eq. (13.22), we obtain

$$\begin{aligned}\log P(D|\alpha, \beta, M) &= \log(Z_F) - \log(Z_D(\beta)) - \log(Z_W(\alpha)) \\ &= \frac{n}{2}\log(2\pi) - \frac{1}{2}\log\det(\mathbf{H}^{MP}) - F(\mathbf{x}^{MP}) - \frac{N}{2}\log\left(\frac{\pi}{\beta}\right) - \frac{n}{2}\log\left(\frac{\pi}{\alpha}\right) \\ &= -F(\mathbf{x}^{MP}) - \frac{1}{2}\log\det(\mathbf{H}^{MP}) + \frac{N}{2}\log(\beta) + \frac{n}{2}\log(\alpha) + \frac{n}{2}\log 2 - \frac{N}{2}\log\pi\end{aligned}$$

We will consider first the second term in this expression. Since $\mathbf{H}$ is the Hessian of F in Eq. (13.4), we can write it as $\mathbf{H} = \nabla^2 F = \nabla^2(\beta E_D) + \nabla^2(\alpha E_W) = \beta\mathbf{B} + 2\alpha\mathbf{I}$, where $\mathbf{B} = \nabla^2 E_D$. If we let λ^h be an eigenvalue of $\mathbf{H}$ and λ^b be an eigenvalue of $\beta\mathbf{B}$, then $\lambda^h = \lambda^b + 2\alpha$ for all corresponding eigenvalues. Now we take the derivative of the second term in the above equation with respect to α. Since the determinant of a matrix can be expressed as the product of its eigenvalues, we can reduce it as shown below, where $\mathrm{tr}(\mathbf{H}^{-1})$ is the trace of the inverse of the Hessian $\mathbf{H}$.

$$\frac{\partial}{\partial\alpha}\frac{1}{2}\log\det\mathbf{H} = \frac{1}{2\det\mathbf{H}}\frac{\partial}{\partial\alpha}\left(\prod_{k=1}^{n}\lambda^{h}\right)$$

$$= \frac{1}{2\det\mathbf{H}}\frac{\partial}{\partial\alpha}\left(\prod_{i=1}^{n}(\lambda_i^b + 2\alpha)\right)$$

$$= \frac{1}{2\det\mathbf{H}}\left[\sum_{i=1}^{n}\left(\prod_{j\neq i}(\lambda_j^b + 2\alpha)\right)\frac{\partial}{\partial\alpha}(\lambda_i^b + 2\alpha)\right]$$

$$= \frac{\sum_{i=1}^{n}\left(\prod_{j\neq i}(\lambda_j^b + 2\alpha)\right)}{\prod_{i=1}^{n}(\lambda_i^b + 2\alpha)}$$

$$= \sum_{i=1}^{n}\frac{1}{\lambda_i^b + 2\alpha} = \mathrm{tr}(\mathbf{H}^{-1})$$

Next, we will take the derivative of the same term with respect to β. First, define the parameter γ, as shown below, and expand it for use in our next step. The parameter γ is referred to as the effective number of parameters.

$$\gamma \equiv n - 2\alpha\,\mathrm{tr}(\mathbf{H}^{-1})$$

$$= n - 2\alpha\sum_{i=1}^{N}\frac{1}{\lambda_i^b + 2\alpha} = \sum_{i=1}^{n}\left(1 - \frac{2\alpha}{\lambda_i^b + 2\alpha}\right) = \sum_{i=1}^{n}\left(\frac{\lambda_i^b}{\lambda_i^b + 2\alpha}\right) = \sum_{i=1}^{n}\frac{\lambda_i^b}{\lambda_i^h}$$

Now take the derivative of $\frac{1}{2}\log\det(\mathbf{H}^{MP})$ with respect to β.

$$\begin{aligned}
\frac{\partial}{\partial\beta}\frac{1}{2}\log\det\mathbf{H} &= \frac{1}{2\det\mathbf{H}}\frac{\partial}{\partial\beta}\left(\prod_{k=1}^{n}\lambda_k^h\right) \\
&= \frac{1}{2\det\mathbf{H}}\frac{\partial}{\partial\beta}\left(\prod_{i=1}^{n}(\lambda_i^b+2\alpha)\right) \\
&= \frac{1}{2\det\mathbf{H}}\left[\sum_{i=1}^{n}\left(\prod_{j\neq i}(\lambda_j^b+2\alpha)\right)\frac{\partial}{\partial\beta}(\lambda_i^b+2\alpha)\right] \\
&= \frac{\frac{1}{2}\sum_{i=1}^{n}\left(\prod_{j\neq i}(\lambda_j^b+2\alpha)\left(\frac{\lambda_i^b}{\beta}\right)\right)}{\prod_{i=1}^{n}(\lambda_i^b+2\alpha)} \\
&= \frac{1}{2\beta}\sum_{i=1}^{n}\frac{\lambda_i^b}{\lambda_i^b+2\alpha} = \frac{\gamma}{2\beta}
\end{aligned}$$

where the fourth step is derived from the fact that λ_i^b is an eigenvalue of $\beta\mathbf{B}$, and therefore the derivative of λ_i^b with respect to β is just the eigenvalue of $\mathbf{B}$ which is λ_i^b/β.

Now we are finally ready to take the derivatives of all terms in $\log P(D|\alpha,\beta,M)$ and set them equal to zero. The derivative with respect to α will be

$$\begin{aligned}
\frac{\partial}{\partial\alpha}\log P(D|\alpha,\beta,M) &= \frac{\partial}{\partial\alpha}F(\mathbf{w}^{\mathrm{MP}}) - \frac{\partial}{\partial\alpha}\frac{1}{2}\log\det(\mathbf{H}^{\mathrm{MP}}) + \frac{\partial}{\partial\alpha}\frac{n}{2}\log\alpha \\
&= \frac{\partial}{\partial\alpha}(\alpha E_W(\mathbf{w}^{\mathrm{MP}})) - \mathrm{tr}(\mathbf{H}^{\mathrm{MP}})^{-1} + \frac{n}{2\alpha^{\mathrm{MP}}} \\
&= -E_W(\mathbf{w}^{\mathrm{MP}}) - \mathrm{tr}(\mathbf{H}^{\mathrm{MP}})^{-1} + \frac{n}{2\alpha^{\mathrm{MP}}} = 0
\end{aligned}$$

Rearranging terms, and using our definition of γ, we have

$$\begin{aligned}
E_W(\mathbf{w}^{\mathrm{MP}}) &= \frac{n}{2\alpha^{\mathrm{MP}}} - \mathrm{tr}(\mathbf{H}^{\mathrm{MP}})^{-1} \\
2\alpha^{\mathrm{MP}}E_W(\mathbf{w}^{\mathrm{MP}}) &= n - 2\alpha^{\mathrm{MP}}\mathrm{tr}(\mathbf{H}^{\mathrm{MP}})^{-1} = \gamma \\
\alpha^{\mathrm{MP}} &= \frac{\gamma}{2E_W(\mathbf{w}^{\mathrm{MP}})}
\end{aligned}$$