

Fundamentals of Computational Neuroscience

Second Edition



Thomas P. Trappenberg

OXFORD

Fundamentals of Computational Neuroscience

This page intentionally left blank

Fundamentals of Computational Neuroscience

SECOND EDITION

Thomas P. Trappenberg
Dalhousie University

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Great Clarendon Street, Oxford OX2 6DP

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide in

Oxford New York

Auckland Cape Town Dar es Salaam Hong Kong Karachi
Kuala Lumpur Madrid Melbourne Mexico City Nairobi
New Delhi Shanghai Taipei Toronto

With offices in

Argentina Austria Brazil Chile Czech Republic France Greece
Guatemala Hungary Italy Japan Poland Portugal Singapore
South Korea Switzerland Thailand Turkey Ukraine Vietnam

Oxford is a registered trade mark of Oxford University Press
in the UK and in certain other countries

Published in the United States
by Oxford University Press Inc., New York

© Oxford University Press, 2010
© Thomas Trappenberg, 2002

The moral rights of the author have been asserted
Database right Oxford University Press (maker)

First edition published 2002
Second edition published 2010

All rights reserved. No part of this publication may be reproduced,
stored in a retrieval system, or transmitted, in any form or by any means,
without the prior permission in writing of Oxford University Press,
or as expressly permitted by law, or under terms agreed with the appropriate
reprographics rights organization. Enquiries concerning reproduction
outside the scope of the above should be sent to the Rights Department,
Oxford University Press, at the address above

You must not circulate this book in any other binding or cover
and you must impose the same condition on any acquirer

British Library Cataloguing in Publication Data
Data available

Library of Congress Cataloging in Publication Data
Data available

Typeset by the author
Printed in Great Britain
on acid-free paper by
CPI Anthony Rowe

ISBN 978-0-19-956841-3
1 3 5 7 9 10 8 6 4 2

Oxford University Press makes no representation, express or implied, that the drug
dosages in this book are correct. Readers must therefore always check the product
information and clinical procedures with the most up-to-date published product
information and data sheets provided by the manufacturers and the most recent codes of
conduct and safety regulations. The authors and the publishers do not accept responsibility
or legal liability for any errors in the text or for the misuse or misapplication of material in
this work. Except where otherwise stated, drug dosages and recommendations are for the
non-pregnant adult who is not breast-feeding.

*Dedicated to
my parents Dr Rüdiger and Marga
my children Kai and Nami
and my wife Kanayo*

This page intentionally left blank

Foreword

I am fortunate to have the opportunity to present to you the second edition of ‘Fundamentals’ with major reorganizations and additions. My goal for this edition was to increase the focus of presentations while adding discussions on important new developments. For example, I find it increasingly enlightening to view the brain as an anticipatory memory system, and I have added a discussion of this theory to the introductory chapter and to the last chapter, which contains now, in addition, several related sections, including introductions and discussions of Boltzmann machines, generative models, Bayesian models and adaptive resonance theory.

Chapter 2 now includes an improved emphasis on synaptic functions, and the popular spiking model of Izhikevich was added to Chapter 3. Chapter 4 is entirely dedicated to synaptic plasticity, while Chapter 5 summarizes more biological background on system levels and explores networks of spiking neurons. A brief introduction to a support vector machine, which is currently considered the best machine learning model, is added to Chapter 6. Chapter 7 is now dedicated to cortical maps, with a major enhancement of the discussion of tuning curves, self-organizing maps, and dynamical neural field theory. The point attractor networks discussed in Chapter 8 are now a natural extension of the models discussed in Chapter 7. Finally, new and important system-level models have been added to the last two chapters, and the appendices have been revised.

To make this edition more useable as textbook, I have included simulation sections with program examples and some example exercises in most chapters. Additional classroom resources are available on the web site for this book, including the figures in pdf format, example slides, and program versions for two open source alternatives to MATLAB, Octave and SciLab. I am aware that courses in this area can take many different forms, depending on the background of students and the emphasis of material chosen by the instructor. Thus, all material might not be necessary for your course, and some guidance to the most essential reading is included in the introductory chapter. Not all sections include the extensive discussion that might be necessary in classroom settings, but it was important for me to mention some issues without lengthy explanations, to show directions for further studies while keeping the book to a reasonable length.

This page intentionally left blank

Acknowledgements

It is difficult to express my gratitude to many of my colleagues since so many have contributed through valuable publications, enlightening talks, or stimulating discussions – too many to list them all. But I am specifically thankful for direct comments on the previous text or on new drafts by Alan Fine, Steve Grossberg, Alexander Hanuschkin, Geoff Hinton, Jason Satel, Michael Schmitt, Dominic Standage, Fumio Yamazaki, and Si Wu. I am also grateful for the hospitality I received at the Bernstein Centre for Computational Neuroscience in Freiburg, Germany, and at Future University, Hakodate, Japan, during my sabbatical during which most of the revisions were made.

This page intentionally left blank

Preface

Computational neuroscience is still a young and dynamically developing discipline, and some choice of topics and presentation style had to be made. This text introduces some fundamental concepts, with an emphasis on neuronal models and network properties. In contrast to most of the research literature, the book tries to highlight the reasons behind common assumptions and simplifications used in models.

The themes included in this book are chosen to provide some path through the different levels of description of the brain. The chapters of the book can be divided into three major parts as shown in Fig. 1. Chapters 2–4 focus on basic mechanisms of neuronal information-processing, which include the change of membrane potentials through ion channels, spike generations, and synaptic plasticity. The next part of the book describes the information processing capabilities of basic networks, including feedforward and competitive recurrent networks. The last part describes some examples of combining such elementary networks, and describes more system-level models of the brain.

Most models in the book are quite general and are aimed at illustrating basic mechanisms of information processing in the brain. In the research literature, the basic elements reviewed in this book are often combined in specific ways to model specific brain areas. Our hope is that the study of the basic models in this book will enable the reader to follow some of the recent research literature in computational neuroscience.

While we tried to emphasize some important concepts, we did not want to give the impression that the chosen path is the only direction in computational neuroscience. Therefore, we sometimes mention some issues without extensive discussion. These comments are intended to increase the reader's awareness of some issues and to provide some keywords to facilitate further literature searches. Also, while some examples of specific brain areas are mentioned in this book, a comprehensive review of models in computational neuroscience is beyond the scope of this text.

Mathematical formulas and programming examples

This book includes mathematical formulas and concepts. However, we use these mathematical concepts strictly as practical tools, and we tried to avoid more theoretical discussions, including detailed proofs. Mathematical formulas are intended to be descriptive in this book, and we tried to avoid mathematical notations that are too detailed. We are well aware that some readers might not have extensive experience with such concepts and formulations. However, we did not try to avoid mathematical formulations for several reasons. Foremost, the notations of mathematics allow a precision in formulations which

Basic neurons
Chapter 2: Membrane potentials and spikes
Chapter 3: Simplified neurons and population nodes
Chapter 4: Synaptic plasticity
Basic networks
Chapter 5: Random networks
Chapter 6: Feedforward network
Chapter 7: Competitive networks
Chapter 8: Point attractor networks
System-level models
Chapter 9: Modular models
Chapter 10: Hierarchical models

Fig. 1 This book can be divided into three parts. The first three chapters describe the basic information-processing elements, the second part focuses on basic networks of such elements, and the third part gives examples of system-level models.

are lengthy to achieve with plain written language. Also, the mathematical formulas can often be translated directly into programs and other quantitative evaluations. There is no reason to be afraid of formulas, and it is important to see beyond the symbols and to understand their meaning.

Much of the mathematics used in this book is based on notations used to simplify the descriptions. This includes the use of vector and matrix notations, which will drastically shorten the specification of network models. Appendix A includes a review of such notations. This appendix also includes a description of some functions used in this book. We recommend some tutorials on such materials to allow students to move beyond these technicalities.

Formulas are the basis for quantifying models in this book, and mathematical formulas allow a precise implementation on a computer or analytical treatment. Most models in this book describe the change of a quantity with time, such as a membrane potential or synaptic weight values. Equations that describe such changes are called *differential equations*. A comprehensive knowledge of the theory of differential equations is not required for understanding this book, but it does require one to learn a specific example of differential equations, that of a *leaky integrator*. A basic knowledge of the numerical approaches to solving differential equations is essential for this book and many other dynamic modelling approaches. Appendix B reviews some of the techniques and methods of differential equations and their numerical integration.

Another mathematical theory, that of random numbers, is very useful in computational neuroscience and should be taught in such a course. In neuroscience (as in other disciplines), we often get different values each time we perform a measurement, and random numbers describe such situations. We often think of these circumstances as noise, but it is also useful to think about random variables and statistics in terms of uncertainties. Learning and reasoning in uncertain circumstances is a fundamental requirement of the brain, and we will argue that mental functions can be viewed as probabilistic reasoning. A review of the basic concepts of probability theory are included in Appendix C. Also, probability theory is the basis of information theory which has some impact on computational neuroscience. Appendix D discusses this topic further.

This book also includes a few examples of powerful analytical techniques to give the reader a flavour of some of them. Not every neuroscientist has to perform such calculations themselves, but it is necessary to comprehend the general ideas to be able to get support from those who specialize in such techniques, if required in your own research. However, it is instructive when studying this book to perform some numerical experiments yourself. We therefore included an introduction to a modern programming environment that is very much suited for many of the models in neuroscience. Writing programs and creating advanced graphics can be learned easily within a short time, even without extensive prior computer knowledge. Some basic programs are included in the text in sections labelled *Simulations*. All the experiments are discussed in the text beforehand, so that the study of these sections is only necessary if the reader wants to explore the specific details of the simulations.

References

This book does not provide a historical account of the development of ideas in computational neuroscience. Indeed, extensive references have been avoided where possible to concentrate on describing fundamental ideas. References to the original research literature are only provided when following corresponding examples closely, or when the topics discussed in the book cannot be found in the standard references mentioned in the additional reading list at the end of each chapter.

The additional reading lists, which include some brief comments on the references, are intended to guide further reading and to include some specific references to the few papers that are discussed explicitly in the text and are not referenced in corresponding figure captions. Thus, any further research should be accompanied with a more exhaustive literature review of specific topics, which might well point out more recent developments in this area. The World Wide Web provides us with plenty of tools to find leads to well recognized papers, and we did not attempt to provide an extensive biography of the history of ideas in this area. It should be understood that the intention of this book is to introduce ideas of the computational neuroscience community, and that we do not claim that all of these ideas are our own. We apologize for not mentioning many other research papers that have had major impacts in the field and that have guided a lot of our thinking.

This book is about neuroscience, and several issues mentioned in this book can be found in basic textbooks of neuroscience, in particular some basic descriptions of neurons and brain organization. I want to encourage each reader who is not familiar with these subjects to consult such, more standard, neuroscience books. In contrast to standard textbooks in neuroscience, we concentrate in this book on the computational aspects that are often not the major focus of other books. This book, instead, focuses on the modelling aspects, in particular the teaching of modelling, how to make useful abstractions, as well as how to use tools to analyse models, and some major contributions from such modelling studies. In contrast to other books on the technological aspects of neural networks, this book is intended to outline the relation of theoretical concepts or particular technological solutions to brain functions.

A guided tour through the chapters

Chapter 1 is a general discussion of the challenges and strategies of computational neuroscience. It includes a discussion of the role of models and abstractions, and reviews Marr's distinctions of different levels of analysis. Finally, a brief outline is given of a high-level theory the brain, that the brain is an anticipatory memory system.

Chapter 2 describes the workhorses of the brain – neurons – and their specific information-processing capabilities. The discussion includes the famous model of spike generations by Hodgkin and Huxley, and also focuses on the important information-processing mechanisms of dendrites and synapses. The chapter ends by alluding to the idea behind compartmental models, which are a major component of computational neuroscience. While some of you

might dedicate your research to this level, and should thus consult more specific books on these topics, we follow the path to network processing with simplified *point neurons*. As argued above, we highly recommend reading through the remainder of the book even if you are concentrating your research on neuronal levels, since constraints from system-level information processing in the brain are increasingly relevant to the neuronal level.

Chapter 3 is an introduction to some of the more popular models of spiking point neurons, as well as descriptions of population activities. These models are, in principle, fairly basic and could be introduced in a few paragraphs. The chapter therefore includes more extensive discussion of some properties of such model neurons, such as their response variability with noise. This discussion is included because it provides an opportunity to introduce some notions of random variables which are useful in describing brain processes. However, these discussions are secondary to the importance of the basic models, and a more thorough reading of these parts can be delayed until a deeper study of this area is desired. Such, more complementary or continuative, subject sections are marked with asterisks throughout the book. This chapter includes a brief discussion of how information is presented in neural activity, often referred to as neural codes. This is an area where information theory has contributed greatly to our understanding, and some further discussion is included in Appendix D.

Chapter 4 is about one of the most fascinating and far-reaching brain mechanisms, that of synaptic plasticity. The chapter is divided roughly into three parts, a general introduction to the essence of plasticity, that of building associations; one part that looks a bit further into the biophysical basis of synaptic plasticity; and a final part which formulates these findings in terms that can be used in network applications.

Chapter 5 is also divided roughly into three parts. The first part discusses some more anatomical background, which will become important in the following chapters. This chapter then starts to investigate properties of networks of neurons, starting with small random networks which reveal interesting problems of how activity can spread through networks via diverging-converging chains. The last part introduces some more realistic network simulations of spiking neurons which are able to demonstrate activity patterns that are so characteristic for activity in live tissue.

Chapter 6 is dedicated to the discussion of mapping networks. Such networks are fundamental for network processing in the brain, and this area of mainly feedforward processing has hugely influenced thinking in cognitive neuroscience. It has also been an area that has shaped machine learning research over the past three decades. The chapter includes a discussion of learning rules which put neural plasticity to action. We also take this opportunity to discuss more recent developments in machine learning, which have revolutionized the area of neural networks.

Chapter 7 returns to more specific brain processing architectures and information representation. After introducing the general model with the example of tuning curves, the chapter divides the more detailed discussions into two parts. In the first part, we discuss the exciting area of self-organizing maps, which form cortical representations of concepts through learning. The latter part of the chapter discusses the dynamics within these recurrent networks,

which give rise to many observations made in physiology and psychophysics.

Chapter 8 discusses similar architectures as in the previous chapter, in the more classic framework of recurrent associative networks. This is a fascinating area where methods from statistical physics have contributed so much and which has brought so many scientists from other fields into neuroscience. While these discussions can be found in many books on neural networks, this chapter includes a discussion of sparse representation, which is essential in brain processing and is easily overlooked in the standard formulation of such networks.

Chapter 9 moves toward more system-level models. This chapter has two main sections. In the first section, we discuss combining basic networks, starting with combining feedforward networks and then combining recurrent networks into modular designs. The second section focuses on control theory, and, in particular, on another major ingredient of learning, reinforcement learning.

Chapter 10 provides some examples of system-level theories. The chapter starts by stressing the importance of hierarchical representations to form invariant representations. It then discusses some examples of system-level models, one in attentive vision, and one about more general workspace processing. Finally, we discuss a theory that regards the brain as an anticipating memory system. After an outline of the principal thinking behind such theories, we discuss some more specific implementations and related models.

The **appendices** are intended to remind the reader about essential scientific concepts used frequently in the book. These include some basic mathematical notions, particularly vector and matrix notations. Also, most of the simulation programs are numerical integrations. It is not necessary to delve deeply into the theory of differential equations, and this appendix introduces the problem sufficiently to understand the applications in this book. As mentioned before, notions of probability theory are frequently used in this book. While many textbooks in statistics concentrate on specific techniques of hypothesis testing, our use is mainly concerned with random variables and density functions. The appendix on information theory, an application area of probability theory, is mainly aimed at furthering some of the discussions in the book. Finally, the appendix on MATLAB®¹ and comparable programming environments is intended to get you started running the example simulations in the book.

¹MATLAB is a registered trademark of The MathWorks, Inc.

Using this book in a course

Each chapter of the book is designed to provide introductory material to a specific topic area. The figures and programs of the book are provided in electronic format on the book's web site², and these web pages also provide some additional resources for teachers and students. The first eight chapters include a few examples of exercises, typically with some questions testing the student's understanding of an important concept and some small simulation exercises. Exercises in the last two chapter are omitted since a course should typically include a student project at this stage. It is advisable to spend some time on reviewing the mathematical concepts and to introduce programming with MATLAB. Corresponding material is included in the appendices. Such background material could be taught in tutorials to students with varying strengths in these topics.

²Web site with accompanying resources including programs are at <http://www.cs.dal.ca/~tt/fundamentals>

Not all sections of the chapters are required for an introductory course, while advanced courses could use *sections marked with the symbol* \diamond after the title to delve deeper into some material. Chapter 2 contains material which should be, to a large extent, familiar to neuroscience students, but might be essential background for students from other disciplines. Our experience has shown that synaptic and spiking mechanisms are often confused, and changes have been made in this edition to better stress these different components. A major focus of our treatment is the modelling of the biophysical processes, and simulations of such models help to deepen understanding of these processes. This book mentions compartmental models and alternative conductance-based models only very briefly. While introductory classes could skip over these discussions, advanced courses could extend these discussions with additional literature.

Essential material in Chapter 3 covers the basic spiking models and the population model. The discussion of noise models could be part of advanced reading and could be combined with a review of basic probability theory. Synaptic plasticity, discussed in Chapter 4, is so essential that this chapter contains a very general introduction to the essence of forming associations, and then discusses different rules. Most of this material should be covered in all classes, although the rest of the book uses mostly population models to simplify the discussions, and the examples of using plasticity rules in Section 4.4 could be skipped, although the Gaussian distribution of weights with Hebbian learning of random patterns is used later and could be explained then.

Chapter 5 is the start of the second part of the book with a focus on networks. This chapter starts with more biological facts on brain organization, and continues to describe some basic effects of random networks. The middle section on netlets could be skipped in a first reading, but the last section provides an interesting example of a spiking network which will be augmented later with Hebbian learning. The concept of mapping networks (Chapter 6) is essential for many models in computational neuroscience, and these networks can be used to demonstrate some fundamental issues in statistical learning theory. The most essential parts of this chapter are covered in the first two sections, whereas the next two sections cover additional reading with machine learning content. While machine learning is not intended to be the primary focus of this book, some of these discussions might be useful for advanced classes.

In Chapter 7, the Kohonen map, and basic DNF model, are essential, but the formal analysis of attractor states (Section 7.3.4) and Section 7.4 could be left for further reading. Some issues of population coding (Section 7.5) should also be mentioned. The concept of attractor networks (Chapter 8) is a major contribution of theoretical neuroscience and is therefore essential reading. However, some of the formal analysis, such as the signal-to-noise analyses, and the dynamical system point-of-view (Section 8.4), are not essential in a first reading.

The last two chapters are a collection of higher-level concepts. The book starts with examples of basic combinations of networks, although instructors might choose other examples. Even if these basic examples are skipped, a comprehensive course should mention reinforcement learning (Section 9.6) as an important ingredient in system-level learning. Similarly, Chapter 10 starts

with discussions of system-level models that could be replaced with alternative examples. However, the later sections on generative models, adaptive resonance theory and probabilistic reasoning (Section 10.3), are important concepts that should be mentioned. Further suggestions and submissions of material to be included on the web pages of the book are more than welcome.

This page intentionally left blank

Contents

1	Introduction	1
1.1	What is computational neuroscience?	1
1.1.1	Tools and specializations in neuroscience	2
1.1.2	Levels of organization in the brain	3
1.2	What is a model?	5
1.2.1	Phenomenological and explanatory models	6
1.2.2	Models in computational neuroscience	7
1.3	Is there a brain theory?	8
1.3.1	Emergence and adaptation	9
1.3.2	Levels of analysis	11
1.4	A computational theory of the brain	13
1.4.1	Why do we have brains?	13
1.4.2	The anticipating brain	14
	Exercises	16
	Further reading	17
I	Basic neurons	19
2	Neurons and conductance-based models	21
2.1	Biological background	21
2.1.1	Structural properties	22
2.1.2	Information-processing mechanisms	23
2.1.3	Membrane potential	24
2.1.4	Ion channels	25
2.2	Basic synaptic mechanisms and dendritic processing	26
2.2.1	Chemical synapses and neurotransmitters	26
2.2.2	Excitatory and inhibitory synapses	26
2.2.3	Modelling synaptic responses	27
2.2.4	Non-linear superposition of PSPs	32
2.3	The generation of action potentials: Hodgkin–Huxley equations	33
2.3.1	The minimal mechanisms	34
2.3.2	Ion pumps	35
2.3.3	Hodgkin–Huxley equations	35
2.3.4	Numerical integration	37
2.3.5	Refractory period	38
2.3.6	Propagation of action potentials	41
2.3.7	Above and beyond the Hodgkin–Huxley neuron: the Wilson model ◊	41

2.4	Including neuronal morphologies: compartmental models	46
2.4.1	Cable theory	47
2.4.2	Physical shape of neurons	48
2.4.3	Neuron simulators	49
	Exercises	50
	Further reading	50
3	Simplified neuron and population models	53
3.1	Basic spiking neurons	53
3.1.1	The leaky integrate-and-fire neuron	54
3.1.2	Response of IF neurons to very short and constant input currents	55
3.1.3	Activation function	57
3.1.4	The spike-response model	59
3.1.5	The Izhikevich neuron	61
3.1.6	The McCulloch–Pitts neuron	63
3.2	Spike-time variability ◊	64
3.2.1	Biological irregularities	64
3.2.2	Noise models for IF neurons	67
3.2.3	Simulating the variability of real neurons	67
3.2.4	The activation function depends on input	69
3.3	The neural code and the firing rate hypothesis	70
3.3.1	Correlation codes and coincidence detectors	71
3.3.2	How accurate is spike timing?	73
3.4	Population dynamics: modelling the average behaviour of neurons	74
3.4.1	Firing rates and population averages	74
3.4.2	Population dynamics for slow varying input	75
3.4.3	Motivations for population dynamics ◊	76
3.4.4	Rapid response of populations	78
3.4.5	Common activation functions	79
3.5	Networks with non-classical synapses: the sigma–pi node	81
3.5.1	Logical AND and sigma–pi nodes	82
3.5.2	Divisive inhibition	82
3.5.3	Further sources of modulatory effects between synaptic inputs	83
	Exercises	84
	Further reading	84
4	Associators and synaptic plasticity	87
4.1	Associative memory and Hebbian learning	87
4.1.1	Hebbian learning	88
4.1.2	Associations	89
4.1.3	Hebbian learning in the conditioning framework	91
4.1.4	Features of associators and Hebbian learning	93
4.2	The physiology and biophysics of synaptic plasticity	94
4.2.1	Typical plasticity experiments	94
4.2.2	Spike timing dependent plasticity	96
4.2.3	The calcium hypothesis and modelling chemical pathways	97

4.3	Mathematical formulation of Hebbian plasticity	99
4.3.1	Spike timing dependent plasticity rules	99
4.3.2	Hebbian learning in population and rate models	100
4.3.3	Negative weights and crossing synapses	104
4.4	Synaptic scaling and weight distributions	105
4.4.1	Examples of STDP with spiking neurons	106
4.4.2	Weight distributions in rate models	109
4.4.3	Competitive synaptic scaling and weight decay	110
4.4.4	Oja's rule and principal component analysis	113
	Exercises	116
	Further reading	116
II	Basic networks	117
5	Cortical organization and simple networks	119
5.1	Organization in the brain	119
5.1.1	Large-scale brain anatomy	120
5.1.2	Hierarchical organization of cortex	121
5.1.3	Rapid data transmission in the brain	123
5.1.4	The layered structure of neocortex	124
5.1.5	Columnar organization and cortical modules	125
5.1.6	Connectivity between neocortical layers	127
5.1.7	Cortical parameters	128
5.2	Information transmission in random networks ◊	130
5.2.1	The simple chain	130
5.2.2	Diverging–converging chains	130
5.2.3	Immunity of random networks to spontaneous background activity	131
5.2.4	Noisy background	132
5.2.5	Information transmission in large random networks	133
5.2.6	The spread of activity in small random networks	134
5.2.7	The expected number of active neurons in netlets	134
5.2.8	Netlets with inhibition	135
5.3	More physiological spiking networks	137
5.3.1	Random network	137
5.3.2	Networks with STDP and polychrony	140
	Exercises	142
	Further reading	142
6	Feed-forward mapping networks	143
6.1	The simple perceptron	143
6.1.1	Optical character recognition (OCR)	143
6.1.2	Mapping functions	145
6.1.3	The population node as perceptron	147
6.1.4	Boolean functions: the threshold node	148
6.1.5	Learning: the delta rule	150
6.2	The multilayer perceptron	155

6.2.1	The update rule for multilayer perceptrons	157
6.2.2	Generalization	159
6.2.3	The generalized delta rules	160
6.2.4	Biological plausibility of MLPs	163
6.3	Advanced MLP concepts ◊	165
6.3.1	Kernel machines and radial-basis function networks	165
6.3.2	Advanced learning	166
6.3.3	Batch versus online algorithm	168
6.3.4	Self-organizing network architectures and genetic algorithms	169
6.3.5	Mapping networks with context units	170
6.3.6	Probabilistic mapping networks	172
6.4	Support vector machines ◊	173
6.4.1	Large-margin classifiers	173
6.4.2	Soft-margin classifiers and the kernel trick	175
	Exercises	178
	Further reading	179
7	Cortical feature maps and competitive population coding	181
7.1	Competitive feature representations in cortical tissue	181
7.2	Self-organizing maps	183
7.2.1	The basic cortical map model	183
7.2.2	The Kohonen model	184
7.2.3	Ongoing refinements of cortical maps	188
7.3	Dynamic neural field theory	190
7.3.1	The centre-surround interaction kernel	191
7.3.2	Asymptotic states and the dynamics of neural fields	192
7.3.3	Examples of competitive representations in the brain	195
7.3.4	Formal analysis of attractor states ◊	198
7.4	‘Path’ integration and the Hebbian trace rule ◊	202
7.4.1	Path integration with asymmetrical weight kernels	202
7.4.2	Self-organization of a rotation network	204
7.4.3	Updating the network after learning	204
7.5	Distributed representation and population coding	205
7.5.1	Sparseness	206
7.5.2	Probabilistic population coding	207
7.5.3	Optimal decoding with tuning curves	209
7.5.4	Implementations of decoding mechanisms	210
	Exercises	212
	Further reading	213
8	Recurrent associative networks and episodic memory	215
8.1	The auto-associative network and the hippocampus	215
8.1.1	Different memory types	215
8.1.2	The hippocampus and episodic memory	218
8.1.3	Learning and retrieval phase	219
8.2	Point-attractor neural networks (ANN)	219
8.2.1	Network dynamics and training	220

8.2.2	Signal-to-noise analysis ◊	224
8.2.3	The phase diagram	227
8.2.4	Spurious states and the advantage of noise ◊	230
8.2.5	Noisy weights and diluted attractor networks	232
8.3	Sparse attractor networks and correlated patterns	233
8.3.1	Sparse patterns and expansion recoding	234
8.3.2	Control of sparseness in attractor networks	235
8.4	Chaotic networks: a dynamic systems view ◊	238
8.4.1	Attractors	239
8.4.2	Lyapunov functions	240
8.4.3	The Cohen–Grossberg theorem	242
8.4.4	Asymmetrical networks	243
8.4.5	Non-monotonic networks	246
	Exercises	246
	Further reading	247
III	System-level models	249
9	Modular networks, motor control, and reinforcement learning	251
9.1	Modular mapping networks	251
9.1.1	Mixture of experts	252
9.1.2	The ‘what-and-where’ task	253
9.1.3	Product of experts	256
9.2	Coupled attractor networks	257
9.2.1	Imprinted and composite patterns	258
9.2.2	Signal-to-noise analysis	259
9.3	Sequence learning	262
9.4	Complementary memory systems	264
9.4.1	Distributed model of working memory	264
9.4.2	Limited capacity of working memory	265
9.4.3	The spurious synchronization hypothesis	266
9.4.4	The interacting-reverberating-memory hypothesis	268
9.5	Motor learning and control	270
9.5.1	Feedback controller	271
9.5.2	Forward and inverse model controller	272
9.5.3	The cerebellum and motor control	273
9.6	Reinforcement learning	274
9.6.1	Classical conditioning and the reinforcement learning problem	275
9.6.2	Temporal delta rule	277
9.6.3	Temporal difference learning	278
9.6.4	The actor–critic scheme and the basal ganglia	282
	Further reading	286
10	The cognitive brain	289
10.1	Hierarchical maps and attentive vision	289

10.1.1 Invariant object recognition	289
10.1.2 Attentive vision	291
10.1.3 Attentional bias in visual search and object recognition	294
10.2 An interconnecting workspace hypothesis	295
10.2.1 The global workspace	295
10.2.2 Demonstration of the global workspace in the Stroop task	297
10.3 The anticipating brain	298
10.3.1 The brain as anticipatory system in a probabilistic frame- work	299
10.3.2 The Boltzmann machine	302
10.3.3 The restricted Boltzmann machine and contrastive Hebb- ian learning	304
10.3.4 The Helmholtz machine	306
10.3.5 Probabilistic reasoning: causal models and Bayesian net- works	308
10.3.6 Expectation maximization	310
10.4 Adaptive resonance theory	313
10.4.1 The basic model	313
10.4.2 ART1 equations	316
10.4.3 Simplified dynamics for unsupervised letter clustering	317
10.5 Where to go from here	319
Further reading	321
A Some useful mathematics	323
A.1 Vector and matrix notations	323
A.2 Distance measures	325
A.3 The δ -function	326
B Numerical calculus	327
B.1 Differences and sums	327
B.2 Numerical integration of an initial value problem	327
B.3 Euler method	328
B.4 Higher-order methods	330
B.5 Adaptive Runge–Kutta	331
Further reading	334
C Basic probability theory	335
C.1 Random numbers and their probability (density) function	335
C.2 Moments: mean, variance, etc.	336
C.3 Examples of probability (density) functions	338
C.3.1 Bernoulli distribution	338
C.3.2 Binomial distribution	338
C.3.3 Chi-square distribution	338
C.3.4 Exponential distribution	339
C.3.5 Lognormal distribution	339
C.3.6 Multinomial distribution	339
C.3.7 Normal (Gaussian) distribution	339
C.3.8 Poisson distribution	340

C.3.9 Uniform distribution	340
C.4 Cumulative probability (density) function and the Gaussian error function	340
C.5 Functions of random variables and the central limit theorem	341
C.6 Measuring the difference between distributions	342
C.6.1 Marginal, joined, and conditional distributions	343
Further reading	344
D Basic information theory	345
D.1 Communication channel and information gain	345
D.2 Entropy, the average information gain	348
D.2.1 Difficulties in measuring entropy	349
D.2.2 Entropy of a spike train with temporal coding	350
D.2.3 Entropy of a rate code	351
D.3 Mutual information and channel capacity	353
D.4 Information and sparseness in the inferior-temporal cortex	354
D.4.1 Population information	354
D.4.2 Sparseness of object representations	356
D.5 Surprise	357
Further reading	359
E A brief introduction to MATLAB	361
E.1 The MATLAB programming environment	361
E.1.1 Starting a MATLAB session	362
E.1.2 Basic variables in MATLAB	363
E.1.3 Control flow and conditional operations	366
E.1.4 Creating MATLAB programs	369
E.1.5 Graphics	370
E.2 A first project: modelling the world	371
E.3 Octave	373
E.4 Scilab	375
Further reading	377

This page intentionally left blank

Introduction

1

This introductory chapter outlines the major focus, specific tools, and strategies of computational neuroscience. We argue that theoretical and computational studies are important in order to comprehend and guide experimental investigation. The nature and role of models in research is discussed. A model is an abstraction of the real systems used to investigate specific questions, and this chapter outlines the guiding principles for models used throughout the book. In addition, a high-level theory of how the brain works is included in this chapter to guide some discussions in the following chapters. The last section of this chapter includes a short guide to the book, outlining the path we are taking while embarking on a fascinating scientific venture.

1.1 What is computational neuroscience?

The scientific area now commonly called computational neuroscience uses distinct techniques and asks specific questions aimed at advancing our understanding of the nervous system. A brief definition might be:

Computational neuroscience is the theoretical study of the brain used to uncover the principles and mechanisms that guide the development, organization, information-processing and mental abilities of the nervous system.

Thus, computational neuroscience is a specialization within neuroscience. Neuroscience itself is a scientific area with many different aspects. Its aim, and therefore that of computational neuroscience, is to understand the nervous system, in particular the part concentrated in the skull, that we call the brain. The brain is studied by researchers who belong to diverse disciplines, including physiology, psychology, medicine, computer science and mathematics, to name but a few. Neuroscience emerged from the realization that interdisciplinary studies are vital to further our understanding of the brain. While huge progress in our understanding of brain functions has been made in the past few decades, there are many questions that we can only tackle by combined efforts: How does the brain work? What are the biological mechanisms involved? How is it organized? What are the information-processing principles used to solve complex tasks such as perception? How did it evolve? How does it change during the lifetime of organisms? What is the effect of damage to particular areas and the possibilities of rehabilitation? What are the origins of degenerative diseases and possible treatments? These are questions asked by neuroscientists in many different sub-fields, using a multitude of different research techniques.

1.1 What is computational neuroscience?	1
1.2 What is a model?	5
1.3 Is there a brain theory?	8
1.4 A computational theory of the brain	13
Exercises	16
Further reading	17

1.1.1 Tools and specializations in neuroscience

Many techniques are employed within neuroscience to answer those important questions about brain operations. These include genetic manipulations; recording of cell activities in cultured cells, brain slices, or even whole brains; optical imaging; non-invasive functional brain imaging; psychophysical measurements; and computational simulations. Each of these techniques is complicated and laborious enough to justify a specialization of neuroscientists in particular techniques. Therefore, we speak of neurophysiologists, cognitive scientists, and anatomists. It is, however, vital for any neuroscientist to develop a basic knowledge of all major techniques, so he or she can comprehend and utilize the contributions made in other specializations. The significance of any technique has to be evaluated in view of its specific problems and limitations, as well as the specific aim of the technique. Computational neuroscience is one of the areas in neuroscience with increasing importance, and a basic comprehension of this field is becoming essential for all neuroscientists.

Computational and theoretical neuroscience develops and tests hypotheses of the functional mechanisms of the brain. A major focus in this area is therefore the development and evaluation of *models*. Computational neuroscience can be viewed as a specialization within theoretical neuroscience that employs computers to simulate models. The major reason for using computers is that the complexity of models in this area is often beyond analytical tractability. For such models we have to employ carefully designed numerical experiments to be able to compare the models to experimental data. However, we do not want to restrict our studies to this tool, because analytical studies can often give us a much deeper and more controlled insight into the features of models and the reasons behind numerical findings. Whenever possible, we try to include analytical as well as numerical techniques, and we will use the term computational neuroscience synonymously with theoretical neuroscience. The term 'computational' also stresses that we are interested, in particular, in the computational and information-processing aspects of brain functions.

Although computational neuroscience is theoretical by its very nature, it is important to bear in mind that models must be gauged on experimental data; they are otherwise useless for understanding the brain. Only experimental measurements of the real brain can verify 'what' the brain actually does. In contrast to the experimental domain, computational neuroscience tries to speculate 'how' the brain operates. Such speculations are developed into hypotheses, realized into models, evaluated analytically or numerically, and tested against experimental data. Also, models can often be used to make further predictions about the underlying phenomena.

Ideally, we want to integrate experimental facts from different levels of investigation into a coherent model of how the brain works. The interaction between different disciplines, computational studies and experiments, is illustrated in Fig. 1.1. Important experimental input to computational neuroscience comes from neuroanatomy, such as the morphology and functional connectivity of brain structures; from neurophysiology, in which the behaviour of single neurons is investigated; and from psychology, in which behavioural effects are studied with psychophysical experiments. Computational neuroscientists

use mathematical models for the description of experimental facts, borrowing methods from a wide variety of disciplines, such as mathematics, physics, computer science and statistics. Investigations of hypotheses with the aid of models leads to specific predictions that have to be verified experimentally. The comparison of model predictions with experimental data can then be used to refine the hypotheses and to develop more accurate models, or even models that can shed light on different phenomena. The studies within computational neuroscience can also help to develop applications such as advanced analysis of brain-imaging data, technical applications that utilize brain-like computations, and ultimately, better treatment of patients with brain damage and other brain-related disorders.

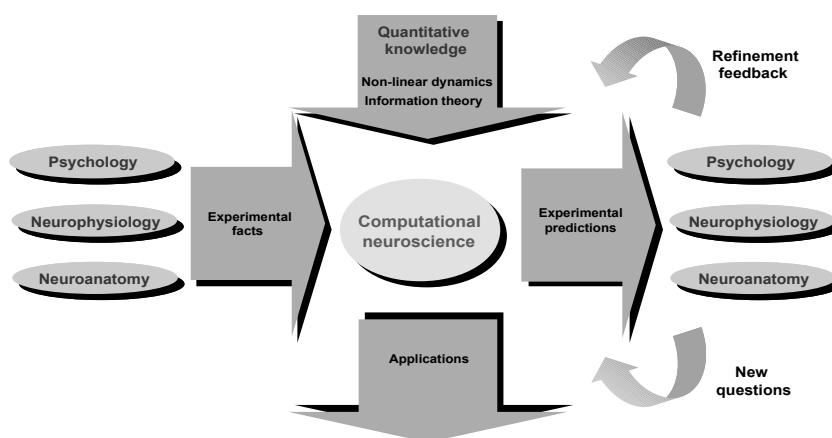


Fig. 1.1 Illustration of the role of computational neuroscience in the integration of experimental facts from different levels of investigation. The models developed in computational neuroscience make predictions that can be verified experimentally. The close comparison of experiments with model predictions can then be used to make refinements to the models (or may even lead to the development of new approaches) that can further our understanding of brain systems and can make new predictions that have to be verified experimentally. Also, such research frequently leads to new applications, such as advanced analysis techniques or new patient treatment techniques. [Adapted from Gustavo Deco, personal communication.]

1.1.2 Levels of organization in the brain

The nervous system has many levels of organization on spatial scales ranging from the molecular level of a few Angstrom ($1\text{\AA} = 10^{-10}\text{m}$), to the whole nervous system on the scale of over a metre. Biological mechanisms at all these levels are important for the brain to function. Different levels of organization in the nervous system are illustrated in Fig. 1.2. An important structure in the nervous system is the *neuron*, which is a cell that is specialized for signal processing. Depending on external conditions, neurons are able to generate electric potentials that are used to transmit information to other cells to which they are connected. Mechanisms on a subcellular level are important for such information-processing capabilities. Neurons use cascades of biochemical reactions that have to be understood on a molecular level. These include, for example, the transcription of genetic information which influences information-processing in the nervous system. Many structures within neurons can be identified with specific functions. For example, *mitochondria* are structures important for the energy supply in the cell, and *synapses* mediate information transmission between cells. The complexity of a single neuron,

and even isolated subcellular mechanisms, makes computational studies essential for the development and verification of hypotheses. It is possible today to simulate morphologically reconstructed neurons in great detail, and there has been much progress in understanding important mechanisms on this level.

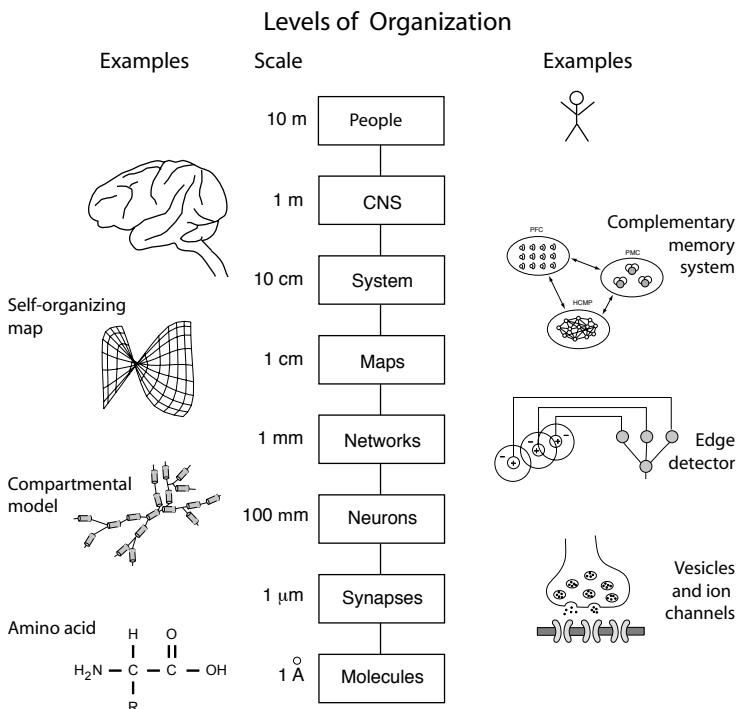


Fig. 1.2 Some levels of organization in the central nervous system on different scales [adapted from Churchland and Sejnowski, *The computational brain*, MIT Press (1992)].

However, single neurons certainly do not tell the whole story. Neurons contact each other and thereby compose *networks*. A small number of interconnected neurons can exhibit complex behaviour and enable information-processing capabilities not present in a single neuron. Understanding networks of interacting neurons is a major domain in computational neuroscience. Networks have additional information-processing capabilities beyond that of single neurons, such as representing information in a distributed way. An example of a basic network is the *edge detector* formed from a centre-surround neuron as proposed by *Hubel and Wiesel*. The illustrated levels above the one labelled ‘Networks’ in Fig. 1.2 are also composed of networks, yet with increasing size and complexity. An example on the level termed ‘Maps’ in Fig. 1.2 is a *self-organizing topographic map*, which is part of an important discussion in this book.

The organization does not stop at the map level. Networks with a specific architecture and specialized information-processing capabilities are composed into larger structures that are able to perform even more complex information-processing tasks. System-level models are important in understanding higher-order brain functions. The central nervous system depends strongly on the

dynamic interaction of many specialized subsystems, and the interaction of the brain with the environment. Indeed, we will see later that active environmental interactions are essential for brain development and function.

Although an individual researcher typically specializes in mechanisms of a certain scale, it is important for all neuroscientists to develop a basic understanding of the functionalities of different scales in the brain. Computational neuroscience can help the investigations at all levels of description, and it is not surprising that computational neuroscientists investigate different types of models at different levels of description. Computational methods have long contributed to cellular neuroscience, and computational cognitive neuroscience is now a rapidly emerging field. The contributions of computational neuroscience are, in particular, important to understand non-linear interactions of subprocesses. Furthermore, it is important to comprehend the interactions between different levels of description, and computational methods have proven very useful in bridging the gap between physiological measurements and behavioural correlates.

1.2 What is a model?

Modelling is an integral part of many scientific disciplines, and neuroscience is no exception. The more complex a system is, the more we have to make simplifications and build example systems to provide insights into aspects of the complex system under investigation. The term ‘model’ appears frequently in many scientific papers, and describes a vast variety of constructs. Some papers present a single formula as a model, some papers fill several pages with computer code, and some describe with words a hypothetical system. We need to understand what a model is, and, in particular, what the purposes of models are.

It is important to distinguish a model from a hypothesis or a theory. Scientists develop hypotheses of the underlying mechanisms of a system that have to be tested against reality. In order to test a specific feature of a hypothesis, we build a model that can be evaluated. Sometimes we try to mimic real systems under artificial means, in order to be able to test the systems by different conditions, or to make measurements that would not be possible in a ‘real’ system. A model is hence a simplification of a system in order to test particular aspects of the system or hypothesis. A brief explanation of a model, which is useful to remember throughout this book and in research, is:

A model is an abstraction of a real-world system to demonstrate particular features of, or investigate specific questions about, the system. Or, in more scientific terms, a model is a quantification of a hypothesis to investigate the hypothesis.

A good example of this is the use of models in the field of architecture. Small-scale paper models of buildings, or computer graphics (Fig. 1.3) generated with sophisticated three-dimensional graphics packages, can be used to get a first impression of the physical appearance and aesthetic composition of a design. A model has a particular purpose attached to it and has to be viewed in this light.



Fig. 1.3 A computer model of a building which gives a three-dimensional impression of the design.

A model is not a recreation of the ‘real’ thing. The paper model of a house cannot be used to test the stability of the construction, a purpose for which a building engineer uses different models. In such models, it is important to scale down physical properties of the building materials regardless of the physical appearance, such as the colour of the building.

1.2.1 Phenomenological and explanatory models

In science, we typically represent experimental data in the form of graphs, and then seek to describe these data points with mathematical functions (Fig. 1.4). An example of this is the ‘modelling’ of response properties (tuning curves) of neurons in the *lateral geniculate nucleus* (LGN), which can be fitted with a specific class of functions called *Gabor functions* by adjusting the parameters of these functions. Gabor functions are therefore said to be ‘models’ of the receptive fields in the LGN. Of course, this *phenomenological model* does not tell us anything about the biophysical mechanisms underlying the formation of receptive fields and why cells respond in this particular way, so such a ‘model’ seems rather limited. Nevertheless, it can be useful to have a functional description of the response properties of LGN cells. Such parametric models are a shorthand description of experimental data that can be used in various ways. For example, if we want to study a model of the primary visual cortex, to which these cells project, then it is much easier to use the parametric form of LGN responses as input to the cortical model, rather than including further complicated models of the earlier visual pathway in detail.

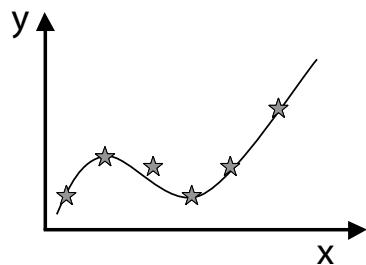


Fig. 1.4 The graph shows some data points, plotted with star symbols, such as data derived from experimental measurements. Also shown is a model represented by the line, and this curve fits the data reasonable well. The curve can be derived from a simple mathematical formula that fits the data points (phenomenological model), or result from more detailed models of the underlying system.

As scientists, we want to find the roots of natural phenomena. The explanations we are seeking are usually deeper than merely parameterizing experimental data with specific functions. Most of the models in this book are intended to capture processes that are thought of as being the basis of the information-processing capabilities of the brain. This includes models of single neurons, networks of neurons, and specific architectures capturing brain organizations. Models are used to study specific aspects of a hypothesis or theory, but can also help to interpret experimental data.

A major role of models in science is to illustrate principles underlying natural phenomena on a conceptual level. Sometimes the level of simplification and abstraction is so crude that scientist talk about *toy models*, although this terminology undermines slightly the importance of such models in science. The simplifications made in such models might be necessary to employ analytical methods to analyse such models at a depth that is not possible in more realistic models. The educational importance of these toy models should not be underestimated, in particular, in demonstrating principal mechanisms of natural phenomena. It is easy to complicate things, but the real scientific challenge is to simplify theoretical concepts.

The current state of neuroscience, often still exploratory in nature, frequently makes it difficult to find the right level of abstraction to properly investigate hypotheses. Some models in computational neuroscience have certainly been too abstract to justify claims derived from them. On the other hand, there is a great danger in keeping too many details that are not essential for the scientific argument. Models are intended to simplify experimental data, and thereby to

identify which details of the biology are essential to explain particular aspects of a system. Modelling the brain is not a contest in recreating the brain in all its details on a computer. It is questionable how much more we would comprehend the functionality of the brain with such complex and detailed models. Models must be carefully constructed, and reasonable simplification are the result of careful scientific investigations and insight into natural processes. It is not always possible to justify all assumptions adequately, but it is important to at least clarify which assumptions have been made and to argue about them. The purpose of models is to better comprehend the functionality of complex systems, and simplicity should be a major guide in designing appropriate models. This philosophy is sometimes called the *principle of parsimony*, also known as *Occam's razor*. Basically, we want the model as simple as possible, while still capturing the main aspects of data that the model should capture. We will see this principle in action throughout this book, and discuss the issue specifically in conjunction with machine learning in Chapter 6.

1.2.2 Models in computational neuroscience

Where do we start when trying to explain brain functions? Do we have to re-build the brain in its entirety with computational techniques in order to understand it? Currently, many neuroscientists are collaborating in an ambitious project, called the *Blue Brain Project*, to implement in a computer, with as much detail as possible, neocortical circuits. This type of detailed project can show us which emergent phenomena can be expected in the brain, or where gaps in our knowledge require further investigation. However, even if it was be possible to simulate a whole brain on a computer, with all the details from biochemistry to large-scale organization, this would not explicitly mean that we have better explanations of brain functions. What we are looking for, at least in this book, is a better comprehension of brain mechanisms on explanatory levels. It is therefore important to learn about the art of *abstraction*, making suitable simplifications to a system without abolishing the important features we want to comprehend.

The level that is most appropriate for the investigation and explanatory abstraction depends on the scientific question. For example, epileptic seizures are known to be caused by synchronization of whole brain areas, and an understanding of brain dynamics on a systems level is important to comprehend such disorders. This should, of course, not exclude the search for causes on much smaller scales. We know that Parkinson's disease is caused by the death of dopaminergic neurons in the *substantia nigra*, and the role of dopamine in the initiation of motor actions is important to comprehend the full scale of impairments, and to develop better methods of coping with such conditions. The causes of the cell death of dopaminergic neurons are still not known, and it is important to find the reasons, possibly on a genetic level in this case, to enable a real treatment of Parkinson's disease. Therefore, various levels must be investigated, and we have to learn to make connections between the different levels in order to follow how the low-level circumstances, such as mechanisms on a genetic level or biochemical processes in neurons, can influence the characteristics of large-scale systems, such as behaviour of an organism.

1.3 Is there a brain theory?

It is often said that the brain is the most complex system that we know of in nature, and understanding how it works is a large, if not impossible, task. It is true that understanding how the brain works seems difficult when trying to reverse engineer it. However, reverse engineering a system can even be difficult for systems that we created in the first place. For example, imagine measuring the varying states of a transistor in a computer while the computer is running a word processor. Such measurements give important clues from which it should be possible to discover regularities when certain operations are repeated on the computer. Also, such measurements make it possible to discover important principles that must be at work in the digital computer system, such as the discrete nature of information representations. However, it seems a daunting task to recreate a computer program such as a word processor from these data, even if we were able to analyse a large number of measurements of many parts of the computer at the same time. The direct reverse engineering of the brain from data seems even more challenging, given the biological nature of the object. However, we can use the data to understand the principles of brain processing, and we can then use this knowledge to build brain-style information-processing systems. This approach is taken this book.

Correspondingly, there has been some shift in the research approach of the neuroscience community. The past decades have been an era in neuroscience marked by a flood of explorations. Recordings with micro-electrodes from single cells contributed significantly to this exploration, and searching for response properties of neurons is at least as exciting as it must have been for explorers such as Marco Polo to discover new lands. New brain-imaging techniques, such as functional magnetic resonance imaging (fMRI), make it possible to monitor living brains of subjects performing specific mental tasks. Explorations of brain functions with such techniques have been essential in advancing our knowledge in neuroscience. However, while a mountain of data has been gathered for brain functions, using a multitude of techniques, we are now slowly entering a new phase in neuroscience, that of formulating more quantitative hypotheses of brain functions. This shift in the focus of neuroscience research demands some more specific experimental analysis and more dedicated tests of such hypothesis. It is increasingly important to formulate a quantitative hypotheses, and possible alternatives, in such a way that the hypothesis can be tested experimentally.

This new era of neuroscience sounds a lot like quantitative scientific areas such as chemistry or physics, and while a quantitative analysis will generate new breakthroughs in our understanding of brain functions, the ultimate question is if there can be a *brain theory*. We could take the position that in order to understand the brain we need to understand all of the structural details and the current state of a particular brain. This is no different in other scientific areas such as physics. For example, to completely describe the physics of an individual aeroplane we have to know the precise location and form of each nut and bolt and all other structural details up to the amount of dirt on the wings and the details of the air it is flying through. Another example is that of a pot of boiling water, which consists of a lot of individual molecules in a very

dynamic state. Measuring all the microscopic details in the last two examples seems impossible. Yet, we have a fairly good understanding of the process of boiling water and why an aeroplane can fly. This was not possible overnight but is the result of dedicated scientific research over the past few centuries. Important for the success in physics, chemistry, and other scientific disciplines, was the realization of the right level of description or the right level of abstraction of a problem. For example, we learned to describe the average behaviour of the molecules in an ideal gas, which gave us the fundamentals of thermodynamics and ultimately led to a better understanding of the mechanisms of flowing air that enables engineers to construct more efficient aeroplane. We know today about the essential quantum nature of atomic and subatomic interactions, but a description of Mount Everest on this level is not reasonable. There are geological theories of mountain formation that are more appropriate than employing quantum theory to these questions.

There is no reason to conjecture that the brain cannot be tractable with similar scientific rigour to that developed in other disciplines. The brain is certainly more complex than a gas of weakly interacting atoms. However, there are very fundamental questions that we can attack; for example, how a network stores memories that can be recalled in an associative way. Indeed, we have made considerable progress with this question, considering our understanding of associative networks discussed in this book. Another fundamental question is why the brain is relatively stable, while still being able to adapt to novel environments. Brain theories of this kind are now emerging, and some of these theories will be discussed in this book. It may be too early to talk about a single brain theory, but there is no reason to suspect that theories will advance our understanding of brain functions. Indeed, I think that major breakthroughs have already been made, which need to be brought to a wider audience. The goal of this book is to contribute to this endeavour.

1.3.1 Emergence and adaptation

Standard computers, such as PCs and workstations, have one or more central processors. Each processor is rather complex, with specialized hardware and microprograms implementing a variety of functions, such as loading data into registers, adding, multiplying and comparing data, as well as communicating with external devices. These basic functions can be executed by instructions, which are binary data loaded into a special interpreter module. Complicated data processing can be achieved by writing, often lengthy, programs, which are instructions representing a collection of the basic processor functions. When solving a task with a computer, we have to instruct the machine to follow precisely all the steps that we determined beforehand would solve a particular problem. The sophistication of computers basically reflects the ingenuity of the programmer.

In contrast, information-processing in the brain is very different in several respects. The brain employs simpler processing elements than computers, but lots of them. To explore information processing in networks of neurons, we will mainly use very simple abstractions of real neurons, which we call *nodes* to stress this drastic simplification. These fundamental processing units can be

implemented in hardware, or simulated on a standard computer; for the discussions in this book this does not make a difference. We keep the functionality of nodes as simple as possible for the sake of employing lots of them, typically hundreds, thousands, or even more. The usage of many parallel working processors has motivated the term *parallel distributed processing* in this area. However, with this term one is tempted to think that the processes are independent because only processes that are independent can be processed on different processors in parallel. In contrast to this, a major ingredient of information-processing in the brain is the *interaction of neurons*, and the interaction of neurons is accomplished by assembling them into large networks.

It is the interaction of nodes that enables processing abilities not present in single nodes. Such capabilities are good examples of emergent properties in rule-based systems. *Emergence* is the single most defining property of neural computation, distinguishing it from parallel computing in classical computer science, which is mainly designed to speed up processing by distributing independent algorithmic threads. Interacting systems can have unique properties beyond the mere multiplication of single processor capabilities. It is these types of abilities we want to explore and utilize with neural networks. These system properties are labelled as emergent to stress that we did not encode these properties directly into the system. To better appreciate this, we distinguish the description of a system on two levels, the level of basic rules defining the system, and the level of description aimed at understanding the consequences of such rules. In the study of neural networks we are interested in understanding the consequences of interacting nodes.

Scientific explanations were dominated in the past by the formulation of a set of principles and rules that govern a system. A system can be defined by a set of rules, like in a game. In science, we assume that natural systems are governed by a finite set of rules. The search for these basic rules, or fundamental laws as they are called in this case, was the central scientific quest for centuries. It is not easy to determine such laws, but enormous progress has been made nevertheless. Newton's laws defining classical mechanics, or the Maxwell equations of electromagnetism are beautiful examples of fundamental laws in nature. We do not have a theory of the brain on this level, and some have argued that we might never find a simple set of rules explaining brain functions. However, in many scientific disciplines we are beginning to realize that even with a given set of rules we might still not have a sufficient understanding of the underlying systems. This is analogous to the idea that knowing the rules of the card game Bridge is not sufficient to be a good Bridge player.

Rules define a system completely, and it can therefore be argued that all the properties of a system are encoded in the rules. However, we have to realize that even a small set of rules can generate a multitude of behaviours of the systems, which might be difficult to understand from the basic rules due to the presence of emergent properties. A different level of description might then be more appropriate. For example, thermodynamics can describe appropriately the macroscopic behaviour of systems of many weakly interacting particles, even though the systems are governed by other microscopic rules. On the other hand, there are emergent properties in Newtonian systems that are not well described by classical thermodynamics, such as turbulent fluids. A deeper

understanding of emergent properties is becoming a central topic in the science of the 21st century.

The importance of emergent properties in networks of simple processing elements is not the only extension of traditional information-processing approaches that we think are crucial for understanding brain functions. Another important ingredient is that the brain is an example of an *adaptive system*. In our context, we define adaptation as the ability of a system to adjust its response to external stimuli in dependence of the environment states and the expectations of the system. Humans are a good example of systems that have mastered adaptive abilities and learning. Adaptation and learning is an area that has attracted a lot of interest in the engineering community. The reason for this is that learning systems, which are systems that are able to change their behaviour based on examples in order to solve information-processing demands, have the potential to solve problems where traditional algorithmic methods have not been able to produce sufficient results. Adaptation has two major virtues. One is, as just mentioned, the promise to solve information-processing problems for which explicit algorithms are not yet known. A second virtue is our aim to build systems that can cope with continuously changing environments. A lot of research in the area of neural networks is dedicated to the understanding of learning. Engineering applications of neural networks are not bound by biological plausibility. In contrast to this, we concentrate in this book on biologically plausible learning mechanisms that can help us to comprehend the functionality of the brain.

1.3.2 Levels of analysis

David Marr and *Tomaso Poggio* realized the usefulness of distinguishing between different levels of description when explaining processes, which Marr later explains beautifully in his posthumously published book *Vision*. One issue that Marr raised in his book was that different people need different kinds of explanations. Explanations of brain functions that are satisfactory for a non-specialist can be insufficient for a physiologist or psychologist who is trying to make sense of specific observations in his or her experiments. A computer engineer may require different types of explanations of brain functions again, to be able to implement specific information-processing algorithms. Marr also stressed the difference between understanding computers and understanding computation. Knowing how bits are represented and transformed in a computer is far from understanding high-level applications such as the World Wide Web.

Besides these important considerations for any explanatory theory, Marr made an important distinction between different levels of analysis. These are summarized in Marr's book as:

- (1) **Computational theory:** *What is the goal of the computation, why is it appropriate, and what is the logic of the strategy by which it can be carried out?*
- (2) **Representation and algorithm:** *How can this computational theory be implemented? In particular, what is the representation for the input and output, and what is the algorithm for the transformation?*

(3) **Hardware implementation:** *How can the representation and algorithm be realized physically?*

The most abstract and general level, that of a computational theory, is concerned with what a process is trying to achieve, and what the principal approach to the solution is. In his book, Marr discusses the example of a cash register, which is adding up the price of goods brought to the checkout counter. This explanation answers the question of what the cash register is doing, but we need also to ask why the cash register is doing it (adding the numbers) in this specific way. The answer to this question is that the rules of adding numbers, rather than multiplying them, encapsulates what we think is appropriate for this process; that prices of individual goods should just add up, independent, for example, of the order in which the goods are presented. Marr assigns great importance to this level and explains in his book:

'To phrase the matter in another way, an algorithm is likely to be understood more readily by understanding the nature of the problem being solved than by examining the mechanism (and hardware) in which it is embodied.'

Having a clear theory of what the brain, or specific brain processes, are trying to accomplish can be a powerful research guide and needs to be considered more in neuroscientific research.

The next level of description is concerned with how the computation, specified on the computational level, is realized on an algorithmic level. Marr considered brain research as an information-processing problem, and he clearly was aware of the duality between representing and processing information. That is, in order to process information, this information has first to be represented in a specific form. For example, numbers can be represented in binary form or with Roman numbers. Representations are important since the specific algorithm used for implementing a process usually depends on the representation. Many different representations are possible, and there are usually many algorithms for each type of representation which can accomplish the task. Different representations can drastically influence algorithms, and different algorithms can have quite different properties, such as robustness or efficiency. Theories in computational neuroscience have strongly focused on representations, on which specific algorithms are built. It should also be mentioned that there are examples that use the arguments in the opposite direction, such as deriving representations from constraints on algorithms, such as robustness. Clearly, the brain must use specific representations, and specific algorithms, and it is the goal of computational neuroscience to help find them.¹

¹In the definition of this representational and algorithmic level, Marr speaks specifically of input and output representations, and the process of transforming, or mapping, input to output. We will soon discuss why this mapping view is not adequate for brain functions. In a later section of his book, Marr discusses the difficulty in finding invariant features from changing data. This is an example that is difficult in a mapping framework but which can be resolved easily in the computational theory of the brain presented below. But this criticism should not distract from the importance of this level of explanation.

The third level of Marr's scheme is concerned with the physical realization of the algorithms and representation. This is the level to which neuronal biophysics and physiology can speak most directly. Again, there might be several possible physical realizations of a specific algorithm, and we want to understand the physical realizations in the brain. This level of explanation does not only back up the level of explanations above, but is also important in its own right for specific interventions. The different levels are weakly related, and inspiration and constraints from different levels of analysis can guide research on other levels.

Since this book is intended to be an introductory guide, we follow mainly a bottom-up approach, through learning first about some physical properties of the nervous system, how information is represented in the brain, and some algorithmic theories of solving specific problems. A broad knowledge of such issues is necessary for any advanced work in neuroscience. However, as Marr suggested, it is important to guide research more specifically through computational theories. We therefore outline in the following section a broad computational theory of the brain to which we return in more detail in the final chapter.

1.4 A computational theory of the brain

1.4.1 Why do we have brains?

One of the first questions we might want to ask is why we have a brain at all. While this question has likely been asked and answered in many different ways, by many people, the ideas of *Daniel Wolpert* are particularly revealing. Wolpert studies sensorimotor control; how brains are able to supervise appropriate movements in a complex sensory world. He notes that plants live stable lives without a brain and he also tells the story of a little sea creature, called the sea squirt, which is hatched with a small nervous system and swims through the ocean until it settles down on some rock, after which it digests its brain. So, although there might be some bias from his sensimotor perspective, it seems that animals need a brain to move.

If we accept this answer for now, we can ask why creatures with a brain want to move around. We can answer this question by stating that the creature might want to move around to find food and sexual partners to enable survival of its species and to make evolutionary progress. Even without these concrete suggestions, we can think about some objective the creature wants to achieve. Mathematically, we view this as maximizing an objective function. With this view, we can imagine that nervous systems developed into more and more sophisticated systems that help to achieve the evolutionary objectives of moving organisms. The human brain is thereby, arguably, the most developed example of such evolved systems, even inventing machines to travel. Thus, a more formal answer to what the brain is doing might be:

The brain produces goal-directed behaviour to maximize our probability of survival.

If individual brains exist to maximize the organism's survival capacity, then why could it happen that a teenager gets drunk and falls off a cliff? The reason is that maximizing survival probability acts on a population, not on an individual level. The brain implements a specific strategy for each individual, which itself may not be perfect. Furthermore, experimenting with the unknown is essential for finding new, and ultimately better, solutions, so that the adventurous behaviour of a teenager can ultimately help our society.

1.4.2 The anticipating brain

The next question to ask is how the brain achieves the ability to produce specific goal-directed behaviour. Or, in Marr's view of a computational theory, what is the brain specifically doing and why in its specific way. We could again follow the sensorimotor view and state that the ability to produce goal directed movements enables a creature to react to specific environmental situations. The brain helps in this task by analysing sensory representations of the environment and mapping them to appropriate motor actions. It is possible to imagine that very complex mappings can be implemented in neural tissue. Indeed, we argue in Chapter 6 that feedforward networks are universal approximators, meaning that any functional relation between an input vector, representing sensory states, and an output vector, representing motor actions, can be learned by such networks.

The study of sensory–motor mappings has been important in neuroscientific investigations. However, rather than concentrating on fairly direct response functions, our aim in this book is to understand sophisticated human actions enabled by the brain, and ultimately the entire human mind. The central thesis outlined here is that sophisticated human thought processes can not be achieved by simple feedforward mapping systems, but require sophisticated feedback systems.

To illustrate the importance of feedback systems further, let us take a concrete example. Imagine that we go into a concert hall to listen to a symphony. Let us consider a brief visual experience when entering the concert hall. There are chairs which certainly look different to chairs at home, though people usually have no problem recognizing them as chairs, even in a very brief moment. One can do this even with very brief image presentations. *Simon Thorpe* has shown that visual stimuli of very short durations, only lasting a few tens of milliseconds, are enough to enable the categorization of some of the content of visual stimuli. How is this possible?

Visual scene analysis is quite challenging from a information-processing perspective. The human eye contains around 130 million photoreceptors, so that a visual snapshot on the retina would have 15 MB (megabytes) of data if we only consider binary responses of each photoreceptor. While this is roughly the magnitude of modern digital cameras, it is puzzling that we can not achieve any comparable scene analysis with modern computer systems, despite the fact that computers are more than a million times faster (a computer can easily compute several billion instructions per second, or one instruction in less than a nanosecond (ns), which is very fast compared to the millisecond (ms) time scale of a neuron). The often-suggested answer, that the brain is a large parallel computer, is not sufficient, since only a small number of steps can be processed in the brain in a reasonable time between sensation and perception/response. For example, a common guide for neural processing time is around 10 ms per synaptic stage, such that only 10 to 100 processing steps can be assumed for a task in which humans form percepts or start reacting to stimuli. The problem is that we don't know an algorithm that can work with so few steps. Even basic steps in traditional image processing, such as segmenting an image to separate objects from the background, can take thousands of computational steps.

How, then, can the fast perception of the brain be achieved? Our thesis here is that fast perception can only be achieved through foreknowledge, and that the human brain functions as a predictive or *anticipatory memory system*. In short, the idea is that we already have a rich knowledge of our commonly encountered environment, and that we can extrapolate this knowledge quickly to individual circumstances. Thus, the brief version of our computational theory can be outlined as:

The brain is an anticipating memory system that learns to represent expectations of the world, which can be used to generate goal-directed behaviour.

People know about chairs, and possibly even concert halls, so analysis of the visual scene, when entering the concert hall, becomes tractable. We will see that central to this thesis is some form of top-down processing, on top of bottom-up processing for acquiring environmental evidence. Simple bottom-up systems, as described by feedforward networks, also called *perceptrons*, are insufficient for human-like information-processing. The brain is not like most machines that simply react predictably to external commands, it is a more sophisticated system which can interact with the environment. For this we need top-down processing and an active engagement of the system with the environment, which also make brains dependent on their history and current states.

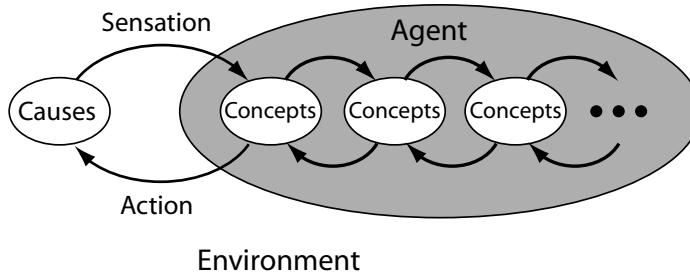


Fig. 1.5 Basic outline of a high-level model of the brain describing a fundamental hypothesis of brain processing as an anticipatory, hierarchical memory system.

An outline of some important components of such an anticipating system, which we will formalize more fully in Chapter 10, is shown in Fig. 1.5. It is important that the organism acts in an environment through sensations and actions. *Causes* in the environment, such as objects or sequences of events, cause specific sensations to the organism. These sensations activate learned representations in a hierarchical organization within the organism. While the first level of processing is aimed at closely matching peripheral sensations, each additional level is trained to represent increasingly abstract concepts learned from the environment. Such a layered representation is sometimes called a *deep representation*. Concepts are learned, and self-organize, through actions that the organism takes in the environment.

While concepts on different levels can change over time, activations of higher-level concepts will also influence expectations of activating specific lower-level concepts. The anticipating system tries to match sensations through activating appropriate concepts on different hierarchical levels. This anticipation of

sensations via top-down influences is crucial for the operational capabilities of organisms. Anticipation solves, for example, the time constraints in perception mentioned above, since not all information has to be processed bottom-up to activate the hypothesis of which objects are in the environment. It reduces the complexity of discovering invariant features in the environment, since such knowledge can be learned during development and is part of the expectations of specific sensations.

Also, it is important that organisms act in their environments. For example, objects can be touched, and rotated, to verify whether concepts activated by initial sensations conform with the additional information. This active hypothesis testing can dramatically reduce the learning problem by focusing on relevant data. Organisms can actively explore the environment, which is crucial in forming concepts in the brain. Consequences of the *action–perception loop*, that is, the importance of actions for perception, are only beginning to be explored by the computational research community.

Finally, it is appropriate to introduce and use probabilistic descriptions when talking about many brain processes. We will see that this is simply convenient, on the one hand, but is also required, for the following reasons. Many processes in nature, including brain processes, are inherently noisy. This is reflected by the fact that the outcomes of repeated measurements are always fluctuating. There is considerable debate about the origin of noise in nature, whether noise is a fundamental property, an *irreducible indeterminacy*, or if noise is only based on *epistemic limitations*, such as limitations in the measurement process or the limitations of controlling ‘hidden variables’. However, regardless of this philosophical debate, the introduction of the mathematical construct of *random variables*, and the corresponding formal system of *probability theory*, was a major advancement in science. Clearly, the human brain cannot control or accurately determine all the factors in its environment. Thus, epistemic limitations alone are reason enough for the brain to process information probabilistically. Random variables and probability theory are therefore not only a useful description of noisy brain processing, but the fundamental objects described by these concepts may well be fundamental quantities considered by the brain.

Exercises

- (1.1) Give three examples of models and briefly describe what makes them models.
- (1.2) Give four examples of neuroscientific issues at different levels of analysis, as illustrated in Fig. 1.2.
- (1.3) What is the relationship between theoretical and experimental studies in neuroscience?
- (1.4) Make a model of an aeroplane and explain which features are modelled with your solution.

Further reading

One of the first books that discussed a wide array of computational issues of brain functions was Churchland and Sejnowski (1992). The most advanced book in this area is Dayan and Abbott (2001), which goes beyond the level of the discussions in this book and contains important additional topics, as well as more rigorous mathematical treatment of some of the topics mentioned in this book. While the book by Dayan and Abbott is essential reading for practitioners in computational neuroscience, we also recommend some more general books for a broader audience, which capture important aspects in brain science, and can help to keep the larger picture in mind. Among those books are, foremost, the book by Jeff Hawkins (2004), which beautifully explains the computational theory advocated in this book. The recent book by Norman Doidge (2007) beautifully describes the amazing consequences of brain plasticity. Glimcher (2003) contains a wonderful description of the history of the mind–body philosophy, a great description of physiological studies, as well as some great discussion of applying game theory to neuroscience. Finally, there

is a good collection of articles on Scholarpedia (Izhikevich, 2009), which are a good initial reading to explore important topics in neuroscience.

Patricia S. Churchland and Terrence J. Sejnowski (1992), *The Computational Brain*, MIT Press.

Peter Dayan and Laurence F. Abbott (2001), *Theoretical Neuroscience*, MIT Press.

Jeff Hawkins with Sandra Blakeslee (2004), *On Intelligence*, Henry Holt and Company.

Norman Doidge (2007), *The Brain that Changes Itself: Stories of Personal Triumph from the Frontiers of Brain Science*, James H. Silberman Books.

Paul W. Glimcher (2003), *Decisions, Uncertainty, and the Brain: The Science of Neuroeconomics*, Bradford Books.

Eugene M. Izhikevich (2006), Scholarpedia, <http://www.scholarpedia.org>.

This page intentionally left blank

Part I

Basic neurons

This page intentionally left blank

Neurons and conductance-based models

2

Neurons are specialized cells that enable specific information-processing mechanisms in the brain. This chapter summarizes the basic functionality of neurons and outlines some of their fascinating biochemical processes. These include an outline of chemical synapses, the parametrization of the response of the postsynaptic membrane potential to synaptic events, the origin of action potentials and how they can be modelled using elegant differential equations introduced by Hodgkin and Huxley, generalizations of such conductance-based models, and compartmental models that incorporate the physical structure of neurons. The review in this chapter is intended to justify simplifications of following models and to mention some of the neuronal characteristics that may be relevant in future research.

2.1 Biological background	21
2.2 Basic synaptic mechanisms and dendritic processing	26
2.3 The generation of action potentials: Hodgkin–Huxley equations	33
2.4 Including neuronal morphologies: compartmental models	46
Exercises	50
Further reading	50

2.1 Biological background

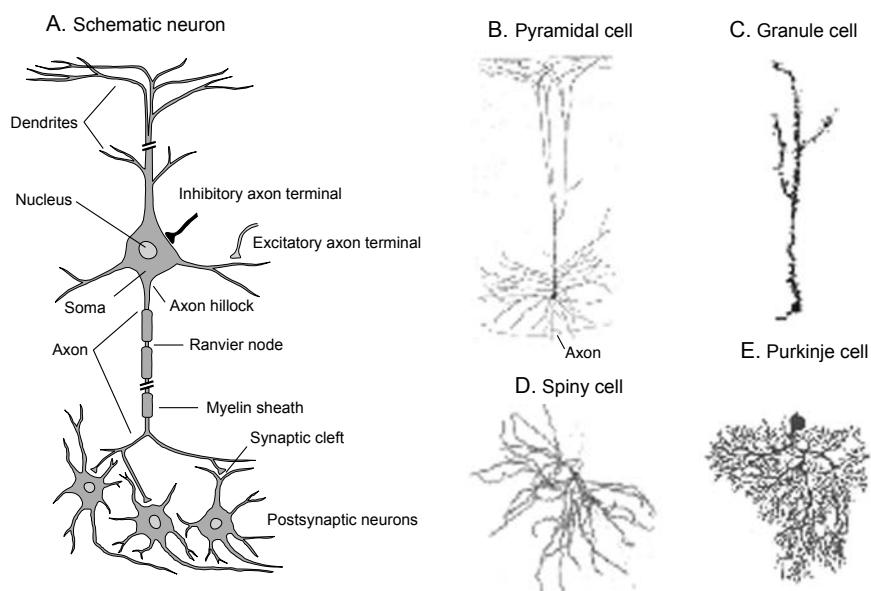
The central nervous system is primarily made up of two principal cell types, *neurons* and *glial* cells. Glial cells far outnumber neurons, and while glial cells have been seen as mainly providing supporting functions, there is increasing evidence of many more active processing functions of different types of glial cells. For example, the study of sophisticated processing mechanisms through neuron–glia interactions and signalling through glial networks is increasing. Glial cells can also regulate synaptic plasticity and possibly blood flow determined by synaptic activity. However, this chapter will concentrate on the information processing provided by neuronal networks. In order to describe the networks of neuron-like elements, we first have to gain some insight into the working of single neurons. Important issues include mechanisms of information transmission within single neurons and between neurons, and how the biologically complex neurons can be modelled sufficiently to answer specific scientific questions. The following review only mentions some of the sophisticated computational abilities of neurons, which some readers may want to study in more detail using the literature outlined at the end of this chapter. The review outlines briefly some of the computational approaches used to describe single neurons with biophysical and morphological details. The study of the biophysical machinery in single neurons is an active area within computational neuroscience. While most of the remainder of the book is based on strongly simplified models to investigate network properties, the main purpose of the discussion in this chapter is to gain an appreciation of the complexity in single neuron processes.

Neurons are biological cells and therefore have common features of other cells, including a *cell body* (also called a *soma*), a *nucleus* containing *DNA*, *ribosomes* assembling proteins from genetic instructions, and *mitochondria* providing energy for the cell. In contrast to other cells, neurons specialize in signal processing utilizing special electrophysical and chemical processes. Much progress has been made in unveiling some of those complex processes. Only some basic processes are reviewed here.

2.1.1 Structural properties

There are many different types of neurons, with differences in size, shape, or physiological properties. However, there are many general features of neurons, which are summarized here by describing a generic neuron, as illustrated in Fig. 2.1A, while outlining where variations are common. The major structural features are a *cell body* (or *soma*) and root-like extensions called *neurites*. The neurites are further distinguished into the receiving fibres of neurons, called *dendrites* from the Greek word *dendron* meaning tree, and one major outgoing trunk called an *axon*, which is the Greek word for axis.

Fig. 2.1 (A) Schematic neuron that is similar in appearance to pyramidal cells in the neocortex. The components outlined in the drawing are typical for most major neuron types [adapted from Kandel, Schwartz, and Jessell, *Principles of neural science*, McGraw-Hill, 4th edition (2000)]. (B–E) Examples of morphologies of different neurons. (B) Pyramidal cell from the motor cortex [from Cajal, *Histologie du système nerveux de l'homme et des vertèbres*, Maloine, Paris (1911)], (C) Granule neuron from the olfactory bulb of a mouse [from Greer, *J. Comp. Neurol.* 257: 442–52 (1987)], (D) Spiny neuron from the caudate nucleus [from Kitai, in *GABA and the basal ganglia*, Chiara and Gessa (eds), Raven Press (1981)], (E) Golgi-stained Purkinje cell from the cerebellum [from Bradley and Berry, *Brain Res.* 109: 133–51 (1976)].



The shape of neurons can vary considerably, and morphological criteria have been used as a scheme for classifying neurons. Some examples of shapes of neurons are illustrated in Fig. 2.1B–E. The most abundant neocortical neurons are categorized into *pyramidal neurons* and *stellate neurons*. Pyramidal neurons, of which an example is shown in Fig. 2.1B, are by far the most abundant neuronal type in the neocortex, accounting for as much as 75–90% of the neurons therein. They are characterized by a pyramid-shaped cell body with an axon usually extending from the base of the pyramidal cell body (*basal base*). The

dendrites of the pyramidal cell can be divided into a far-reaching dendritic tree extending from the apical tip of the cell body into the upper layer of the cortex (discussed further in Chapter 5), and the basal dendrites with more local organizations. These neurons form contacts with other neurons with asymmetrical appearances and are thought to be excitatory.

The two other classes of neuronal types that are common in the neocortex have star-like appearances and are therefore called stellate neurons. Dendrites of *spiny stellate neurons* (as well as dendrites of pyramidal cells) are covered with many boutons, called *spines*. Typically, spiny stellate neurons have an excitatory effect on other cells. In contrast, the *smooth stellate neurons* are not covered with many spines and form inhibitory connections with other neurons that have a symmetrical appearance.

It should be clear that the division of neocortical neurons into pyramidal and stellate neurons is only a rough categorization of neuronal types in the neocortex. Other divisions can be made, although the distinction between different classes is often not easy to make on their spatial appearance alone. Some neurons that have been given other names include *Martinotti cells*, which are found mainly in the deeper layers of the neocortex and grow axons that extend commonly into the outer layer; *basket cells*, which are smooth stellate cells that synapse preferentially on the cell body of pyramidal cells; and *chandelier cells*, which synapse preferentially on the initial segment of pyramidal axons. Structural properties have computational consequences that must be explored. Later in this chapter we outline how cell morphologies can be incorporated in specific simulations of single neurons.

2.1.2 Information-processing mechanisms

Neurons can receive signals from many other neurons, called *efferents*, typically on the order of 10,000 neurons. Some neurons have many fewer incoming fibres such as motor neurons in the brainstem, and there are examples of cells receiving many more inputs, such as pyramidal cells in the hippocampus, which have been estimated to receive on the order of 50,000 inputs. The sending neurons contact the receiving neuron at specialized sites, either at the cell body or the dendrites. The English physiologist *Charles Sherrington* named these sites *synapses*. Various mechanisms are utilized to transfer information between the *presynaptic* neuron (the neuron sending the signal) and the *postsynaptic* neuron (the neuron receiving the signal). The general information-processing feature of synapses is that they enable signals from a presynaptic neuron to alter the state of a postsynaptic neuron, which eventually triggers the generation of an electric pulse in the postsynaptic neuron. This electric pulse, the important *action potential*, is usually initiated at the root of the axon, the *axon-hillock*, and subsequently travels along the axon. Every neuron has one axon leaving the soma, although the axon can branch and send information to different regions in the brain. For example, some neurons spread information to neurons in neighbouring areas of the same neural structure with axon branches called *axon collaterals*, while other axons can reach through *white matter* to other brain areas. White matter contains no neuron bodies, only axons and *oligodendrocytes*, special glial cells that provide the myelin sheath for axons as

described later. At the receiving site the axons commonly split into several fine branches, so-called *arbors* (Latin for tree), with *axon terminals* or *axon boutons* (French for buttons) forming part of the synapses to other neurons or muscles.

2.1.3 Membrane potential

The ability of a neuron to vary its intrinsic electric potential, the *membrane potential*, is important for the signalling capabilities of a neuron. The membrane potential, which we shall denote by V_m , is defined as the difference between the electric potential within a cell and its surroundings. The inside of a cell is neutral when positively charged ions, called *cations*, are bound to negatively charged ions, called *anions*. The origin of this potential difference comes from a different concentration of ions within and outside a cell. For example, the concentration of potassium (K^+) ions is around twenty times larger within a cell compared to the surrounding fluid. This concentration difference encourages the outflow of potassium by a diffusion process as long as the membrane is permeable to this ion. The neuronal membrane is, however, not permeable to the anions that typically bind to potassium, so the diffusion process leaves an excess of negative charge inside a neuron and an excess of positive charge in the surrounding fluid of the neuron.

The increasing electrical force resulting from the increased difference in the membrane potential will eventually be strong enough to match the force generated by the concentration difference of potassium within and outside of the cell. Hence, the electrical force will balance the diffusion process, and the neuron settles at an equilibrium state. The resulting potential is formally described by the *Nernst equation* which relates the work that is necessary to compensate for the diffusion gradient to the logarithm of the ratio of the concentrations, the absolute temperature, and some system-specific parameters. Calculating the equilibrium potential for potassium channels in a typical neuron results in a value around $E_K = -80$ mV, which is called the *reversal* potential for the channel. Only a very small percentage of K^+ ions have to leave the neuron in order cause the membrane potential. Similar processes for other ions, in particular sodium (Na^+) which has a concentration in the surroundings of a neuron exceeding the concentration within the neuron, eventually lead to the *resting potential* of a neuron, which is typically around $V_{rest} = -65$ mV.

Of course, it is probably more exciting and important to study how the membrane potential can be altered to transmit information. The membrane potential can be altered by a variety of mechanisms for which different neurons are specialized. For example, sensory neurons are specialized in converting external stimuli into electrical signals; examples include tactile neurons, which are sensitive to pressure, and photoreceptor neurons in the retina, which are sensitive to light. The information transmission between neurons, with which we are mainly concerned in this book, is achieved in a variety of ways. For example, *electrical synapses* or *gap-junctions* consist of special conducting proteins that allow a direct electromagnetic signal transfer between two neurons. However, the most common type of connection between neurons in the central nervous system is the *chemical synapse*, which will be described below.

2.1.4 Ion channels

The permeability of cell membranes to certain ions is achieved via *ion channels*. Ion channels are special types of proteins embedded in the membranes of cells (see Fig. 2.2). These proteins form pores that enable specific ions to enter or to leave cells. Common ions involved in such processes within the nervous system are sodium (Na^+), potassium (K^+), calcium (Ca^{2+}), and chloride (Cl^-). The ion channels that drive the resting potential of neurons are usually open all the time (see Fig. 2.2A). It is hence appropriate to call them *leakage channels*. We will shortly meet other types of ion channels that can open and close under various conditions.

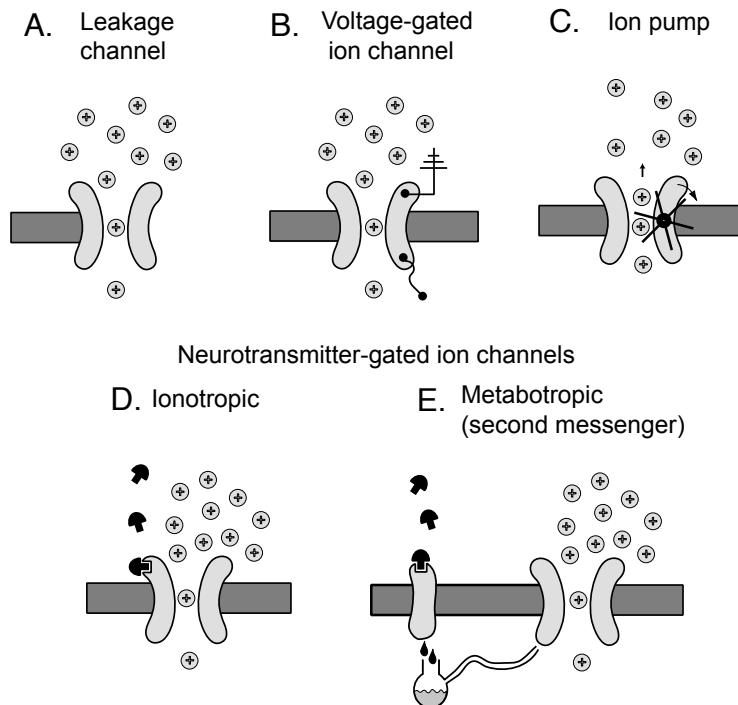


Fig. 2.2 Schematic illustrations of different types of ion channels. (A) Leakage channels are always open. (B) The opening of voltage-gated ion channels depends on the membrane potential. This is indicated by a little wire inside the neuron and a grounding wire outside the neuron. Such ion channels can, in addition, be neurotransmitter-gated (not shown in this figure). (C) Ion pumps are ion channels that transport ions against the concentration gradients. (D) An ionotropic neurotransmitter-gated ion channel opens when a neurotransmitter molecule binds to the channel protein, which, in turn, changes the shape of the channel protein so that specific ions can pass through. (E) A metabotropic synapse influences ion channels only through secondary messengers.

Information transmission within and between cells is largely based on ion channels. The following section describes these mechanisms in more detail, starting with a discussion of chemical synapses and followed by a discussion of the generation of action potentials in Section 2.3. In Section 2.4 we put the pieces together and describe the major functionality of neurons, using a compartmental model. The remainder of the chapter contains a brief discussion of extensions of the basic mechanisms, while the Chapter 3 concentrates on simplifications that enable us to study the behaviour of large networks of neurons.

2.2 Basic synaptic mechanisms and dendritic processing

2.2.1 Chemical synapses and neurotransmitters

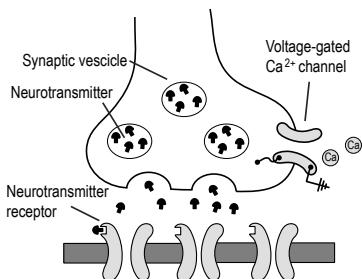


Fig. 2.3 Schematic illustration of chemical synapses.



Fig. 2.4 Electron microscope photo of a synaptic terminal.

Chemical synapses, schematically outlined in Fig. 2.3 (an electron micrograph of a synaptic terminal is shown in Fig. 2.4), are the workhorses of neuronal information transfer in the mammalian nervous system. They consist of a specialized extension on an axon, the *axon terminal*, and specific receiving sites on dendrites. Axon terminals of chemical synapses have the ability to synthesize special chemicals called *neurotransmitters*, concentrated and stored in *synaptic vesicles*. The arrival of an action potential triggers the probabilistic release of neurotransmitters. Released neurotransmitters drift across the *synaptic cleft*, a small gap of only a few micrometres ($1\mu\text{m} = 10^{-6}\text{m}$) between the axon terminal and the dendrite of the postsynaptic neuron.

Many different neurotransmitters have been identified in the nervous system. Common neurotransmitters in the central nervous system include small organic molecules such as glutamate (Glu) or gamma-aminobutyric acid (GABA). Dopamine (DA) is another neurotransmitter that has attracted a lot of attention for its role in motivation, attention, and learning. Synaptic transmission at neuromuscular junctions, important for initiating muscle movements, is primarily mediated by acetylcholine (ACh), and sensory nerves utilize a variety of other neurotransmitters.

Neurotransmitter-gated ion channels are special types of ion channels in postsynaptic dendrites that open and close under the regulation of neurotransmitters. Without the presence of the neurotransmitter the size of the pore is too small to let ions flow through it. These ion channels open when neurotransmitters bind to specific receiving sites of the ion channels, after which ions of the right size and shape can flow through them, changing the membrane potential of the cell in turn. The response in the membrane potential is called the *postsynaptic potential* (PSP). A larger quantity of neurotransmitter released by the presynaptic neuron can trigger a stronger response in the postsynaptic membrane potential, but this relationship does not have to be linear; twice the amount of neurotransmitter or twice the number of synapses involved in the overall synaptic event does not necessarily increase the membrane potential twofold.

The effects of neurotransmitters depend on the receptor type, so that the same neurotransmitter can have different effects in postsynaptic neurons. Receptors that are tightly coupled to an ion channel are called *ionotropic* and are illustrated in Fig. 2.2D. Such neurotransmitter-gated ion channels typically open rapidly after binding neurotransmitters. In contrast, *metabotropic* synapses, stylized in Fig. 2.2E, only influence ion channels through secondary messengers and are thus much slower and less specific.

2.2.2 Excitatory and inhibitory synapses

Different types of neurotransmitters and their associated ion channels have different effects on the state of receiving neurons. An important class of neuro-

transmitters opens channels that will allow positively charged ions to enter the cell. These neurotransmitters therefore trigger the increase of the membrane potential that drives the postsynaptic neurons toward their excited state. The process (and sometimes simply the synapse) is therefore said to be *excitatory*. Synaptic channels gated by the neurotransmitter Glu are a very common example of such excitatory synapses with different types of receptors. One fast ionotropic Glu receptor is called AMPAR because it can be activated with the synthetic chemical α -amino-3-hydroxy-5-methylisoxazole-4- propionic acid. Another Glu receptor is called NMDAR because it can be activated by the synthetic chemical *N*-methyl-D-aspartate; it is much slower and voltage dependent. The voltage dependence comes from the fact that these channels are blocked by magnesium ions (Mg^{2+}) in the neuron's resting state, even when Glu is bound. The Mg^{2+} ions can only be removed through depolarization of the membrane potential such as through backpropagating action potentials (BAPs) explained below. Only after the removal of magnesium can sodium and calcium, the two major ions that can pass through this channel, enter the neuron. In Chapter 4 we discuss the importance of these mechanisms for synaptic plasticity.

In contrast to excitatory synapses, some neurotransmitter-gated ion channels can drive the postsynaptic potential towards the resting potential or inhibit the effect of excitatory synapses. Such synapses are collectively called *inhibitory*. A prominent example of inhibitory synapses uses the neurotransmitter GABA with a fast receptor called $GABA_A$ and a slower receptor called $GABA_B$. The neurotransmitter DA has several receptor types, some of which are excitatory and some of which are inhibitory. Inhibition can be *subtractive* in the sense that it lowers the membrane potential. However, inhibition can also be *divisive* in the sense that it modulates the effect of excitation. For example, $GABA_A$ receptors have no effect on the membrane potential if the membrane potential is at rest, so it does not reduce the potential further. Inhibitory synapses close to the cell body can have such modulatory (multiplicative) effects on summed excitatory postsynaptic potentials (EPSPs). Such a form of inhibition is also called *shunting inhibition*.

2.2.3 Modelling synaptic responses

As mentioned above, the variation of a membrane potential initiated from a presynaptic spike can be described as a *postsynaptic potential* (PSP). Excitatory synapses increase a membrane potential in what is called the *excitatory postsynaptic potential* (EPSP). In contrast, *inhibitory postsynaptic potentials* (IPSPs) are responses that lower the membrane potential. The form of IPSPs can often be modelled similar to EPSPs but with negative currents. We describe here only a generic implementation of ion-channel mediated PSP.

The experimentally measured time course of a PSP following a presynaptic event, in particular for fast excitatory AMPA receptors and inhibitory GABA receptors, can be parameterized by the function

$$\Delta V_m^{\text{non-NMDA}} = At e^{-t/t^{\text{peak}}}, \quad (2.1)$$

where ΔV_m denotes the difference between the membrane potential and the resting potential, A is an amplitude parameter, and t^{peak} is the time for which

this function reaches its maximal value. Synaptic responses of the above types are typically fast, with peak times around 0.5–2ms. The above function is sometimes called the *alpha-function* in neuroscience. In this function we did not include the typical synaptic delay of the PSP after the firing of a presynaptic neuron. This delay is caused by the time it takes for a neurotransmitter to be released in an axon terminal, the duration of the diffusion process of the neurotransmitter, and the time necessary to open the channels. All of these processes are typically fast (often less than 1 ms) and can be added directly in network simulations.¹

¹While we consider here synaptic delays as negligible, we will argue in Chapter 5 that latencies between presynaptic and postsynaptic events, termed axonal delays, can be large and variable.

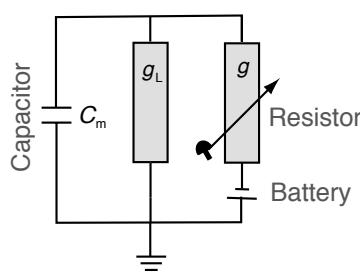
The alpha-function can be implemented with a basic model often used to describe chemical synapses. We thereby treat a segment of dendrite as a simple *compartment* that is a leaky capacitor (a capacitor with a constant resistor) together with a neurotransmitter-gated resistor, as illustrated in Fig. 2.5A. Each resistor is also supplied with a battery representing the concentration difference of ions within and outside the cell (Nernst potentials). The combined effects of the different components can easily be expressed by taking the conservation of electric charge into account. This is formalized in *Kirchhoff's law*,

$$c_m \frac{dV_m(t)}{dt} = -I, \quad (2.2)$$

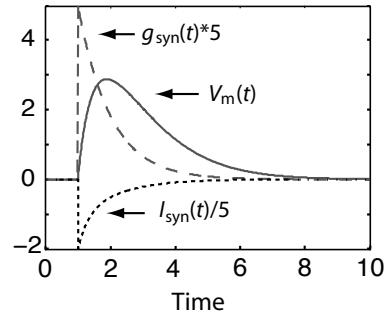
where c_m is the capacitance of the dendritic compartment and the minus sign on the right-hand side is a convention in neuroscience where currents are measured as flowing from outside the cell to inside the cell.

Fig. 2.5 (A) Electric circuit of a simple dendritic compartment to simulate a PSP with capacitance c_m , conductance g_L of the leakage channel, time dependent conductance g of the transmitter-gated ion channel and battery describing the equilibrium potential of the transmitter-gated ion channel. There is no battery for the leakage channel as the resting potential of circuit is set to zero. (B) Results for the numerical integration of the dendritic compartment model. Shown are time courses of the conductance $g(t)$, the synaptic current $I_{syn}(t)$, and the membrane potential $V(t)$. The conductance and current are scaled for better visibility, and the units correspond to the numerical values used in the simulations discussed below.

A. Electric circuit of basic synapse



B. Time course of variables



Eqn 2.2 is a linear differential equation. A differential equation describes the infinitesimal change of a quantity (the voltage in this example) with time, and this change is given by a function on the right-hand side. Solving a differential means to derive the time course of this quantity,

$$\frac{dx(t)}{dt} = f(x, t, \dots) \Rightarrow x(t) = ? \quad (2.3)$$

We will mainly solve these equations numerically with simulation programs. Numerical integration in this circumstance simply means to sum all the changes,

as given by the function on the right-hand side, from an initial state. Models are often formulated with differential equations, and is therefore recommended to study the brief introduction to differential equations given in Appendix B in case the reader is not so familiar with these concepts.

In the synaptic model, the current I is produced by two channels, a leakage channel with conductance (inverse of resistance) g_L and a reversal potential of zero (the resting potential of this compartment as we measure everything relative to this voltage), and the neurotransmitter-gated synaptic ion channel with time-varying conductance $g_{\text{syn}}(t)$ and reversal potential E_{syn} . The relation between the electric potential, the current, and the conductance is given by *Ohm's law* ($I = gV$),

$$I(t) = g_L V(t) - g_{\text{syn}}(t)(V_m(t) - E_{\text{syn}}). \quad (2.4)$$

The time course of the time-varying synaptic conductance is modelled further with a simple model of the average channel dynamics. In this model we assume that many channels open immediately after binding neurotransmitters, but close stochastically like radioactively decaying material. This open-and-decay process is modelled as

$$\tau_{\text{syn}} \frac{dg_{\text{syn}}(t)}{dt} = -g_{\text{syn}}(t) + \delta(t - t_{\text{pre}} - t_{\text{delay}}), \quad (2.5)$$

where the delta function $\delta()$ is a convenient way to express a term that has an infinitely large contribution in an infinitely small time interval, when its argument is zero. The contribution to the sum (or integral) is finite (see Appendix A.3). In the case above, a contribution is made at time t_{delay} after a presynaptic spike that occurred at t_{pre} . At other times, this equation results simply in an exponential decay with time constant τ_{syn} (see Appendix B).

Below we discuss a MATLAB program that implements the equations in this section and solves them numerically. The result of this numerical integration after a neurotransmitter binding at time $t = 1$ is shown in Fig. 2.5B. The time evolution of $g(t)$ is shown with a dashed line and is precisely an exponential decay. The opening of the channel produces a synaptic current I_{syn} through the neurotransmitter-gated ion channel (dotted line), where the amount does not only depend on the conductance of the ion channel, but also on the voltage of the membrane relative to the reversal potential of this channel. Thus, if the synaptic channel stays open, then the system comes to an equilibrium where the synaptic current matches the leakage current, and the voltage stabilizes at an intermediate value between zero and the reversal potential. However, the neurotransmitter-gated channels also close rapidly so that the synaptic compartment eventually decays to its resting state.

The above model and its interpretation is a crude approximation of the synaptic processes. In reality, there is a complicated interplay between different ion channels and even subunits of receptors. The parameters $c_m, g_0, E_{\text{syn}}, \tau_{\text{syn}}$ have therefore no direct relations to specific ion channels and can be chosen arbitrarily to fit experimental data. Dendritic processing is further enriched by several other mechanisms, such as interdendritic calcium waves and BAPs related to active spiking mechanisms, as discussed in the next section. Glial cells can also influence synapses, for example by regulating vesicle release through

calcium signalling. The above generic synaptic model does not attempt to explain the detailed workings of synapses, but rather, should be seen as a phenomenological model intended to parameterize experimental findings for convenience. This model is, however, already sufficient to explore many features of spiking neurons and networks and is sufficient for most of the discussions in this book. If necessary, more details of the underlying biochemical and physical mechanisms can be incorporated into the model, but, as we have stressed in Chapter 1, our aim here is not to recreate all details of the brain, but rather to extract essential mechanisms that explain brain functions.

Simulation

A short introduction to programming in MATLAB is given in Appendix E, and some discussion on numerical integration techniques is included in Appendix B. The simulations in Fig. 2.5B were done with the program listed in Table 2.1, which is included as file `EPSP.m` in folder `Chapter2`. This program is not only a demonstration of a basic conductance-based model, but is also the first example of a simulation program based on numerical integration. The principle algorithm implemented in this program is to solve an ordinary differential equation of the form eqn 2.6 with a discrete iterative process,

$$x(t + \Delta t) = x(t) + \Delta t f(x, t, \dots), \quad (2.6)$$

which calculates the value of the variable at the next time index (after one time step) from the value at the current time index plus the time step multiplied by the change as specified in the function f . Since it is the first major simulation program in this book, we describe this program line-by-line in the following.

In MATLAB, every text that follows the percentage sign, `%`, is a comment and will not be interpreted by the MATLAB interpreter. When using the double percentages, `%%`, the text until the next double percentage will be highlighted as block in the MATLAB editor, which can help with navigations in the code. We typically start the program with a comment that briefly states the purpose of the program in Line 1. Also, we frequently use Line 2 to clear the workspace and figures, and to switch on the hold mode when plotting several lines into one figure. Empty lines, such as Line 3, are a further aid to represent the code more clearly.

The specifics for this simulation program start at Line 4 with a block of definitions. In Line 5, we set some values for constants used in the program. This includes the capacitance of the membrane, `c_m`, the conductance of the leakage channel, `g_L`, the time constant which sets the time scale of the opening (or better closing) of the synaptic ion channel, `tau_syn`, the reversal potential of the synaptic ion channel, `E_syn`. The last parameter is the integration time step, `delta_t`, which sets the (time) resolution of the simulation. Note that all the parameters are numbers and do not have units. A numerical simulation program can not handle physical units and we have to make sure that the values are in the appropriate units. For example, we can interpret the time scale in terms of milliseconds (ms), so that the integration time step is $\Delta t = 0.01$ ms. The conductance of the leakage channel could be 1 Siemens (S), which is the inverse of the resistance measured in Ohms. A more typical value would be

Table 2.1 Program EPSP.m

```

1 %% Synaptic conductance model to simulate an EPSP
2 clear; clf; hold on;
3
4 %% Setting some constants and initial values
5 c_m=1; g_L=1; tau_syn=1; E_syn=10; delta_t=0.01;
6 g_syn(1)=0; I_syn(1)=0; v_m(1)=0; t(1)=0;
7
8 %% Numerical integration using Euler scheme
9 for i=2:10/delta_t
10    t(i)=t(i-1)+delta_t;
11    if abs(t(i)-1)<0.001; g_syn(i-1)=1; end
12    g_syn(i)= g_syn(i-1) - delta_t/tau_syn * g_syn(i-1);
13    I_syn(i)= g_syn(i) * (v_m(i-1)-E_syn);
14    v_m(i) = v_m(i-1) - delta_t/c_m * g_L* v_m(i-1) ...
15        - delta_t/c_m * I_syn(i);
16 end
17
18 %% Plotting results
19 plot(t,v_m); plot(t,g_syn*5,'r--'); plot(t,I_syn/5,'k:');

```

that the combined leakage channels in a square centimeter (cm^2) of membrane would be around some miliSiemens (mS). The conductance of other channels have then to be specified in the same units, and the other quantities have to be chosen consistently.

In Line 6, we set initial values for the variables that will change with time. These quantities include the conductance of the synaptic ion channel, $g_{\text{syn}}(1)$, which is set to zero so that it is closed at the beginning of the simulation. Consequently, the corresponding current, $I_{\text{syn}}(1)$, is also zero. While we calculate the current from the conductance and membrane potential later, we include the value explicitly here for the first time index, $i=1$, since we start the iteration below at time index $i=2$ (the index after one time step). We also start with a zero membrane potential at the first time index, $v_m(1)=0$, and set the clock for the first time index to zero, $t(1)=0$.

The numerical integration of the differential equations, which specify the model, start at Line 9. The numerical integration is done using the Euler method (see Appendix B), which simply adds the changes to the initial states for each integration step. The counter for the time step (index) is given by the variable i . The loop specifies that it starts with the value of 2 and is incremented by 1 until the index is equal to $10/0.01=1000$. This time corresponds to 10 ms, or more precisely to 9.99 ms since we set the time for the first time index to zero. The actual time is updated in each step in Line 10 and kept in the vector t .

The next two lines implement eqn 2.5. Line 11 simulates the opening of the

synaptic ion channel after binding of neurotransmitters at time $t = 1$ ms by setting the conductance at the previous time step to one, which is represented mathematically as the delta function in eqn 2.5. We override the value of the previous time step because this value is used in the next line to calculate the actual value at this current time step. The if statement might have an unexpected form. One might have suspected a statement like `if t(i) == 1;`. The problem with this statement is that the left-hand side and the right-hand side have to be exactly the same, bit for bit. For example, `1.000000000001` is not equal to 1. To make sure that the if statement catches the time in the right vicinity, we ask instead that the absolute difference between `t(i)` and 1 should be less than a small value.

Line 12 is the simulation of the dynamics of the ion channel without the delta function part in eqn 2.5. The initial value of this conductance is zero, and it will stay like this until we force it to open at $t = 1$ ms in Line 11. After this it will decay since the code specifies that the new value of the conductance will be ‘one minus a little bit’ of the value at the previous time. The reduction of this value is proportional to the value itself, which is the characteristics of an exponential decay. This can be seen in the solution, the dashed line in Fig. 2.5B. As argued above, we can think of this dynamics as the average number of open channels when the channels close randomly like in a radioactive decay. This simulation does thus not include the precise dynamics of individual ion channels, although we could build such models.

An open synaptic channel will result in a synaptic current when the membrane potential is different from the reversal potential of this synaptic channel. This current is calculated with Ohm’s law in Line 13. This current itself will change the membrane potential. The change of the membrane potential is given by Kirchhoff’s law (the differential equation eqn 2.2), where the current is now made up of the synaptic current and the leakage current. The leakage current is written in Line 14 and the synaptic current in Line 15, and this example also demonstrates how program statements can be extended to more than one line.

The loop over the integration steps ends at Line 16, so that the results can be plotted with the instructions on Line 19.

2.2.4 Non-linear superposition of PSPs

Electrical potentials have the physical property that they superimpose as the sum of the individual potentials. Many of the models discussed in the literature incorporate linear superposition of synaptic input, meaning that inputs are simply summed up in neuronal models.

However, it is important to keep in mind that the membrane potential of a neuron depends on several other factors, such as the physical shape of the dendrites, the voltage dependence of some channels, and other interactions between synaptic ion channels. Non-linear interactions between synaptic inputs can add greatly to the information-processing capabilities of neural networks. We will discuss some examples in this book, but this area is still largely unexplored.

In this section we will only give an impression of the complex dendritic mechanisms that are still largely unexplored. Many more issues are under active research, including intercellular calcium signalling and interactions of

synapses with genetic information. Most of the current computational models on a network level have instead concentrated on much more stereotyped spiking mechanisms that we describe next. However, the large possibility of sophisticated dendritic computation has led some people, like Carver Mead, to speculate that dendritic computation might lead to the building of truly intelligent systems.

2.3 The generation of action potentials: Hodgkin–Huxley equations

As discussed in the previous section, the dendritic membrane potential of a neuron can be altered by the release of neurotransmitters from a presynaptic neuron that opens specific synaptic ion channels and allows ions to leave or enter the neuron. This section describes how the change in the membrane potential can ultimately trigger the generation of an action potential, often simply called a spike, in compartments of the neuron with different voltage-sensitive ion channels. These mechanisms are traditionally implicated in axonal processing, although it is increasingly being realized that active spike generation also happens in dendrites. These BAPs are crucial in at least some forms of plasticity, as discussed in Chapter 4, but might have many more roles in dendritic information processing.

A prototypical form of action potential, as measured in the giant axon of a squid by Alan Hodgkin and Andrew Huxley, is shown in Fig. 2.6. It is characterized by a sharp increase (*depolarization*) of the membrane potential to positive values, followed by a sharp decrease in the membrane potential, undershooting (*hyperpolarizing*) the resting potential of the neuron before returning to resting potential.

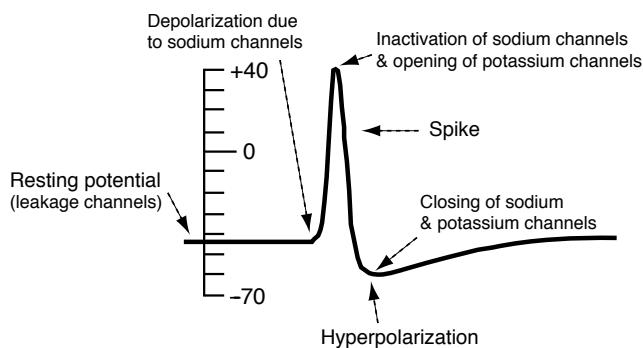


Fig. 2.6 Typical form of an action potential; redrawn from an oscilloscope picture from Hodgkin and Huxley (1939).

It was a major scientific accomplishment by Hodgkin and Huxley when they quantitatively described the form of action potentials with equations summarized in this section. It is not only the beauty of the description that made them famous, but also the fact that they quantified the process leading to the generation of action potentials with mathematical terms that were later identified with ion channels. Indeed, their model made specific predictions that

could be verified experimentally and by so doing guided further research and many discoveries.

2.3.1 The minimal mechanisms

A least two types of voltage-dependent ion channels and one type of static ion channel are necessary for the generation of a spike, as illustrated in Fig. 2.7. In contrast to the neurotransmitter-gated ion channels discussed in the previous section, voltage-dependent ion channels open and close as a function of the voltage of the membrane, as symbolized in Fig. 2.2B. A voltage-dependent sodium channel is responsible for the rising phase of action potentials. When neurotransmitter-gated ion channels depolarize neurons sufficiently, voltage-dependent sodium (Na^+) channels open, leading to an influx of Na^+ due to the negative potentials and the lower concentration of Na^+ within the cell. The domination of the sodium channel shifts the membrane potential close to the sodium resting potential, which is around $V_m = +65 \text{ mV}$.

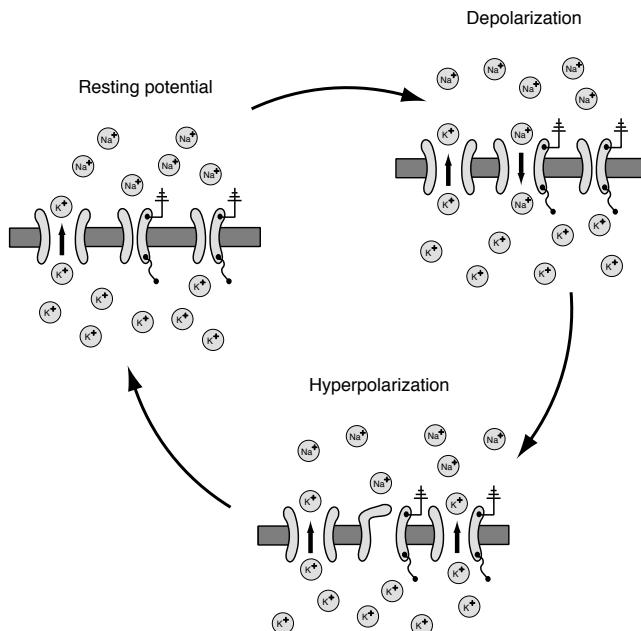


Fig. 2.7 Schematic illustration of the minimal mechanisms necessary for the generation of a spike. The resting potential of a cell is maintained by leakage channels through which potassium ions can flow as a result of concentration differences between the inside of the cell and the surrounding fluid. Voltage-gated sodium channels allow the influx of positively charged sodium ions and thereby the depolarization of the cell. After a short time sodium channels are blocked and voltage-gated potassium channels open. This results in hyperpolarization of the cell. Finally, the hyperpolarization causes the inactivation of voltage-gated channels and a return to the resting potential.

Two contributing processes cause the subsequent falling phase. First, sodium channels inactivate due to blockade of the channels by a part of the protein making up the channels, around 1 ms after the opening of the channels. Second, another type of voltage-dependent channels, potassium (K^+) channels, open, leading to an efflux of K^+ . In contrast to voltage-dependent sodium channels, which open nearly instantaneously after crossing the threshold, voltage-dependent potassium channels open about 1 ms after the initial depolarization of the action potential. The dominance of potassium channels drives the membrane potential close to the potassium equilibrium of around $V_m = -80 \text{ mV}$,

undershooting the normal resting potential of the neuron. This hyperpolarization of neurons relative to the resting potential, V_{rest} , of neurons causes the voltage-dependent potassium channels to close and the voltage-dependent sodium channels to deactivate, so that the normal resting potential, V_{rest} , of neurons is eventually recovered.

2.3.2 Ion pumps

Repeated generation of action potentials results in a repeated efflux of K^+ and influx of Na^+ . If a neuron repeated the process, as described so far, many times, it would decrease the potassium concentration and increase the sodium concentration within the cell repeatedly, eventually leading to the failure to generate action potentials. Another type of ion channel, not included in Fig. 2.7, is therefore vital for a cell. These specific ion channels are called *ion pumps* and are illustrated schematically in Fig. 2.2C. Ion pumps can transfer specific ions against their concentration level between the inside and outside of neurons. The price to pay is the large amount of energy necessary for this process. Ion pumps are estimated to account for about 70% of the total metabolic consumption of neurons, or about 15% of the total energy consumption in humans, while the brain accounts for one-fifth of our total energy consumption.

2.3.3 Hodgkin–Huxley equations

Hodgkin and Huxley quantified the process of spike generation with a set of four coupled differential equations, formalizing their measured findings of the giant axon of a squid. The net movement of ions across the membrane is a current that we label with the symbol I_{ion} . The number and permeability of open ion channels determine the electric conductance (the inverse of the resistance), which we label with \bar{g}_{ion} . The bar over the g is used here to indicate that this value can change and is therefore a function of several factors. In the following we express the membrane potential relative to the resting potential and label it with V . The relation between this electric potential, the current, and the conductance is given by Ohm's law,

$$I_{\text{ion}} = \bar{g}_{\text{ion}}(V - E_{\text{ion}}), \quad (2.7)$$

where E_{ion} is the equilibrium potential for the channel when taking the ion concentrations within and outside the cell into account. This corresponds to a channel-specific battery, as shown in Fig. 2.8. The equilibrium potential in the above formula is also expressed relative to the resting potential of the neuron.

As discussed above, a major component of the general framework for generating an action potential is the fact that the K^+ and Na^+ channels are voltage dependent. Hodgkin and Huxley introduced empirically three dynamic variables, n , m , and h , to describe this voltage dependence and the dynamics of the channels. The variable n describes the activation of the (then unknown) potassium channels, m describes the activation of the sodium channels, and h describes the inactivation of the sodium channels. The voltage and time dependence of the conductances is then described in the model by modulating constant conductance values, g_{ion} (without bar), with the voltage and time

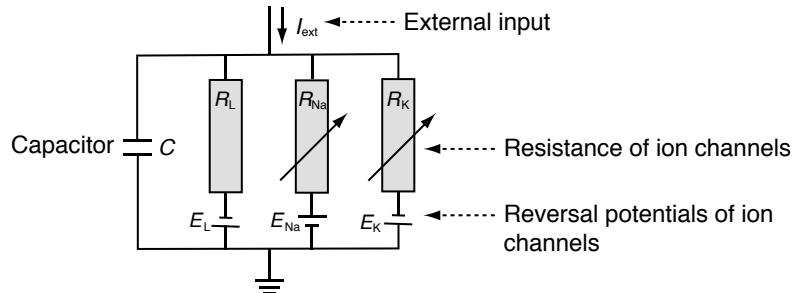


Fig. 2.8 A circuit representation of the Hodgkin-Huxley model. This circuit includes a capacitor on which the membrane potential can be measured and three resistors with their own batteries, modelling the ion channels; two are voltage-dependent and one is static.

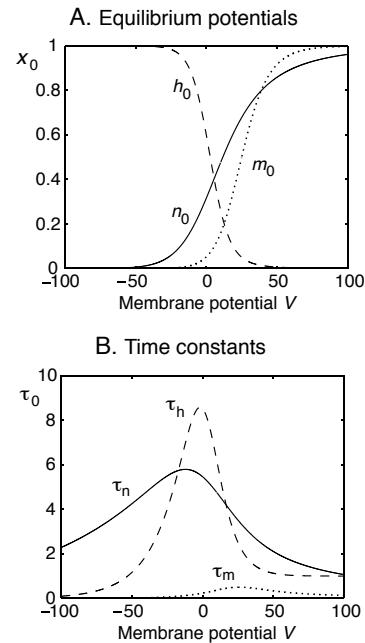


Fig. 2.9 (A) The equilibrium functions and (B) time constants for the three variables n , m , and h in the Hodgkin-Huxley model with parameters used to model the giant axon of the squid.

dependent variables $n(V, t)$, $m(V, t)$, and $h(V, t)$,

$$\bar{g}_K(V, t) = g_K n^4 \quad (2.8)$$

$$\bar{g}_{Na}(V, t) = g_{Na} m^3 h. \quad (2.9)$$

Hodgkin and Huxley chose the dynamics and voltage dependence of the variables so that the experimental data could be approximated reasonably well. They formulated this dynamic with an elegant set of three first-order differential equations, one for each variable. All three differential equations have the same form, expressed by the formula

$$\frac{dx}{dt} = -\frac{1}{\tau_x(V)}[x - x_0(V)] \quad (2.10)$$

where each letter x should be substituted by each of the variables n , m , and h . The particular dynamics of the variables and the specific form of the functional dependence of x_0 and τ_x on the voltage were choices made by Hodgkin and Huxley. They chose these specific forms to get a reasonable fit to the experimental data. At this point we will not write down the lengthy expressions for the functions $\tau_x(V)$ and $x_0(V)$ together with all the values of the included parameters that Hodgkin and Huxley chose. Instead, we have included all these details in the program `hh.m` provided in Table 2.2 below. The voltage dependence of these functions, which describes the voltage dependence of the potassium and sodium channels, is illustrated in Fig. 2.9. The leakage channel is static, and the corresponding conductance, $\bar{g}_L = g_L$, is therefore a constant.

The final ingredient that we have to consider is the fact that neurons store electric charges. This is formally expressed as a capacitance, C . The Hodgkin-Huxley model can thus be represented with electronic components, as illustrated in Fig. 2.8, consisting of a capacitor in parallel with three resistors, each supplied with their own battery setting their reversal potential. One of the resistors is constant, while the other two can change depending on the state of the system. The combined effects of the different components can be expressed by considering the conservation of electric charge (Kirchhoff's law),

$$C \frac{dV}{dt} = - \sum_{\text{ion}} I_{\text{ion}} + I(t). \quad (2.11)$$

This formula, a first-order differential equation, describes the change of the membrane potential with time. $I(t)$ is an external current, for example, the

current from the neurotransmitter-gated ion channels. We can now put all pieces together by inserting the three ionic currents discussed above into eqn 2.11 and adding the three differential equations of the form 2.10, resulting in the standard four differential equations of the Hodgkin–Huxley model,

$$C \frac{dV}{dt} = -g_K n^4 (V - E_K) - g_{Na} m^3 h (V - E_{Na}) - g_L (V - E_L) + I(t) \quad (2.12)$$

$$\tau_n(V) \frac{dn}{dt} = -[n - n_0(V)] \quad (2.13)$$

$$\tau_m(V) \frac{dm}{dt} = -[m - m_0(V)] \quad (2.14)$$

$\tau_h(V) \frac{dh}{dt} = -[h - h_0(V)]$

We discussed the generation of action potentials by sodium, potassium, and leakage channels. It should be clear that this mechanism is a result of the simultaneous working of many such ion channels. The conductances in the Hodgkin–Huxley equations are hence the net result of individual ion channels in the membrane. The Hodgkin–Huxley model makes thereby the assumption that individual ion channels work independently of each other, an assumption that was recently called into question. Thus, the basic model discussed here is also a minimalistic model to explain the basic mechanisms of spike generation.

The densities of the ion channels have to exceed a certain threshold in order to be able to generate an action potential. These conditions normally occur in the axonal membrane, although active action potentials have now been observed in dendrites. The axon can typically be excited at any location when the axon is not *myelinated* as this blocks the ion channels. However, *in vivo* (which is Latin for ‘within the living’), neurons typically initiate action potentials in the axon-hillock.

2.3.4 Numerical integration

The Hodgkin–Huxley equations can be easily integrated numerically, and a MATLAB program to accomplish this with the parameters used originally by Hodgkin and Huxley to model the axon of the giant squid is given below. Results of this numerical integration are shown in Figs 2.10 and 2.11. These simulations demonstrate that a constant external current with sufficient strength (here $I_{ext} = 10 \text{ mA/cm}^2$) leads to a constant firing of the neuron with a stereotyped waveform, as shown in Fig. 2.10. It is remarkable how the form of this simulated action potential captures the essential details of action potentials in real neurons, such as the one shown in Fig. 2.6.

When increasing external current, the number of spikes in a fixed interval, also called *rate*, increases. The function that describes the rate in dependence of input will be called the *activation function* in this book. This function is sometimes also called the *gain function*. Simulation results for the activation function of a Hodgkin–Huxley neuron are plotted in Fig. 2.11. The result reveals two more characteristics of the basic Hodgkin–Huxley neuron. First, in the basic model, shown with the solid line in the graph, onset of firing follows a sharp threshold around $I_{ext} = 6 \text{ mA/cm}^2$. Second, there is a fairly limited range of frequencies when the axon fires, starting at about 53 Hz and

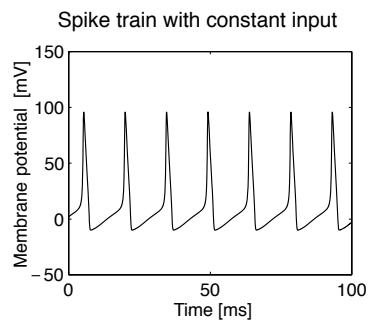


Fig. 2.10 A Hodgkin–Huxley neuron responds with constant firing to a constant external current of $I_{ext} = 10 \text{ mA/cm}^2$.

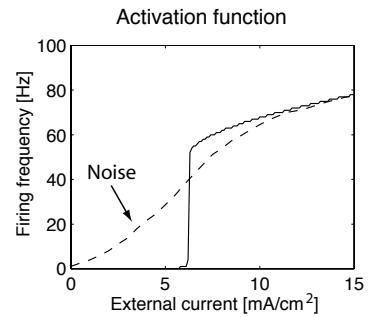


Fig. 2.11 The dependence of the firing rate on the strength of the external current shows a sharp onset of firing around $I_{ext}^c = 6 \text{ mA/cm}^2$ (solid line). High-frequency noise in the current has the tendency to ‘linearize’ these response characteristics (dashed line).

increasing only slightly with increasing external current. This is interesting because it shows that not all neuronal frequencies are equally likely.

Simulation

The file `hh.m` (see Table 2.2) contains the program for the Hodgkin–Huxley model using the Euler method for the numerical integration (see Appendix B.3). We first defined some parameters, the maximal conductances and battery voltages as used by Hodgkin and Huxley. We placed them into vectors, which makes it possible to write the later equations in compressed form using matrix notation. After the initialization of some more variables we enter a big loop over the time steps over which we want to integrate the Hodgkin–Huxley equations. For each time step, the voltage of the membrane potential may have changed and we have first to calculate the parameters τ_x and x_0 of eqn 2.10 for each of the parameters m , n , and h . These are functions that were chosen by Hodgkin and Huxley. They actually defined them in terms of functions α_x and β_x that are directly related to τ_x and x_0 as stated in the program. The particular values of some parameters in these functions are the ones originally used by Hodgkin and Huxley.

After we have calculated τ_x and x_0 for the given voltage of the membrane we are ready to integrate eqn 2.10 using a simple Euler method (see Appendix B) by replacing this continuous equation with the discrete version

$$x(t + \Delta t) = (1 - \frac{\Delta t}{\tau_x})x(t) + \frac{\Delta t}{\tau_x}x_0, \quad (2.15)$$

which is directly translated into MATLAB as

```
x=(1-dt./tau).*x+dt./tau.*x_0.
```

Notice that we have used dots before the multiplication and division symbols. This indicates that the following operation should act on each individual component instead of the usual matrix multiplications and divisions. The final steps are to use Ohm's law to calculate the corresponding synaptic currents and to update the membrane potential from the synaptic currents and the external current that flow within the small time step Δt for which all of these calculations are done. We have included several time steps before recording the membrane potential over time in order to equilibrate the system first so that our results do not depend on the initial conditions. The results of these simulations are shown in Fig. 2.12B. We started the external current sufficiently early to elicit a spike at $t = 10$, and the model neuron started to depolarize and to spike shortly thereafter. This is followed by a hyperpolarization phase, and the constant external current can only elicit the next spike after some time. The external input was removed at $t = 40$ before the model neuron was able to generate a third spike.

2.3.5 Refractory period

There is a maximum firing rate with which any neuron can respond. The reason for this is that the inactivation of the sodium channels make it impossible to

Table 2.2 Program hh.m

```

1 %% Integration of Hodgkin--Huxley equations with Euler method
2 clear; clf;
3 %% Setting parameters
4 % Maximal conductances (in units of mS/cm^2); 1=K, 2=Na, 3=R
5 g(1)=36; g(2)=120; g(3)=0.3;
6 % Battery voltage ( in mV); 1=n, 2=m, 3=h
7 E(1)=-12; E(2)=115; E(3)=10.613;
8 % Initialization of some variables
9 I_ext=0; V=-10; x=zeros(1,3); x(3)=1; t_rec=0;
10 % Time step for integration
11 dt=0.01;
12 %% Integration with Euler method
13 for t=-30:dt:50
14     if t==10; I_ext=10; end % turns external current on at t=10
15     if t==40; I_ext=0; end % turns external current off at t=40
16 % alpha functions used by Hodgkin-and Huxley
17 Alpha(1)=(10-V)/(100*(exp((10-V)/10)-1));
18 Alpha(2)=(25-V)/(10*(exp((25-V)/10)-1));
19 Alpha(3)=0.07*exp(-V/20);
20 % beta functions used by Hodgkin-and Huxley
21 Beta(1)=0.125*exp(-V/80);
22 Beta(2)=4*exp(-V/18);
23 Beta(3)=1/(exp((30-V)/10)+1);
24 % tau_x and x_0 (x=1,2,3) are defined with alpha and beta
25 tau=1./(Alpha+Beta);
26 x_0=Alpha.*tau;
27 % leaky integration with Euler method
28 x=(I-dt./tau).*x+dt./tau.*x_0;
29 % calculate actual conductances g with given n, m, h
30 gnmh(1)=g(1)*x(1)^4;
31 gnmh(2)=g(2)*x(2)^3*x(3);
32 gnmh(3)=g(3);
33 % Ohm's law
34 I=gnmh.*(V-E);
35 % update voltage of membrane
36 V=V+dt*(I_ext-sum(I));
37 % record some variables for plotting after equilibration
38 if t>=0;
39     t_rec=t_rec+1;
40     x_plot(t_rec)=t;
41     y_plot(t_rec)=V;
42 end
43 end % time loop
44 %% Plotting results
45 plot(x_plot,y_plot); xlabel('Time'); ylabel('Voltage');

```

A. MATLAB editor Window with hh.m program

```

1 % Integration of Hodgkin-Huxley equations with Euler method
2 clear; clc;
3 % Setting parameters
4 % Maximal conductances (in units of mS/cm^2); 1=K, 2=Na, 3=K
5 g(1)=3.6; g(2)=12.0; g(3)=0.3;
6 % Battery voltage (in mV); 1=n, 2=m, 3=h
7 E(1)=-12; E(2)=+15; E(3)=+10.413;
8 % Initialisation of some variables
9 x(1)=0; V=-10; x=zeros(1,3); x(3)=1; t_rec=0;
10 % Time step for integration
11 dt=0.01;
12 % Integration with Euler method
13 for t=300:dt:500
14     if t>=10; I_ext=10; end % turns external current on
15     if t>=40; I_ext=0; end % turns external current off
16     % alpha functions used by Hodgkin-and-Huxley
17     alpha(1)=(10-V)/100*(exp((10-V)/10)-1);
18     alpha(2)=(25-V)/100*(exp((25-V)/10)-1);
19     alpha(3)=0.07*exp(-V/20);
20
21 % Data function, read from Hodgkin-and-Huxley

```

B. MATLAB figure window with results of the simulation

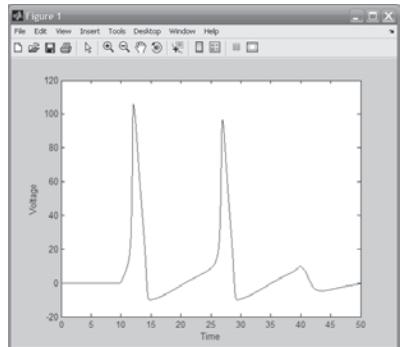


Fig. 2.12 (A) MATLAB Editor Window and (B) Figure Window for simulating the model by Hodgkin and Huxley with program hh.m included in folder Chapter2.

initiate another action potential for about 1 ms. This time is called the *absolute refractory period*, which limits the firing rates of neurons to a maximum of about 1000 Hz because then only a maximum of 1 spike in a millisecond, or 1000 spikes in a second, can be generated. Due to the hyperpolarizing action potential it is also relatively difficult to initiate another spike during this time period. This *relative refractory period* reduces the firing frequency of neurons even further. Very fast spiking frequencies have been measured in brainstem neurons, with frequencies sometimes exceeding 600 Hz. In contrast, cortical neurons typically respond with much lower frequencies, sometimes with only a few spikes per second. This cannot be explained by the refractory periods alone.

It is not obvious why cortical neurons fire with such low frequencies and with such irregularity. However, many circumstances influence the firing times of neurons. Among them are the details of charge transmission within dendrites, interactions between PSPs, other types of ion channels, and additional mechanisms not included in the Hodgkin–Huxley equations. Also, very importantly, we have to consider interactions within a network, specifically with inhibitory interneurons. Furthermore, a constant external current is physiologically unrealistic in the working brain, so one has to consider more realistic driving currents to get a better feel for typical response characteristics of a neuron.

To demonstrate how simple alterations in the basic mechanisms can alter the response characteristics of a Hodgkin–Huxley neuron, some results of further simulation are added in Fig. 2.11 with a dashed line. These simulations included some high frequency (1000 Hz) white (Gaussian distributed) noise. Such high-frequency noise could, for example, simulate low-frequency inputs from many presynaptic neurons. This noise was added to the main input current which was constant as in the previous simulations. The results indicate that such noise has the tendency to ‘linearize’ the f_I curve, resulting in the sigmoid-shaped function plotted as a dashed line in Fig. 2.11. Such response curves are important to model the information transmission in networks, and we will discuss such response curves further at the end of the next chapter when we can compare such curves to population responses.

2.3.6 Propagation of action potentials

Once an action potential is initiated, it will rapidly increase the membrane potential of neighbouring axon sites and thus travel along the axon with a low speed of around 10 m/s. This form of signal transportation within an *active membrane* is loss-free; the signal is regenerated at each point of the axon membrane and the signal will not deteriorate with distance. This is of major importance since axons can sometimes be very long, on the order of metres. On the other hand, this form of loss-free signal transportation is much slower than simple current transportation within a conductor, such as in an electrical wire or cable, and also consumes a lot of energy, as we mentioned above. It seems that nature responded to these problems by covering axons with a sheath of *myelin*. This sheath is created by a special type of glial cells, *oligodendrocytes* in the central nervous system and *Schwann cells* in the peripheral nervous system. Myelin sheaths change the electrical properties of axons to allow fast, but lossy, signal transfer within axons. However, myelin sheets are regularly interrupted by so-called *Ranvier nodes*, at which points the action potential can regenerate through the active membrane mechanism. It seems that nature combines, in this way, the advantages of fast and ‘cheaper’ signal transmission with regenerative amplifiers. It is interesting to note that a large part of the myelination happens after an initial organizational phase of the nervous system, which occurs in humans around their second year of age.

2.3.7 Above and beyond the Hodgkin–Huxley neuron: the Wilson model ◊

The Hodgkin–Huxley equations have taught us the basic concepts of the generation of action potentials. However, in contrast to giant axons of squids, mammalian neurons have additional types of ion channels that enable more complex neuronal responses. At least a dozen types of ion channels can be involved in the spike generation of human neocortical neurons. It is fairly straightforward to include other ion channels in models to describe more specifically the electrical properties of specific neurons. Some modelling papers include therefore many coupled differential equations to model many different channels in one compartment. Such models are important to investigate detailed response characteristics of neurons and to investigate the interactions between the different channels. On the other hand, simplifications are also common when studying networks of neurons. Simplifications are often necessary to enable large-scale investigations, and, as discussed in Chapter 1, our useful scientific strategy is also to keep models as simple as possible. In the final section of this chapter we will provide some basic examples to illustrate that the original Hodgkin–Huxley equations can be modified in several ways, either to simplify the systems for large-scale studies, or to include more of the details of real neurons.

Models that reduce the dimensionality of Hodgkin–Huxley systems have been advanced for many years. For example, we can simplify the Hodgkin–Huxley model by using the results shown in Fig. 2.9. From Fig. 2.9A we see that the rate of inactivation of Na^+ channels is approximately reciprocal to the opening of K^+ channels. We can thus reduce the dimensionality of the model by setting

$h = 1 - n$. Furthermore, from Fig. 2.9B we can see that the time constant τ_m for the dynamic variables m is small for all values of V . The dynamic for this variable is thus fast and can be approximated with the corresponding equilibrium values, $m_0(V)$. These simplifications reduce the Hodgkin–Huxley model to a two-dimensional system with an action potential very similar to that of the original Hodgkin–Huxley neurons. The model can be further simplified for neocortical neurons. Neocortical neurons often show no inactivation of the fast Na^+ channel, so h can be set to $h = 1$ (or, equivalently, $n = 0$ within our approximations). Following these approximations *Hugh Wilson* showed that a simplified model with only two differential equations,

$$C \frac{dV}{dt} = -g_K R(V - E_K) - g_{\text{Na}}(V)(V - E_{\text{Na}}) + I(t) \quad (2.16)$$

$$\tau_R \frac{dR}{dt} = -[R - R_0(V)], \quad (2.17)$$

still behaves similarly to Hodgkin and Huxley's system of four differential equations. The Wilson model described by eqns 2.16 and 2.17 combined the Na^+ and leakage channels into a new single Na^+ channel by including a linear term in the description of the voltage dependence of the new channel. The symbol of the only remaining dynamic modulation variable, R , describes the recovery of the membrane potential.

So far, we have only approximated the principal mechanisms that have already been described by Hodgkin and Huxley. In the following we include two more types of ion channels that seem essential for the more diverse firing properties of mammalian neocortical neurons. The first channel is a cation influx channel with more graded influx characteristics described by a dynamic gating variable T . Such channels, in particular Ca^{2+} channels, are a major ingredient of mammalian nervous systems. The more graded Ca^{2+} influx is mainly responsible for the more graded response characteristics of neocortical neurons compared to those of the giant axon of the squid. The second channel describes a slow hyperpolarizing current, such as that of a common Ca^{2+} -mediated K^+ channel, with a dynamic gating variable H . We will see that this type of channel is essential for the generation of a more complex firing pattern based on spike frequency adaptation. The complete Wilson model of mammalian neocortical neurons includes these types of channels, and is given by

$$C \frac{dV}{dt} = -g_{\text{Na}}(V)(V - E_{\text{Na}}) - g_K R(V - E_K) - g_T T(V - E_T) - g_H H(V - E_H) + I(t) \quad (2.18)$$

$$\tau_R \frac{dR}{dt} = -[R - R_0(V)] \quad (2.19)$$

$$\tau_T \frac{dT}{dt} = -[T - T_0(V)] \quad (2.20)$$

$$\tau_H \frac{dH}{dt} = -[H - 3T(V)], \quad (2.21)$$

with polynomial parametrizations of the voltage dependence of the effective conductances,

$$g_{\text{Na}}(V) = 17.8 + 0.476V + 33.8 \cdot 10^{-4}V^2 \quad (2.22)$$

$$R_0(V) = 1.24 + 0.037V + 3.2 \cdot 10^{-4}V^2 \quad (2.23)$$

$$T_0(V) = 4.205 + 0.116V + 8 \cdot 10^{-4}V^2. \quad (2.24)$$

Despite drastic simplifications, such as neglecting the typical inactivation of the Ca^{2+} channels, this model is able to approximate mammalian spike characteristics in great detail. This includes the shape of single spikes, as well as all major classes of spike characteristics, such as regular spiking neurons (RS), fast spiking neurons (FS), continuously spiking neurons (CS), and intrinsic bursting neurons (IB), by choosing appropriate values of the remaining constants. This is shown next.

Simulations of the Wilson model can be done with the programs discussed below. All of the following simulations were done with fixed values of the constants as suggested by Wilson, namely $E_{\text{Na}} = 50$ mV, $E_{\text{K}} = -95$ mV, $E_{\text{T}} = 120$ mV, $E_{\text{R}} = E_{\text{K}}$, $C = 100 \mu\text{F}/\text{cm}^2$, $g_{\text{Na}} = 1$, $g_{\text{K}} = 26$, $\tau_{\text{T}} = 14$ ms, $\tau_{\text{R}} = 45$ ms. The simulations also use a constant external driving current, $I_{\text{ext}} = 1$ nA. Ignoring the slow, calcium-mediated potassium channel ($g_{\text{H}} = 0$) and setting $\tau_{\text{R}} = 1.5$ ms, $g_{\text{T}} = 0.25$ results in constant rapid spiking, as shown Fig. 2.13A. Such spike trains are typical for inhibitory interneurons in the mammalian neocortex when stimulated with a constant current.

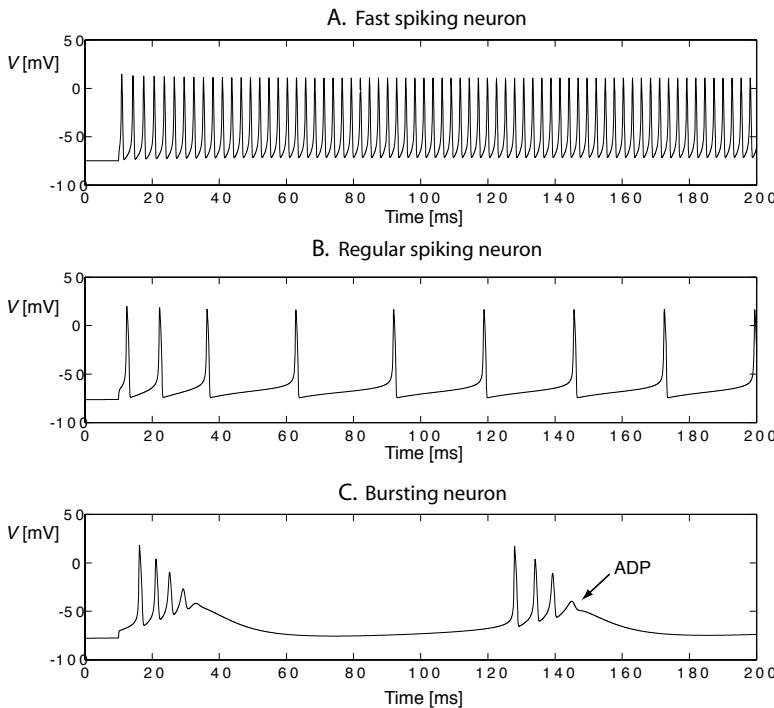


Fig. 2.13 Simulated spike trains of the Wilson model. (A) Simulates fast spiking neuron (FS) typical of inhibitory neurons in the mammalian neocortex ($\tau_{\text{R}} = 1.5$ ms, $g_{\text{T}} = 0.25$, $g_{\text{H}} = 0$). (B) Simulated regular spiking neuron (RS) with longer interspike time intervals ($\tau_{\text{R}} = 4.2$ ms, $g_{\text{T}} = 0.1$). The slow, calcium-mediated potassium channel ($g_{\text{H}} = 5$) is responsible for the slow adaptation in the spike frequency. (C) This graph demonstrates that even more complex behaviour, typical of mammalian neocortical neurons, can be incorporated in the Wilson model. The parameters ($\tau_{\text{R}} = 4.2$ ms, $g_{\text{T}} = 2.25$, $g_{\text{H}} = 9.5$) result in a typical bursting behaviour, including a typical after-depolarizing potential (ADP) [see also Wilson, *J. Theor. Biol.* 200: 375–88 (1999)].

Excitatory neurons typically have more complex behaviour, with slower spike rates and elongated spikes. In the following simulations, we therefore use a time constant of $\tau_{\text{R}} = 4.2$ ms. The inclusion of a slow, calcium-mediated potassium channel ($g_{\text{H}} = 5$), with a slightly smaller calcium conductance ($g_{\text{T}} = 0.1$),

results in the spike train shown in Fig. 2.13B. The slow hyperpolarizing channel has the important effect of reducing the firing rate after initial stimulation of the neuron. This *spike rate adaptation* is sometimes called *fatigue*. The firing rates of such regularly spiking neurons are often reduced to about half of their initial value in about 50 ms, and some neurons even cease firing altogether.

Another important class of neurons, showing short bursts interleaved with silent phases, can be simulated by increasing the slow hyperpolarizing conductance ($g_H = 9.5$) as well as the calcium conductance ($g_T = 2.25$). The firing response of such a neuron is shown in Fig. 2.13C. Such *bursting neurons* typically show a long-lasting *after-depolarizing potential* (ADP), which is also present in the Wilson model.

Simulation

Every set of ordinary differential equations (ODE) can easily be integrated with the Euler method. In program `wilson_euler.m`, included in directory `Chapter2`, we have implemented this for the Wilson model discussed above. Some results of this program are shown in Fig. 2.14A. The low-order Euler method can, however, be numerically unstable as discussed in Appendix B. In short, the low-order Euler method only represents the continuous differential closely if we use a very small time step in the numerical integration. Fig. 2.14A shows results for two different integration time steps, $\Delta t = 0.01$ and $\Delta t = 0.06$. The question of how small the time step should be becomes obvious. In

A. Wilson model with Euler method B. Wilson model with higher-order ODE solver

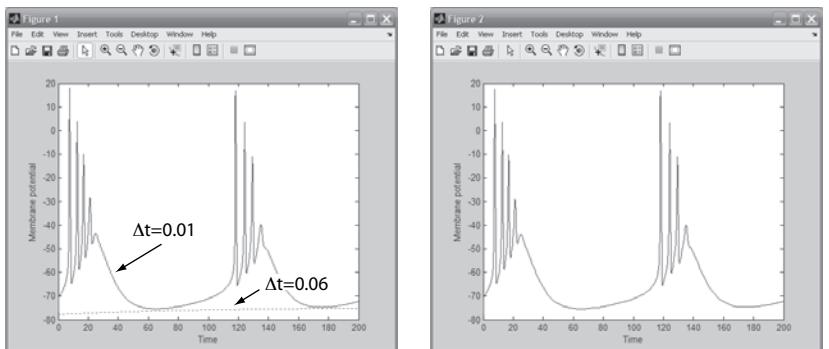


Fig. 2.14 (A) MATLAB *Figure Window* produced with program `wilson_euler.m` with two different integration time steps Δt . (b) Results of the program `wilson.m` that solves the same model with a higher-order ODE solver.

practice we can make some numerical experiments in which we alter the size of the time step considerably. If the numerical results are stable with regard to the change of the step width, we can gain some confidence that the results are generic. However, there are many advanced integration methods. For example, we can use some information on the curvature of the function instead of using a linear interpolation between integration points. Such higher-order methods generally produce more accurate results when integrated with the same time step. The time step should also be chosen properly in each domain of the functions we want to integrate. If the integral is not changing very much over time, then it is reasonable to use a large time step. In contrast, if the integral

Table 2.3 Program wilson.m

```

1 %% Integration of Wilson model with multistep ODE solver
2 clear; clf;
3
4 %% Integration:
5 %1: Equilibration: no external input;
6 y0=zeros(1,4); y0(4)=-1; param=0; I_ext=0; tspan=[0 100];
7 [t,y]=ode45('wilson_ode',tspan,y0,[],I_ext);
8
9 %2: Integration with external input;
10 y0=y(size(t,1),:); param=0; I_ext=1; tspan=[0 200];
11 [t,y]=ode45('wilson_ode',tspan,y0,[],I_ext);
12
13 %% Ploting Results
14 plot(t,100*y(:,4)); xlabel('Time'); ylabel('Membrane potential');

```

Table 2.4 Function wilson_ode.m

```

1 function ydot=wilson_ode(t,y,flag,I_ext)
2 % ode for wilson equations
3 %% parameters of the model: 1=K,R; 2=Ca,T; 3=KCa,H; 4=Na;
4 g(1)=26; g(2)=2.25; g(3)=9.5; g(4)=1; g=g';
5 E(1)=-.95; E(2)=1.20; E(3)=E(1); E(4)=-.50; E=E';
6 tau(1)=1/4.2; tau(2)=1/14; tau(3)=1/45; tau=tau';
7
8 V=y(4); y3=y(1:3);
9
10 x0(1)=1.24 + 3.7*V + 3.2*V^2;
11 x0(2)=4.205 + 11.6*V + 8 *V^2;
12 x0(3)=3*y(2);
13 x0=x0';
14
15 ydot=tau.* (x0-y3);
16
17 y3(4)=17.8 + 47.6*V +33.8*V^2;
18
19 I=g.*y3.* (y(4)-E);
20 ydot(4)=I_ext-sum(I);
21 return

```

is changing rapidly we have to use a smaller time step. Several methods are available for choosing the time step as a function of the performance of the algorithm. Such methods are generally known as *adaptive time step methods*.

All we have said is common knowledge in numerical mathematics and details can be found in many books. An advantage of MATLAB is that many of these algorithms are implemented as functions that can be used directly within MATLAB. A demonstration using a MATLAB-native ODE solver to integrate the Wilson equations is included in the file `wilson.m` (see Table 2.3). A frequently used MATLAB ODE solver, based on the Runge–Kutta algorithm with adaptative time steps, is invoked with the function call

```
[t,y]=ode45('wilson_ode',tspan,y0,[],I_ext);
```

The MATLAB ODE solver function returns the value of the function y , defined through the differential equation

$$\frac{dy}{dt} = f(y), \quad (2.25)$$

at times t . The function f , which incorporates all the details of our model, is specified in a function file whose name is passed as the first argument in the call to the ODE solver. In our case we encapsulate the right-hand side of eqn 2.25 in the file `wilson_ode.m` (see Table 2.4). Translating the continuous formulas in the text to programs is very easy. We no longer have to calculate the corresponding discrete version of the equation and can write a function that very much resembles our equations in the book. The form of the parameter list for MATLAB ODE solvers is generic in MATLAB, so it is possible to change the ODE solver by simply changing the name in the function call. For example, `ode45`→`ode15s` changes the numerical integration algorithm to a solver which is more appropriate for stiff problems. The graph produced by the program `wilson.m` is shown in Fig. 2.14B. This version of the program runs faster than the program `wilson_euler` with the necessary small time step.

2.4 Including neuronal morphologies: compartmental models

We have discussed the effects of neurotransmitter-gated ion channels, leakage channels, and voltage-dependent ion channels on the membrane potential of a neuron. To get a comprehensive view of the state of a neuron we must also include the conductance properties and the physical shape of the neuron. In contrast to axons with active membranes able to generate action potentials, dendrites have been seen historically more like passive conductors,² analogous to long cables. The physics of conducting cables was worked out in the mid-19th century by *Lord Kelvin*, enabling the first transatlantic communication cables. Many researchers have since applied this theory to neural transmission. For example, *Wilfrid Rall*, who has contributed much to the theory and its applications. Taking these missing links into account, we will outline the basic idea behind compartmental models in this section. Such modelling alone is an active (and very well developed) area of computational neuroscience (see ‘Further reading’ at the end of this chapter).

²Dendrites also have active ion channels generating BAPs. Non-linear effects in dendrites are also important. However, many general features of dendrites in a subthreshold regime can be studied using passive cable equations, so this approximation is used to outline the principal idea behind compartmental modelling.

2.4.1 Cable theory

To get a feeling for some equations involved in compartmental modelling, we will briefly outline the *cable equation*. This equation describes the spatio-temporal variation of an electric potential along a cable-like conductor driven by an injected current I_{inj} . For an idealized one-dimensional linear cable, the equation is given by

$$\lambda^2 \frac{\partial^2 V_m(x, t)}{\partial x^2} - \tau_m \frac{\partial V_m(x, t)}{\partial t} - V_m(x, t) + V_0 = R_m I_{\text{inj}}(x, t). \quad (2.26)$$

This is a partial differential equation,³ second-order in space and first-order in time. The solution of this equation, $V_m(x, t)$, describes the potential of the cable at each location of the cable and how it varies at each location with time. The constant λ describes the physical properties of the cable and has the dimensions of $\Omega \text{ cm}$. For example, a cylindrical cable of diameter d , in an equipotential surrounding, has a lambda parameter of

$$\lambda = \sqrt{\frac{dR_m}{2R_i}}, \quad (2.27)$$

where R_m is the specific resistance of the membrane and R_i is the specific intracellular resistance of the cable. A specific resistance is the resistance of a piece of material with constant cross-section, divided by the volume of the resistor. Membrane resistance is, of course, much larger than the intracellular resistance, which explains why most of the current flows within the dendritic tree. Note that the lambda parameter changes with the diameter of the cable. The time constant τ_m also depends on the physical properties of the cable and is given by

$$\tau_m = R_m C_m, \quad (2.28)$$

where R_m is the specific resistance of the membrane, or the inverse of the leakage conductivity, and C_m is the specific capacitance, the capacitance per unit area.

The solutions of eqn 2.26 also depend on the form of the injected current $I_{\text{inj}}(x, t)$, and analytical solutions can only be given for some simple examples which are nevertheless instructive.⁴ Is is clear that the cable potential should reach a stable configuration (distribution within the cable) if the injected current is not time dependent. For example, if we consider a semi-infinite cable that starts at $x = 0$ with a fixed potential $V_0 = R_m I_0$ and extends to infinity thereafter, then the potential decays exponentially with

$$V_m = V_0 e^{-\frac{x}{\lambda}}. \quad (2.29)$$

This exponential decay is, of course, a problem for information transmission⁵ and the signal in a long cable has to be amplified periodically in order for it to be transmitted over large distances.

If we include voltage-dependent ion channels as in the Hodgkin–Huxley equations, we have a voltage-dependent injected current. In this case we have to solve the *non-linear cable equation* where $I_{\text{inj}}(x, t)$ in eqn 2.26 is replaced with $I_{\text{inj}}(V_m(x, t), t)$ which depends explicitly on V_m , which in turn varies spatially

³The equation includes partial derivatives symbolized by ∂ instead of total derivatives symbolized by d . Partial derivatives only consider explicit dependencies on the parameters and ignore implicit dependence on other parameters. This differential equation is formally a hyperbolic differential equation in case you want to look it up in mathematics books.

⁴The equation can be integrated numerically in more general situations as discussed later.

⁵In the case of signal transmission we have to solve the cable equation in the presence of a time-varying current. However, the general features of a decaying signal also hold in this case.

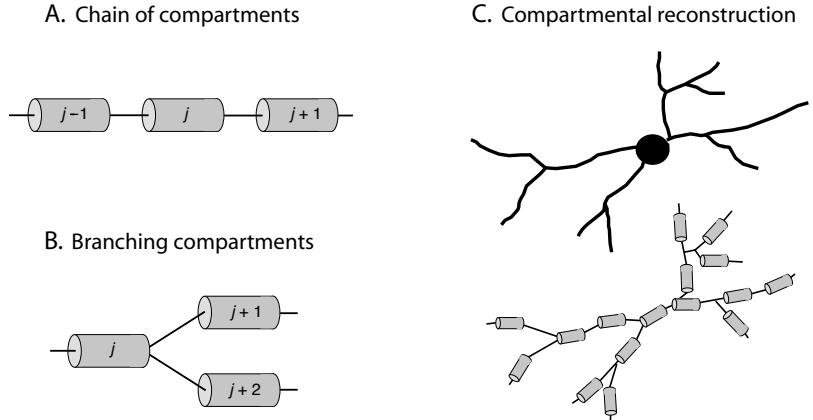


Fig. 2.15 Short cylindrical compartments describing small equipotential pieces of a passive dendrite or small pieces of an active dendrite or axon when including the necessary ion channels. (A) Chains and (B) branches determine the boundary conditions of each compartment. (C) The topology of single neurons can be reconstructed with compartments, and such models can be simulated using numerical integration techniques.

and temporally. No general analytical solution can be given for this equation, but we can use numerical integration techniques to solve the equation for particular cases.

2.4.2 Physical shape of neurons

The cable equations depend on the physical properties of the cable through the parameter λ , and it is time to take the physical shape of neurons into account. The first step in solving the cable equations for cables with structures other than a simple homogeneous linear cable is to divide the cable into small pieces (for example, small cylindrical cables), where each piece has to be small enough so that the potential within each unit is approximately constant. We call this unit a *compartment* (see Fig. 2.15A). Each compartment is governed by a cable equation for a finite cable, which is a first-order differential equation in time, as we have seen above. For small compartments we can assume a constant potential within the compartment. However, we also have to take the boundary conditions into account. This can be done by replacing the continuous space x with discrete spatial locations labelled with indices, such as x_j , and replace the spatial differentials with differences,

$$\frac{\partial^2 V(x, t)}{\partial x^2} \rightarrow \frac{V_{j+1} - 2V_j(t) + V_{j-1}(t)}{(x_{j-1} - x_j)^2}. \quad (2.30)$$

With similar formulas we can take branching cables (as illustrated in Fig. 2.15B) into account. A neuron is therefore simulated with a discrete model of small cylinder-like cables as illustrated in Fig. 2.15C. This *compartmental model* is governed by a set of first-order differential equations in time, since we have replaced the spatial differentials with difference equations. The number of compartments should be large enough to accurately represent a neuron. In practice, the number of compartments needed to get good approximations of the real neuron depends on the complexity of the neuron. Models with a few hundred to several thousand compartments have been used in the literature to represent single neurons very accurately.

2.4.3 Neuron simulators

The set of coupled differential equations defining a compartmental model can be solved numerically. While numerically solving differential equations is, in principle, straightforward, as discussed in Appendix B, there are many details to be considered for efficient numerical integration. These include the stability of solutions and efficient use of computational resources. It is therefore useful to employ special-purpose software that implements appropriate numerical techniques so that computational neuroscientists can concentrate on the biological questions. Compartmental models have often been analysed numerically by replacing their equations with equivalent representations of electrical components and using circuit simulation programs such as SPICE to solve the systems. Today there are also several public-domain simulation packages that specialize in compartmental modelling. Popular examples include the simulator GENESIS (GEneral NEural SImulation System), developed at Caltech (see <http://www.genesis-sim.org>).

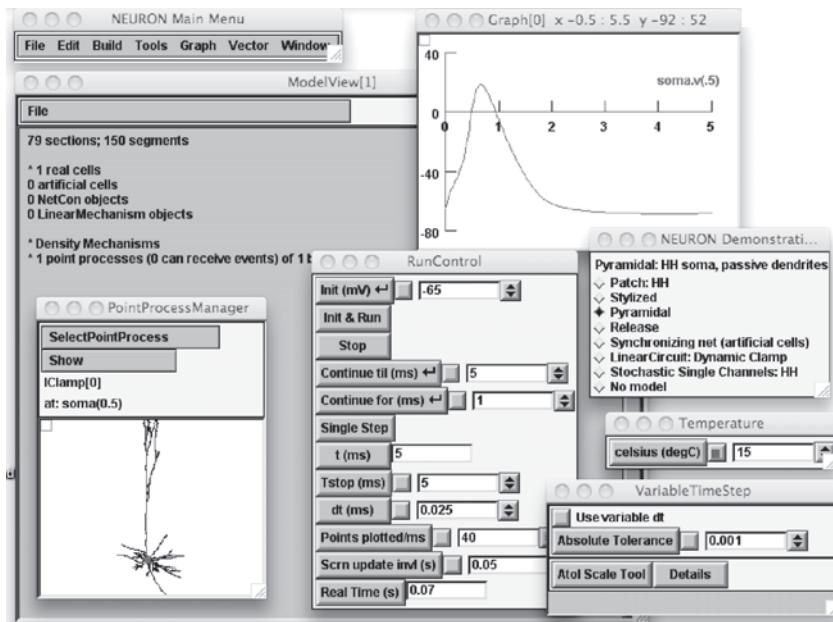


Fig. 2.16 Example of the NEURON simulation environment (see <http://www.neuron.yale.edu/neuron/>). The example shows a simulation with a reconstructed pyramidal cell, in which a short current pulse was injected.

Another simulator called NEURON (see <http://neuron.duke.edu>), was developed by M. L. Hines, J. W. Moore, and N. T. Carnevale from Duke and Yale Universities. Its distribution package includes an example of a three-dimensional reconstruction of a pyramidal cell, as shown in Fig. 2.16. In the shown example, we injected a 20 nA current at $t = 1$ ms for 0.5 ms at the location on the dendrite indicated by the blob and cursor in the graphical outline of the cell. We then measured the response at the soma, which can be seen in the graph window. The passive dendrite passed the current on to the soma with some delay and a more gradual response. The new versions of such simulation programs also include many specific cellular mechanisms and some support for

network simulations. All of these simulation packages include tutorials that are highly recommended for further studies.

Exercises

- (2.1) What is the difference between an EPSP and an action potential?
- (2.2) Modify the simulation program for a synapse (Table 2.1) to show the time course of the EPSP when the synapse is stimulated with neurotransmitters every 20 ms.
- (2.3) A membrane has capacitance $1000 \mu\text{F}$ and a voltage-gated and time-dependent ion channel. This channel has a reversal potential of -1 mV and supports an inward current of positive ions with conductance $g_0 = 1 \text{ S}$ in its base state. When the membrane potential exceeds 0.5 mV , this channel opens for an additional inward current of negative ions with conductance $g_{vt} = 5\text{S}$ for a time window of $\Delta t = 1 \text{ ms}$. Write a simulation program which shows the time course of the membrane potential.
- (2.4) Why is the resting potential different in Fig. 2.6 and Fig. 2.10? Change the Hodgkin–Huxley program to reproduce the scale as measured in the physiological experiments.
- (2.5) Use the Hodgkin–Huxley program to plot the current-response (activation) function as shown in Fig. 2.11.

Further reading

The book by Bear, Connors, and Paradiso (2006) is a very readable introduction to neuroscience, and the text by Kandel, Schwartz, and Jessell (2000) is a standard reference, which is particularly good for looking up specific topics. A standard reference in neurobiology, which has large portions dedicated to the description of single neurons, is Shepherd (1994). It includes not only the basic processes sketched in this chapter, but also neuromodulators and developmental issues.

The following books have a more computational focus. Koch (1999) is one of the most comprehensive discussions of the biophysics of single neurons, a ‘must have’ when doing research with extended biophysical details. The book edited by Koch and Segev (1998) has a specific computational focus and is likely the best book explaining many techniques in computational neuroscience, in particular with detailed neuronal models. This collection of articles from specialists in the corresponding areas is one of the best references for the basic information processing of neurons, including cable theory, active dendrites, ion-channel

dynamics, and compartmental models. The first edition of this book, with different contributions, published in 1989, is also worthwhile reading. The amazing book by Tuckwell contains a detailed analysis of single neurons, cable theory, and stochastic modelling with a much more mathematical focus. Finally, I drew largely on the paper by Wilson (1999) as an example of conductance-based models beyond the Hodgkin–Huxley description. His book is a wonderful example of how dynamical systems theory can provide insights into neural modelling.

Mark F. Bear, Barry W. Connors, and Michael A. Paradiso (2006), *Neuroscience: exploring the brain*, Lippincott Williams & Wilkins, 3rd edition.

Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell (2000), *Principles of neural science*, McGraw-Hill, 4th edition.

Gordon M. Shepherd (1994), *Neurobiology*, Oxford University Press, 3rd edition.

- Christof Koch (1999), *Biophysics of computation; information processing in single neurons*, Oxford University Press.
- Christof Koch and Idan Segev (eds.) (1998), *Methods in neural modeling*, MIT Press, 2nd edition.
- C. T. Tuckwell (1988), *Introduction to theoretical neurobiology*, Cambridge University Press.
- Hugh R. Wilson (1999) *Spikes, decisions and actions: dynamical foundations of neuroscience*, Oxford University Press. See also his paper in Journal of Theoretical Biology 200: 375–88, 1999.

This page intentionally left blank

Simplified neuron and population models

3

In keeping with the minimalistic approach to modelling, this chapter introduces further abstractions of neurons that are frequently used to study the importance of spike timings in networks of such elements. A common model of simplified neurons used in network simulations is the *integrate-and-fire (IF) model*, and we will also discuss some advanced variations of such models. The IF model is comprised of a subthreshold leaky-integrator dynamic, a firing threshold, and a reset mechanism. We will demonstrate that the variability in the firing times of such model neurons is often much less than the variability in real neurons, and we outline some *noise models* that can be incorporated into the deterministic models to increase the variability in neuronal response. In addition, this chapter introduces models that describe the collective behaviour of groups of neurons, also known as *population models* or *rate models*. Population nodes are the basis for many system-level investigations in cognitive neuroscience and are used frequently in later sections of this book.

3.1 Basic spiking neurons

Many computational studies in the literature are based on strongly simplified versions of the mechanisms present in real neurons. The reasons for such simplifications are twofold. On the one hand, such simplifications are often necessary to make computations with large numbers of neurons tractable. On the other hand, it can also be advantageous to highlight the minimal features necessary to enable certain emergent properties in the networks. It is, of course, an important part of every serious study to verify that the simplifying assumptions are appropriate in the context of the specific question to be analysed via a model.

The form of spikes generated by neurons is very stereotyped and it is therefore unlikely that the precise details of spike shape are crucial for information transmission in a nervous system on the level discussed in this book.¹ In contrast, spike timing certainly has some influence on the processing of spiking elements in networks that we will explore further below. Thus, for the questions we are going to ask, it is not important to describe the precise form of a spike, only the integration of synaptic input leading to a generation and the following recovery process of a spike. We therefore neglect the detailed ion-channel dynamics and concentrate instead on approximating the dynamic integration of synaptic input, spike timing, and resetting after spikes.

3.1 Basic spiking neurons	53
3.2 Spike-time variability ◇	64
3.3 The neural code and the firing rate hypothesis	70
3.4 Population dynamics: modelling the average behaviour of neurons	74
3.5 Networks with non-classical synapses: the sigma–pi node	81
Exercises	84
Further reading	84

¹Proteins, which make up ion channels, have to be maintained constantly and could thus be altered through intercellular processes. While we will discuss this somewhat for dendritic plasticity in Chapter 4, intercellular control of spiking mechanisms is beyond the scope of this book.

3.1.1 The leaky integrate-and-fire neuron

The main effects of sodium and leakage channels, which are important in the rising phase of a spike, is that they drive the neuron to higher voltages with input and let the voltage decay to its resting potential otherwise. Neglecting the non-linearities captured by the non-linear voltage dependence of sodium channels, we can approximate the subthreshold dynamics of the membrane potential with a simple linear differential equation according to Kirchoff's law (see eqns 2.11 and 2.2),

$$\tau_m \frac{dv(t)}{dt} = -(v(t) - E_L) + RI(t), \quad (3.1)$$

where E_L is the resting potential. Such a system is also called a *leaky integrator*. We have introduced a time constant τ_m determined mainly by the capacitance of a one-compartment model and the average conductances of the sodium and leakage channels. The input current $I(t)$ can have different sources, such as artificially applied current in experimental settings or synaptic input. We will specify this later for specific models.

While the subthreshold buildup of the membrane potential is modelled in some detail, a major simplification is made once the membrane potential is large enough to trigger an action potential. At this point we used the full set of Hodgkin–Huxley equations to model the form of the spike in Chapter 2. However, we are now only interested in the fact that a spike was generated at time t^f when the threshold ϑ was reached. Mathematically we can write this as

$$v(t^f) = \vartheta. \quad (3.2)$$

Assuming that there is a fixed threshold is only a crude approximation of real spike generation. The Hodgkin–Huxley equations already show that the *value of no return*, which is the amount of buildup from which spiking can not be prevented by turning off the stimulus, depends on the form (or curvature) of current buildup. Also, some cortical neurons *in vivo* display a strong variability of this threshold as well as other variabilities that are ignored here.

To complete the model we also have to reset the membrane potential after a neuron has fired. Several mechanisms can be considered here which will be explored further below. A particularly simple choice is to reset the membrane potential to a fixed value v_{res} immediately after a spike, for example,

$$\lim_{\delta \rightarrow 0} v(t^f + \delta) = v_{\text{res}}, \quad (3.3)$$

where δ is an infinitesimally small time step in the continuous formulation, or a finite time step in discrete versions, as used in numerical simulations. We can incorporate an absolute refractory time by holding the membrane potential constant, $v = v_{\text{res}}$ for a certain period of time after the spike. A more graded refractoriness can be simulated with a smoother functional description instead of the sudden reset of the membrane potential, but most of the simulations discussed in this book will not depend crucially on these details. Equations 3.1–3.3 define the (leaky) *integrate-and-fire neuron*, although a better name would probably be a leaky integrate-and-fire-and-reset neuron. It is also often

abbreviated as *LIF neuron* or *IF neuron*. Such a model neuron is illustrated schematically in Fig. 3.1.

While the form of the IF neuron as presented here is rather general, this neuron model is commonly used in conjunction with specific input currents for specific modelling studies. For example, we can think of this current as supplied externally in experiments or we could describe the current which is produced from specialized processes of sensory cells. In many network simulations, we are mainly concerned with the sum of synaptic currents generated by firings of presynaptic cells. This sum of synaptic currents depends on the *efficiency* of individual synapses, which are described by a *synaptic strength value*, w_j , for each synapse, labelled by an index j representing each presynaptic neuron. We also call this value the *connection strength*, *synaptic efficiency*, or simply *weight value*. In the following, we assume that there are no interactions between synapses so that we can write the total input current to the neuron under consideration as the sum of the individual synaptic currents where each postsynaptic response to a presynaptic event, written as an α -function that we discussed in Section 2.2.3, is multiplied by a weight value (synaptic efficiency)

$$I(t) = \sum_j \sum_{t_j^f} w_j \alpha(t - t_j^f). \quad (3.4)$$

The variable t_j^f denotes the firing time of the presynaptic neuron of synapse j , in contrast to the firing time of the postsynaptic neuron above, which has no index. Thus, we parameterized the time course of synaptic response with the function $\alpha(t)$ such as used in eqn 2.1. Alternatively, we could use a conductance-based model of synapses as in eqns 2.4–2.5. The major simplification of IF neurons is the spike generation model.

3.1.2 Response of IF neurons to very short and constant input currents

The whole set of equations describing an IF neuron are non-linear due to the reset of the membrane potential after each spike, and typically have to be integrated numerically. However, the dynamic equation for the subthreshold membrane potential leading up to a spike, eqn 3.1, is mathematically an *inhomogeneous linear differential equation* that can be solved analytically. To simplify the following equations we will set $E_L = 0$ so that the membrane potential v is measured relative to this resting potential. We discuss the solutions in this section for some instructive case.

We start with the case of an IF neuron that is driven by a very short input pulse (a very short input current $I(t)$ which is not sufficient to elicit a spike²). Say we start at a situation where the initial potential shortly before time $t = 0$ is

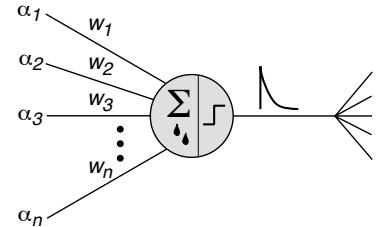


Fig. 3.1 Schematic illustration of a leaky integrate-and-fire neuron. This neuron model integrates (sums) the external input, with each channel weighted with corresponding synaptic weighting factors w_i , and produces an output spike if the membrane potential reaches a firing threshold.

²Mathematically we write this short input pulse as a delta function $I(t) = \delta(t)$ as is further described in Appendix A.3. It is hard to imagine that this form can be realized in nature; it would mean having an input pulse that is infinitely strong but has zero duration. However, it is possible to define the delta function properly as a limiting process of a finite pulse, for example, as a Gaussian bell curve, and making the duration smaller while keeping the integral over this function constant. With this limiting process, we can solve the differential equation 3.1.

equal to zero. We then apply a very short input pulse that drives the potential to at time $t = 0$ to $v(t = 0) = 1$. After this, there is no remaining input current, so that eqn 3.1 becomes a simple homogeneous differential equation for all times $t > 0$, given by

$$\tau_m \frac{dv(t)}{dt} + v(t) = 0. \quad (3.5)$$

The solution of this differential equation is an exponential function that can easily be verified by inserting the result into the differential equation. The membrane potential decays exponentially after a short external current is applied,

$$v(t) = e^{-t/\tau_m}. \quad (3.6)$$

The time scale of this decay is given by the time constant τ_m .

Another example where we can integrate the subthreshold dynamic equations is an IF neuron driven by a constant input current that is low enough to prevent firing. This will lead, after some transient time, to a stationary state where the membrane potential will not change any further because the input current no longer changes. When the membrane potential does not change we have

$$\frac{dv}{dt} = 0. \quad (3.7)$$

Inserting this into eqn 3.1 yields the equilibrium potential for large times given by

$$v = RI. \quad (3.8)$$

We chose the condition where the input current is small enough to not elicit a spike. This condition can now be specified further as $RI < \vartheta$. We only calculated the equilibrium solution of the membrane potential after a constant current had been applied for a long time. The differential equation for constant input can also be solved for all times after the constant current $I_{\text{ext}} = \text{const}$ is applied, as in

$$v(t) = RI(1 - e^{-t/\tau_m} + \frac{v(t = 0)}{RI}e^{-t/\tau_m}), \quad (3.9)$$

which can be verified by inserting this solution into the differential equation. This solution has two parts: the last term describes the exponential decay of potential at $v(t = 0)$, as before, and the first term describes the increase of the membrane potential due to the input current. This solution is illustrated in Fig. 3.2A. The neuron does not fire in this case because we have used a relatively small input current, relative to the firing threshold of the neuron, $RI < \vartheta$. In contrast, if the mean external current is larger than the threshold, $RI > \vartheta$, then the neuron fires regularly with a constant time between spikes, the *interspike interval*. This is shown in Fig. 3.2B from simulations discussed next.

Simulation

An example program to simulate a leaky IF neuron with the Euler method is listed in Table 3.1. The corresponding file `if_sim.m` is included in folder

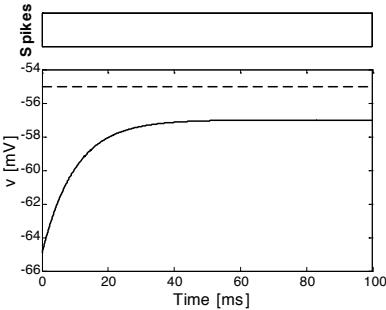
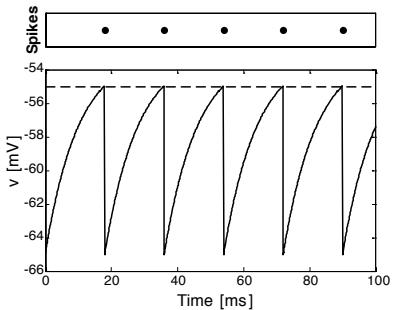
A. External input $RI_{\text{ext}} = 8 \text{ mV} < \text{threshold}$ **B. External input $RI_{\text{ext}} = 12 \text{ mV} > \text{threshold}$** 

Fig. 3.2 Simulated spike trains and membrane potential of a leaky integrate-and-fire neuron. The threshold is set at $\vartheta = 10 \text{ mV}$ and is indicated as a dashed line. (A) Constant input with strength $RI_{\text{ext}} = 8 \text{ mV}$, which is too small to elicit a spike. (B) Constant input of strength $RI_{\text{ext}} = 12 \text{ mV}$, strong enough to elicit spikes in regular intervals. Note that we did not include the form of the spike itself in the figure but simply reset the membrane potential while indicating that a spike occurred by plotting a dot in the upper figure.

Chapter 3 of the book resource page on the web. This program was used to produce the plots in Fig. 3.2.

Some parameters are set in Lines 4–9, and the meaning is indicated as comments after the assignment statements. The starting value for the time step counter, `t_step`, and the initial membrane potential are set in Line 12. The loop which starts at Line 13 integrates the equations until $t = 100$. For each of the steps in the loop, we increment the `t_step` counter, which is a integer value, to record the values during integration. In Line 15, we check if the membrane potential is larger than the threshold. Thus, the variable `s` indicates if there is a spike at this time step. The spiking variable, `s`, is used in the next line to execute two separate parts of the code. If `s=1`, the membrane potential is set to the resting potential since the second part of the code is zero. Otherwise, the leaky integrator dynamic, eqn 3.1 is executed. Such a way of coding conditional statements in MATLAB can be faster and more intuitive than using explicit `if` statements.

After the integration, the code plots the results with the remaining code. This example program shows how to place several plots into one figure window with the command `subplot`. There are different ways to use this command as specified when typing `help subplot` in the command window. In the shown example, we specified directly locations in coordinates of the MATLAB figure window. We chose this specific presentation with two subplots to highlight that only the subthreshold dynamics and the time of the spike are described by the basic IF model. However, sometimes the membrane potential is set to high values at the time of spiking before resetting it to the reset potential, in order to generate more common graphs.

3.1.3 Activation function

We can calculate the time an IF neuron needs for a constant input current to reach threshold, called the *first passage time*, from the solution in eqn 3.9. We can choose the time scale so that the last spike occurs at $t = 0$, which is also the time at which the membrane potential is reset, $v(t = 0) = v_{\text{res}}$. The first passage time, t^f , is then given by the time when the membrane potential

Table 3.1 Program if_sim.m

```

1  %% Simulation of (leaky) integrate-and-fire neuron
2  clear; clf;
3
4  %% parameters of the model
5  dt=0.1;          % integration time step [ms]
6  tau=10;          % time constant [ms]
7  E_L=-65;         % resting potential [mV]
8  theta=-55;       % firing threshold [mV]
9  RI_ext=12;        % constant external input [Ohm*mA=mV]
10
11 %% Integration with Euler method
12 t_step=0; v=E_L;
13 for t=0:dt:100;
14     t_step=t_step+1;
15     s=v>theta;
16     v=s*E_L+(1-s)*(v-dt/tau*((v-E_L)-RI_ext));
17     v_rec(t_step)=v;
18     t_rec(t_step)=t;
19     s_rec(t_step)=s;
20 end
21
22 %% Plotting results
23 subplot('position',[0.13 0.13 1-0.26 0.6])
24 plot(t_rec,v_rec);
25 hold on; plot([0 100],[-55 -55],'-');
26 axis([0 100 -66 -54]);
27 xlabel('Time [ms]'); ylabel('v [mV]')
28
29 subplot('position',[0.13 0.8 1-0.26 0.1])
30 plot(t_rec,s_rec,'.', 'markersize',20);
31 axis([0 100 0.5 1.5]);
32 set(gca,'xtick',[],'ytick',[])
33 ylabel('Spikes')

```

reaches the *firing threshold*, $v(t^f) = \vartheta$. This is calculated from eqn 3.9 to be

$$t^f = -\tau_m \ln \frac{\vartheta - RI}{v_{\text{res}} - RI}, \quad (3.10)$$

where \ln is the natural logarithm (logarithmus naturalis), which is the inverse of the exponential function. The inverse of the first passage time defines the *firing rate*,

$$\bar{r} = (t^{\text{ref}} - \tau_m \ln \frac{\vartheta - RI}{v_{\text{res}} - RI})^{-1}, \quad (3.11)$$

where we included an absolute refractory time t^{ref} . This function is the *activation function* of an IF neuron which is illustrated in Fig. 3.3 for several values of the reset potential, v_{res} , and absolute refractory time, t^{ref} . This function quickly reaches asymptotic linear behaviour after a quick onset with external currents exceeding the threshold value. A threshold-linear function is often used to approximate the activation function of IF neurons.

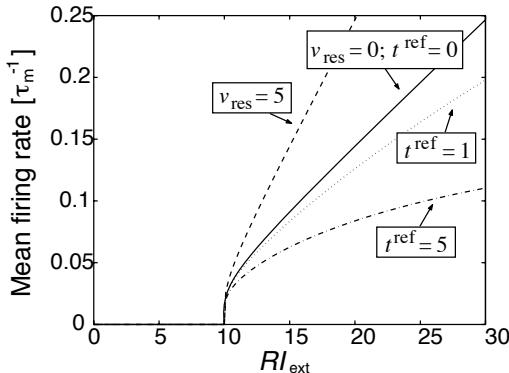


Fig. 3.3 Activation function of a leaky integrate-and-fire neuron for several values of the reset potential v_{res} and refractory time t^{ref} . The firing rate is given in units of the inverse time constant. The neuron fires only when the input is strong enough for the membrane potential to reach threshold ($RI_{\text{ext}} = 10$ in this case.) The mean firing rate increases for larger reset potentials and shorter refractory times.

The activation function of a neuron can be measured experimentally, similar to the simulations by applying constant external currents of different magnitude to an isolated neuron in a *Petri dish*. Such *in vitro* recordings can be used to examine the principal neuronal parameters that we need to understand for biologically faithful simulations. However, we have already seen that such activation functions also depend on noise in the system, as was illustrated for a Hodgkin–Huxley neuron in Fig. 2.11. We will now discuss some noise models that can be used to revise this activation function.

3.1.4 The spike-response model

In the previous section we integrated the equation describing simple IF neurons with constant currents driving the neurons, and we calculated the responses of the neurons to very short current pulses. When thinking about information transmission in neurons we are much more interested in time-varying input currents, such as those generated by presynaptic spike trains. We can formally integrate the dynamic differential equation 3.1 for any time-dependent inputs. To show this, we use our previous results showing the response of neurons

to very short external current pulses. The idea of the solution is to express an arbitrary external current stream $I(t)$ as a collection of many very short pulses modulated by the strength of the input signal at the particular time. The response to each current pulse is an exponential function, as we have seen before. The general solution can thus be expressed as a sum over all the exponential responses to very short current pulses. The sum is actually an integral because we have a term for each infinitesimal time step. The time course of an IF neuron in response to an arbitrary input current $I(t)$ can be written as an integral equation,

$$v(t) = R \int_0^\infty e^{-s/\tau_m} I(t-s) ds. \quad (3.12)$$

The integration variable s stands for all the times that precede the time t , and current pulses at such times influence the membrane potential at time t by an amount that depends exponentially on the time distance s . This means that more recent spikes have a larger influence on the membrane potential than more distant spikes.

The input current $I(t)$ can be generated with electrodes penetrating a neuron, and we could apply different forms of this current to study the responses of neurons. The following is an outline of the equations specifically for the interesting case of synaptic inputs generated by presynaptic firing. We need to take into account the resetting mechanisms in the IF neurons, which we ignored in eqn 3.12. We will account for the response in the membrane potential following a presynaptic spike and the change in the membrane potential following a postsynaptic spike by using two separate terms, labelled with ϵ and η , respectively. The description of the membrane potential is therefore split into the following components³

$$v(t) = \sum_j \sum_{t_j^f} w_j \epsilon(t - t^f, t - t_j^f) + \sum_{t^f} \eta(t - t^f). \quad (3.13)$$

This equation includes a sum over several ϵ terms, one for each synapse multiplied by the corresponding weight value. The changes in membrane potential described by the ϵ terms can depend, in general, on the last postsynaptic spike at time t^f if the ion channels are voltage-dependent. The main feature encoded in this ϵ term is the influence of the postsynaptic membrane potential on the firing history of the individual presynaptic spikes t_j^f , as determined by eqn 3.12. We can include in this term the specific form of the input current generated by a synaptic event that we parametrized by the α -function in Chapter 2. This can be written as

$$\epsilon(t - t_j^f) = R \int_0^\infty e^{-s/\tau_m} \alpha(t - t_j^f - s) ds, \quad (3.14)$$

for a synaptic input at synapse j .

The second term in eqn 3.13 (the η term) describes the reset of the membrane potential after a postsynaptic spike has occurred. We can write this in a form similar to that of the expression for the synaptic response, by including a short negative delta current with the strength of the threshold

$$RI_{\text{res}} = -\vartheta \delta(t - t^f). \quad (3.15)$$

³The membrane potential of the leaky integrate-and-fire model can be expressed as the sum of the individual potentials η and ϵ due to the linearity of the differential equation 3.1.

The integral 3.12 for this special reset current is given by

$$\eta(t - t^f) = -\vartheta e^{-(t-t^f)/\tau_m}. \quad (3.16)$$

The firing time t^f , written without an index, is the firing time of the postsynaptic neuron, given by solving the equation

$$v(t^f) = \vartheta, \quad (3.17)$$

in contrast to the presynaptic neuron firing times that have an index, t_j^f . Note that the functions η and ϵ can often be solved analytically for particular choices of reset mechanisms and α -functions, respectively. Some examples are summarized in Table 3.2. The integrated form of the IF neurons, defined by eqns 3.17 and 3.13, is called the *spike-response model*. We will use this model to derive descriptions of the average behaviour of populations of neurons later in this chapter.

3.1.5 The Izhikevich neuron

The subthreshold dynamic of leaky IF neurons, eqn 3.1, is linear in v and is therefore beneficial for analytic calculations and is efficient in numerical implementations. The drawbacks of the basic IF model are that it might not sufficiently approximate⁴ the subthreshold dynamic of a neuronal membrane potential and that it does not include the variety of response patterns seen in real neurons. While there are, of course, more detailed models, as we discussed in Chapter 2, their computational demand is often prohibitive for use in large-scale network simulations. However, *Eugene Izhikevich* has proposed an intermediate model which is computationally efficient while still being able to capture a large variety of response properties of real neurons.

This model neuron is described by two coupled differential equations, one for the membrane potential v , and one for a recovery variable u ,

$$\frac{dv(t)}{dt} = 0.04v^2(t) + 5v(t) + 140 - u + I(t) \quad (3.18)$$

$$\frac{du(t)}{dt} = a(bv - u), \quad (3.19)$$

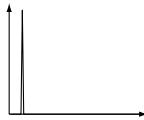
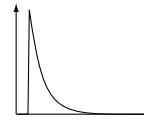
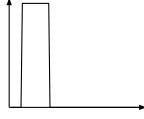
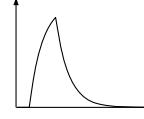
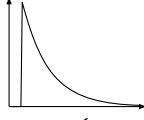
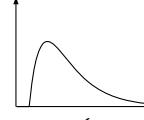
along with the corresponding reset conditions,

$$v(v > 30) = c \text{ and } u(v > 30) = u - d. \quad (3.20)$$

The equations describe the membrane potential v in mV for a given external current I in mA, and the time scale corresponds to ms, and includes four parameters a, b, c , and d . In contrast to the IF neuron, and more consistent with real neurons and Hodgkin–Huxley type models, this model does not have a constant firing threshold. The reset voltage of $v = 30$ is far above the regime where spike generation could be stopped by the removal of external current. The model does, therefore, better incorporate the critical regime of spike initiation. Furthermore, through the inclusion of a recovery variable that models the inactivation of sodium channels and hyperpolarizing potassium channels, typical spike patterns of biological neurons can be simulated. The

⁴Whether it is sufficient or not depends on the scientific question under investigation.

Table 3.2 Examples of spike response functions (right column) describing the membrane potential in response to the corresponding forms of postsynaptic potentials (left column) used in simulations and analytical models. The firing time of the presynaptic neuron is t^f , and t^d is the duration of the α -pulse in the second example. τ^m and τ^s are time constants

Effective potential after a single presynaptic spike	Spike-response function
 $\alpha(t) = \delta(t - t^f)$	 $\epsilon(t) = \begin{cases} 0 & \text{for } t \leq t^f \\ e^{-\frac{(t-t^f)}{\tau^m}} & \text{for } t > t^f \end{cases}$
 $\alpha(t) = \begin{cases} 0 & \text{for } t \leq t^f \\ \frac{1}{\tau_m} & \text{for } t^f < t \leq t^d \\ 0 & \text{for } t > t^d \end{cases}$	 $\epsilon(t) = \begin{cases} 0 & \text{for } t \leq t^f \\ 1 - e^{-\frac{(t-t^f)}{\tau_m}} & \text{for } t^f < t \leq t^d \\ (1 - e^{-\frac{t^d}{\tau_m}}) e^{-\frac{(t-t^f-t^d)}{\tau_m}} & \text{for } t > t^d \end{cases}$
 $\alpha(t) = \begin{cases} 0 & \text{for } t \leq t^f \\ \frac{1}{\tau_s} e^{-\frac{(t-t^f)}{\tau_s}} & \text{for } t > t^f \end{cases}$	 $\epsilon(t) = \begin{cases} 0 & \text{for } t \leq t^f \\ \frac{e^{-\frac{(t-t^f)}{\tau_m}} - e^{-\frac{(t-t^f)}{\tau_s}}}{1 - (\tau_s/\tau_m)} & \text{for } t > t^f \end{cases}$

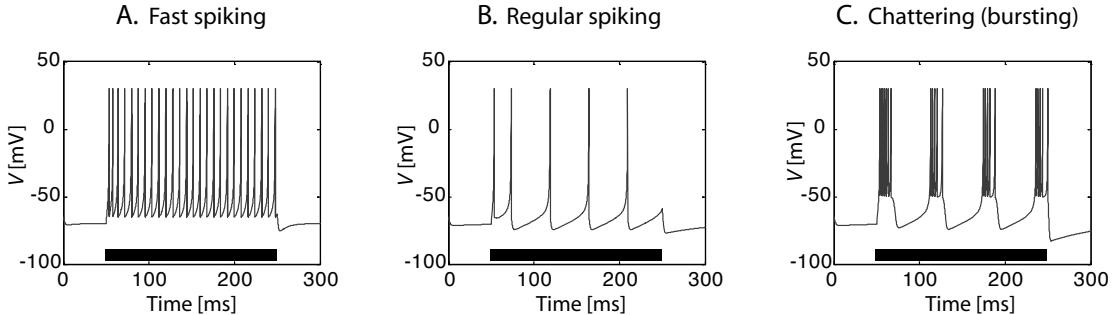


Fig. 3.4 Different spike pattern of a Izhikevich neuron by varying some parameters around their base values $a = 0.02$, $b = 0.2$, $c = -65$ and $d = 2$. The bar on the bottom of the graphs indicates the times at which a constant current of 10 mA was applied. (A) Fast spiking results from a fast recovery constant, $a = 0.1$. (B) Regular spiking with a large reset value of the recovery variable, $d = 8$. (C) Bursting behaviour simulated with $d = 2$ and a high value for the voltage reset, $c = -50$.

constant values in eqn 3.18 are chosen so that the model can fit a large variety of neural behaviour.

The four parameters a , b , c , and d can be set to simulate different type of neurons. Fig. 3.4 shows examples with the recommended base values of $a = 0.02$, $b = 0.2$, $c = -65$ and $d = 2$, unless stated otherwise. The parameter a sets the time scale of the recovery variable with low values corresponding to slow recoveries. For example, a value of $a = 0.1$ results in the fast spiking shown in Fig. 3.4A, where the bar at the bottom indicates the interval when a current of $I = 10$ was applied. Such fast spiking with little spike-time adaptation is similar to the basic IF neuron with a small time constant and is common in some types of inhibitory neurons. More common for excitatory neurons in the cortex is a spike rate with fatigue to constant input, like that shown in Fig. 3.4B. To simulate this neuron, the base parameters were used except for the parameter describing the reset of the recovery variable, d , which was set to $d = 8$. An interesting bursting behaviour (Fig. 3.4C) can be seen when using a high value for the voltage reset parameter, $c = -50$, and moderate recovery variable reset of $d = 2$. The parameter b , not varied in these simulations, describes the sensitivity of the recovery variable to fluctuations of the membrane potential. This model is used in network simulations discussed in Section 5.3 where the corresponding code can be found.

3.1.6 The McCulloch–Pitts neuron

Before leaving the section which introduces spiking neurons, we should finally mention one of the oldest and simplest neuron models, which was introduced by Warren McCulloch and Walter Pitts in 1943. This model has the form of a logical unit, which sums input values, x_i^{in} , to determine the net input,

$$h = \sum_i x_i^{\text{in}}. \quad (3.21)$$

The unit becomes active if the net input is larger than a threshold value, Θ ,

$$x^{\text{out}} = \begin{cases} 1 & \text{if } h > \Theta \\ 0 & \text{otherwise} \end{cases}. \quad (3.22)$$

Many information-processing capabilities of neural networks can be discussed and demonstrated with McCulloch–Pitts units. Before using this model in later chapters we should therefore say a few words about the interpretation of this model. While McCulloch and Pitts certainly had neurons in mind, it is clear that this model does not capture the precise time course of a membrane potential. Rather, this model should be seen as a model for operations in discrete time steps. While this time step could, in principle, be very small to capture rapidly changing signals, it is also clear that McCulloch and Pitts considered computational tasks of neurons rather than the description of physical properties of neurons. A better view of the relevant scale of the time step is therefore several tens of milliseconds, and we could interpret activity during this time step as the occurrence of one or more spikes during this interval. If we want to take different numbers of spikes in such a time step into account, then we might want to change the activation function (eqn 3.22) to allow a more graded output. The output would then represent a temporal average of spike counts.

The use of temporal averaging by the brain has been questioned, since responses would then be much longer than observed experimentally. However, we could go a step further in our abstraction and consider the logical unit of McCulloch and Pitts as a reflection of a population of functional connected neurons which will become active in response to a specific input. Activation of this population does then reflect the presence of certain input patterns, and we can imagine that individual neurons can contribute to a number of populations, as envisioned by *Donald Hebb* for his cell assemblies. The population approach is an important contribution to brain modelling. Such models are formally introduced in Section 3.4, and the computational abilities of such logical units are explored in Chapter 6. While the interpretation of the the McCulloch–Pitts neuron is not always clear, this should not distract from the importance of this model to illuminate computational principles in brain processing, including spike processing in a discrete time model.

3.2 Spike-time variability ◊

3.2.1 Biological irregularities

Neurons in the brain do not fire regularly, rather, they seem extremely noisy. Neurons that are relatively inactive emit spikes with low frequencies that are very irregular. Also, high-frequency responses to relevant stimuli are often not very regular. Single-cell recordings transmitted to a speaker sound very much like the irregular ticking of a Geiger counter when exposed to radioactive material. A histogram of the interspike intervals of one cortical cell is shown in Fig. 3.5A. This prefrontal cell fired around 15 spikes/s without a noticeably task-relevant pattern. The interspike interval distribution shows great variability. The firing pattern of this cortical cell is therefore not well described with the constant interspike intervals generated by the simple IF neurons studied

in Section 3.1. A convenient measure of the variability of spike trains is the *coefficient of variation*, which is defined by the ratio of the standard deviation σ and the mean μ ,

$$C_V = \frac{\sigma}{\mu}. \quad (3.23)$$

This value is $C_V \approx 1.09$ for the cell data shown in Fig. 3.5A. Recordings in the brain often show a high value of variability such as $C_V \approx 0.5–1$ for regularly spiking neurons in the *primary visual cortex* (V1) and the *medial temporal lobe* (MT), as pointed out by Softky and Koch (1993). Regular firing of an IF neuron has an inconsistent value of $C_V = 0$.

The distribution of the interspike intervals shown in Fig. 3.5A shows an exponentially decaying tail after a rapid onset determined by the refractory period. The exponential distribution, or more formally the *probability density function* of this distribution, given by

$$\text{pdf}^{\text{exponential}}(x; \lambda)(x) = \lambda e^{-\lambda x}, \quad (3.24)$$

has only one parameter, $\lambda = 1/b$, where b is equal to the mean, and, at the same time, the standard deviation of an exponentially distributed random variable. The coefficient of variation for the exponential distribution is therefore $C_V = 1$. The number of events, when the time between events is exponentially distributed, is given by the Poisson distribution,

$$\text{pdf}^{\text{Poisson}}(x; \lambda)(x) = \sum_{i=1}^x \lambda^i \frac{e^{-\lambda}}{i!}, \quad (3.25)$$

and spike trains in computational studies are often generated with a *Poisson process*. A Poisson process is a process that results in a variable being Poisson distributed. An important characteristic of a Poisson process is that there is no memory of past events. That is, at any given time there is an equal likelihood that the next spike occurs. This is, of course, not the case in the refractory time, which accounts for reductions in the number of small interspike intervals. The exponential distribution is shown in Fig. 3.5B, together with a corresponding simulation of a modified Poisson process which is described in more detail below.

The heuristic modelling of spike trains discussed so far does not describe the processes producing the noise. Also, there are several further factors that we need to take into account, including inhibition in the network and other contributing factors from irregularities in the biological system. To begin with, the input to the system, such as sensory input from the environment, is not constant as is assumed in the simulations above. Moreover, on a single-cell level we expect many structural irregularities to contribute to the irregular behaviour of neurons. These include the diffuse propagation of neurotransmitters across the synaptic cleft, the opening and closing of ion channels, the propagation of the membrane potential along dendrites with varying geometries, the nature of biochemical processes, and the probabilistic nature of transmitter releases by axonal spikes. It is beyond the scope of this book, and beyond the scope of most models in computational neuroscience, to try to describe the nature of these irregularities in detail. Instead, we often incorporate the sum of all these irregularities into our models by including noise in the simulations.

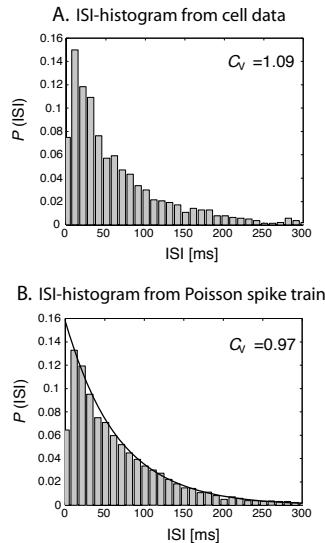


Fig. 3.5 Normalized histograms of interspike intervals (ISIs). (A) Data from recordings of one cortical cell (Brodmann's area 46) that fired without task-relevant characteristics with an average firing rate of about 15 spikes/s. The coefficient of variation of the spike train is $C_V \approx 1.09$ [data courtesy of Stefan Everling]. (B) Simulated data from a Poisson-distributed spike train in which a Gaussian refractory time has been included. The solid line represents the probability density function of the exponential distribution when scaled to fit the normalized histogram of the spike train. Note that the discrepancy for small interspike intervals is due to the inclusion of a refractory time.

Table 3.3 Program poisson_spiketrain.m

```

1  %% Generation of Poisson spike train with refractoriness
2  clear; clf;
3  fr_mean=15/1000;      % mean firing rate
4  %% generating poisson spike train
5  lambda=1/fr_mean;      % inverse firing rate
6  ns=1000;                % number of spikes to be generated
7  isi1=-lambda.*log(rand(ns,1)); % generation of expo. distr. ISIs
8  %% Delete spikes that are within refractory period
9  is=0;
10 for i=1:ns;
11     if rand>exp(-isi1(i)^2/32);
12         is=is+1;
13         isi(is)=isi1(i);
14     end
15 end
16 %% Ploting histogram and calculating cv
17 hist(isi,50);           % Plot histogram of 50 bins
18 cv=std(isi)/mean(isi) % coefficient of variation

```

Simulation

Before moving on to discuss specific noise models, we show here briefly a program, listed in Table 3.3 and implemented in file `poisson_spiketrain.m`, to generate Poisson spike trains. Poisson spike trains are characterized by exponentially distributed interspike intervals. Exponentially distributed random numbers can be generated from uniformly distributed random numbers by taking the negative logarithm of the uniform random number and multiplying it by the parameter of the distribution (see Appendix C on how to transform random numbers). This can be implemented with the MATLAB command line

```
r=-lambda*log(rand(n1,n2))
```

To simulate refractoriness of a simulated neuron producing such spike trains, we deleted in the program the firing times with a certain probability that depends on the interspike interval. This does not affect most of the spikes in a low-frequency spike train, but removes some of the unrealistically large number of small interspike intervals. This program also features another specialized plotting routine, that of plotting histograms with the command `hist`. Using this function without assigning it to variable bins data produces a bar plot. The functions `hist` can, however, be used more generally to bin values, and a separate function, the MATLAB function `bar`, can be used to produce bar plots. The program produces figures similar to the one used in Fig. 3.5B.

3.2.2 Noise models for IF neurons

How can we include noise in the neuron models to describe some of the stochastic processes within neuronal responses? For simplicity, we will discuss this for the IF model, although similar procedures can be applied to other models. We want to determine the stochastic firing times of neurons. In order to do so we can use three principal methods to include a stochastic component in the IF model. These are illustrated in Fig. 3.6.

The noise models commonly used for IF neurons can be summarized as:

- (1) **Stochastic threshold:** We can replace the threshold, which the membrane potential has to pass in order to generate a spike, with a noisy threshold

$$\vartheta \rightarrow \vartheta + \eta^{(1)}(t). \quad (3.26)$$

- (2) **Random reset:** We can reset the membrane potential to a random reset potential

$$v^{\text{res}} \rightarrow u^{\text{res}} + \eta^{(2)}(t). \quad (3.27)$$

- (3) **Noisy integration:** The integration mechanisms in neurons can be noisy such that the leaky integrator of the IF neurons may be better described by a stochastic differential equation (which is equivalent to saying that the integrator is not noisy but it integrates noisy inputs)

$$\tau_m \frac{dv}{dt} = -v + RI_{\text{ext}} + \eta^{(3)}(t). \quad (3.28)$$

With appropriate choices for the distribution function of the random variables $\eta^{(1)}$, $\eta^{(2)}$, and $\eta^{(3)}$ we can produce equivalent results for the stochastic processes of a neuron, although the same probability distribution for each random variable can produce different results for each noise model. In practice, we want to choose distributions that are appropriate for capturing experimental data, so which noise model to choose is more a question of convenience. For analytical treatments it is often most convenient to use a random threshold model. Although there is little evidence that the firing thresholds of real neurons change over time, this noise model is equivalent to the other noise models, so modelling of stochastic processes in neurons can be done in this way. Numerical studies frequently use noisy input to model stochastic processes in the brain. This model has the simple interpretation of noisy synaptic transmission that can be observed in real neuronal systems. Analytical treatments of this model are difficult, although it is straightforward to integrate noise in this fashion in numerical studies, as shown below.

3.2.3 Simulating the variability of real neurons

While the choice of the noise model depends primarily on convenience, an important remaining question is the appropriate choice of the random process, including the probability distribution, and the time scale on which those fluctuations are relevant. We cannot give general answers to these questions as the particular choice depends strongly on the nature of the question and the particular neural system under investigation. Appropriate choices have to be

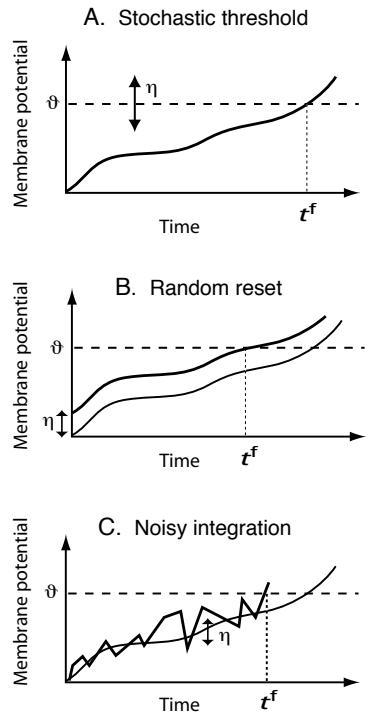


Fig. 3.6 Three different noise models of integrate-and-fire neurons. (A) Stochastic threshold, (B) random reset, and (C) noisy integration. t^f is the time of firing, ϑ the firing threshold, and η is a random variable [adapted from W. Gerstner, in *Pulsed neural networks*, Maass and Bishop (eds), MIT Press (1998)].

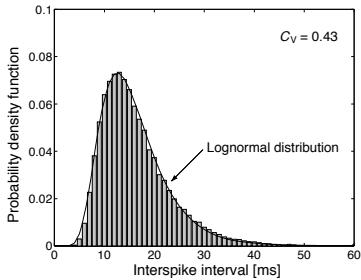


Fig. 3.7 Simulated interspike interval (ISI) distribution of a leaky IF neuron with threshold $\vartheta = 10$ mV and time constant $\tau_m = 10$ ms. The underlying spike train was generated with noisy input around the mean value $R\bar{I} = 12$. The fluctuations were therefore distributed with a standard normal distribution. The resulting ISI histogram is well approximated by a lognormal distribution (solid line). The coefficient of variation of the simulated spike train is $C_V \approx 0.43$.

made to fit experimental data whenever the precise form of fluctuations is relevant. In the following, we only give a flavour of the effects of including noise in the IF model by considering noisy input (noise model 3). In the following example we use a normally distributed input current produced by adding white noise to a constant input current,

$$I_{\text{ext}} = \bar{I}_{\text{ext}} + \eta \quad \text{with } \eta \in N(0, 1). \quad (3.29)$$

Normally distributed input signals are a good approximation when we consider independent synaptic input from many equally distributed neurons as stated by the *central limit theorem* (see Appendix C). The interspike interval (ISI) distribution of an IF neuron with mean input $R\bar{I}_{\text{ext}} = 12$ mV and threshold $\vartheta = 10$ mV is shown in Fig. 3.7.

This distribution is approximated well by a *lognormal distribution*,

$$\text{pdf}^{\text{lognormal}}(x; \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\log(x)-\mu)^2}{2\sigma^2}}. \quad (3.30)$$

A fit of the data to this distribution is shown as a solid line in Fig. 3.7. The coefficient of variation of these data is $C_V \approx 0.43$, approaching the lower end of cortical variability of interspike intervals. This is remarkable since we considered only one source of noise. We can, for example, assume this noise results from the noisy internal mechanisms of integration within the neuron, and we could then consider, in addition, noisy input from the variability in the input spike trains themselves. There are several other ways to increase the variability of spikes in simple IF neurons, such as using partial and noisy reset after a spike has occurred. This effectively increases the gain in the relation between input current and output spike frequency and has been argued to describe the variability seen in experiments very well.

The last example we want to discuss is an IF neuron that is driven by independent presynaptic spike trains with exponential interspike intervals (Poisson spike trains). Results from such simulations are shown in Fig. 3.8. There, we have taken as input 500 Poisson-distributed spike trains with refractory corrections as discussed above. The mean firing rate of the input neurons was set to 20 Hz, which was lowered slightly to 19.3 Hz due to the Gaussian refractory time with a 2 ms time constant. We took only 500 independent presynaptic spike trains into account as they should represent the fraction of presynaptic neurons that are active in a particular task (for example, 10% of the inputs to a neuron with 5000 presynaptic neurons). Each presynaptic spike was set to elicit an EPSP in the form of an α -function (eqn 2.1) with amplitude $w = 0.5$ and time constant of 2 ms for all synapses. The synaptic input of all the input spike trains was then large enough to keep the average incoming current, that is, the sum over all EPSPs, larger than the firing threshold of the neuron.

The sum of the EPSPs for the first 1000 ms in this experiment is illustrated by the upper curve in Fig. 3.8A, and the firing threshold is indicated as a dashed line. The average exceeds the firing threshold of the IF neuron, and this, in turn, results in a regular firing pattern with a high firing rate of around 118 Hz. This result is similar to the response of an IF neuron to a constant input as discussed in Section 3.1.2. The ISI distribution, plotted in Fig. 3.8B, has low variance, and the coefficient of variation of this model neuron is only

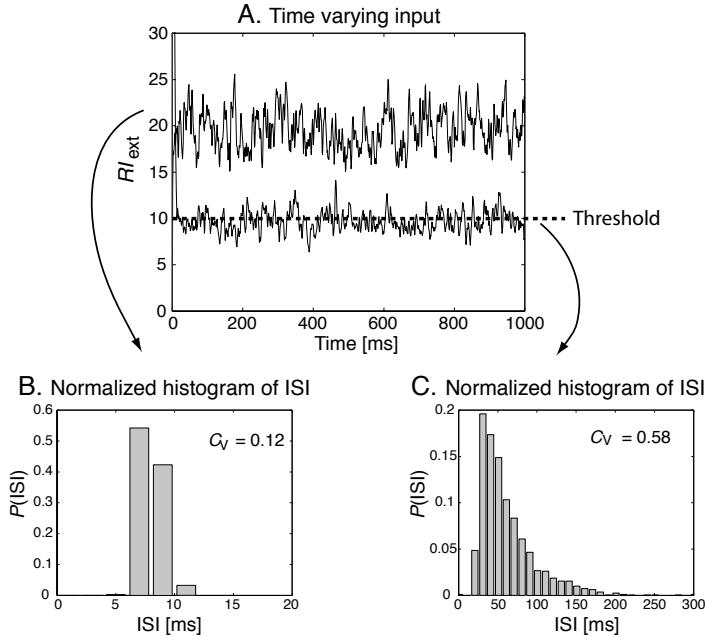


Fig. 3.8 Simulations of an IF neuron that has no internal noise but is driven by 500 independent incoming spike trains with a corrected Poisson distribution. (A) The sums of the EPSPs, simulated by an α -function for each incoming spike with amplitude $w = 0.5$ for the upper curve and $w = 0.25$ for the lower curve. The firing threshold for the neuron is indicated by the dashed line. The ISI histograms from the corresponding simulations are plotted in (B) for the neuron with EPSP amplitude of $w = 0.5$ and in (C) for the neuron with EPSP amplitude of $w = 0.25$.

$C_V = 0.12$. Note that these simulations did not include any noise in the neuron model itself; they only included noise in the driving input currents. Noise in the neuron model would further increase the coefficient of variation.

If we lower the effect of each incoming spike by lowering the amplitude of the α -function to $w = 0.25$, then we get an average postsynaptic current, which is shown for the first 1000 ms of the simulation in the lower curve of Fig. 3.8A. The average sum of EPSPs was $RI_{ext} = 9.7$ mV in this simulation, which is just below the firing threshold of the neuron. The fluctuations due to the random processes of the incoming spike trains are then crucial, and the neuron shows an irregular firing pattern with an average firing rate of 16 Hz. The corresponding ISI histogram is shown in Fig. 3.8C. The coefficient of variation is $C_V = 0.58$, exceeding the lower boundary found in experiments. The question is, of course, how a neuron can adjust the combined strength of incoming spikes to function in this balanced, more biologically realistic, regime. It is rather difficult to achieve the right level of input strength, as it seems to require a fine-tuning of the strength parameter by hand. The question of *synaptic scaling* must be investigated further, and possible factors that need to be considered include inhibition in networks, synaptic plasticity, as well as explicit scaling mechanisms.

3.2.4 The activation function depends on input

A final, important, point is that the activation or gain function of a neuron depends on variations in the input spike train. This can be shown analytically for an IF neuron that is driven by a noisy current with normally distributed

values. The IF dynamics of eqn 3.1 is then a stochastic differential equation as we have argued before in connection with noise model 3. The stochastic differential equations of the leaky IF neuron can still be solved formally. The first passage time as given in eqn 3.10 is then a random variable for which we can calculate the *mean first passage time*. These calculations were carried out by *Tuckwell* (see ‘Further reading’ in Chapter 2), who showed that the average firing rate for a stochastic IF neuron is given by

$$\bar{r} = (t^{\text{ref}} + \tau_m \int_{(v^{\text{res}} - R\bar{I}_{\text{ext}})/\sigma}^{(\vartheta - R\bar{I}_{\text{ext}})/\sigma} \sqrt{\pi} e^{v^2} [1 + \text{erf}(v)] dv)^{-1}, \quad (3.31)$$

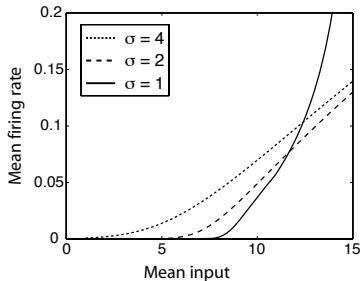


Fig. 3.9 The activation function of an IF neuron driven by an external current that is given a normal distribution with mean $\mu = R\bar{I}$ and variance σ^2 . The reset potential was set to $v_{\text{res}} = 5\text{mV}$ and the firing threshold of the IF neuron was set to $\vartheta = 10\text{mV}$. The three curves correspond to three different variance parameters.

where σ is the variance of the Gaussian signal and erf is the error function as described in Appendix C, eqn C.13. The firing rate is not only a function of the mean input current, but also depends on higher moments (for example, variance, skewness, etc.) of the distribution describing the input signal. The important result to keep in mind is that the mean firing time of an IF neuron depends on the precise form of the input spike train, not only on the mean firing rate of the inputs,

$$\bar{r} = \bar{r}(\mu, \sigma, \dots). \quad (3.32)$$

Some examples of the mean firing rates of an IF neuron as a function of the mean firing rates, μ , of presynaptic spike trains, as specified by eqn 3.31, are illustrated in Fig. 3.9. This figure shows the response curves for three different values of the standard deviation *sigma* of presynaptic firings. The activation function for a low-variance input spike train has a sharp transition, as we have seen before in Fig. 3.3, and the firing rate of the neuron will soon approach its maximal firing rate, as determined by the inverse of the absolute refractory time for means of the input current exceeding the firing threshold. Also note that the effective threshold, the point where the strong increase of the firing rate with external input starts, is lower than the hard threshold imposed on the IF neuron model. With increasing variance, the strong non-linear response is ‘linearized’, similarly to the linearization by noise in the Hodgkin–Huxley model (see Fig. 2.11). In the population models introduced below, we can take the structure of the driving signals into account with activation functions that depend on the mean input current as well as the higher moments of the distribution of the input signals. This is, unfortunately, often neglected in the application of such models in the literature.

3.3 The neural code and the firing rate hypothesis

Neuroscientists try to decipher the *neural code* by searching for reliable correlations between firing patterns and behavioural consequences, or correlations, between sensory stimuli and neural activity patterns. The most robust findings typically show modulations of the firing rate of neurons with sensory stimuli. For example, one of the first scientists to use microelectrode recordings in the 1920s, after sufficient amplification of the small electrical signals became available through vacuum tubes, was the English physiologist *Edgar Douglas*

Adrian. One of the first effects he recognized was that the number of spikes of a neuron often increases when increasing the strength of a stimulus. An increasing firing rate is easily detectable and still dominates the neurophysiological search for stimuli that ‘drive’ a neuron.

An example of a rate code that was explored by Adrian is that of the stretch receptor on the frog muscle which increases with increasing weights on the muscle, as shown in Fig. 3.10. In general, firing rates of sensory neurons increase considerably in a short time interval following the presentation of an effective stimulus to the recorded neuron. The response of a neuron to various stimuli is sometimes captured in *response curves*. For example, the response curve of a sensory neuron in the primary visual cortex, also called the *tuning curve* of the neuron, is shown in Fig. 3.11. Simple cells in this visual area respond to orientations of bars moved through the *receptive field* of the neuron, which is the area in the physical world for which this neuron is responsive. While firing rates dominate physiological descriptions of neuronal responses, and much of the further discussions in this book, we should not overlook the fact that other parts of spike patterns can convey information. We therefore take a short side tour into this topic.

3.3.1 Correlation codes and coincidence detectors

Information processing in the brain usually includes modulations of firing rates. This is not very surprising as we expect a variation in the number of spikes with varying input currents. However, we want to ask if only the firing rate is used in the brain to convey information. A rare example where the firing rate does not show the relevant information is shown in Fig. 3.12. The figure displays the response of two neurons in the primary auditory cortex to a 4 kHz tone with an amplitude envelope shown at the top. Fig. 3.12B shows the average firing rate at 5 ms intervals over many trials for each neuron. No significant variation of the firing rate can be seen in either of the neurons. However, some stimulus-locked variation in the relative spiking of the two neurons can be seen when plotting the rate of spikes from one neuron that occurred within a short fixed interval of the spiking of the other reference neuron (Fig. 3.12C). This rate indicates the probability of co-occurrence of the spikes of the two neurons, which is a nice example where behavioural correlates can only be seen in the correlation between the firing patterns of two neurons.

In contrast, for a receiving leaky IF neurons, the close temporal proximity of spikes is relevant. A *perfect integrator*, which sums the input of synaptic input without leakage, is illustrated for the case of two driving inputs in Fig. 3.13A. The membrane potential accumulates the synaptic currents triggered by the presynaptic spikes and therefore counts the number of presynaptic spikes since the last reset of the membrane potential. In the case of a leaky integrator, illustrated in Fig. 3.13B, the membrane potential decays after an increase caused by presynaptic spikes, and the neuron becomes sensitive to the relative timing of spikes of different presynaptic neurons. Such a leaky integrator with a small time constant can be used as a *coincidence detector*.

A neuron generates a spike after the membrane potential reaches a threshold, at which time the membrane potential must be reset (not included in Fig. 3.13).

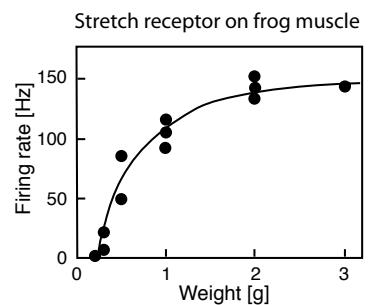


Fig. 3.10 Data from Adrian’s original work showing the firing rate of a frog’s stretch receptor on a muscle as a function of the weight load applied to the muscle [redrawn from Adrian, *J. Physiol. (Lond.)* 61: 49–72 (1926)].

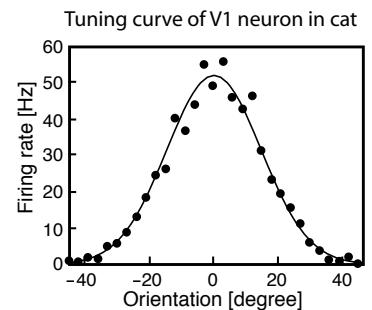


Fig. 3.11 Response (tuning curve) of a neuron in the primary visual cortex of the cat as a function of the orientation of a light stimulus in the form of a moving bar [data from Henry, Dreher, and Bishop, *J. Neurophys.* 37: 1394–1409 (1974)].

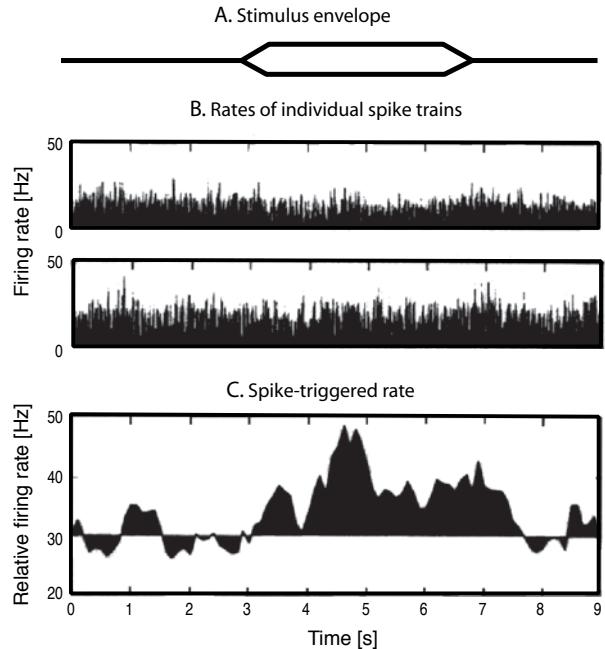


Fig. 3.12 An example of the response of some neurons in the primary auditory cortex that do not show significant variations in response to the onset of a 4 kHz tone with the amplitude envelope shown in (A). (B) Average firing rates in 5 ms bins of two different neurons. (C) Spike-triggered average rate that indicates some correlation between the firing of the two neurons that is significantly correlated to the presentation of the stimulus [from DeCharms and Merzenich, *Science* 381: 610–13 (1996)].

With the thresholds indicated by dashed lines in Fig. 3.13, both neurons, the integrator and the coincidence detector, would fire at the same time. However, the reason for the firing would be very different in the two cases. In the first case, that of a perfect integrator, the neuron fires because four presynaptic spikes occurred since the last firing of the neuron. This neuron would also fire at this time if one of the two last simultaneous spikes occurred at an earlier time. In contrast, the spike of the leaky integrator only occurs because of the occurrence of two simultaneous presynaptic spikes. With a higher firing threshold, larger than the effect of the sum of two simultaneous EPSPs, it is also possible to employ such a leaky integrator neuron to detect the coincidence of more than two spikes. The time window of coincidence depends on the decay constant and the time course of the EPSPs. Thus, temporal proximities of spikes can make a difference in the information processing of the brain.

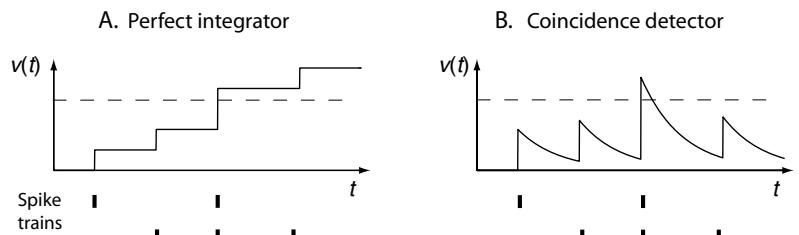


Fig. 3.13 Schematic illustration of (A) a perfect integrator and (B) a leaky integrator that can be utilized as coincidence detector. In this example the membrane potential $v(t)$ integrates short current pulses of the two spike trains shown at the bottom.

3.3.2 How accurate is spike timing?

It is a widely held belief that neural spiking is not very reliable, and that there is a lot of variability in neuronal responses. Another demonstration of spike-time variability, from experiments by Buračas and colleagues, is shown in Fig. 3.14 for responses of a cell in the middle temporal area (MT) that responds to the movement of visual stimuli. The top graph of Fig. 3.14A shows the response of the neuron in several trials to a stimulus with constant velocity, as indicated at the bottom. The middle graph shows the trial average. The data indicate a reliable initial neuronal response. After this initial response the neuron still fires rapidly; however, the times of these spikes are different in each trial. Some people search for a neural code in the firing pattern of neurons in response to a constant pattern. The data in Fig. 3.14A indicate that there might not be much information in the continuing firing pattern of the neuron due to the enormous variability.

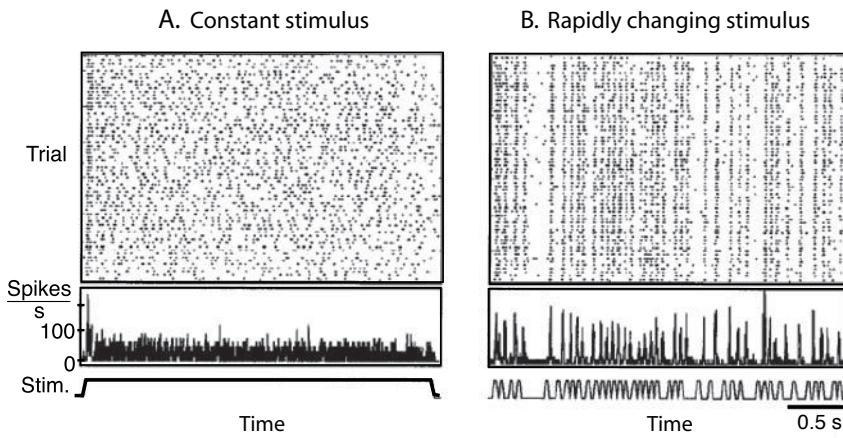


Fig. 3.14 Spike trains (top) and average response over trials (middle) of an MT neuron to a visual stimulus with either constant velocity (A) or altering velocity (B) as indicated in the bottom graph [adapted from Buračas *et al.*, *Neuron* 20: 959–69 (1998)].

Data like those shown in Fig. 3.14A lead to the impression that neuronal spike times are very variable and imprecise. However, this has to be viewed in relation to the experimental situation. Fig. 3.14B shows the response of the same neuron to a rapidly varying stimulus. The response of the neuron follows the variations in the stimulus rapidly with little jitter. Each neuron does not always respond with a spike to the changing velocity of the stimulus, and some spikes still occur in between the stimulus changes. However, the majority of spikes follow the changing stimulus rapidly. Thus, sensory events can elicit a rapid cascade of neuronal activity through a neuronal network which, in turn, can lead to rapid recruitment of neurons along processing pathways. In other words, populations of neurons can rapidly convey information in a neural network.

3.4 Population dynamics: modelling the average behaviour of neurons

Simulation of networks of many thousands of spiking neurons has become tractable on recent computer systems. However, even with the increasing power of digital computers, we are barely able to simulate neural systems with spiking neurons on the scale of functional units in the brain, not to mention models on a scale comparable to that of the central nervous system. Furthermore, as discussed in Chapter 1, our aim is not so much to reconstruct the brain in all its detail, but rather to extract the principles of its organization and functionality. Many of the models in computational neuroscience, in particular on a cognitive level, are therefore based on descriptions that do not take the individual spikes of neurons into account, but instead describe the average activity of neurons or neural populations.

In this section we introduce population models and discuss their relationship to populations of spiking neurons. The aim is to highlight under what conditions these common approximations are useful and faithful descriptions of neuronal characteristics. It is clear that *rate models* cannot incorporate all aspects of networks of spiking neurons. However, many of the principles behind information processing in the brain can be illuminated on the level of population models, and many of the features of population models have been confirmed with spiking neurons.

3.4.1 Firing rates and population averages

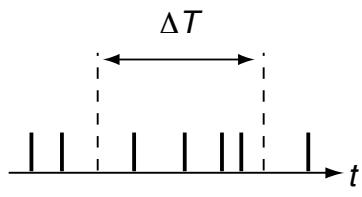


Fig. 3.15 Temporal average of a single spike train with a time window ΔT that has to be large compared to the average interspike interval.

It is important to recognize that neurophysiological recordings of single cells have been very successful in correlating the firing rates of single neurons with the behavioural perceptions, or responses, of a subject. It is common in physiological studies to derive an instantaneous firing rate with the help of a sliding window in the spike train. For example, as illustrated in Fig. 3.15, we can estimate the average temporal spike rate of a neuron with a fixed window of size ΔT ,

$$\begin{aligned}\nu(t) &= \frac{\text{number of spikes in } \Delta T}{\Delta T} \\ &= \frac{1}{\Delta T} \int_{t-\Delta T/2}^{t+\Delta T/2} \delta(t' - t^f) dt',\end{aligned}\quad (3.33)$$

where t^f is the firing time of the neuron. This defines, for small time windows, the *instantaneous firing rate*. It is also possible to calculate a weighted average with different *kernel functions* which often give smoother results. For example, it is common in physiological studies to use a Gaussian window that does not have the sharp boundaries of the rectangular time window above. The firing rate of a neuron at time t is then defined by

$$\nu(t) = \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{+\infty} \delta(t' - t^f) e^{(t'-t)^2/2\sigma^2} dt'. \quad (3.34)$$

The physiological estimates of the firing rates of single neurons often include an average over several trials under equal experimental conditions. Such averages

over repetitions are necessary because the firing of a neuron in a single trial is very noisy. This is a valid experimental estimation procedure as it has been found that such averages correlate well with behavioural responses. However, it is also clear that averaging over several trials is not an option for the brain, where responses to a single stimulus must be possible. Also, the temporal averaging explained above can only be employed by the brain in a limited way since the time windows of averaging have to be much smaller than the typical response time of the organism, which can be on the order of 100 ms. Thus, mainly population activity is relevant for information processing in the brain.

The brain does not rely on the information of a single spike train. Indeed, such reliance would make the brain very vulnerable to damage or reorganization. We therefore conjecture that there must be a subpopulation, or *pool of neurons*, with similar response properties, as illustrated in Fig. 3.16. The neurons in these subpopulations of a cortical module may act in a statistically similar way. This would explain why single-neuron recordings have been so successful in correlating single-neuron measurements to behavioural findings. Thus, we conjecture that the temporal average of single neurons measured in repeated physiological experiments approximates the neuro-computationally relevant average population activity $A(t)$ of neurons,

$$\begin{aligned} A(t) &= \lim_{\Delta T \rightarrow 0} \frac{1}{\Delta T} \frac{\text{number of spikes in population of size } N}{N} \\ &= \lim_{\Delta T \rightarrow 0} \frac{1}{\Delta T} \int_{t-\Delta T/2}^{t+\Delta T/2} \frac{1}{N} \sum_{i=1}^N \delta(t' - t_i^f) dt'. \end{aligned} \quad (3.35)$$

Since this average employs a sum over many neurons (in contrast to eqn 3.33), it is possible to use much smaller time windows. In the limit of very small time windows, we can also write the last equation in differential form,

$$A(t)dt = \frac{1}{N} \sum_{i=1}^N \delta(t' - t_i^f), \quad (3.36)$$

which we will use later to derive formulas for population dynamics.

It is difficult to verify this conjecture directly with experiments, as this would demand the simultaneous recording of many thousands of neurons. Recording with multiple electrodes is, to a large extent, beyond a current experimental feasibility, and brain imaging currently averages over too many neurons to be able to verify our conjecture. Local field potentials may come closest to such population recordings, when the populations are spatially localized. The conjecture is, however, supported indirectly by several known experimental facts, such as the existence of cortical columns where neurons with very similar responses are found. The remainder of this chapter is dedicated to the introduction of dynamic models for such populations.

3.4.2 Population dynamics for slow varying input

To describe the average behaviour of a pool of neurons, we divide the population into subpopulations of neurons of the same type, with similar response properties. For simplicity, we assume further that the neurons in this pool have

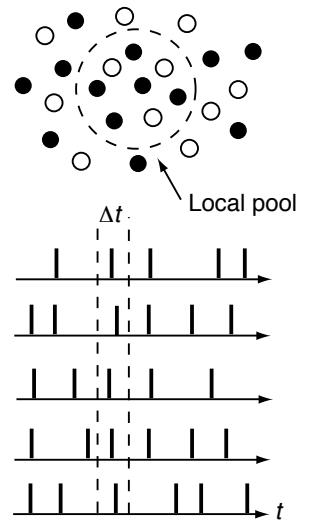


Fig. 3.16 Pool or local population of neurons with similar response characteristics. The pool average is defined as the average firing rate over the neurons in the pool within a relatively small time window [adapted from W. Gerstner, in *Pulsed neural networks*, Maass and Bishop (eds), MIT Press (1998)].

the same membrane time constant τ_m , the same mean numbers and synaptic efficiencies of afferents, receive the same input current I^{ext} , and do not interact. We now guess that the dynamics of such a neuronal pool can be described by a leaky integrator similar to the subthreshold dynamics of the individual IF neurons,

$$\tau \frac{dA(t)}{dt} = -A(t) + g(RI^{\text{ext}}(t)). \quad (3.37)$$

The function g is a *population activation function*, that describes the influence of the external current on the activation of the pool. We will discuss several activation functions in the next section. For now, it is sufficient to have a linear function $g(x) = x$ in mind. It is clear that the dynamic eqn 3.37 can only be an approximation of the pool dynamics. The motivation for this description is that, when at any instance in time only a small number of neurons in the pool is firing, we expect that the population dynamics are mainly characterized by the subthreshold dynamics to slowly varying input currents. The time constant in eqn 3.37 for slowly varying input currents should then be close to the average membrane time constants of the neurons in the pool.

The description given in eqn 3.37 is particularly appropriate when we analyse *asymptotic stationary states* of neuronal networks. *Stationary states* are states that do not change under the dynamics of the system, and by asymptotic we mean that these are the states after the initial transient behaviour has levelled off. Saying that the states do not change under the dynamics of the system can be expressed mathematically as $dA/dt = 0$. Including this in eqn 3.37 yields

$$A(t) = g(RI^{\text{ext}}(t)). \quad (3.38)$$

This is, of course, only true for constant input because the state cannot be stationary with varying input. However, the formula should also hold, to a good approximation, with slowly varying input. Many information-processing capabilities of neural networks have been studied in this limit, as shown in later chapters.

3.4.3 Motivations for population dynamics ◊

The population dynamic given by eqn 3.37 is the basis for many studies in cognitive neuroscience, and much research has gone into its justification. A lot of such studies use techniques developed in statistical physics, since the task has some similarity to deriving thermodynamic descriptions from the underlying physical movement of molecules. The principle is to start with a large number of coupled differential equations (spiking neurons in our case) and to find methods and approximations so that the large set of coupled equations can be replaced by a dynamic equation of the mean. Wilson and Cowan (1972) were among the first to argue in this way, and Brunel and Wang (2001) derived more advanced mean field models for neurons with different ion channels. We will not look at these derivations in detail, but we will outline briefly such arguments, as advanced by *Wulfram Gerstner* and *Leo van Hemmen*, which are generalizations of the Wilson–Cowan integral equations.

Recall from Section 3.1.4 that the membrane potential in the spike response model is given by

$$v_i(t) = \sum_{t^f} \eta(t - t^f) + \sum_j \sum_{t_j^f} w_{ij} \epsilon(t - t_j^f), \quad (3.39)$$

where we have ignored the possible dependence of the ϵ term on the postsynaptic firing, but included new indices i for the postsynaptic neurons, as we are now interested in a population of such neurons. We will again assume a specific kind of population of N neurons in which each neuron reacts in a similar way to input and has, on average, a synaptic efficiency specified by the constant w_0 ,

$$w_{ij} = \frac{w_0}{N}. \quad (3.40)$$

We also assume that there is no spike-time adaptation in the neurons, that the total number of neurons in the population stays constant, and that we, formally, have an infinitely large population of neurons in which only the means of the random variables matter. Using the definition of the population average (eqn 3.36) we can thus express the mean influence of the postsynaptic potential using the rate of the population as

$$v_\epsilon(t) = w_0 \int_0^\infty \epsilon(t') A(t - t') dt'. \quad (3.41)$$

Finally, we have to take noise into account, which can be done with either of the noise models mentioned in Section 3.2.2. In general, we can define a probability density $P_v(t|t^f)$ with which a neuron fires at time t when it has a membrane potential v and has fired previously at times t^f . The membrane potential does depend on the population rate, as specified in eqn 3.41. The population rate at time t can thus be expressed as the sum (or integral) of all the population rates at previous times, multiplied by the probabilities of firing at the corresponding times,⁵

$$A(t) = \int_{-\infty}^t P_v(t|t^f) A(t^f) dt^f. \quad (3.42)$$

This dynamic is exact in the limit of an infinite pool of neurons, and the equation can be solved for particular probability densities, $P_v(t|t^f)$, derived from the noise models. It is possible to use such descriptions directly in simulations, although it requires the calculation of large sums, which is computationally demanding. It is therefore still desirable to derive approximations of population dynamics in differential form, and the derivation from the integral equations makes it possible to precisely specify the assumptions that have to be made. *Wolfram Gerstner* has discussed this issue in great detail (see ‘Further reading’). He has shown that, in the *adiabatic limit*, in which the cell assembly responds only slowly to slowly varying inputs that do not cause specific collective phenomena of the neurons in the cell assembly (such as synchronization or phase locking), the population dynamics can be approximated with differential equations in the form of eqn 3.37 with an activation function of the form

$$g(x) = \frac{1}{t^{\text{ref}} - \tau \log(1 - \frac{1}{\tau x})}, \quad (3.43)$$

⁵See Gerstner in *Neural Computation* 12:43–89, 2000, and references therein, for details.

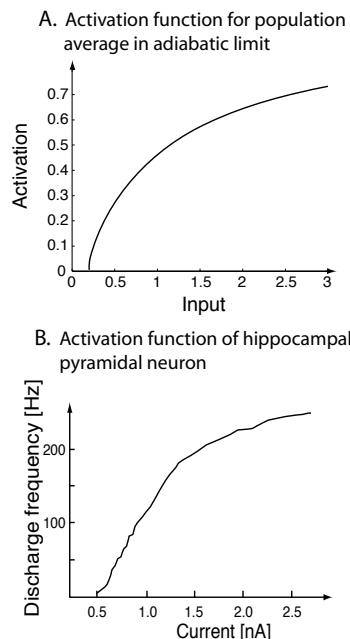


Fig. 3.17 (A) The activation function of eqn 3.43 that can be used to approximate the dynamics of a population response to slowly varying inputs (adiabatic limit). (B) Examples of physiological activation functions from a hippocampal pyramidal cell. The discharge frequency is based on the inverse of the first interspike interval after the cell started to respond to rectangular current pulses with different strength [redrawn from Lanthorn, Storm, and Anderson, *Exp. Brain Res.* 53: 431–43 (1984)].

where t^{ref} is an absolute refractory time. This result tells us that the average activation function of the population is similar to the activation function of a single IF neuron (compare to eqn 3.11), although only in the adiabatic limit. This activation function, eqn 3.43, is plotted in Fig. 3.17A with an absolute refractory time of $t^{\text{ref}} = 1$ ms and a time constant of $\tau = 5$ ms.

Some activation functions of single neurons that have been measured electrophysiologically have shown similar response characteristics. An example of an activation function measured from a hippocampal pyramidal cell is shown in Fig. 3.17B. The figure shows the instantaneous firing rate (discharge frequency) of such a neuron in response to a 1.5 s rectangular current stimulus, with different amplitudes measured in nanoamperes (nA). The instantaneous firing rate was derived from the inverse of the first interspike interval. The neuron quickly adapted to the stimulus (not shown in the figure) so that instantaneous firing based on subsequent spikes would be much smaller. Keep in mind that we are comparing two different definitions of activation functions in Fig. 3.17, a population response in Fig. 3.17A to an averaged single neuron response in Fig. 3.17B. The similarity of the forms could be the result of the particular way in which the measurements are performed. Activation functions are typically measured in experiments with isolated neurons, which could account for the similarity of this response to the average response of a population to slowly varying inputs. Such experimental measurements neglect possible interactions *in vivo* that could alter the effective gain function considerably. It is important to keep in mind the different interpretations of activation functions used in the following models. Most often we will consider population models with slowly varying inputs.

We described in this section the line of thought for deriving population models from averaging over populations. While these *bottom-up* arguments are of great importance and have resulted in many advancements in the field, it is clear that many approximations and assumptions have to be made in order to allow an analytical treatments of such systems. However, we should not forget that many simulation studies have shown that findings of population models can be reproduced with models of spiking neurons. Furthermore, many examples in the remainder of this book demonstrate that population, or rate, models can capture behavioural data and characteristics of neural processing in the brain. The *top-down* motivation of such models, the ability of the models to capture a wide variety of neuronal data, should also be taken into account for such models.

3.4.4 Rapid response of populations

We stressed several times that the population dynamics of eqn 3.37 should only hold for slowly varying inputs, and can indeed break down under several circumstances. To demonstrate this, we consider a pool of equivalent IF neurons all with the same time constant $\tau_m = 10$ ms, and all receiving the same noisy input $I^{\text{ext}} = \bar{I}^{\text{ext}} + \eta$ with $\eta = N(0, 1)$. A simulation of such a ‘network’ of independent model neurons (no connections between the neurons) is shown in Fig. 3.18. In this simulation, we have switched the external input, at time $t = 100$ ms, from a low magnitude $R\bar{I}^{\text{ext}} = 11$ mV to a higher magnitude

$R\bar{I}^{\text{ext}} = 16$ mV. The spike count follows the jump in the input almost instantaneously. The reason for this is that at each instant of time there is a subset of neurons that are close to threshold. These neurons can respond quickly to the input, and the other neurons have time to follow quickly. The population rate calculated with the population dynamics in eqn 3.37 follows only slowly this change of input when the time constant is set to $\tau = \tau_m$. The simulation demonstrates that very short time constants, much shorter than typical membrane time constants, have to be considered when using this model to approximate the dynamics of population responses to rapidly varying inputs.

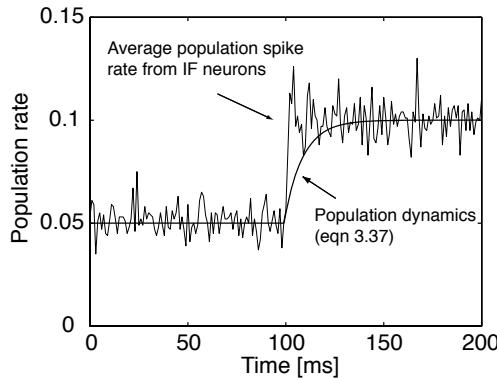


Fig. 3.18 Simulation of a population of 1000 independent integrate-and-fire neurons with a membrane time constant $\tau_m = 10$ ms and threshold $\vartheta = 10$ mV. Each neuron receives an input with white noise around a mean $R\bar{I}^{\text{ext}}$. This mean is switched from $R\bar{I}^{\text{ext}} = 11$ mV to $R\bar{I}^{\text{ext}} = 16$ mV at $t = 100$ ms. The spike count of the population almost instantaneously follows this jump in the input, whereas the average population rate, calculated from eqn 3.37 with a linear activation function, follows this change of input only slowly when the time constant is set to $\tau = \tau_m$.

We have used noise in the models of spiking neurons while leaving any noise out of the model for the population average. The reason for this is that the noise is essential for the argument with spiking neurons. We have not included noise in the population dynamics so as to outline the average response of the node. Adding noise introduces fluctuations into this curve that seem to resemble fluctuations in the average of spiking nodes. However, analysis of the mean response would certainly bring the difference to light. Adding noise to the population node just to make it look more realistic would only obscure the general argument. In contrast, the noise in a population of spiking neurons might be essential for information processing in the brain.

While the simulation demonstrates that the simplest population model fails to describe the rapid onset of neuronal responses in populations of spiking neurons, we should also stress that the asymptotic increase of the population response is well captured by the population node. For many models in cognitive neuroscience it is this fact which is modelled and thus sufficient for many explanations of system-level processes. Also, faster responses can be modelled by input-dependent time constants τ or by input that is explicitly dependent on fluctuations in the input as, for example, captured by the mean field analysis of Brunel and Wang.

3.4.5 Common activation functions

While we have already discussed the population activation function in the limit studied by Wulfram Gerstner, it should be clear that more complicated interac-

tions in the neuronal population can, in principle, lead to other such functions g . We also call this function the *activation function*, which is often termed *transfer function* in neural network studies. This function describes the net effect of various processes within a neuronal group that transforms a given input into the specific output of the neuron assembly. As the neural population itself can implement various response characteristics, it is possible that this function can have various forms beyond the monotone activation functions of single neurons discussed before. Some frequently used functions are illustrated in Table 3.4.

The first example is the *linear function*, g^{lin} , which relates the sum of inputs directly to the output. Linear units can often be treated analytically and can provide us with many insights into the working of networks. Many other activation functions can also be approximated by piecewise linear functions. A simple example of such a non-linear function is the *step function*, g^{step} . This function only returns two values and therefore produces binary responses. This activation function was used by McCulloch and Pitts (1943), and a node with this activation function is therefore called a *McCulloch–Pitts node*.

The third example, the *threshold-linear function* g^{theta} , is similar to the linear functions, except that it is bounded from below. Limiting the node activities from below is sensible as negative activity has no biological equivalent. This activation function is a good approximation of the population activation functions derived in the spike response model. Besides the lower limit on the firing rate, it is also biologically sensible to limit the maximal response of the node. This is because the rate of neuronal groups is limited by the refractory times of neurons, and limited number of neurons can be active at any given time. A simple realization of such an activation function is a combination of the threshold-linear function and the step function. However, a more frequently used activation function in this class is the *sigmoid function* g^{sig} , illustrated as the fourth example in Table 3.4. This function bounds the minimal and maximal responses of a node, while interpolating smoothly between these two extremes. This type of activation function is most frequently used in neural network and connectionist models, for reasons that will become apparent in Chapter 6. It also approximates the activation function of noisy IF neurons well, as we saw in Chapter 2. The last function in the table is very different to the former in the sense that it is a non-monotonic function. The example shown is a *radial-basis function*, one of a group of general functions that are symmetrical around a base value. The particular example g^{gauss} is the famous Gaussian bell curve.

Suffice it to say that these are only a few examples of possible activation functions. We have only outlined the general shape of these functions, and it should be clear that these shapes can be modified. The functions often include parameters with which some characteristics of the functions can be changed, such as the slope and offset of a function. For example, we can generalize the sigmoid function in the table with

$$g^{\text{sig}} = \frac{1}{1 + \exp(-\beta(x - x_0))}, \quad (3.44)$$

Table 3.4 Examples of frequently used activation functions and their basic implementation in MATLAB

Type of function	Graphical represent.	Mathematical formula	MATLAB implementation
Linear		$g^{\text{lin}}(x) = x$	<code>x</code>
Step		$g^{\text{step}}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{elsewhere} \end{cases}$	<code>floor(0.5*(1+sign(x)))</code>
Threshold-linear		$g^{\text{theta}}(x) = x \Theta(x)$	<code>x.*floor(0.5*(1+sign(x)))</code>
Sigmoid		$g^{\text{sig}}(x) = \frac{1}{1+\exp(-x)}$	<code>1./(1+exp(-x))</code>
Radial-basis		$g^{\text{gauss}}(x) = \exp(-x^2)$	<code>exp(-x.^2)</code>

where increasing the value of the parameter β increases the steepness of the curve, and x_0 shifts the curve along the abscissa. Keep in mind that these are not the only possible forms of activation functions. Networks of nodes with different activation functions have different characteristics and it is worth exploring the dependence of network characteristics on the specific activation functions. However, it is also interesting to note that many of the information-processing capabilities of networks of such nodes do not depend critically on the precise form of the activation function. Several types of activation function lead to similar network abilities in the sense that we do not have to fine-tune the functions in order to demonstrate network abilities.

3.5 Networks with non-classical synapses: the sigma-pi node

We introduced population nodes in the last section by assuming an assembly of neurons that behave with similar response profiles. We also assumed that such nodes are summing all weighted inputs, reflecting the additive characteristics of currents. However, we already mentioned in Chapter 2 that single neurons show also non-linear interactions between different input channels; for example, in the case of shunting inhibition. It is therefore likely that population nodes can interact in a non-linear way. This chapter introduces briefly a basic non-linear (multiplicative) interaction between population nodes as this will be used

in some of the models described later in this book.

3.5.1 Logical AND and sigma-pi nodes

An example of a strong non-linear interaction between two ion channels is a spiking neuron with a firing threshold that requires at least two spikes in some temporal proximity to each other. A single spike alone, in this case, cannot initiate a spike. Only if two spikes are present within the time interval, on the order of the decay time of EPSPs, can a postsynaptic spike be generated. This corresponds to a logical AND function. We can represent such an AND function with a multiplicative term of two binary presynaptic terms, r_1 and r_2 , with values one, indicating that a presynaptic spike at a particular synapse is present, or zero, indicating that no presynaptic spike is present. Thus, the dependence of the activation of such a node from two inputs is described by a multiplicative term such as $h = r_1 r_2$.

We can generalize this idea for population models. Let us consider two independent presynaptic spike trains with average firing rates r_j^{in} and r_k^{in} , respectively. The average firing rate determines the probability of having a spike in a small interval. The probability of having two spikes of the two different presynaptic neurons in the same interval is then proportional to the product of the two individual probabilities. This forms the basis of a simple model of non-linear interactions between synapses in a rate model that includes non-linear interactions between synapses. The activation of a node i in this model is given by

$$h_i = \sum_{jk} w_{ijk} r_j^{\text{in}} r_k^{\text{in}}, \quad (3.45)$$

where the weight factor w_{ijk} describes now the overall strength of two combined synaptic inputs. The activation of postsynaptic neurons in this model depends on the sum (mathematically depicted by the Greek letter Σ) of multiplicative terms (mathematically depicted by the Greek letter \prod), and such a node is therefore called a *sigma-pi node*.⁶ The model can be generalized to other forms of non-linear interactions between synaptic channels by replacing the product of the presynaptic firing rates with some function of presynaptic firing rates, for example, $r_j^{\text{in}} r_k^{\text{in}} \rightarrow g(r_j^{\text{in}} r_k^{\text{in}})$, but the simple multiplicative model already incorporates essential features of non-linear ion channel interactions that are sufficient for most of the models discussed in this book. Note that we can also view such interactions as modulatory effects of one synaptic input on the other synaptic input.

3.5.2 Divisive inhibition

Equation 3.45 is an example of two non-linear interacting excitatory channels. An analogous model of an interaction between an excitatory synapse and an inhibitory synapse can be written as

$$h_i = \sum_{jk} w_{ijk} r_j^{\text{excitatory}} / r_k^{\text{inhibitory}}. \quad (3.46)$$

This is an example of divisive inhibition which models the effects of shunting inhibition as mentioned in Chapter 2. Inhibitory synapses on the cell body

⁶The product of four terms $x_1 x_2 x_3 x_4$ can be written with this shorthand notation as $\prod_{i=1}^4 x_i$. We have not included this notation in our equation as we have only two factors in the product and the notation would actually make the formula look more cluttered.

often have such shunting effects (see Fig. 3.19A), whereas inhibitory synapses on remote parts of the dendrites are better described by subtractive inhibition. It is possible that synapses in between have effects that interpolate between these two extremes.

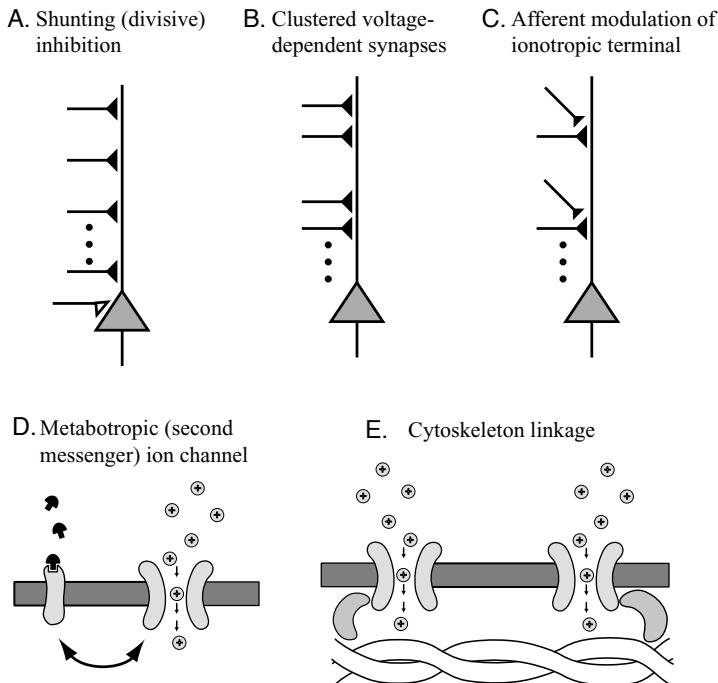


Fig. 3.19 Some sources of non-linear (modulatory) effects between synapses as modelled by sigma-pi nodes. (A) Shunting (divisive) inhibition, which is often recorded as the effect of inhibitory synapses on the cell body. (B) The effect of simultaneously activated voltage-gated excitatory synapses that are in close physical proximity to each other (synaptic clusters) can be larger than the sum of the effect of each individual synapse. Examples are clusters of AMPA and NMDA type synapses. (C) Some cortical synaptic terminals have nicotinic acetylcholine (ACh) receptors. An ACh release of cholinergic afferents can thus produce a larger efflux of neurotransmitter and thereby increase EPSPs in the post-synaptic neuron of this synaptic terminal. (D) Metabotropic receptors can trigger intracellular messengers that can influence the gain of ion channels. (E) Ion channels can be linked to the underlying cytoskeleton with adapter proteins and can thus influence other ion channels through this link.

3.5.3 Further sources of modulatory effects between synaptic inputs

Interactions between synapses can result from a variety of other sources in biological neurons. An obvious source is voltage-dependent synapses that are in physical proximity to each other as illustrated in Fig. 3.19B. A good example are NMDA receptors that are blocked for low membrane potentials. If the membrane potential is raised by an EPSP from another non-NMDA synapse in its proximity, then it is possible that the blockade is removed so that the combined effect of a non-NMDA synapse together with the NMDA synapse is much larger than the sum of the activation through each synapse alone. The physical proximity is necessary as the EPSP in a passive dendrite decays with distance, so that the non-linear effects are largest for nearby synapses. The non-linear effects can also reach larger distances between synapses with local active dendrites that can activate more remote dendrites through active spike propagation.

Examples of a direct influence of specific afferents on the release of neurotransmitters by presynaptic terminals are also known. This is illustrated

schematically in Fig. 3.19C. An example is cholinergic afferents that emit acetylcholine (ACh). This can bind to ACh receptors at a presynaptic terminal that opens calcium channels. The excess of calcium in the presynaptic terminal then triggers an enhanced release of neurotransmitters by a presynaptic action potential. This is an example of the modulation of an ionotropic synapse. A more diffuse modulation can be achieved with metabotropic receptors (see Fig. 3.19D). Such receptors initiate intracellular chemical reactions leading to second messengers that can influence the efficiency or gain of different ion channels. A more direct linkage between ion channels is possible through adapter proteins that link ion channels to the underlying cytoskeleton in a neuron (Fig. 3.19E). More research is necessary to understand all the details of such mechanisms. We are mainly concerned here with the principal potentials of such modulatory effects between presynaptic neuron activities, and we will employ such mechanisms in some models discussed in later chapters.

Exercises

- (3.1) In Chapter 2, we distinguished synaptic models and spike-generation models. To which type does the leaky integrate-and-fire model belong?
- (3.2) Implement a regular-spiking Izhikevich neuron in MATLAB or a comparable programming environment.
- (3.3) Plot an activation function (the average firing rate versus the input current) for a fast-spiking Izhikevich neuron.
- (3.4) What is the value of the threshold in the activation function of the Izhikevich neuron?
- (3.5) Plot the time course of a dynamic population node (eqn 3.37) that is driven by input which is switched on and off at regular intervals.

Further reading

The book edited by Maass and Bishop (1999) contains a clear introduction to the formulation of spiking networks, as well as many advanced discussions of networks of spiking neurons. Some presentations the present chapter are derived from the first chapter by Wulfram Gerstner, as well has his 2000 paper. The very readable papers by Izhikevich are also recommended. The 2004 paper contains a nice overview of classical spiking neuron models, and the 2003 paper introduces his model. The classic paper by McCulloch and Pitts (1943) is still interesting to read. A formal derivation of a population model is introduced by Wilson and Cowan (1972). The important paper

by Brunel and Wang (2001) includes a detailed mean field theory in the appendix very well, and the paper also provides references to their relevant, earlier work.

Wolfgang Maass and Christopher M. Bishop (eds) (1999), *Pulsed neural networks*, MIT Press.

Wulfram Gerstner (2000), *Population dynamics of spiking neurons: fast transients, asynchronous states, and locking*, in *Neural Computation* 12: 43–89.

Eugene M. Izhikevich (2003), *Simple model of spiking neurons*, in *IEEE Transactions on Neural Networks* 14: 1569–1572.

- Eugene M. Izhikevich (2004), *Which model to use for cortical spiking neurons?*, in *IEEE Transactions on Neural Networks* 15: 1063–1070.
- Warren McCulloch and Walter Pitts (1943), *A logical calculus of the ideas immanent in nervous activity*, in *Bulletin of Mathematical Biophysics* 7: 115–133.
- Hugh R. Wilson and Jack D. Cowan (1972), *Excitatory and inhibitory interactions in localized populations of model neurons*, in *Biophysics Journal* 12:1–24.
- Nicolas Brunel and Xiao-Jing Wang (2001), *Effects of neuromodulation in a cortical network model of working memory dominated by recurrent inhibition*, in *Journal of Computational Neuroscience* 11: 63–85.

This page intentionally left blank

Associators and synaptic plasticity

4

So far, we have neglected some of the most exciting and central mechanisms of brain processing, those of synaptic plasticity and learning in networks. Neurons are connected to form networks, and a neural network is not only characterized by the topology of the network, but also by the *connection strength*, w_{ij} , between two neurons or two population nodes. In this chapter we discuss how connection strengths can be changed in a usage-dependent way through a biological phenomena called *synaptic plasticity*. Synaptic plasticity is the physical basis of learning in neural systems which we will discuss later. Here, we start with a general discussion of *associators*, which summarize the essence of plasticity mechanisms and their importance for cognitive brain processing. We then present the neurophysiological basis of plasticity and some specific models of plasticity. The final part of this chapter discusses some consequences of plasticity rules, in particular weight distributions, and explains some strategies for weight scaling and weight decay. The chapter ends with an application of a basic Hebbian rule with weight decay to *principal component analysis*.

4.1 Associative memory and Hebbian learning

Neural networks are characterized by a collection of connection strengths, w_{ij} , between neurons or population nodes. We can code the network architecture with a large matrix that specifies all the weights between all the nodes in the network. A weight of zero indicates that there is no functioning connection between two neurons. We can loosely differentiate two forms of plasticity:

- **Structural plasticity** is the mechanism describing the generation of new connections and thereby redefining the topology of the network.
- **Functional plasticity** is the mechanism of changing strength values of existing connections.

Also, it is sometimes useful to distinguish *developmental mechanisms* from *adaptations in mature organisms*. The formation and maintenance of neuronal networks in the brain must be, to a large extent, genetically coded. However, not all the details can be coded with the fairly small number of genes in the human genome, estimated to be around 20,000–25,000, not to mention the current view that only a small percentage of these genes are thought to be used to construct particular proteins that guide the functionality of the organisms. Genes can, however, influence attractor substances that guide the

4.1 Associative memory and Hebbian learning	87
4.2 The physiology and biophysics of synaptic plasticity	94
4.3 Mathematical formulation of Hebbian plasticity	99
4.4 Synaptic scaling and weight distributions	105
Exercises	116
Further reading	116

growth of neurites during brain development and the maintenance of synapses, as discussed later. Brain development can also be influenced by environmental circumstances. Adaptive mechanisms are thus thought to be very important for brain functions, at least for the fine tuning of parameters in brain networks.

During neural development of mammalian organisms, it is thought that brains start highly connected and that during infancy many of the connections get pruned. Also, it has been found that there are *critical periods* during which normal development of specific neuronal organizations can be easily disturbed. Providing new synaptic contacts does not only require that axons grow close to dendrites, but that axons develop release sites, that dendrites incorporate synaptic ion channels, and that glial cells provide supporting functionality. While brain development and structural plasticity are important areas of research, many of our discussions in later chapters focus on the ability of adaptive networks in general, without distinguishing between different forms of plasticity that are certainly present in biological organisms. Most of the following discussion is concerned with learning rules that are widely attributed to functional plasticity.

4.1.1 Hebbian learning

The idea that the brain can change and adapt by building associations had already appeared by the end of the 19th century. For example, *Sigmund Freud* proposed the *law of association by simultaneity* in 1888 which very much resembles the general principles explained in this section. However, the central role of associative learning as an organizing principle in the brain became popular only later with an influential book by the Canadian psychologist *Donald O. Hebb*. In his book, *The organization of behaviour*, which was published in 1949, he even speculates on the functional implementation of learning:

'When an axon of a cell A is near enough to excite cell B or repeatedly or persistently takes part in firing it, some growth or metabolic change takes place in both cells such that A's efficiency, as one of the cells firing B, is increased.'

Hebb had no means of observing synapses directly, which makes this hypothesis a wild guess by our standards. But his book had a great influence on many researchers as it was one of the first concrete proposals about how the brain organizes itself to implement the cognitive functions underlying behaviour. In particular, Hebb proposed that cell assemblies, formed dynamically in response to external stimuli, carry out much of the information processing in the brain. The basic physical mechanisms that support these plastic changes in the brain, he argued, are some synaptic changes which he speculated to be 'some growth or metabolic change'. Likely the most important component of his hypothesis for us is the functional basis for this change, that of a functional meaningful correlation between presynaptic and postsynaptic activity. This important insight is summarized as '*what fires together, wires together*'.¹ It is only much more recently that the physiology of synaptic plasticity has been observed in some detail. This newer research shows that Hebb's qualitative description, as quoted above, is remarkably accurate and captures most of the essential

¹While this phrase could be interpreted more towards structural plasticity, it should be understood here in terms of synaptic plasticity, or in Hebbian's sense, in what fires together should wire together more strongly.

organizational principles in the nervous system. Activity-dependent plasticity that depends on pre- and postsynaptic activity is often called *Hebbian plasticity* or *Hebbian learning* in Hebb's honour.

4.1.2 Associations

A major hypothesis quantified by computational models is that synaptic plasticity enables memory and learning, as discussed extensively in later chapters. Human memory is very different from that of digital computers. Memory in a digital computer consists of a device where information is stored, similar to a shelf. When recalling information from the computer memory we have to tell the computer internally where to find this memory using a memory address. Failing to specify the memory address precisely results in complete loss of this memory as it cannot then be recalled. Even an error of one bit in the memory address turns out to be fatal for memory recall. Natural systems cannot work with such demanding precision.

The human memory system is different in many respects from conventional computer memory. We are often able to recall vivid memories of events from small details. For example, when visiting some places of your childhood it is likely that the images of the locations, although not exactly the same as years ago, will trigger vivid memories of friends and certain incidents that occurred at these places; a picture with only part of the face of a friend can be sufficient to recognize him and to recall his name, or, if someone mentions the name of a friend, it is possible to remember certain facial features of this person. In contrast to digital computers, we are able to learn associations that can trigger memories based on related information, that is, only partial information can be sufficient to recall memories. It is this type of memory that we think is essential for human cognitive processes. While computer scientists sometimes call this *content addressable memory*, we will generally call this an *associative memory* or *associator*.

Synaptic plasticity is the necessary ingredient behind forming associations, and we will now illustrate the principal idea of Hebbian associative learning using a simplified neuron as illustrated in Fig. 4.1. We will later use these mechanisms extensively in *associative networks*, illustrated in Fig. 4.2. Such associative architectures were proposed around 1960 independently by *Bernd Widrow (Madaline)* and *Karl Steinbuch (Lernmatrix)*. While we now use the language of neurons, the same arguments apply for associative learning in networks of population nodes, as primarily used in the later chapters of this book. We showed only a few presynaptic axons which synapse on the model node in Fig. 4.1. However, it is important to keep in mind in the following discussion that neurons receive input from a large number of sources, as mentioned before. A cortical neuron receives on the order of 5000–10,000 synapses from other neurons, although only a small subset, fewer than 1%, of active synaptic channels can be sufficient to elicit a presynaptic spike if the synaptic efficiencies are sufficient and synaptic events fall within certain temporal windows.

Let us discuss the example illustrated in Fig. 4.3. There, a certain event, such as the presence of particular smell, is reflected in the firing of a subset of the input channels to the neuron (Fig. 4.3A). We have used only a small

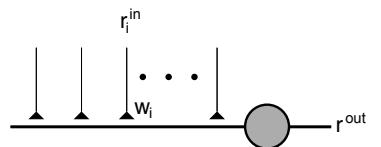


Fig. 4.1 A simplified neuron that receives a large number of inputs r_i^{in} . The synaptic efficiency is denoted by w_i and the output of the node by r^{out} .

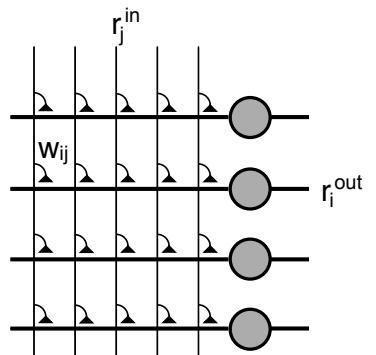


Fig. 4.2 A network of associative nodes. Each component of the input vector, r_i^{in} , is distributed to each neuron in the network. However, the effect of the input can be different for each neuron as each individual synapses can have different efficiency values w_{ij} , where j labels the neuron in the network.

subset of presynaptic neurons to represent the input stimulus in accordance with distributed, yet sparse, coding discussed further below. In Fig. 4.3A we represent the odour stimulus, say the smell from a BBQ when cooking hamburgers, with an input pattern that represents spikes in input channels 1, 3, and 5. The synaptic efficiencies of these input channels are assumed to be large enough to elicit a postsynaptic spike. For example, we can take the spiking threshold to be $\theta = 1.5$ and assume initially that the synaptic weights of only these input channels have the values $w_i = 1$, as shown on the right site of Fig. 4.3A. The input pattern is then sufficient to elicit a postsynaptic spike. This can easily be verified by comparing the internal activation of the neuron,

$$h = \sum_i w_i r_i^{\text{in}} = 3, \quad (4.1)$$

with the firing threshold of the neuron ($\theta = 1.5$). These values of the synaptic weights are also sufficient to enable the neuron to fire in response to partial input. For example, if only channels 1 and 3 are on, then $h = 2 > \theta$ and the neuron still fires. This is very important for brain processes since the input pattern might not always be complete due to noisy processing or partial sensory information. In this way, the brain can achieve *pattern completion*, the ability to complete the representation of a partial stimulus.

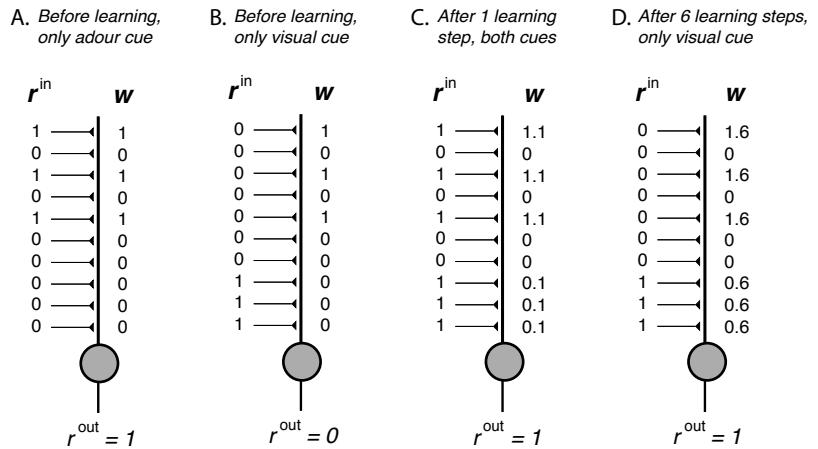


Fig. 4.3 Examples of an associative node that is trained on two feature vectors with a Hebbian-type learning algorithm that increases the synaptic strength by $\Delta w = 0.1$ each time a presynaptic spike occurs in the same temporal window as a postsynaptic spike.

Another stimulus, such as the sight of the hamburger when opening the BBQ, represented in the model as a presynaptic input to the last three incoming channels in Fig. 4.3B, is not able to elicit the response of this neuron with the current values of synaptic weights. In order to associate the visual cue of a hamburger with an odour signalling the smell of a hamburger, we have to modify the synaptic weights. For this we adopt the following strategy:

Increase the strength of the synapses by a value $\Delta w = 0.1$ if a presynaptic firing is paired with a postsynaptic firing.

This *learning rule* modifies the synaptic weights, so that after one simultaneous presentation of both stimuli, the visual cue and the cue representing the odour,

the weights have the values illustrated in Fig. 4.3C. After six consecutive learning iterations, we end up with the synaptic weights shown in Fig. 4.3D. How fast we achieve this point depends on a *learning rate*, which we set to 0.1 in this example.

The visual cue of a hamburger alone is sufficient to elicit the response of the neuron after only 5 or 6 simultaneous presentations of both stimuli. The firing of this neuron is then associated with both the presence of a visual image of a hamburger and the presence of a cue representing the smell of a hamburger. If this neuron, or this neuron in combination with other neurons in a network, elicits a mental image of a hamburger, then the presence of only one cue, the smell or the sight of the hamburger, is sufficient to elicit the mental image of a hamburger. Therefore, the odour and the visual image of a hamburger are associated with its mental image. Such associations are fundamental for many cognitive processes.

4.1.3 Hebbian learning in the conditioning framework

The mechanisms of an associator, as outlined above, rely on the fact that the first stimulus, the odour of the hamburger, was already effective in eliciting a response of the neuron before learning. This is a reasonable assumption since we could start with random networks in which some neurons would be responsive to this stimulus before learning. Thus, we can assume that we have selected such a neuron in our example from a large set of neurons responsive to all kinds of stimuli, based on a random initial weight distribution. Although the synaptic strengths of these input channels change with the learning rule, we call the initial stimulus the *unconditioned stimulus* (UCS) because the response of the neuron to this stimulus was maintained. The main reason for this labelling is that we want to distinguish this input, which is already effective, from the second stimulus, the visual image of the hamburger. For the second input the response of the neuron changes during learning, and so we call the second stimulus the *conditioned stimulus* (CS). The input vector to the system is thus a mixture of UCS and CS as illustrated in Fig. 4.4A.

Several variations of this scheme are known to occur in biological nervous systems. One of these is outlined in Fig. 4.4B. We stressed that it is crucial for the UCS in the previous scheme to elicit a postsynaptic neural response, which was incorporated in our example above by using sufficiently strong synapses in the input channels of the UCS. Alternatively, as illustrated in Fig. 4.4B, the fibres carrying the UCS can have many synapses on to the postsynaptic dendrite that can ensure the firing of the node. For example, mossy fibres in the hippocampus are axons that generate very strong contacts with postsynaptic neurons. Another example is climbing fibres in the cerebellum, which have many hundreds of synapses with a single Purkinje cell. The cerebellum has been implicated in motor learning, and it has been suggested that the UCS provides a motor error signal for such learning mechanisms. We will elaborate on this in Chapter 9.

Another model is illustrated in Fig. 4.4C. Here, we have indicated a specific modulatory mechanism using a separate input to presynaptic terminals, which can induce presynaptic changes. Such mechanisms exist in invertebrates, for

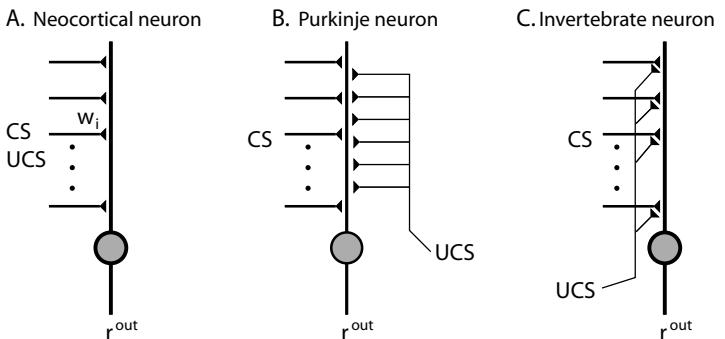


Fig. 4.4 Different models of associative nodes resembling the principal architecture found in biological nervous systems such as (A) cortical neurons in mammalian cortex and (B) Purkinje cells in the cerebellum, which have strong input from climbing fibres through many hundreds or thousands of synapses. In contrast, the model as shown in (C) that utilizes specific input to a presynaptic terminal as is known to exist in invertebrate systems, would have to supply the UCS to all synapses simultaneously in order to achieve the same kind of result as in the previous two models. Such architectures are unlikely to play an important role in cortical processing.

example, in the sensorimotor system of the sea slug *Aplysia*. Such mechanisms could be specific for each presynaptic terminal and would thus be non-Hebbian by definition. To achieve the same results as in the other two models, which depend globally on the firing of the postsynaptic neuron, we would have to supply the UCS signal to all synaptic inputs simultaneously. Such an architecture is much more elaborate to realize in larger nervous systems and seems to be absent in vertebrate nervous systems. This demonstrates that different synaptic mechanisms and information-processing principles may be present in vertebrate and invertebrate nervous systems.

The example of Fig. 4.3 illustrates that it is possible to store information with associative learning; after imprinting an event–response pattern, the response can be recalled from partial information about the event. This is the primary reason that synaptic plasticity is thought to be the underlying principle behind associative memory and other important information-processing abilities in the brain. Of course, the basic learning rule used above has to be extended to incorporate important details not captured in the example. For example, the learning rule outlined in the example increased all the relevant synaptic weights. This *synaptic potentiation* results in an increase of the complete weight vector after some time if the presentation of many different input vectors is coupled with random firing of the postsynaptic node. The synaptic weights would then become extremely large and so the response of the node becomes unspecific to input patterns since it would respond to noisy input. Some form of synaptic weakening, called *synaptic depression*, including some competitive depression between synapses, is therefore important. Also, the temporal structure of the firing pattern can be relevant in the learning rule, as suggested by Hebb. We will review recent experimental findings that verify this hypothesis in the following section.

4.1.4 Features of associators and Hebbian learning

Associations, together with some form of distributed representation discussed further in Chapters 6, lead to important characteristics of networks of associative nodes. These characteristics are essential for brain-style information processing and the modelling of cognitive processes. The following is a brief summary of such characteristics which we will study further in different parts of this book.

Pattern completion and generalization

We have seen that a stimulus capturing only part of a pattern associated with an object can still trigger a memory of that object. This means that networks support recall of details of an object which were not part of the stimulus. This ability to *recall from partial input* is also termed *pattern completion*. It relies on a distributed representation of an object stimulus so that some missing components of a feature vector can be added. Pattern completion is based on the calculation of some form of *overlap*, or *similarity*, of an input vector with a weight vector that represents the pattern on which the node was trained. An example of such a similarity measure is the *dot product* between the pattern input vector and the vector of the synaptic efficiencies. We also need non-linearity in the output function, such as a firing threshold, which is characteristic of neurons and neural populations. Then, as long as the overlap between the input pattern and the trained pattern is large enough, an output is generated that is equivalent to the output of the trained pattern. Thus, the output node responds to all patterns with a certain similarity to the trained pattern, an ability that is called *generalization*.

Prototypes and extraction of central tendencies

A closely related ability of associative nodes, or networks, is the ability to *extract central tendencies*. This occurs when training associative nodes on many similar, but not equivalent, examples. The weight vector then represents an average over these examples, which we can interpret as a *prototype*. Such training sets are typical in natural environments. For example, each person has an individual face, while there are still many common features in all faces. Another reason for training sets with many, slightly different, patterns, is that the training set is derived from the same object, which is represented over time with fluctuating patterns due to some noisy processes in the encoding procedure. The prototype extraction ability of associators can then be used to achieve *noise reduction*, an ability that is important in the case of noisy processing systems, such as the brain and technological applications.

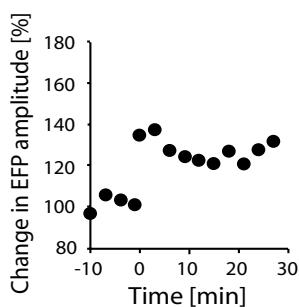
Graceful degradation

The loss, or inaccuracy, of some components of a neural system, such as some synaptic connections or even whole neurons, should not markedly affect the system. Associative networks often degrade gracefully in that a large amount of synapses have to be removed before the system produces a large amount

of error. Even the loss of nodes can be tolerated if the information feeds into an upstream processing layer with associative abilities that can archive pattern completion from partial information. Such *fault tolerance* is essential in biological systems.

4.2 The physiology and biophysics of synaptic plasticity

A. Long term potentiation



B. Long term depression

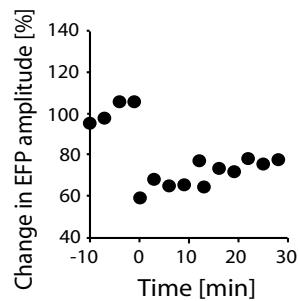


Fig. 4.5 (A) Change in evoked field potential in the striatum after a high-frequency stimulation at time $t = 0$. (B) Demonstration of LTD in similar experiments [adapted from Dang, Yokoi, Yin, Lovinger, Wang, and Li, *Proceedings of the National Academy of Science* 103: 15254-9 (2006)].

In Chapter 2 we reviewed how a spike elicits the release of neurotransmitters in chemical synapses through which presynaptic spikes can influence the state of postsynaptic neurons. In this section, we briefly review the physiology and biochemical mechanisms which are thought to be involved in changing the response characteristics of neurons, with a focus on activity-dependent functional plasticity. We should keep in mind the possibility that several components of the synaptic machinery can change in plasticity experiments. Possible changes include the number of release sites, the probability of neurotransmitter release, the number of transmitter receptors, and the conductance and kinetics of ion channels.

4.2.1 Typical plasticity experiments

Long-lasting changes of synaptic response characteristics, including the average amplitude and latency of EPSPs, were first demonstrated experimentally in 1973 by *Timothy Bliss* and *Terje Lømo* in hippocampus cultures. Some exemplary results of such typical plasticity experiments are shown in Fig. 4.5A. The shown experiments are based on measurement of evoked field potentials (EFPs) in the *striatum* of mice, a subcortical structure involved in motor control. The graph shows the change in EFP amplitude, compared to average initial measurements (here at times from $t = -10$ min to $t = 0$ min). At time $t = 0$, a high-frequency stimulus was applied. The recording of EFP amplitudes after this plasticity-inducing tetanus shows increased EFP amplitude. While the initial, strong response will often decay somewhat after some minutes, many experiments have shown long-lasting effects over hours, days, or more, which is very long on the scale of brain processing. Such changes in synaptic efficacy are therefore termed *long-term potentiation* (LTP) since the average amplitude of the EPSP increased. Other experiments have shown that long-term changes can be of either sign. For example, stimulations with low frequencies at $t = 0$ result in lowering average EFP amplitudes. An example is shown in Fig. 4.5B. Such long-lasting changes are called *long-term depression* (LTD). Such mechanisms can support the associations discussed in the previous section. In particular, LTP can enforce associative responses to a presynaptic firing pattern that is temporally linked to postsynaptic firing, while LTD can facilitate the unlearning of presynaptic input that is not consistent with postsynaptic firing. It is interesting to note that these experiments by Dang *et al.* also showed that preventing LTP in the striatum disrupted learning of motor skills in which the striatum is implicated.

Changes in EFP amplitudes are not the only effects that are measured in

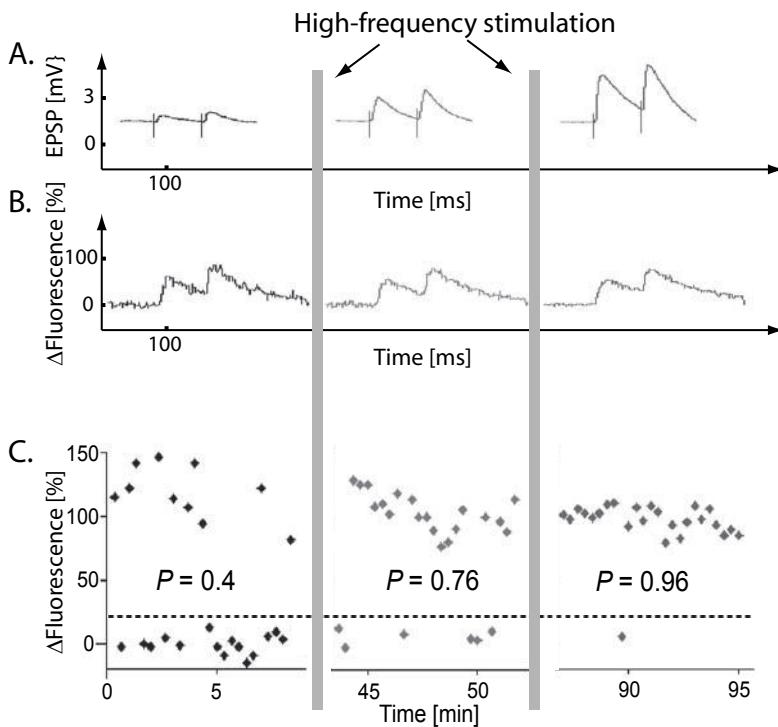


Fig. 4.6 Plasticity experiment in hippocampal slices in which not only EPSPs were measured, but in which, additionally, postsynaptic calcium-dependent fluorescence signals at single synapses were imaged. [Data courtesy of Alan Fine and Ryosuke Enoki].

plasticity experiments. Bliss and Terje Lømo already noted that the latency of evoked postsynaptic spikes responses can be affected. Also, recent experiments by *Alan Fine* and *Ryosuke Enoki* in hippocampal slices of rats have shown that the probability of transmitter release can be modulated in plasticity experiments. Some results of their experiments are illustrated in Fig. 4.6. Fig. 4.6A shows time courses of EPSP responses to pairs of presynaptic stimulations. A pair of probe stimuli was used since it is often observed that the second pulse is more effective than the first pulse in eliciting a postsynaptic response. This is a form of *short-term plasticity* which is called *paired-pulse facilitation*. After measuring the initial response in the neuron, a high-frequency stimulus was applied, indicated by vertical bars in the figure. The increase in EPSP amplitude following the high-frequency stimulation is similar to that seen in the experiments discussed earlier. These experiments also show that a further potentiation was achieved with another high-frequency stimulus.

The investigators also imaged single synapses with calcium-sensitive fluorescent dyes. The time course of changes in the fluorescence are plotted in Fig. 4.6B. While these measurements also show the effects of paired-pulse facilitation, the amplitudes are comparable after each LTP tetanus and so do not show a synaptic increase proportional to the increase in EPSPs. However, they do respond faithfully when transmitter has been released from the presynaptic terminal. An important effect of the LTP tetanus can be seen in Fig. 4.6C. This figure shows the change in fluorescence for several trials in the

different phases of the experiment. The results show that not every presynaptic stimulus produced a postsynaptic response; the fraction of times that a postsynaptic event was observed can be equated with the probability of transmitter release, as mentioned in Section 2.2. Furthermore, the probability of release increased considerably after each high-frequency stimulus. Other experiments by the group have shown that the probability of release can be altered bidirectionally, increasing the probability with high-frequency stimuli and decreasing the probability when the stimuli are paired in a specific way with postsynaptic spikes as discussed in the next section. The advantage of the imaging technique is that the results in Fig. 4.6B and C can be attributed to a single synapse, while the EPSP measurements are likely to measure effects of several synapses. The increase in EPSP amplitude can thus be a population effect, in which the increased probability of getting a postsynaptic event in each of a number of synapses can increase the summed effect of measuring the EPSP in the soma.

4.2.2 Spike timing dependent plasticity

The classic plasticity experiments discussed above typically use high-frequency presynaptic stimulations to induce LTP and lower-frequency stimulations to induce LTD. These stimulations need to be strong enough to ‘drive’ the postsynaptic neuron, reflecting Hebb’s postulate of a causal relation between postsynaptic and presynaptic spikes. *Henry Markram*, and others, have further investigated the necessary causal relation between pre- and postsynaptic spiking by varying the time between externally induced pre- and postsynaptic spikes. An exemplary result of such experiments in hippocampal cultures by *Guo-chiang Bi* and *Mu-ming Poo* is shown in Fig. 4.7A. The measurements were done in a *voltage clamp* so that currents are measured here. The data show the relative changes of EPSC amplitudes for different values of the time differences between pre- and postsynaptic spikes, Δt . These results indicate that there is a critical time window for plasticity of around $|\Delta t| \approx 40$ ms. No synaptic plasticity could be detected for differences of pre- and postsynaptic spike times with absolute values much larger than this time window. Changes in *excitatory postsynaptic current* (EPSC) amplitudes are largest for small positive differences (LTP) or small negative differences (LTD) in the pre- and postsynaptic spike times. The decrease in synaptic plasticity can be fitted reasonably well with an exponential decay of the absolute difference between spike times.

While the data in Fig. 4.7A are obtained from hippocampal cells *in vitro*, similar relations have also been found in neocortical slices, as well as *in vivo*, with similar sizes of plasticity windows. However, the *asymmetrical form of Hebbian plasticity*, shown again schematically on the right in Fig. 4.7A, is not the only possible form for dependence of synaptic plasticity on the relative spike times. Some additional examples are illustrated schematically in Fig. 4.7B–E. Examples are known in which the critical difference window for LTD is much larger compared to that for LTP, as illustrated schematically in Fig. 4.7B. LTP and LTD can also be reversed relative to the difference in pre- and postsynaptic spiking (Fig. 4.7C). For example, positive differences in spike timings induce LTD when synaptic plasticity is triggered in Purkinje cells in the cerebellum. Purkinje cells are known to be inhibitory, and such changes can therefore re-

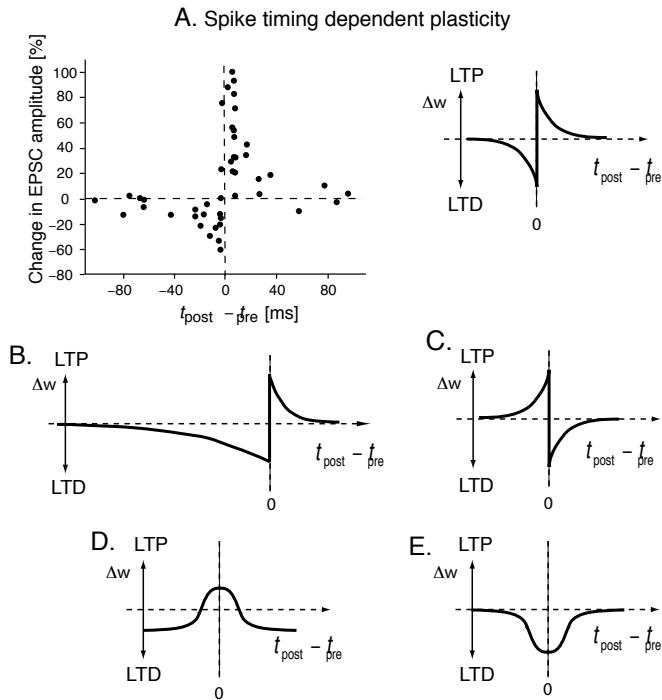


Fig. 4.7 (A) The relative changes in excitatory postsynaptic current (EPSC) amplitudes for various time differences between the time of an induced pre-synaptic spike and the time of an induced post-synaptic spike [data redrawn from Bi and Poo, *Journal of Neuroscience* 18: 10464–72 (1998)]. The form of plasticity is stylized on the right. (B–E) Several examples of the schematic dependence of synaptic efficiencies on the temporal relations between pre- and postsynaptic spikes [adapted from Abbott and Nelson, *Nature Neuroscience* 3: 1178–83 (2000)].

duce inhibition in the targets of Purkinje cells. There are also examples of *symmetrical Hebbian plasticity* illustrated in Fig. 4.7D and E. Such plasticity windows seem more appropriate when considering bursting neurons. It is important for computational neuroscientists to explore the functional roles of the different forms of Hebbian plasticity.

4.2.3 The calcium hypothesis and modelling chemical pathways

The physiological experiments described above illustrate the consequences, in terms of synaptic efficacy, when the synapses are exposed to specific stimulations. While this is most relevant for understanding the functional consequences of synaptic plasticity that we will explore later in this book, it is also important to understand the mechanisms leading to the findings discussed above. Although there are still many unknowns, calcium is an important ingredient. In particular, *Artola* and *Singer*, and earlier *John Lisman*, have advanced the hypothesis that different levels of calcium trigger different biochemical pathways, as illustrated in Fig. 4.8. Calcium triggers the phosphorylation of proteins through enzymes such as CaM kinase II, which can alter the efficiency of existing AMPA receptors and can promote the insertion of new AMPA channels in the long term. Specifically, high levels of calcium promote LTP through *phosphorylation*, whereas moderate levels of calcium lead to *dephosphorylation* and LTD. This is consistent with findings by *Artola* and *Singer* for conditions

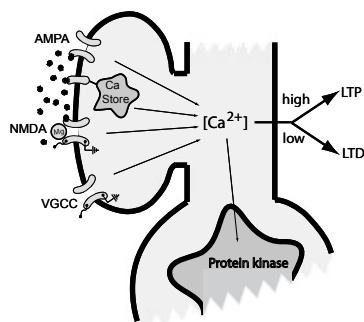


Fig. 4.8 Illustration of the calcium hypothesis which states that high levels of calcium promote LTP, whereas moderate levels lead to LTD. The figure includes common sources of postsynaptic calcium and indicates that calcium drives protein kinases in the nucleus.

leading to LTP and LTD, for which we will describe corresponding learning rules in the next section.

To relate this hypothesis to the activity of neurons, we need to think about the mechanisms that influence postsynaptic calcium levels. There are several sources of postsynaptic Ca²⁺, as schematized in Fig. 4.8. The binding of glutamate can directly influence the postsynaptic calcium level in two ways. First, this occurs through AMPA receptors that gate ion channels with some Ca²⁺ permeability, but the resulting postsynaptic Ca²⁺ level from this source is thought to be moderate. Second, glutamate triggers Ca²⁺ release from internal Ca²⁺ stores. These sources of Ca²⁺ are indirectly triggered by glutamate binding. Glutamate binding also leads to an increase in the membrane potential that can indirectly enable Ca²⁺ entry through *voltage-gated Ca²⁺ channels* (VGCC). Finally, the largest influx of Ca²⁺ is enabled through NMDAR channels. However, as discussed before, NMDAR channels are blocked, even after glutamate release, by magnesium ions (Mg²⁺). Only a strong postsynaptic depolarization removes this blockage, allowing a large increase in postsynaptic Ca²⁺. The necessary, strong depolarization of the membrane potential is likely provided by backpropagating action potentials (BAPs), as mentioned before. Thus, NMDAR channels provide mechanisms for a *Hebbian condition*, those of presynaptic activity (indicated by glutamate binding to NMDAR) and postsynaptic activity (indicated by BAPs).

The hypothesis that NMDAR and postsynaptic Ca²⁺ are involved in synaptic plasticity is backed up by demonstrations that the blockade of NMDAR with antagonistic drugs, or prevention of the rise in postsynaptic calcium concentrations, abolishes synaptic plasticity. Note that we have discussed mainly postsynaptic mechanisms of long-lasting plasticity. However, as mentioned before, there is evidence that presynaptic modifications are important for synaptic plasticity and learning. For example, the above-mentioned modulation of transmitter release probabilities can support long-lasting postsynaptic changes or contribute to learning and information processing in its own way. It is also possible that postsynaptic processes influence presynaptic processes through glial cells which wrap around synapses and interact with both the pre- and postsynaptic sides through direct chemical and electrical couplings and other Ca²⁺ channels. A direct presynaptic influence is also possible through *retrograde messengers* such as *nitric oxide*. Such mechanisms for conveying information back from the postsynaptic terminal to presynaptic boutons are not required in invertebrate systems with specific terminals on presynaptic boutons (Fig. 4.4C). Presynaptic changes in synaptic plasticity seem to be important in such systems through the temporally linked transmitter release of the UCS input and the activation of a specific presynaptic terminal.

It is possible to model the intracellular chemical pathways related to synaptic plasticity directly. An example of chemical pathways in D1 receptors of spiny neurons of the striatum is shown in Fig. 4.9, as summarized by Nakano, Doi, Yoshimoto and Doya. These synapses are particularly interesting since they are regulated not only by glutamate, but also by dopamine (DA) input from the substantia nigra. Experiments by Jeff Wickens and colleagues have shown that stimulation of cortical input alone results in LTD. Only in combination with DA input could LTP be induced. Nakano *et al.* modelled the *signalling network*

shown in Fig. 4.9 based on AMPA receptor trafficking, insertion of AMPA receptors when phosphorylated by kinases and removal when dephosphorylated by phosphatases while regulated by DA and Ca²⁺ concentration. The authors found DA- and Ca²⁺-dependent plasticity, with low levels leading to LTD and high levels to LTP. We will come back to such dependencies in Section 4.3.2.

4.3 Mathematical formulation of Hebbian plasticity

In neural network models, synaptic plasticity is described by a change of weight values, w , between nodes. Hence, the weight values are not static but can change over time. Unfortunately, we do not know much about the precise time course of synaptic changes, and most experiments can only explore modifications after certain training intervals. It is thus appropriate to express the variation of weight values only after time steps, Δt , in a discrete fashion as,

$$w_{ij}(t + \Delta t) = w_{ij}(t) + \Delta w_{ij}(t_i^f, t_j^f, \Delta t; w_{ij}). \quad (4.2)$$

We have indicated here the dependence of the weight changes on various factors. For activity-dependent synaptic plasticity it is clear that weight changes should depend on the firing times, t^f , of the pre- and postsynaptic neurons. We included in this formula the possibility that the weight changes depend on the current value of the synaptic strength. *Weight-dependent plasticity* is an important factor, discussed further below, since the strength of biological synapses can certainly only vary within some interval. Different instantiations of the generic rule in eqn 4.2 are discussed in this section.

4.3.1 Spike timing dependent plasticity rules

To explore the consequences of experimental findings of synaptic plasticity in networks of spiking neurons, experimental data are often interpreted with heuristic models that parameterize certain aspects of the experimental findings. For example, *spike timing dependent plasticity* (STDP) rules are commonly used to describe plasticity with spiking neurons. As argued above, spike timing is essential for plasticity. Unfortunately, the amount of synaptic change depends on many factors when simply viewed as a function of pre- and post-synaptic activity. Bi and Poo used well-isolated pairings of single pre- and postsynaptic spikes to demonstrate STDP curves, as shown in Fig 4.7A. We have already discussed the calcium hypothesis of synaptic plasticity, and it is therefore expected, and seen in experiments, that different spike patterns can result in different plasticity curves. The following rules only apply strictly for isolated spike pairings. Additional specifications have to be included if, for example, multiple spikes or overlapping bursts are considered. The specific theoretical interpretation of the experimental data by Bi and Poo has thus to be applied cautiously. The rule is discussed here mainly because it has gained much influence in computational neuroscience,

In the situation of STDP as explored by Bi and Poo, we can write the rule for the synaptic efficiency change in dependence on the relative timing between

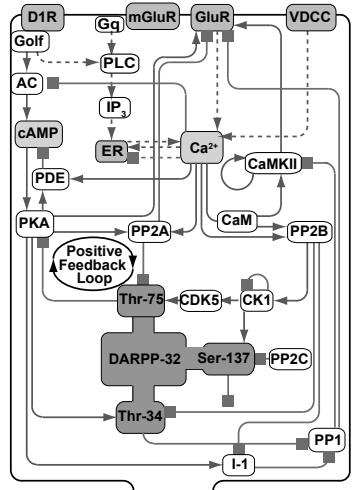


Fig. 4.9 Signalling pathways in the D1 receptors of striatal spiny neurons [adapted from Nakano, Doi, Yoshimoto, and Doya, *Society for Neuroscience* 676.5 (2007)].

a postsynaptic spike at t^{post} and presynaptic spike at t^{pre} as

$$\Delta w_{ij}^{\pm} = \epsilon^{\pm}(w) K^{\pm}(t^{\text{post}} - t^{\text{pre}}). \quad (4.3)$$

This form includes a weight-dependent learning rate, $\epsilon^{\pm}(w)$, and term that depends on the relative spike timings, $K^{\pm}(t^{\text{post}} - t^{\text{pre}})$. Also, we allowed for different functional forms of LTP (labelled with superscript '+') and LTD (labelled with superscript '-'). Following the experimental findings shown in Fig. 4.7A, the *kernel* function $K^{\pm}(t^{\text{post}} - t^{\text{pre}})$ is commonly taken to have an exponential form,

$$K^{\pm}(t^{\text{post}} - t^{\text{pre}}) = e^{\mp \frac{t^{\text{post}} - t^{\text{pre}}}{\tau^{\pm}}} \Theta(\pm[t^{\text{post}} - t^{\text{pre}}]). \quad (4.4)$$

We also allowed for different decay scale in potentiation and depression with different constants τ^+ and τ^- . $\Theta(x)$ is a threshold function that restricts LTP and LTD to the correct domains of positive and negative differences between pre- and postsynaptic spiking, respectively.

The basic asymmetrical Hebb rule (eqn 4.3 with kernel of eqn 4.4) suggests that when a postsynaptic spike follows a presynaptic spike in a small time window of a few milliseconds, then the synapse is potentiated. The next question, not addressed with the data in Fig 4.7A, is about what happens with repeated spike pairings. It is clear that the process can not go on forever. Synaptic efficacies must be restricted between a lower value (typically zero) and some upper value. A weight-dependent learning rate, $\epsilon^{\pm}(w)$, was therefore included in the formulation of eqn 4.3. The precise form of the weight-dependence is not yet established experimentally, and discussions of different forms of weight-dependent learning rates has become a major topic in theoretical studies. For example, a learning rule with a constant learning rate, also called an *additive learning rule*, is often considered. Such a learning rule might be reasonable, at least in some interval of synaptic weights,

$$\epsilon^{\pm} = \begin{cases} a^{\pm} & \text{for } w_{ij}^{\min} \leq w_{ij} \leq w_{ij}^{\max} \\ 0 & \text{otherwise} \end{cases}, \quad (4.5)$$

where a^{\pm} are constants for LTP and LTD. We call this rule an *additive rule with absorbing boundaries*, since theoretical weight changes outside the plasticity interval are absorbed by the boundaries. An alternative rule is a *multiplicative rule*, with more graded non-linearity when approaching the boundaries,

$$\epsilon^+ = a^+(w^{\max} - w_{ij}) \quad (4.6)$$

$$\epsilon^- = a^-(w_{ij} - w^{\min}). \quad (4.7)$$

It is also possible to consider other functional forms of learning rates. Distributions of synaptic weights can depend on the precise form of weight dependence, as discussed further below.

4.3.2 Hebbian learning in population and rate models

In models that describe the average behaviour of single neurons (*rate models*) or cell assemblies (*population models*), we cannot incorporate the spike timings.

We have therefore to fall back on a description of plasticity that depends only on the activity of pre- and postsynaptic nodes. The essence of Hebbian learning rules is then captured by the observation that the plasticity depends on the correlation of pre- and postsynaptic firing. Of course, these models can then not be used to investigate specific spike timing dependent issues, but as we discussed in the Chapter 1, different models are necessary for different types of questions. However, it is also a good idea to confirm some assumptions of population models with spiking models.

Most Hebbian plasticity rules used in population models have a functional form that can be expressed as

$$\Delta w_{ij} = \epsilon(t, w)[f_{\text{post}}(r_i)f_{\text{pre}}(r_j) - f(r_i, r_j, w)] \quad (4.8)$$

between a postsynaptic node i with a firing rate r_i , and a presynaptic node j with firing rate r_j . We included several parameters (constants or functions) in order to summarize popular rules in population models. The term ϵ represents the overall strength of changes and is often called the *learning rate*. The learning rate can depend on various factors, such as the weight values themselves, to confine weight values to a prescribed interval. Also, the learning rate is sometimes made explicitly time-dependent to describe, for example, external modulation of synaptic plasticity or developmental plasticity with different plasticity domains.

The functions f_{pre} and f_{post} depend on the firing rates of the pre- and postsynaptic nodes, respectively. For example, *Eduardo Renato Caianiello* was among the first² to specify Hebb's postulate mathematically with linear functions for f_{pre} and f_{post} in what he called the *mnemonic equation*,

$$\Delta w_{ij} = \epsilon(w)[r_i r_j - f(w)]. \quad (4.9)$$

The term f describes *weight decay*. We ignore weight decay in the following versions of Hebbian rules and discuss this separately in the next section. Therefore, keep in mind that the rules below have to be augmented with weight scaling in some form to achieve stable learning. For simplicity, we consider constant values for the learning rate in the following. The most basic Hebbian rule has then the form

$$\Delta w_{ij} = \epsilon r_i r_j, \quad (4.10)$$

which captures the correlation between the variables r_i and r_j . However, the functions f_{pre} and f_{post} typically include some *plasticity thresholds*, as seen in brain experiments. For example, in the generic form of a Hebbian rule introduced by *Terrance Sejnowski*, these thresholds are set equal to the average firing rates $\langle r_i \rangle$, where the angular brackets are used to denote the average of the quantity enclosed over many trials with different stimuli. The corresponding Hebbian learning rule,

$$\Delta w_{ij} = \epsilon(r_i - \langle r_i \rangle)(r_j - \langle r_j \rangle), \quad (4.11)$$

²Stephen Grossberg had already discovered such systems in the late 1950s, but his papers were not published until 10 years later.

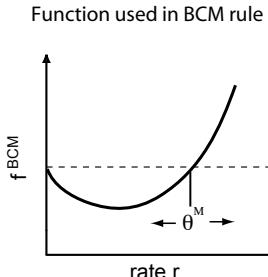


Fig. 4.10 Function as used in BCM theory with a modifiable threshold θ^M . BCM theory was invented by [Bienenstock, Cooper, and Munro [*Journal of Neuroscience* 2:32–48 (1982)]].

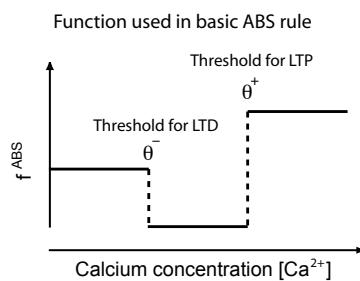


Fig. 4.11 A piecewise linear step function that provides three domains for synaptic plasticity, no change for $x < \theta^-$, LTD for $\theta^- < x < \theta^+$, and LTP for $x > \theta^+$. The dependent variable, x , is assumed to be calcium concentration in the Ca^{2+} hypothesis of Lisman [*Proceeding of the National Academy of Science* 86: 9574–8 (1989)] and Artola and Singer [*Trends in Neuroscience* 16: 480–7 (1993)].

is called the *covariance rule* since the average change of the weight value with this rule,

$$\begin{aligned}\langle \Delta w_{ij} \rangle &= \epsilon \langle (r_i - \langle r_i \rangle)(r_j - \langle r_j \rangle) \rangle \\ &= \epsilon (\langle r_i r_j \rangle - \langle r_i \rangle \langle r_j \rangle),\end{aligned}\quad (4.12)$$

is proportional to the covariance between pre- and postsynaptic firing. This is also called a cross-correlation function when ϵ is a proper normalization constant. This learning rule captures the basic suggestion of Hebb in that LTP is induced when the presynaptic firing rate and the postsynaptic firing rate are both systematically below, or above, their average firing rates, and that LTD is induced when one of the firing rates is below average while the other remains above.

We can distinguish the two conditions of LTD even further. When the firing rate of a presynaptic node is above average while the firing rate of the postsynaptic node is below average, then the weight of this specific synapse should be reduced as it cannot be relevant for the functional relevant postsynaptic firing. This only concerns a specific synapse, and this form of plasticity is therefore termed *homosynaptic LTD* (with Greek prefix *homo* = same). In contrast, when the postsynaptic node is active above the average firing rate, then all synapses of presynaptic nodes that are firing below average should be reduced to make the neuron respond more specifically to relevant input. We call this form of plasticity *heterosynaptic LTD* (with Greek prefix *hetero* = other) to stress that this type of plasticity concerns all the synapses of a postsynaptic neuron.

The covariance rule of eqn 4.11 captures the basic Hebbian learning necessary for building associators. It is in this sense a minimal model for Hebbian plasticity, which we will use frequently in later discussions. However, some variations of the basic Hebbian rule are important to notice, specifically rules for synaptic plasticity commonly called *BCM* theory after their inventors *Elie Bienenstock, Leon Cooper, and Paul Munro*. The basic plasticity rules in the BCM theory has the form

$$\Delta w_{ij} = \epsilon (f^{\text{BCM}}(r_i; \theta^M)(r_j) - f(w)). \quad (4.13)$$

The function f^{BCM} is outlined in Fig. 4.10. Low rates lead to LTD whereas high rates lead to LTP. Most importantly, the threshold between LTD and LTP, θ^M , is modifiable and can depend on the rate of the postsynaptic neuron. For example, the threshold becomes higher when the postsynaptic neuron becomes more active. Such a sliding threshold has not only been observed experimentally, but is also an important ingredient in keeping the system stable with this learning rule. BCM theory has been applied in particular to developmental plasticity, as in the formation of cortical maps.

A new interpretation of the BCM rule in terms of the calcium hypothesis was provided by *Alain Artola* and *Wolf Singer*, based on earlier work by Artola, Bröcher and Singer. The *ABS model* follows the calcium hypothesis by assuming a lower $[\text{Ca}^{2+}]$ threshold, Θ^- , to induce LTD and a higher $[\text{Ca}^{2+}]$ threshold, Θ^+ , to induce LTP. This hypothesis can be written in a form similar to the BCM rule above,

$$\Delta w_{ij} = \epsilon ([\text{Ca}^{2+}]) (f_{\text{ABS}}([\text{Ca}^{2+}]; \theta^+, \theta^-) - f(w)). \quad (4.14)$$

A simple example of a function f_{ABS} is outlined in Fig. 4.11. This function is constant in the different domains set by the threshold values,

$$f_{\text{ABS}}(x) = \begin{cases} 0 & \text{for } x < \theta^- \\ -1 & \text{for } \theta^- < x < \theta^+ \\ 1 & \text{for } x > \theta^+ \end{cases}, \quad (4.15)$$

although it is possible that there is a concentration dependence within these domains. This formulation in eqn 4.14 is only indirectly dependent on the activities of the pre- and postsynaptic nodes since the calcium concentration $[Ca^{2+}]$ is a function of several processes. For example, as reviewed in Fig. 4.8, postsynaptic calcium concentration is mediated by activation of different voltage-dependent and ligand-gated calcium channels, including NMDA and non-NMDA channels, and internal calcium stores. The calcium concentration depends on postsynaptic depolarization, which in turn depends on activation in the presynaptic cell. The activity dependence of the calcium level can be a complicated function of the presynaptic activities. However, if we assume a monotone, and mainly linear, relation between activity and calcium concentration, then we can write a simplified model such as

$$\Delta w_{ij} = \epsilon(f_{\text{ABS}}(r_i; \theta^-, \theta^+) \text{sign}(r_j - \theta^{\text{pre}})), \quad (4.16)$$

where θ^{pre} is the minimal presynaptic activation for potentiation. We could consider refined versions of f_{ABS} where the amount of potentiation and depression is varied with activity, but the essence of associative plasticity is already captured in this basic rule.

All the above rules must be augmented with mechanisms to keep weights bounded. For example, we can incorporate weight-dependent learning rates or weight-decay terms, as discussed further in Section 4.4. In the simplest case, we could just restrict the number of learning steps, or confine the weights to a certain interval. However, we can also consider more dynamic mechanisms, such as the activity-dependent threshold in the BCM theory.

Simulation

Here we will briefly show that the basic Hebbian plasticity rule can be implemented compactly in MATLAB. The basic Hebbian rule for rate models, eqn 4.10, states that a term

$$\Delta w = ba \quad (4.17)$$

is added to the previous weight w , where a is the firing rate of the presynaptic node and b is the firing rate of the postsynaptic node, both for a given training example. We can incorporate all the synapses on a postsynaptic node by collecting the weight values in a vector \mathbf{w} , and the rates of the presynaptic nodes in a column vector \mathbf{a} . We use a row vector to collect the weights on a single postsynaptic node, but we always write rates of nodes as column vectors. Similarly, we can also consider a column vector, \mathbf{b} , of the training rates of all postsynaptic neurons. The learning rule can then be written as

$$\Delta \mathbf{w} = \mathbf{b}\mathbf{a}'. \quad (4.18)$$

We used thereby the prime symbol to indicates the transpose of a vector which converts a column vector into a row vector. The weights \mathbf{w} are now collected in a matrix since we multiply a column vector with a row vector. For example, lets consider three presynaptic nodes with rates $a = (1, 2, 3)'$ and two postsynaptic nodes with rates $b = (4, 5)'$. The weight matrix is then

$$\mathbf{w} = \begin{pmatrix} 4 \\ 5 \end{pmatrix} (1, 2, 3) = \begin{pmatrix} 4 & 8 & 12 \\ 5 & 10 & 15 \end{pmatrix}. \quad (4.19)$$

That is, the weight matrix has components, w_{ij} , where the first index labels positions in a column (the index of the postsynaptic node) and the second index labels positions in a row (the index of the presynaptic node),

$$\mathbf{w} = \begin{pmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1n^{\text{in}}} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2n^{\text{in}}} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ w_{n^{\text{out}}1} & w_{n^{\text{out}}2} & w_{n^{\text{out}}3} & \dots & w_{n^{\text{out}}n^{\text{in}}} \end{pmatrix} \quad (4.20)$$

The number n^{in} is the number of presynaptic neurons, and n^{out} is the number of postsynaptic neurons. A useful trick is to remember that the first label stands for 'to' and the second label for 'from', as in $w_{\text{to from}}$, or $w_{\text{post pre}}$.

If we serially apply all training vectors, then the change in weight values is the sum of all the changes from individual training examples. To write this mathematically, we collect all the column vectors of different training examples into an array, so that we get matrices \mathbf{a} and \mathbf{b} with components $a_{i\mu}$ and $b_{i\mu}$ for training example μ . The Hebbian weight matrix after training all the examples is then given by adding to the initial matrix \mathbf{a}^0 the sum over all the training examples, which can be written for the components,

$$w_{ij} = w_{ij}^0 + \sum_{\mu} b_{i\mu} a_{i\mu}. \quad (4.21)$$

The second term on the right-hand side corresponds to a matrix multiplication of the form $\mathbf{b}\mathbf{a}'$. Thus, eqn 4.18 corresponds to training all patterns with an initial weight matrix of $\mathbf{a}^0 = 0$. This formula can be written directly into MATLAB. For example, in program `weightDistribution.m`, discussed further in Section 4.4.2 and displayed in Table 4.1, the training is done in Line 9 with

```
w=(rPost-ar)*(rPre-ar)';
```

where `rPost` and `rPre` are matrices for postsynaptic and presynaptic rates of all training examples, respectively, and the constant `ar` is the average rate of the training examples. This rule is therefore an implementation of the covariance rule, eqn 4.11.

4.3.3 Negative weights and crossing synapses

A basic fact that should be observed in learning rules is that synaptic values are always positive, since these values represent weights. However, as discussed in

Section 2.2, IPSPs reduce the overall membrane potential. Thus, in neurons or population nodes that sum all the inputs, we can describe inhibitory synapses with negative weights. Of course, the type of synapse depends on the specific receptor type and cannot change. Thus, synaptic learning rules should not change the sign of weight values. While this is obvious for synapses in single neurons, either in spiking neurons or in rate models where the node represents the average frequency of single neurons, the situation for weights in population nodes is not so obvious.

As argued in Section 3.4, a population node represents a collection of neurons, and the ‘synapses’ of such a node describe the average influence of one population of neurons on to another population of neurons. These populations could, in principle, contain excitatory and inhibitory neurons in what we call *mixed population nodes*. The influence of one mixed population on to another could be either excitatory or inhibitory. Moreover, learning in the system, and synaptic plasticity within the mixed populations, could even change the overall effect of one population on to another. For example, a neuron becoming inhibitory even when starting from an excitatory value. Thus, in population models it is often common not only to interpret negative weights as inhibitory, but even to allow a domain crossing of such weight values. Such domain crossings certainly do not mean that excitatory receptors become inhibitory or vice versa. Learning with such models could describe developmental processes in which new synapses in the population might become active, or even new synapses being formed. In principle, one should no longer speak of synapses in population models, but the term has become common when describing contacts in neural networks.

In later chapters we will discuss common models with mixed population nodes. However, many population models do separate excitatory and inhibitory populations. We will later argue about the differences and appropriateness of these formulations using some specific examples, in particular in models of cortical maps in Chapter 7 and auto-associative memory models in Chapter 8. As with all models, the appropriateness has to do with the scientific questions that the model is designed to answer, as well as the ability to capture the effects under investigation sufficiently. The mixed population node is sufficient for most of the discussions in this book. Such models are typically more compact and easier to study analytically. In contrast, models with separate inhibitory and excitatory nodes are easier to relate to biological networks. Such models can also include properties not present in the simplified mixed population versions, but most mechanisms of brain process discussions in this book can be followed with exemplary models of mixed population nodes.

4.4 Synaptic scaling and weight distributions

Repeated application of additive association rules results in runaway synaptic values. Many modelling examples include a simple restriction of the range for synaptic values, either explicitly, or through a finite number of training steps used in the simulations. While such approximations are sometimes sufficient and appropriate in terms of the modelling objective, real neural systems need some balance and competition between synapses to allow stable learning in

diverse sensory situations. We therefore have to consider competitive synaptic scaling. We start this section with example applications of some of the above rules and demonstrate some resulting weight distributions.

4.4.1 Examples of STDP with spiking neurons

Synaptic efficiencies are continuously changing as long as learning rules are applied. However, to study the consequences of these rules we can consider the distribution of weights, and how they change with time. In general, since we typically consider learning many random patterns, we can think of weight changes as stochastic processes. There have been many methods developed to analyse stochastic systems, such as *Fokker–Planck equations* which describe the time evolution of the probability density function of systems like the one considered here. Such methods can be used to describe the drift and possible equilibrium distributions of synaptic weights. Synaptic distributions can also be explored with direct applications of the plasticity rules in simulations, as shown next.

We will now consider the basic STDP rule of eqn 4.3 with kernel 4.4. It is clear that the strength of synapses that consistently drive the spiking of a postsynaptic neuron are increased with this rule. The learning of synapses can be supervised, as in experiments, if we force the postsynaptic spiking to be in close temporal relation to a presynaptic spike train. In contrast, spikes of presynaptic neurons that are uncorrelated with the postsynaptic spiking can occur before, or after, the postsynaptic spike. Such uncorrelated events can therefore elicit random events of potentiation or depression. If the overall effect of depression for such events is stronger than average potentiation, then synaptic weights decrease over time for patterns that should not be associated with the firing of the postsynaptic neuron. We thus expect a bimodal distribution for such basic association rules, with some synapses tending toward the maximal possible strength, while others decay to zero.

To demonstrate this in more detail, we will now follow work of *Sen Song*, *Kenneth Miller*, and *Larry Abbott* and study a single IF neuron with 1000 excitatory synapses that are individually driven by presynaptic Poisson spike trains with average firing rates of 20 Hz. When we start the simulations we set all the synaptic weights to large values. As a consequence, the postsynaptic neuron fires with large frequencies in a very regular manner because the average synaptic input exceeds the firing threshold of the neurons, as discussed in Chapter 3. We then apply an additive STDP rule with marginally stronger LTD than LTP. The average synaptic weight decreases under these conditions, resulting in neuronal responses with lower firing rates and an increased coefficient of variation, as illustrated in Fig. 4.12A. Interestingly, some synapses survive and become the major driving source of the neuron. The weight distribution resulting after 5 min of simulated time using the additive STDP rule is shown in Fig. 4.12B. There are many synaptic weights clustering around very small values, with a few synapses that have large synaptic weights approaching the upper limit imposed in the simulations. Analytical results also indicate that weight distributions with the additive Hebbian rule have bimodal distributions with peaks approaching the limiting values.

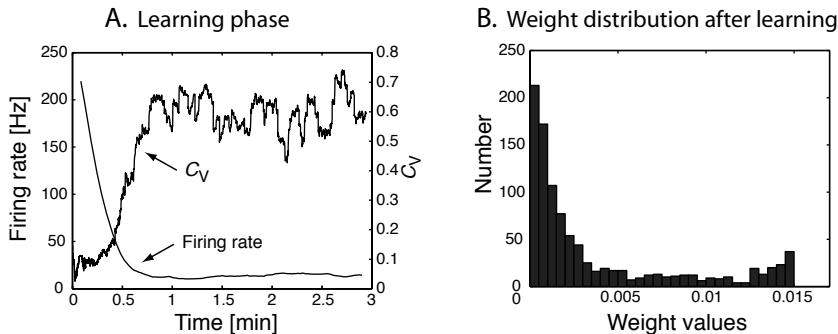


Fig. 4.12 (A) Firing rate and C_V , the coefficient of variation, of an IF neuron that is driven by 1000 excitatory Poisson spike trains while the synaptic efficiencies are changing according to an additive STDP rule with asymmetrical Gaussian plasticity windows. (B) Distribution of weight values after 5 min of simulated training time (which is similar to the distribution after 3 min). The weights were constrained to be in the range of 0–0.015. The distribution has two maxima, one at each boundary of the allowed interval. [These simulations follow closely the work of Song, Miller, and Abbott, *Nature Neuroscience* 3: 919–26 (2000).]

It is worth analysing the above experiments in more detail. In Chapter 3, we discussed the fact that IF neurons respond with regular firing to random spike trains if the synaptic efficiencies are strong enough so that the average current is larger than the firing threshold of the neuron (see Fig. 3.8). The firing time of the IF neuron in this mode is mainly determined by the average input current, so that each individual presynaptic spike is not solely responsible for a postsynaptic spike. We can quantify the average responsibility of presynaptic spike for a postsynaptic spike by measuring the cross-correlation function between them. To do this, we define a quantity $s(\Delta t)$ that has the value $s = 1$ if a spike occurs in the time interval Δt , and is zero otherwise ($s = 0$). With this spike train representation it is straightforward to quantify the correlation between the presynaptic spikes and the postsynaptic spikes using the *cross-correlation function*,

$$C(n) = \langle s^{\text{pre}}(t)s^{\text{post}}(t + n\Delta t) \rangle - \langle s^{\text{pre}} \rangle \langle s^{\text{post}} \rangle, \quad (4.22)$$

where the expressions enclosed in angular brackets denote the average over time of that quantity. The correlation is zero if the pre- and postsynaptic spike trains are independent (that is, $\langle s^{\text{pre}}s^{\text{post}} \rangle = \langle s^{\text{pre}} \rangle \langle s^{\text{post}} \rangle$). The correlation is larger than zero when there is, on average, more incidence of close relations between pre- and postsynaptic spikes. The quantity is negative if presynaptic spikes result in a consistent reduction of postsynaptic spikes (anti-correlation).

Measurements of such cross-correlations confirm that a single input spike train has little correlation with the output spike train when the IF neuron is in the regular firing regime. The measured average cross-correlations corresponding to the above simulation with initial weight values of $w = 0.015$ is shown in Fig. 4.13 with star symbols and interpolated with a dashed line. This resulted in regular firing of the postsynaptic neuron with a frequency of around 270 Hz. The average cross-correlation in these simulations is consistent with zero if we consider the results within their variance indicated by error bars. The similar

values of the average cross-correlations for positive and negative time shifts, Δt , indicate that the occurrence of a presynaptic spike before a postsynaptic spike is equally as likely as the occurrence of a presynaptic spike after a postsynaptic spike. The equal average occurrence of presynaptic spikes before and after a postsynaptic spike means that, statistically, LTP occurs as much as LTD. The average behaviour of the neuron would thus never change if the effects of LTP and LTD were equal. However, if the effect of LTD is a little bit stronger than that of LTP, such that the area under the LTD branch of STDP (Fig. 4.7) is larger than the area under the LTP branch, then we get a reliable decrease of the weight values over time.

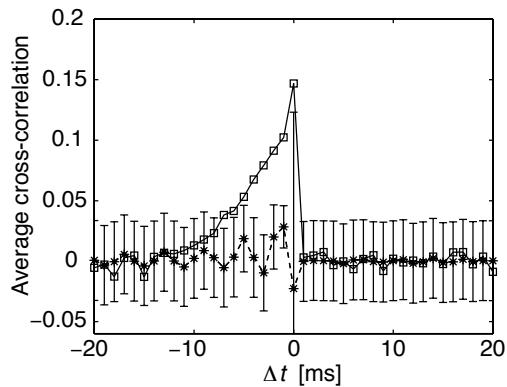


Fig. 4.13 Average cross-correlation function between presynaptic Poisson spike trains and the postsynaptic spike train (averaged over all presynaptic spike trains) in simulations of an IF neuron with 1000 input channels. The spike trains that lead to the results shown with stars were generated with each weight value fixed to 0.015. The cross-correlations are consistent with zero when considered within the variance indicated by the error bars. The squares represent results from simulations of the IF neuron driven by the same presynaptic spike trains as before, but with the weight matrix after Hebbian learning shown in Fig. 4.12. Some presynaptic spike trains cause postsynaptic spiking with a positive peak in the average cross-correlation functions when the presynaptic spikes precede the postsynaptic spike [see also Song and Abbott, *Neurocomputing* 32 & 33: 523–8 (2000)]. No error bars are shown in this curve for clarity.

The average cross-correlations between all presynaptic spikes and the postsynaptic spikes with the weights after learning, as shown in Fig. 4.12, are plotted in Fig. 4.13 with squares. These weights caused a much lower and more realistic frequency of the postsynaptic spike train around 18 Hz. The peak for small negative values of Δt in the cross-correlation curve indicates that some presynaptic spike trains were responsible for eliciting a postsynaptic spike. This increased correlation of some spike trains supports the LTP necessary to stabilize the corresponding synapses. *Song and Abbott*, who first elaborated the arguments outlined here, demonstrated that a steady state, with a reasonably large coefficient of variation, can be reached for different firing frequencies in the input spike trains. Of course, the time it takes to reach the steady state after changing the input frequency depends on the time it takes to modify the synaptic weights. Such biological details have not yet been determined

experimentally. In the case of a very fast transition to a steady state, this would mean that the neuron could also adapt to a change in the firing frequencies of inputs, something that has been termed *gain control*. However, too rapid a permanent change of synaptic weights has other disadvantages for the stability of learning.

While the above results with spiking neurons are encouraging, there are still many questions concerning the biological details of the STDP rules, specifically with regard to scaling mechanisms and the form of weight dependence. Also, to consider the stability of the learning process further, we have to consider such learning rules in networks of neurons. Before turning to networks in the next chapter, we need briefly to discuss scaling in rate models.

4.4.2 Weight distributions in rate models

For some discussions later in this book, it is useful to know the weight distribution of the Hebbian covariance rule (eqn 4.11), which is the dominant rule in population or rate models. In many studies using this rule, random patterns are used to represent the input patterns of the presynaptic nodes. The random numbers for postsynaptic responses are assumed to be independent of these inputs, at least during the learning phase. This is a reasonable assumption since learning is aimed at establishing correlations between input and output pattern, and learning should cease when this correlation is established. We thus consider independent firing rates, r_i and r_j , for the pre- and postsynaptic nodes.

If we take the basic, additive, Hebbian covariance rule of eqn 4.11 and start with weight values of $w_{ij}^0 = 0$, then we can write the weight values after learning N_p patterns as

$$w_{ij} = \frac{1}{\sqrt{N_p}} \sum_{\mu} (r_i^{\mu} - \langle r_i \rangle)(r_j^{\mu} - \langle r_j \rangle). \quad (4.23)$$

Here, we have used a learning rate of $\epsilon = 1/\sqrt{N_p}$ so that the width of the weight distribution does not change with the number of training patterns, for reasons we will see shortly. After subtracting the mean from each random number for the pre- and postsynaptic firing rates, we end up multiplying random variables with zero mean and variance σ^2 . Those new random variables are again random variables, which can have a different distribution than the distributions used to generate the firing pattern, as explained in Appendix C. It is important here that the learning rule of eqn 4.23 is a sum over N_p independent random variables. Then, the central limit theorem (see Appendix C.5) tells us that the sum is a random variable whose distribution approaches a Gaussian distribution with mean zero and variance σ^2/N_p , at least for very large sums. This is the reason for the specific normalization in eqn 4.23, since the width of the Gaussian distribution is proportional to the square root of the variance.

A Gaussian distribution is indeed what we find in numerical simulations of this rule. To demonstrate this, we will use exponentially distributed firing rates for the input and output patterns in a numerical experiment. The exponential distribution was chosen to mimic experimental findings, as discussed in Appendix D (see Fig. D.7). However, different distributions, such as the frequently studied patterns made out of equally distributed binary numbers,

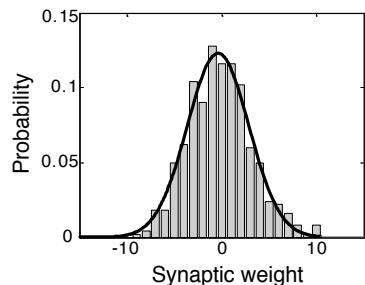


Fig. 4.14 Normalized histograms of weight values from simulations of 500 synapses on to a population node after training 1000 exponentially distributed random patterns with the Hebbian covariance learning rule 4.11. A fit of a Gaussian distribution to the data is shown as the solid line.

produce similar results. A frequency histogram that plots the relative number of weight values of many weights after 1000 learning steps, one for each pattern, is shown in Fig. 4.14, and the program is discussed below. We have included a fit of the simulated weight distribution to the Gaussian function that shows that the data are well approximated by a Gaussian distribution, as we predicted above.

Simulation

The following program, listed in Table 4.1, was used to produce the data for Fig. 4.14. We used 500 presynaptic nodes and one postsynaptic node. This network is trained on 1000 patterns. Training is done with the Hebbian covariance rule in Line 9, as explained in Section 4.3.2. We used matrix notations that are a direct representation of the mathematical formulas. We recommend the use of matrix operations whenever possible since this will produce code with considerably higher performance.

This program uses exponentially distributed firing rates similar to rate distributions of real neurons. It is straightforward to replace the corresponding lines in the program with other rate vectors (for example `rPre=rand(nn,npat);` etc.), and to test that the resulting weight distributions do not depend on the choice of rate vectors. Exponentially distributed numbers are thereby produced by a function of uniformly distributed variables. A function of a random variable produces another random variable with another probability density function. This new probability density function can be calculated with the help of cumulative density function as described in Appendix C.5. Random numbers from an arbitrary distribution can always be produced by appropriate sampling, such as choosing uniform random numbers covering at least all possible numbers of the distribution and accepting them with a probability proportional to the function itself. Finally, the statistics toolbox in MATLAB contains functions to produce random numbers for many other distributions.

Several normalizations are used in this program. In Line 10, we use a standard normalization to keep the width of the distribution independent of the number of training patterns. In Line 13, we divide w by the total number of nodes, since we use 500 nodes to create the histogram, and in Line 14, we normalize the histogram so that we can compare it to probability distributions, which must sum to one. The program also demonstrates how a function, in this instance the Gaussian function as specified in the MATLAB function file listed in Table 4.2, can be fitted to data points using the function `lsqcurvefit` of the *Optimization Toolbox*. There are further fitting functions available in the *Statistics Toolbox*, such as `nlinfit` and the corresponding interactive tool `nlintool`.

4.4.3 Competitive synaptic scaling and weight decay

The discussion above showed that we can constrain the width of weight distributions with small learning rates. Many demonstrations with computer simulations only use a small number of training patterns, so that runaway synapses are not a problem. However, maintaining a balance between synapses is ex-

Table 4.1 Program weightDistribution.m

```

1 %% Weight distribution of Hebbian synapses in rate model
2 clear; clf; %clear workspace and figure
3 nn=500; npat=1000; %number of nodes and patterns
4 %% Random pattern; firing rates are exponential distributed
5 ar=40; %average firing rate of pattern
6 rPre =-ar.*log(rand(nn,npat)); %exponential distr. pre rates
7 rPost=-ar.*log(rand(1,npat)); %exponential distr. post rate
8 %% Weight matrix
9 w=(rPost-ar)*(rPre-ar)'; %Hebbian covariance rule
10 w=w/sqrt(npat); %standard scaling to keep variance constant
11 %% Histogram plotting
12 x=-10:1:10;
13 [n,x]=hist(w/nn,x); %calculate histogram
14 n=n/sum(n); %normalizaton to get probability distribution
15 h=bar(x,n); set(h,'facecolor','none');
16 %% Fit normal ditribution to data
17 a0=[0 5];
18 a=lsqcurvefit('normal',a0,x,n);
19 n2=normal(a,-15:0.1:15);
20 hold on; plot(-15:0.1:15,n2,'r')

```

Table 4.2 Function normal.m

```

1 function y=normal(a,x);
2 % returns values of normalized Gaussian function with parameters a
3 y=1/(sqrt(2*pi)*a(2))*exp(-(x-a(1)).^2./(2*a(2).^2));
4 return

```

tremely important to build reliable and responsive systems. We can not discuss this topic in its entirety at this point, since network properties, such as inhibition, contribute. Here, we will discuss possible mechanisms for competitive synapses in single nodes. We discussed in Section 4.4.1 an example of synaptic scaling with additive STDP and restricted weights, where the number of strong channels is reduced for higher-frequency inputs. This is biologically realistic, as it has been found with fluorescent labelling that the number of active channels does vary in such situations. Some experiments have also demonstrated more directly that synaptic efficiencies depend on the average postsynaptic activity. For example, blocking some inhibitory channels of a neuron, which would increase its activity, scales down synaptic efficiencies, whereas forced reduction of postsynaptic activity leads to an increase of synaptic efficiencies. So far, the biological mechanisms of synaptic scaling are not completely understood. A simple implementation based on limited resources, which keep the summed synaptic strength constant, seems unlikely. However, there are new physiological findings that have the potential of solving the puzzle, such as regulation mechanisms of glial cells.

Although the precise biological mechanisms are not yet clear, we can model competitive synapses in various ways. Most of these mechanisms are mathematically equivalent to normalizing the weight vector. This normalization can be observed either strictly, or only dynamically on average (asymptotically). An example for a strict observation of the norm is to divide each weight value by the sum of all weights after each update,

$$w_{ij} \leftarrow \frac{w_{ij}}{\sum_j w_{ij}}, \quad (4.24)$$

which keeps the length of the weight vector to exactly one. This formula only holds when weight values are positive. In the case of positive and negative values, we need to divide by the square root of the sum of squares. In the formula, we used an arrow to indicate that the original weight values are replaced with the normalized values, instead of introducing a new symbol for the weights. The normalization in eqn 4.24 is an example of multiplicative (or divisive) scaling, since the rate of change depends on each weight value. While it is easy to implement in simulations, this normalization scheme is problematic since it is a non-local operation to sum all the weight values, which is computationally expensive and biologically unrealistic. Much more useable are dynamic implementations, which do not keep the weights strictly constant at each time step, but can enable useful scaling with local implementations.

The most direct way to model scaling is to include a separate weight dynamic on a time scale much longer than the synaptic plasticity discussed so far. For example, as recommended by *Mark van Rossum*, we can measure a running average, $a(t)$, of postsynaptic activity with a leaky integrator,

$$\tau_s \frac{da(t)}{dt} = -a(t) + \delta(t - t^{\text{spike}}), \quad (4.25)$$

where we used again the δ -function described in Appendix A.3. The time constant, τ_s , of this process should be on the order of minutes, or more. This average amplitude can then be used to scale the weights in a multiplicative

way, for example like

$$\frac{dw_{ij}(t)}{dt} = \beta w_{ij}(t)(a_{\text{goal}} - a(t)) + \gamma w_{ij}(t) \int_0^t dt(a_{\text{goal}} - a(t)), \quad (4.26)$$

where β and γ are constants. The first term on the right-hand side drives the weight toward a goal value, a_{goal} , and the second term minimizes the remaining error over time.

Another possibility, which is particularly elegant and useful for theoretical reasons, is to include such scaling dynamics together with the above-discussed plasticity in a form called *weight decay*. Such scaling mechanisms are particularly appropriate and useful in rate models since the independent variable is already a rate that we want to stabilize. Several decay terms have been discussed in the literature. For example, a simple weight decay can be added to the basic Hebbian rule,

$$\Delta w_{ij} = r_i r_j - c w_{ij}, \quad (4.27)$$

where the decay amplitude, c , has to be chosen appropriately. However, it is difficult to find a balance between appropriate reinforcement of this synapse and the constant decay term. It is therefore more appropriate to make this constant dependent on the rate of the output node. The Scottish neural network pioneer *David Willshaw* suggested the rule

$$\Delta w_{ij} = r_i(r_j - w_{ij}), \quad (4.28)$$

which often works quite well. This also looks similar to the basic Hebbian rule where w is the average postsynaptic rate that defines the postsynaptic plasticity threshold. Of course, it is possible to use other decay amplitudes, c , as functions of the postsynaptic rate, which can change how fluctuations around the mean are treated. For example, the Finnish scientist *Erkki Oja* suggested a quadratic form,

$$\Delta w_{ij} = r_i r_j - (r_i)^2 w_{ij}, \quad (4.29)$$

which is particularly useful, as discussed next. Weight decay terms can be derived from penalty terms in objective functions when deriving learning rules as minimizers of objective functions, as discussed in Chapter 6. Finally, as already mentioned, *Bienenstock, Cooper, and Munro* long ago suggested a scaling scheme that seems to have more biological realism. They pointed out that the threshold for LTP depends on the time-averaged recent activity of a neuron. Only postsynaptic activity that exceeds this threshold can induce LTP, otherwise LTD is induced.

4.4.4 Oja's rule and principal component analysis

The Hebbian rule with a weight decay term, as suggested by Oja (eqn 4.29), is particularly interesting as this not only normalizes the weight vector to unit length, but also has further theoretical advantages. Let us study the consequence of this rule on a linear node with two input channels (Fig. 4.15) that is driven by random input taken from a two-dimensional probability distribution

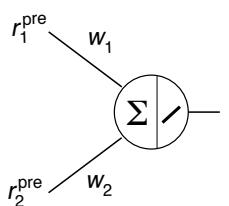


Fig. 4.15 A linear summation node with two input channels used in the simulations of the principal component example.

with zero mean. The details of this computer experiment are outlined below. The training examples of the pairs $(r_1^{\text{pre}}, r_2^{\text{pre}})$ define points in the $r_1^{\text{pre}} - r_2^{\text{pre}}$ plane. These training examples are shown as dots in Fig. 4.16. Before applying Oja's learning rule we initialized the weight vector with random values. The end point of the randomly chosen initial weight vector is shown as a cross in the figure. The trajectory of the end point of the weight vector during learning, when applying Oja's rule, is shown as the line extending from the cross that marks the initial weight values. At the end of the training session, after the network has been trained on a large number of sample points from the distribution, the weight vector converged to the point indicated in the figure as an arrow. The final weight vector has a length of $|w| = 1$.

While the weight normalization is important, as discussed above, Oja's rule also implements efficiently an algorithm called *principal component analysis* (PCA). The purpose of PCA is to describe high-dimensional data in a reduced yet meaningful way. Such data reduction can be illustrated with the example shown in Fig. 4.16. The direction of the largest variance of the data in this figure, which is the direction given by the weight vector after training, is called the *first principal component*. The variance in the perpendicular direction, which is called the *second principal component*, is less. In higher dimensions, the next principal components are in further perpendicular directions with decreasing variance along the directions. If one would be allowed to use only one quantity to describe the data, then one can choose values along the first principal component, since this would capture an important distinction between the individual data points. Of course, we lose some information about the data, and a better description of the data can be given by including values along the directions of higher-order principal components. Describing the data with all principal components is equivalent to a transformation of the coordinate system and thus equivalent to the original description of the data.

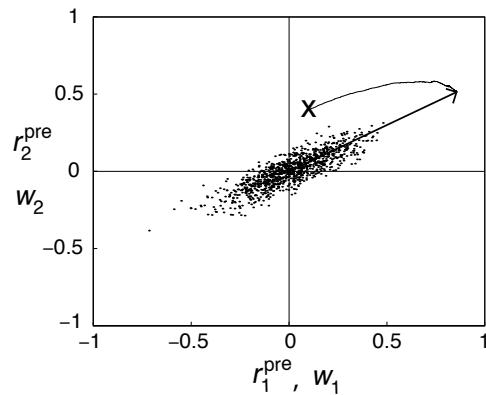


Fig. 4.16 Simulations of a linear node trained with Oja's rule on training examples (indicated by the dots) drawn from a two-dimensional probability distribution with mean zero. The weight vector with initial conditions indicated by the cross converges to the weight vector (thick arrow), which has length $|w| = 1$ and points in the direction of the first principal component.

Training a population node with Oja's rule results in a training vector along the first principal component. This is an example of the extraction of central tendencies, or prototypes, as mentioned in Section 4.1.4. The prototype is here the direction of the first principal component, and the values of individual data points along this direction distinguish each individual from the prototype. A

single node can only express a single quantity describing the input data, but it is also possible to design networks in which other nodes represent further principal components. Principal component analysis is based on the assumption that the components of the data vectors are statistically independent. While this is true in the example shown in Fig. 4.16, since we chose the components independently in the program, components of natural patterns are often correlated. For example, let us think about the variables in the data sets as representing the age and height of children. The positive correlation between these values is apparent in the data, but we can also ask which combinations of these values provide independent quantities for describing the data. This form of analysis is called *independent component analysis* (ICA), and much progress has been made in implementing such an analysis with neural networks.

Simulation

An example using an associative node with Hebbian learning and weight decay to perform PCA is given in program `oja.m` (Table 4.3). The network for this particular two-dimensional problem has only one linear node with two input channels. This architecture is specified by the initial weight matrix with arbitrary values in Line 3. In Line 4, a rotation matrix is defined to, which is used later to generate training examples. The training over 1000 training examples is done in the loop between Lines 6 and 12. In each of these training steps, a new training point is chosen in Line 7 and plotted in Line 8, the node output is calculated in Line 9, and this postsynaptic value is used in the program implementation of the learning rule eqn 4.29 in Line 10 with a learning rate of 0.1. In Line 11 we record the weight trajectory for later plotting. Note that

Table 4.3 Program `oja.m`

```

1 %% Linear associator with Hebb and weight decay: PCA a la Oja
2 clear; clf; hold on
3 w=[0.1;0.4]; % (arbitrary) starting value
4 a=-pi/6; rot=[cos(a) sin(a);-sin(a) cos(a)]; % rotation matrix
5 %% Training
6 for i=1:1000
7     rPre=0.05*randn(2,1).*[4;1]; rPre=rot*rPre; %training examples
9     plot(rPre(1),rPre(2),'.') % plot training point
8     rPost=w'*rPre; % network update
10    w=w+0.1*rPost*(rPre-rPost*w); % Hebbian training
11    w_traj(:,i)=w; % recording of weight history
12 end
13 %% Plotting results
14 plot(w_traj(1,:),w_traj(2,:),'r')
15 plot([0 w(1)], [0 w(2)],'k','linewidth',2)
16 plot([-1 1],[0 0],'k'); plot([0 0],[-1 1],'k')
17 axis([-1 1 -1 1]);

```

the training data are first chosen from a two-dimensional Gaussian distribution with standard deviation of 4 in one direction, and standard deviation of 1 in the other direction. This produces this elongated shape of data points, which is rotated by using the rotation matrix.

The trajectory of the weights is plotted in Line 14 with a red line, while the end vector of the weight is plotted with a thick black line in Line 15. Line 16 adds some axis, while Line 17 sets the axes limits. This program was used to produce figure Fig. 4.16. We only later added an arrowhead, a large X indicating the initial weight values, and some labels.

Exercises

- (4.1) In the example of learning associations shown in Fig. 4.3, the unconditioned and conditioned stimuli were always present without noise. Can such a model neuron learn associations with incomplete stimuli, where on average only one of the corresponding three channels is active?
- (4.2) Is spike timing dependent plasticity (STDP) only dependent on the spike timing?
- (4.3) Write a program that implements STDP for one synapse and show how the synaptic value changes with repeated synaptic events.
- (4.4) The program `weightDistribution.m` of Table 4.1 uses exponential distributed rate values of presynaptic neurons and the postsynaptic neurons. What is the resulting weight distribution if these rate values are chosen from a Poisson distribution? Explain.
- (4.5) Replace in program `oja.m` of Table 4.3 the normalization of weights with the Oja rule by the Willshaw rule. Plot how well the different methods approximate the direction of the first principal component during the learning process.

Further reading

The article by Abbott and Nelson (2000) is part of a special issue of *Nature* dedicated to computational theories. The article itself gives a good overview of asymmetrical Hebbian learning and synaptic scaling. The article by Artola and Singer (1993) is a classic, outlining the calcium hypothesis of synaptic plasticity. It contains references to earlier work by Lisman and by the authors together with S. Bröcher. The article by Rossum, Bi, and Turrigiano is a nice example of a computational neuroscience paper, which also contains the use of the Fokker–Planck equation to calculate weight distributions.

Laurence F. Abbott and Sacha B. Nelson (2000), *Synaptic plasticity: taming the beast*, in *Nature Neuroscience (suppl.)* 3: 1178–83.

Alain Artola and Wolf Singer (1993), *Long-term depression of excitatory synaptic transmission and its relationship to long-term potentiation*, in *Trends in Neuroscience* 16: 480–7.

Mark C. W. van Rossum, Guo-chiang Bi, and Gina G. Turrigiano (2000) *Stable Hebbian learning from spike timing-dependent plasticity*, in *Journal of Neuroscience* 20(23): 8812–21.

Part II

Basic networks

This page intentionally left blank

Cortical organization and simple networks

5

This chapter follows a top-down approach to provide further background of brain organizations. It summarizes some *anatomical organizations*, starting with a global brain anatomy before moving to more detailed organizations of the *layered architecture neocortex*. We also mention a functional organization of cortical maps that will be modelled further in Chapter 7. The second part of this chapter starts exploring networks of neurons. We begin with exploring information transmission in chains of simple neurons. This discussion will show that transmission of information in such networks is not reliable without *diverging-converging chains*. We then study some properties of random networks where *recurrencies* become important, and we will see that balanced activity is not easy to achieve in random recurrent networks.

5.1 Organization in the brain

Mental functions such as perception and learning motor skills are not accomplished by single neurons alone. These functions are an emerging property of specialized networks with many neurons that form the nervous system. The number of neurons in the central nervous system is estimated to be on the order of 10^{12} , and it is demanding to explore such vast systems of neurons. Therefore, rather than trying to rebuild the brain in all its detail on a computer, we aim to understand the principal organization of brains and how networks of neuron-like elements can support and enable particular mental processes. Integration of neurons into networks with specific architectures seem to be essential for such skills. We will explore the computational abilities and specialities of several principal architectures of neural networks in this book.

A thorough knowledge of the anatomy of the brain areas we want to model is essential for any research that attempts to understand brain functions. Although recent research has revealed many important facts about neural organization, it is still often difficult to specify all the components of a model on the basis of anatomical and physiological data alone, and plausible assumptions have to be made to bridge gaps in the knowledge. Even if we can draw on known details, it is often useful to make simplifying assumptions that enable computational tractability or the tracing of principal organizations sufficient for certain functionalities. It is beyond the scope of this book to describe all the details of neuronal organization, and more specialized books and research articles have to be consulted for specific brain areas. We will outline here primarily some principal organizations of *neocortex*, the outer layer of the *cerebral*

5.1 Organization in the brain	119
5.2 Information transmission in random networks ◇	130
5.3 More physiological spiking networks	137
Exercises	142
Further reading	142

cortex often simply referred to as *cortex*. Some examples of other large-scale organizations of the brain and some organizations of specific brain areas such as the *cerebellum* and some *subcortical areas* are given later in this book.

5.1.1 Large-scale brain anatomy

Regions of the neocortex are commonly divided into four *lobes* as illustrated in Fig. 5.1, the *occipital lobe* at the rear of the head, the adjacent *parietal lobe*, the *frontal lobe*, and the *temporal lobes* at the flanks of the brain. Further subdivisions can be made, based on various criteria. For example, at the beginning of the 20th century the German anatomist Korbinian Brodmann identified 52 cortical areas based on their *cytoarchitecture*, the distinctive occurrence of cell types and arrangements, which can be visualized with various staining techniques. Brodmann labelled the areas he found with numbers, as shown in Fig. 5.1. Some of these subdivisions have since been refined, and letters following the number are commonly used to further specify some part of an area defined by Brodmann. Brodmann's cortical map is, however, not the only reference to cortical areas used in neuroscience. Other subdivisions and labels of cortical areas are based, for example, on functional correlates of brain areas. These include behavioural correlates of cortical areas as revealed by brain lesions or functional brain imaging, as well as neuronal response characteristics identified by electrophysiological recordings.

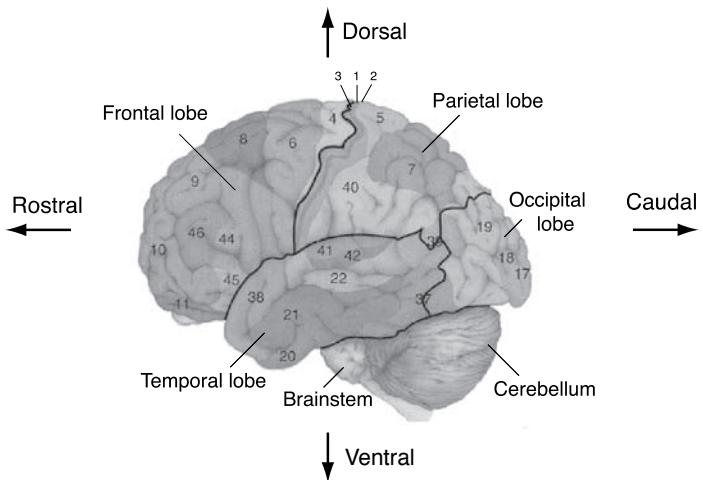


Fig. 5.1 Outline of the lateral view of the human brain including the neocortex, cerebellum, and brainstem. The neocortex is divided into four lobes. The numbers correspond to Brodmann's classification of cortical areas. Directions are commonly stated as indicated in the figure [adapted from Bear, Connors, and Paradiso, *Neuroscience: exploring the brain*, Williams and Wilkins (1996)].

It is, of course, of major interest to establish functional correlates of different cortical areas, a challenge that drives many physiological studies. We might speculate that the diverse functional specialization within the neocortex found with electrophysiological measurements is reflected in major structural differences among the different cortical areas to support specialized mental functions. It is therefore remarkable to realize that this is not the case. Instead, it is found that different areas of the neocortex have a remarkably common neuronal organization. All neocortical areas have anatomically distinguishable layers as

discussed below. The differences in the cytoarchitecture, which have been used by Brodmann to map the cortex, are often only minor compared to the principal architecture within the neocortex, and these variations cannot account solely for the different functionalities associated with the different cortical areas.

The neocortex is different in this respect to older parts of the brain, such as the brainstem, where structural differences are much more pronounced. This is reflected in a variety of more easily distinguishable nuclei. We can often attribute specific low-level functions to each nucleus in the brainstem. In contrast to this, it seems that the cortex is an information-processing structure with more universal processing abilities that we speculate enable more flexible mental abilities. It is therefore most interesting to investigate the information-processing capabilities of neuronal networks with a neocortical architecture.

5.1.2 Hierarchical organization of cortex

A common feature of neocortex is that there are *primary sensory areas* in which basic features of sensory signals are represented, while other areas seem to support more complex representations or mental tasks. Let us highlight this common view of neocortex with the example of vision. The primary visual area that receives major input from the eyes lies in the caudal end of the occipital lobe and is called V1. Information is then transmitted to other visual areas in the occipital lobe before splitting into two major processing streams, the *dorsal stream* along a parietal to frontal pathway, and the *ventral stream* along the temporal lobe. It has been argued that the dorsal stream is specifically adapted to spatial processing, whereas the ventral stream is well equipped for object recognition. We will investigate a model of such *what-and-where* processing later in Section 9.1.2, here we only want to point out that brain scientists try to identify functional specific areas and connections between these areas.

In order to understand how different brain areas work together it is important to establish the anatomical and functional connectivity between brain areas in more detail. Anatomical connections are not easy to establish as it is extremely difficult to follow the path of stained axons through the brain in brain-slices (including the branches that can often have different pathways). This is a daunting task, though it has been done in isolated cases. There are other methods of establishing connectivities in the brain. These include the use of chemical substances that are transported by the neurons to target areas or from target areas to the origin. Functional connectivity patterns, in which we are particularly interested when studying how brain areas work together, can also be established with simultaneous stimulations and recordings in different brain areas. Such experiments show correlations in the firing patterns of neurons in different brain areas if they are functionally connected. Also, some large-scale functional brain organizations can be revealed by brain imaging techniques such as *functional magnetic resonance imaging* (fMRI), which can highlight the areas involved in certain mental tasks. Such studies established clearly that different brain areas do not work in isolation. On the contrary, many specialized brain areas have to work together to solve complex mental tasks.

Some scientists, such as *Van Essen* and colleagues, have long tried to compile experimental data into connectivity maps similar to the one shown in Fig. 5.2.

The specific example was produced by *Claus C. Hilgetag, Mark A. O'Neill, and Malcolm P. Young*. The researchers used a *neuroinformatics* approach. Neuroinformatics is specifically concerned with the collection and representation of experimental data in large databases to which modern *data mining* methods can be applied. Hilgetag and colleagues considered an algorithm that would evaluate many possible configurations, and they found a large set of possible connectivity patterns in the visual cortex satisfying most of the experimental constraints. Each box in Fig. 5.2 represents a cortical area that has been distinguished from other areas on different grounds, typically anatomical and functional. The solid pathways between these boxes represent known anatomical or functional connections. The order from bottom to the top indicates roughly the hierarchical order in which these brain areas are contacted in the information-processing stream, from primary visual areas establishing some basic representations in the brain to higher cortical areas that are involved in object recognition and the planning and execution of motor actions. The authors also took the two basic visual processing pathways in their representation into account, plotting brain areas of the dorsal stream on the left side and the ventral stream on the right side. Note that there are also interactions within these pathways.

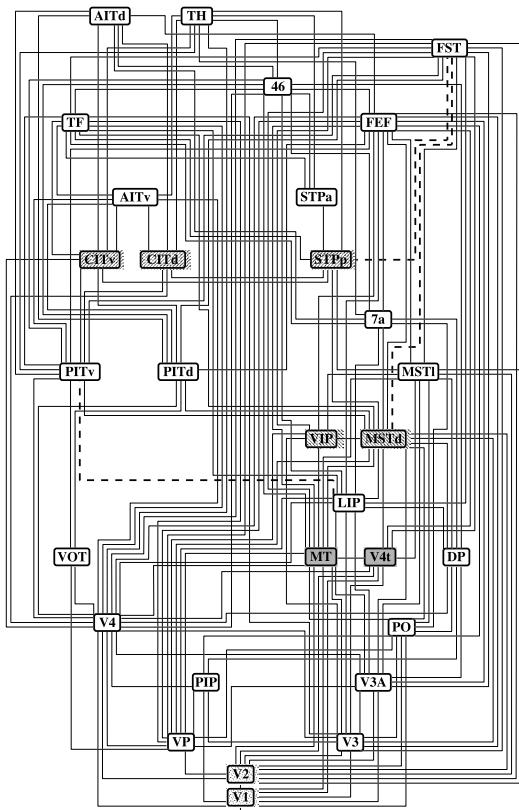


Fig. 5.2 Example of a map of connectivities between cortical areas involved in visual processing [reprinted with permission from C. Hilgetag, M. O'Neill, and M. Young, *Phil. Trans. R. Soc. Lond. B* 355: 71–89 (2000)].

Interestingly, most solutions of the numerical optimization problem have displayed some consistent hierarchical structures. All solutions found violated some of the experimental constraints (dashed line in Fig. 5.2), which is probably based on the inaccuracy of some of the experimental results. Also, the connections indicated are not unidirectional. It is well established that a brain area that sends an axon to another brain area also receives back-projections from the structures it sends to. Such back-projections are often in the same order of magnitude as the forward projections. Interesting examples, not included in Fig. 5.2, are so-called *corticothalamic loops*. The subcortical structure called the *thalamus* was initially viewed as the major relay station through which sensory information projects to the cortex. However, it is becoming increasingly clear that the notion of a pure relay station is too simple as there are generally many more *back-projections* from the cortex to the thalamus compared to the forward projections between the thalamus and the cortex. Some estimates even indicate a number of back-projections that exceed the forward projections tenfold. The specific functional consequences of back-projections between the thalamus and the cortex as well as within the cortex itself are still not well understood. However, such structural features are consistent with reports of the influence of higher cortical areas on cell activities in primary sensory areas, for example, attentional effects in V1.

Two final remarks on cortical maps such as the one illustrated in Fig. 5.2. The connections between brain modules are simply indicated with lines in such figures. However, note that the nature of those connections can be have some structure within the sending or receiving brain areas. For example, the connections could be specific to certain neurons, or there could be an organization in the connections such as in topology preserving maps discussed below. The wiring diagram shown as an example in Fig. 5.2 is therefore limited to highlight a functional modularity in the brain, although the modularity is somewhat imposed by definition. Also, we need to keep in mind that the visual system illustrated in Fig. 5.2 interacts with other areas that are not only dedicated to visual processing.

5.1.3 Rapid data transmission in the brain

From the system level view of the brain it seems that there are many stages of processing in the brain so that achieving even basic tasks like object recognition could take a considerable time. An good illustration of how quickly information can be transmitted through the brain is provided by the research of *Simon Thorpe* and colleagues. They showed that human subjects are able to discriminate the presence or absence of specific object categories, such as animals or cars, in visual scenes that are presented for very short times, as short as 20 ms. The percentage of correct manual responses, which consisted of releasing a button only when an animal was present in a complex image that was presented for 20 ms, is shown for 15 subjects in Fig. 5.3A plotted against the mean reaction time for each subject. The experiment shows some trade-off between reaction time and recognition accuracy, but the important point to note here is the high level of performance for such short presentations of the images.

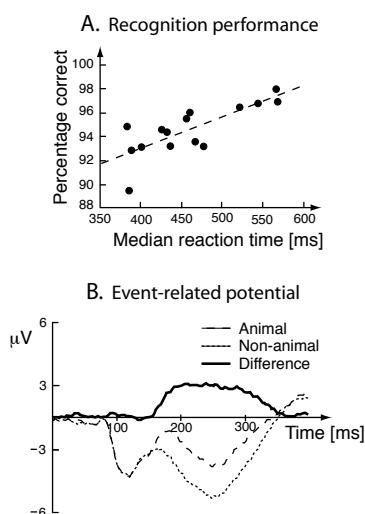


Fig. 5.3 (A) Recognition performance of 15 subjects who had to identify the presence of an animal in a visual scene (presented for only 20 ms) versus their mean reaction time. (B) Event-related potential averaged over frontal electrodes across the 15 subjects [redrawn from Thorpe, Fize, and Marlow, *Nature* 381: 520–2 (1996)].

The ability to recognize objects with these short presentation times is not the only astonishing result in these experiments. The authors also recorded skull EEGs during the experiments. The event-related potential, averaged over frontal electrodes, is shown in Fig. 5.3B, separated for image presentations with and without animals. The average response is not different for the first 150 ms, but becomes markedly different thereafter. The response of the frontal cortex therefore already indicates a correct answer after 150 ms. This is remarkable because for such categorization tasks we know that neural activity has to pass through several layers of brain areas. Each neuron in the processing stream necessary for the categorization path must thus be able to process and pass on information in time intervals of the order of only 10–20 ms or so.

5.1.4 The layered structure of neocortex

Staining of cell bodies or neurites reveals a generally layered structure of the neocortex, as illustrated in Fig. 5.4. We include here some brief comments on experimental staining techniques to clarify assumptions and limitations of such techniques, since these techniques are often crucial for estimating parameters on which models are based. Many different staining techniques can be used to identify neurons or parts thereof. Some staining techniques, such as the *Nissl stain*, colour only the cell body and cannot be used to investigate dendritic or axonal organizations. The *Golgi stain*, based on a silver solution, can be used to visualize more parts of the neuron than those accessible by Nissl staining. When viewing illustrations of such stained tissues it is important to know that only a small percentage of neurons, on the order of only 1–2%, are stained by the Golgi staining method, and different neurons can have different receptivities to this stain. The appearance of neocortical slices visualized by different staining techniques is illustrated in Fig. 5.4A.

In addition to these traditional dyes there is now a variety of other staining techniques including direct *intracellular dye injections* reaching most parts of a neuron, *anterograde staining* that utilizes dyes that are taken up by the cell body and transported down the axons, and *retrograde staining* that utilizes dyes that are taken up by the terminal endings of axons and transported back to the cell body. The former two staining techniques can be used to identify the projection range of neurons, and the latter is useful to highlight the neurons that project into a particular brain area. Mastering such techniques and applying them carefully to get estimates of neuronal populations and dendritic or axonal organizations is a specialization within neuroscience on which computational neuroscientists rely heavily in order to develop biologically faithful models.

Historically, the neocortex is divided into six layers labelled with Roman numerals from I to VI, although more than six layers, commonly 10 including the white matter, can be identified and are included into the historical labelling scheme by further subdivisions. Layer IV is thereby subdivided into IVA, IVB, and IVC, and layer IVC is further subdivided into layers IVC α and IVC β . The extent, or thickness, of the layers varies throughout the neocortex up to a point where some layers are difficult to identify if not absent. Examples of stained slices from different areas within the neocortex are shown in Fig. 5.4B.

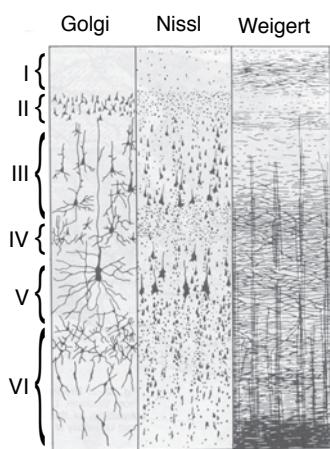
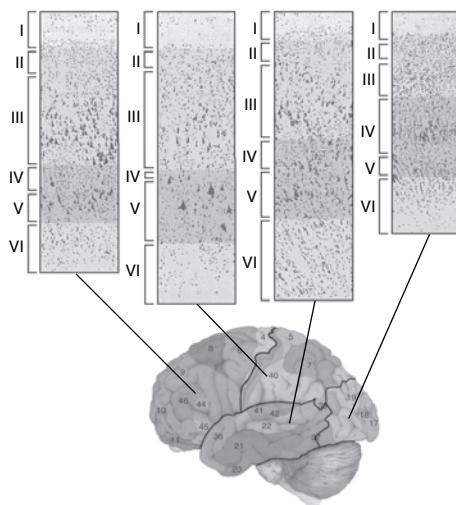
A. Different staining techniques**B. Variation in cortex**

Fig. 5.4 Examples of stained neocortical slices showing the layered structure of the neocortex. (A) Illustration of different staining techniques. [Adapted from Heimer, *The human brain and the spinal cord*, Springer, 2nd edition (1995)]. (B) Different sizes of cortical layers in different areas [adapted from Kandel, Schwartz, and Jessell, *Principles of neural science*, McGraw-Hill, 4th edition (2000)].

The visual appearance in the stained slices defining the layers is dependent on different populations of cell bodies and neurites. Layer I is easily distinguishable as it is mainly lacking in cell bodies and consists mainly of neurites. The other layers are marked by the domination of different cell types.

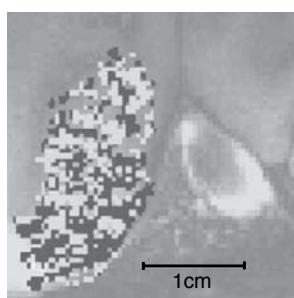
The soma of several neuronal types can be found in each neocortical layer, although the distribution can be used to mark the layers to some extent. As mentioned above, layer I is nearly completely lacking in cell bodies and consists mainly of neurites. Pyramidal cells can be found in most other layers of the neocortex. Layers II and III consist predominantly of small pyramidal cells, although the cells in layer III tend to be larger than those in layer II. Stellate neurons seem, in particular, concentrated around layer IV. In the upper part of this layer (IVA and IVB) one can find a mixture of medium-sized pyramidal cells and stellate cells, whereas the deeper layer (layer IVC) seems to be dominated by stellate neurons. Large pyramidal cells are found predominantly in layer V. A variety of cell types can be found in the deepest layer, layer VI. This includes Martinotti cells and also cells that have elongated cell bodies and are sometimes used to mark this layer. Such cells are sometimes called *fusiform neurons*.

5.1.5 Columnar organization and cortical modules

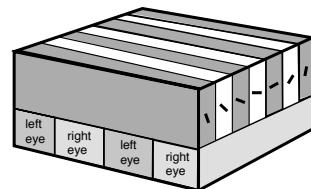
The neuronal organization in the neocortex discussed so far is mainly based on anatomical evidence. There is, in addition, an important functional organization in the neocortex revealed by electrophysiological recordings. These experiments have shown that neurons in a small area of the cortex respond to similar features of an input stimulus. Hubel and Wiesel have investigated such organization in the primary visual (or *striate*) cortex. Neurons in this cortical area respond to visual bars moving in particular directions. More precisely,

neurons in a small cortical column perpendicular to the layers and separated by around 30–100 μm respond to moving bars with a specific retinal position and orientation. These regions are called *orientation columns*. Separate from these arrangements are *ocular dominance columns*, cortical sections that respond preferentially to input from a particular eye (see Fig. 5.5A). The relations of orientation columns and ocular dominance columns are illustrated schematically in Fig. 5.5B. Neurons in small columns in other parts of the cortex also tend to respond to similar stimulus features. For example, cortical columns in the somatosensory cortex each respond to specific sensory modalities such as touch, temperature, or pain. The distribution of neurons with specific response characteristics is hence not purely random in the cortex, but there seems to be some form of organization.

A. Ocular dominance columns



B. Relation between ocular dominance and orientation columns



C. Topographic map of the visual field in primary visual cortex

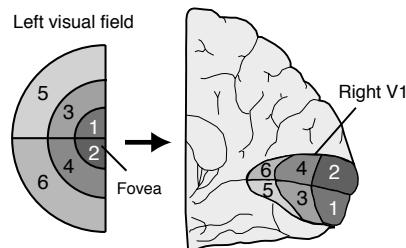
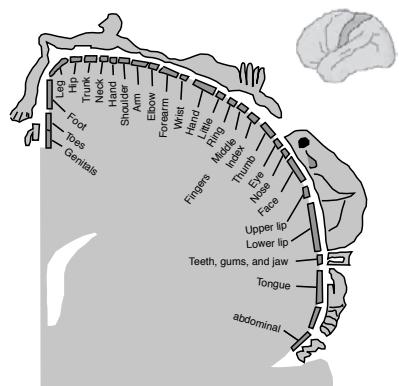


Fig. 5.5 Columnar organization and topographic maps in the neocortex. (A) Ocular dominance columns as revealed by fMRI studies [from Cheng, Waggoner, and Tanaka *Neuron* 32: 359–74, (2001)]. (B) Schematic illustration of the relation between orientation and ocular dominance columns. (C) Topographic representation of the visual field in the primary visual cortex. (D) Topographic representation of touch-sensitive areas of the body in the somatosensory cortex. [(C) and (D) adapted from Kandel, Schwartz, and Jessell, *Principles of neural science*, McGraw-Hill, 4th edition (2000).]

D. Somatosensory map



Hubel and Wiesel called a collection of orientation columns representing a complete set of orientations a *hypercolumn*. They showed that adjacent hypercolumns in the striate cortex respond to visual input from adjacent retinal areas as illustrated in Fig. 5.5C. Central regions of the visual field are represented by a larger cortical area than peripheral areas. The mapping between the visual field and the cortical representation is therefore not area-preserving; the central visual area is overrepresented, a feature that is called *cortical magnification*. However, the map preserves the relationships between adjacent points but not the area. Such maps are commonly labelled as *topographic* in the related literature. We will use this term in a general sense, meaning any map of

a feature space with some systematic relations between points (features) on the map. For example, a *tonotopic map*, which is a map of sound representations, is topographic when adjacent frequencies are represented at adjacent locations in the map. A hypercolumn itself is a topographic map as it contains an ordered representation of orientation, and there are many more examples of such maps in cortex and in subcortical areas. One other example is illustrated in Fig. 5.5D, that of the somatosensory cortex which represents tactile input from different body parts. This cortical area represents again more sensitive areas with larger cortical areas. More examples will appear later in this book, and we will discuss in Chapter 7 mechanisms that can explain how such cortical organization can be formed through experience.

Although we will often simplify cortical organization to discuss more general aspects of information processing in the cortex, it is conceptually important that neurons in small areas of the cortex respond to similar sensory stimuli. For many models in this book it is sufficient to represent the neurons in this area as a single unit, as discussed further below. With such models it is then much easier to explore various brain mechanisms, such as the formation of topographic organizations, as we will do in Chapter 7. In this chapter we continue to outline cortical organization and discuss some consequences of such organization on information processing within the cortex.

5.1.6 Connectivity between neocortical layers

The connectivity pattern within the layered structure of neocortex is becoming increasingly important for computational models. Neurons in layer IV seem to receive a particularly large number of afferents through the white matter from subcortical and other cortical areas. This layer is therefore often viewed as an input layer. Layer V has many large pyramidal cells with axons extending into the white matter. This layer seems therefore to contribute largely to the output of cortical processing. As the white matter is the main pathway between remote cortical areas and, in particular, between cortical and subcortical areas, it is obvious to suggest that the information flowing through the white matter is, to a large extent, responsible for global information transmission in the brain. In contrast, pyramidal cells in layers II and III are thought to be largely responsible for long-range cortico-cortical tangential (lateral) connections. Martinotti cells in the deep layers of the neocortex have axons extending into layer I. These could be responsible for information transfer between adjacent cortical modules from which pyramidal neurons in the upper layers receive synaptic input. Stellate neurons, on the other hand, seem to be more local in their neuritic sphere. The smooth stellate cells are therefore candidates for *inhibitory interneurons*. Their role in the stabilization of cortical processing is an important issue that we will discuss in later sections of this book.

An outline of connectivity patterns within a small column of the neocortex is summarized in Fig. 5.6. This scheme is, of course, only a rough approximation of the many details that are known experimentally. More detailed computational studies have still to be performed to understand the functional role of such organizations in more detail. An example of a model which incorporates laminar circuits is shown in Fig. 5.7. Grossberg and colleagues have now related

Fig. 5.6 Schematic connectivity patterns between neurons in a cortical layer. Open cell bodies represent (spiny) excitatory neurons such as the pyramidal neuron and the spiny stellate neuron. Their axons are plotted with solid lines that end at open triangles that represent the axon terminal. The dendritic boutons are indicated by open circles. Inhibitory (smooth) stellate neurons have solid cell bodies and synaptic terminals, and the axons are represented by dashed lines. [Adapted from Douglas and Martin, in *Synaptic organization of the brain*, Shepherd (ed.), (1990).]

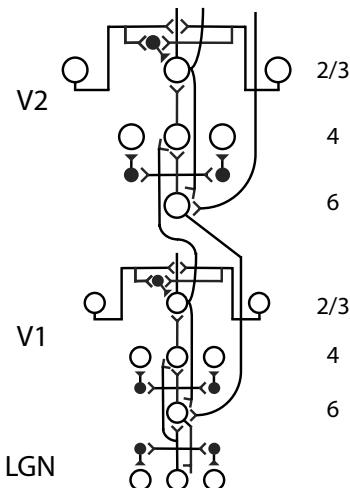
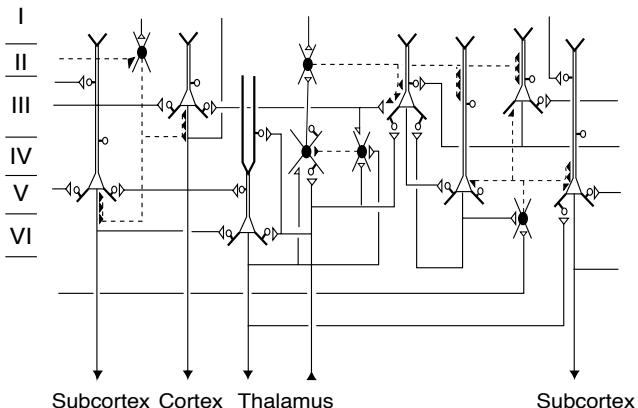


Fig. 5.7 Example of laminar model for the early visual system which attributes specific information processing abilities to the laminar circuits. [With permission from S. Grossberg, see Grossberg, *Spatial Vision* 12: 163-86 (1999).]

such laminar models to many physiological and psychological findings, extending and unifying much of their earlier work. It is thereby interesting to note that, even on this level, cortical areas do not work in isolation. In the example shown it is important to consider the combined layered network of cortical area V1 and V2. Grossberg and colleagues showed that the deep layers (4–6) support thereby item storage, normalization of signals, and contrast enhancement, and that superficial layers (2/3) support grouping of information across processing channels, important factor in forming higher-order representations.

5.1.7 Cortical parameters

It is not feasible, and not a major scientific focus, to extract a detailed wiring diagram of the brain. Even an estimation of cortical parameters, such as the number of neurons in a cortical area, the number of connections and their physiological strength, the composition of an area with neuronal types, etc., is often not easy to extract experimentally. In addition, most of the experimental estimations can only be made for particularly favourable cases from which we have to generalize. The generalizations of such experimental studies are often obscured by considerable variations of such numbers within different cortical areas and between different species. Also, the estimation of such parameters varies considerably with different experimental techniques. You might therefore ask yourself how we can build biologically faithful network models of the brain without the necessary experimental support.

The answer is that we have to approach the study of brain networks from different angles. We will study in this book primarily general network architectures and study the general computational capabilities of such networks. These studies reveal, as we will see throughout the book, that many computational abilities of the networks do not depend critically on specific details and are hence present in a large variety of networks within certain classes. Furthermore, we will discuss mechanisms that guide the development and fine tuning of networks to achieve specific computational tasks. We therefore approach the study of the brain from the perspective of extracting general principles

Table 5.1 Some rough estimates of neocortical parameters [see for example Abeles, *Corticonics: neural circuits of the cerebral cortex*, Cambridge University Press (1991)]

Variable	Value
Neuronal density	40,000/mm ³
Neuronal composition:	
Pyramidal	75%
Smooth stellate	15%
Spiny stellate	10%
Synaptic density	$8 \times 10^8/\text{mm}^3$
Synapses per neuron	1000–20,000
Distribution of synaptic types on pyramidal cell	
Inhibitory synapses	10%
Excitatory synapses from remote sources	45%
Excitatory synapses from local sources	45%
Asynchronous gain (relative synaptic efficiency)	0.003–0.2
Time duration of spike	~ 1 ms
Velocity of spike (myelinated axon of 0.02 mm diameter)	120 m/s
Length of axon	few mm to ~ 1 m
Synaptic cleft	20 nm
Synaptic transmission delay due to diffusion	0.6 ms

that guide the organization of the brain as well as revealing the computational consequences of classes of structurally related networks.

To explain brain functions we have, of course, to concentrate on the classes of networks that are consistent with brain networks. Predictions of models therefore have to be tested carefully with experiments on the real brain. Biologically faithful models can also be guided by experimental estimations of general cortical organization. In Table 5.1 we summarize some rough estimates of neocortical parameters that are good to be aware of when discussing biologically faithful network models. The values presented only indicate an order of magnitude, which is good to keep in mind when developing very general models, and we will see that it is already instructive to study models with some very crude approximations of cortical organization. More specific estimates for specific cortical areas that are modelled should, of course, be taken into account for more specific studies.

How much of the detail of neocortical organization is necessary to explain certain brain functions is difficult to assess and has to be considered for each specific question. Some specifics are certainly essential for very detailed explanations, while we can gain a lot of insight into some information-processing principles in the brain from very general organizational principles. There are also good reasons to believe that the brain itself has to work within general architectures in contrast to very detailed specific architectures coded, for example, genetically. A generally accepted hypothesis is that the brain architecture is based on genetically coded organizational principles on which self-organization mechanisms and experience-based learning act to fine-tune the organization to achieve accurate and flexible behaviour.

5.2 Information transmission in random networks ◇

We now begin to explore some behaviour of networks of simple neuron-like elements. In this section we follow first some thoughts of the neurobiologist *Moshe Abeles* who has analysed cortical connectivity patterns and studied some of the information-processing consequences of such architectures. The networks studied here are mostly random networks with probabilities of synaptic contacts resembling those of cortical organizations. These networks contrast the networks in the following chapters, which typically incorporate learning. We start with some discussions of information transmission in neuronal chains, which have also been advanced as *synfire chains* in the literature.

5.2.1 The simple chain

The simplest network of neurons we can consider is a simple chain of neurons as illustrated in Fig. 5.8A. A spike in the first neuron can propagate through the chain if the strengths of the synaptic connections are sufficient to elicit a postsynaptic spike in each subsequent neuron. Such a mode of information transmission is, however, biologically not reasonable for several reasons. First, a single presynaptic spike is often not sufficient to elicit a postsynaptic spike in the cortex and many other brain areas. At least two or more spikes are often necessary to elicit a postsynaptic spike (see Table 5.1). Second, synaptic transmission is lossy with probabilistic synapses as pointed out in Chapter 4. The probability of release of a sufficient amount of neurotransmitters by the arrival of a presynaptic action potential can be far less than 50%. Third, the death of a single neuron would disrupt the transmission in this chain permanently. Neuronal death is common throughout adult life and is so large that the probability of sustaining complete chains is unrealistically low. The employment of many parallel chains cannot solve the problems realistically as too many redundant chains would be necessary to ensure reliable transmission within the biological parameters mentioned above.

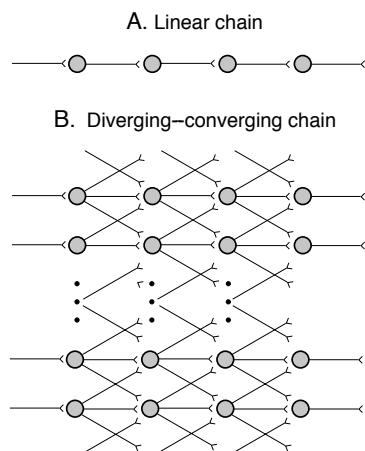


Fig. 5.8 (A) A sequential transmission line of four nodes. Parallel chains are made out of many such sequential transmission lines without connections between them. (B) Diverging-converging chains where each node can contact several other nodes in neighbouring transmission chains.

5.2.2 Diverging-converging chains

The restrictive nature of information transmission in simple chains suggests that the firing of a single neuron should not only depend on a single presynaptic neuron but also that a single neuron should transmit a spike to several other neurons. This is reflected in brain networks where a single neuron is contacted by many other neurons and in turn contacts many other neurons. In addition, the convergence and divergence of information flow through the brain enables more interesting information processing as discussed throughout this book. It is therefore instructive to consider *diverging-converging chains* as illustrated in Fig. 5.8B.

The neurons in each column do not have to be different from neurons in other columns. Rather, the illustration shows the pathways of information flow through time, each column representing a particular step in the information transmission when approximated by a discrete time step scheme. A node that

is involved in information transmission at some time when it gets a signal may get recruited again some time later when it gets some feedback from neurons to which it has projected. Neurons in such feedback loops would be repeatedly drawn in this scheme.

We call the number of neurons in each layer the *width of the chain* and denote it by N . This number can be equal to the number of neurons in a recurrent network. The number of neurons contacted by each neuron is called the *divergence rate* or *multiplicity* of the chain and denoted by m . In contrast, we call the number of connections that each neuron receives the *convergence rate* and denote it by C . We will mainly consider the case of equal divergence and convergence rate, $m = C$, for simplicity, although the analysis of networks with different rates is very similar. The case of $N = m = C$ is a *complete transmission chain*, which we will also call a *fully connected network*. A further parameter on which the transmission of spikes will critically depend in such networks is the synaptic weight, w , of a connection, as discussed in the last chapter. We first consider chains where all connection weights are of the same amplitude. The product of synaptic weight and the multiplicity factor has to be larger than one, $mw > 1$, to enable the transmission of a single spike through this diverging-converging chain.

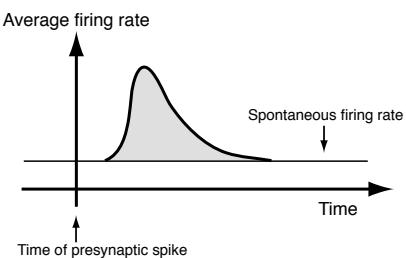
5.2.3 Immunity of random networks to spontaneous background activity

Cortical neurons typically fire with some background activity, which we assume in the following to be of mean 5 Hz with a variance of 3 Hz. We also assume that these spike events are independent. Thus, each neuron fires with the probability of $0.005 = 5/1000$ in each time interval of 1 ms. Let us consider a neuron that has 10,000 excitatory dendritic synapses, that is, it receives input from 10,000 spontaneously firing neurons. The sum of independent random presynaptic spikes arriving in each time interval at each single neuron is then normally distributed with mean $\mu = 10000 * 0.005 = 50$ and variance $\sigma^2 = 0.003$. This follows from the central limit theorem since the sum of N random numbers with mean μ and variance σ^2 is a normally distributed random number with mean $N\mu$ and variance σ^2 . The fluctuations of the arriving spikes from the background firing are very small so that at every time step almost exactly 50 spikes will arrive at the postsynaptic neuron. The neuron is immune to the background firing if it has small enough relative synaptic weights. If we consider that a spike arriving at a synapse with weight $w = 1$ would elicit a spike, then the weight would have to be $w < 1/50 = 0.02$ in order for the neuron to be immune against the background firing. In case of multiple synaptic contacts of a single presynaptic neuron with a postsynaptic neuron, this number should be divided by the multiplicity factor.

To compare these values to experimental data we have to consider how to measure the average synaptic efficiency. A way proposed in the literature is to stimulate a presynaptic neuron while recording from the postsynaptic neuron. It is likely that the postsynaptic neuron fires spontaneously, but the influence of the presynaptic spike should be seen in an average increase of the postsynaptic firing rate after many such trials. This is illustrated in Fig. 5.9. The value of the

area between the synaptic transmission curve and the spontaneous firing rate of the neuron is called the *asynchronous gain*. This value quantifies the average number of extra spikes that are added to the spikes of a postsynaptic neuron by each presynaptic spike. Reported experimental values of the asynchronous gain of synapses at cortical neurons are often in the range of 0.003–0.2 as mentioned in Table 5.1. The synaptic efficiencies of neocortical neurons could hence be small enough to make cortical neurons to a certain extent immune to the background firing of other neocortical neurons.

Fig. 5.9 Schematic illustration of the influence of a single presynaptic spike on the average firing rate of the postsynaptic neuron. The delay in the synaptic transmission curve is caused by some synaptic delay, after which, on average, more postsynaptic spikes are generated within a short time window compared to the spontaneous activity of the neurons [adapted from Abeles, *Corticonics*, p. 97, Cambridge University Press (1991)].



5.2.4 Noisy background

We have argued that the variance in the synaptic current from the background firing of other neurons could be very small. However, it is likely that the faulty transmission within the synapses generates a large variability in the effective postsynaptic currents from the background firing. Let us thus discuss the other extreme, in which the variance of the sum of presynaptic spikes from the background is on the order of the mean of this firing rate. Taking again the parameters from the previous example but considering now a variance of $\sigma^2 = 50$, we want to calculate again how small the synaptic efficiencies have to be in order for the probability of a postsynaptic spike generated by the background firing to be less than a certain value p^{bg} . This can be done by calculating the probability of having more than x simultaneous presynaptic spikes. For the Gaussian distribution in our example this is given by

$$P(n^{\text{spikes}} > x) = \frac{1}{\sqrt{2\pi}\sigma} \int_x^{\infty} e^{-\frac{(y-\mu)^2}{2\sigma^2}} dy \quad (5.1)$$

The integral on the right-hand site defines the *error function* (erf) introduced in Appendix C. Values of this functions are listed typically in statistics books, and this is also implemented in MATLAB. We can write eqn 5.1 thus equivalently as

$$P(n^{\text{spikes}} > x) = \frac{1}{2} [1 - \text{erf}(\frac{x - \mu}{\sqrt{2}\sigma})]. \quad (5.2)$$

The synaptic efficiencies therefore have to be small enough so that

$$x = \mu + \sqrt{2}\sigma \text{erf}^{-1}(1 - 2p^{\text{bg}}) \quad (5.3)$$

simultaneous spikes do not elicit a postsynaptic spike. With the values in our example and the probability $p^{\text{bg}} = 0.1$ for the background activity to elicit a spike we get $x \approx 59$, so that the average synaptic efficiencies have to be $w < 1/59 \approx 0.017$. This is still in agreement with experimental values as mentioned before. Synaptic efficiency values only slightly below this value would allow the neuron to be 90% immune against responding to background firing.

5.2.5 Information transmission in large random networks

If we incorporate the stability in response to strongly fluctuating input current from the background activity of other neurons by considering synaptic efficiencies $w < 0.017$, then we have just estimated that we need at least $59 - 50 = 9$ additional presynaptic spikes on top of the average background firing in the cortex to elicit a meaningful postsynaptic spike. Let us consider a large randomly connected network under these conditions, for example, of size 10^{10} , after injecting enough current into 1000 neurons to force them to spike. Each of these 1000 neurons connects to 10,000 other neurons. The spikes are therefore transmitted to nearly $1000 * 10,000 = 10^7$ different neurons because the chance of overlaps of receiving neurons is only very small in the randomly connected network. Indeed, we should calculate the probability that a single postsynaptic neuron receives two spikes, one from a separate initially stimulated neuron. Let us select one neuron. The probability that this neuron receives a spike is $10^7/10^{10}$. The probability that this neuron receives two spikes is then $(10^7/10^{10})^2 = 10^{-6}$, a very small probability. It follows that the 1000 initial spikes are, on average, not sufficient to elicit secondary spikes if more than one presynaptic spike is needed to elicit a postsynaptic spike.

The failure of spike transmission through a random network is, of course, a consequence of the small number of connections per neuron relative to the large number of neurons in the network, in conjunction with the equal likelihood of each neuron contacting each other neuron in the network. The situation improves in smaller networks. For example, let's consider only a small random network (cortical area) with 1 million (10^6) neurons (each having again 10,000 synapses with other neurons in this set) in which we stimulate 1 neuron; then each neuron has the probability of $10^4/10^6 = 0.01$ of receiving a spike or $1 - 0.01 = 0.99$ of not receiving a spike. If we stimulate 100 neurons, then the probability of not receiving a spike is only $0.99^{100} \approx 0.366$ and the probability of a neuron receiving spikes from two presynaptic neurons is $(1 - 0.366)^2 \approx 0.4$. Of course, the probability that presynaptic spikes will elicit secondary spikes also increases with more focused connectivity patterns as we have to suspect in the cortex (rather than the random connectivities discussed here). Considering a fully diverging-converging chain, we need only to stimulate two of those nodes to elicit a signal transmission because we considered two spikes to be sufficient to elicit a postsynaptic spike.

5.2.6 The spread of activity in small random networks

The probability of contact between two appropriately placed cortical neurons that have large overlaps between the axonal range of the sending neuron and the dendritic tree of the receiving neurons is much higher than we considered in the previous examples. Furthermore, with more specific connectivity, including multiple synapses between two neurons, we can assume higher total synaptic efficiencies between two such closely connected cortical neurons. It is therefore appropriate to assume that only very few active presynaptic neurons can elicit a postsynaptic spike in functionally correlated neurons. It is therefore instructive to study small networks with only a small number of highly efficient synapses. Such networks have been termed *netlets*. Netlets are intended to approximate some aspects of local cortical networks discussed in the following.

Let us first discuss the extreme case where a single presynaptic spike can elicit a postsynaptic spike in all the neurons to which the presynaptic neuron projects efficiently. Let us also assume that the number of these efficiently connected postsynaptic neurons is 10. We consider again a discrete updating scheme with basic time intervals of 1 ms. If we stimulate one neuron at time step $t = 1$ ms then we get 10 spikes at $t = 2$ ms. These 10 spikes will each elicit another 10 spikes so that we have 100 spikes at $t = 3$ ms (as long as the overlap in the target populations is small). We see that the activity of the whole network quickly builds up until all of the neurons are active. The activity of the network becomes smaller if we take some refractoriness into account. For example, we can prevent the model neurons from becoming active in the time step following the time step in which they were last active. This corresponds to an absolute refractory time of 1 ms. We can then expect that around half of the neurons in the netlet are active in any time step because it is likely that a neuron becomes active every second time step.

If we increase the number of spikes that are necessary to elicit a postsynaptic spike by lowering the synaptic weights, then the situation looks a little bit different. A small number of initial spikes will not spread through the network as long as the probability of a postsynaptic neuron receiving the necessary number of spikes simultaneously is low, similar to the arguments we discussed for the large network. However, in small networks we have a larger probability of receiving connections from any other subpopulation of neurons. In addition, if we stimulate initially a relatively large number of neurons in the netlet, then we have a large probability of initiating also secondary and tertiary, etc., spikes. Once we reach this critical number of active spikes we end up again with a quickly spreading activity throughout the network and with nearly half of the neurons active at any time step. Thus, there are two asymptotic modes in such networks: with small initial activity we get an inactive netlet, and with initial large activity we get a nearly maximally active netlet.

5.2.7 The expected number of active neurons in netlets

Random network models such as netlets can be analysed much more systematically. For example, Anninos and colleagues have derived a formula that describes the expected fraction of active nodes a in the netlets as a function

of the fraction of nodes that are active in the previous time interval. This is given by

$$\langle a_{t+1} \rangle = (1 - a_t) \left(1 - e^{-a_t C} \sum_{n=0}^{\Theta-1} \frac{(a_t C)^n}{n!} \right) \quad (5.4)$$

where the angle brackets indicate the expectation value (mean), C is the average number of synapses per neuron, and the firing threshold Θ is the number of presynaptic spikes necessary to elicit a postsynaptic spike. This formula is illustrated for $C = 10$ and various values of Θ in Fig. 5.10. The curves plotted in this figure represent the average expected behaviour when averaging over all possible realizations of netlets. Individual netlets with particular realizations of the connectivity pattern can deviate from these curves, but the curves describe what we expect in most cases with only randomly connected model neurons.

The sum of the fraction of active nodes in two consecutive time steps cannot exceed unity with a refractory time of one time step because an active node cannot be active in the following time step. The *saturation* of the network is therefore given by the decreasing diagonal in Fig. 5.10. The other diagonal indicates when the number of active nodes in one time step is equal to the expected number in the next time step. The number of active nodes at this point does not change; this is called a *steady state*. Network activity always increases in netlets for which part of the curve is above the steady state line. The curve for $\Theta = 1$ in Fig. 5.10A starts with a positive slope so that we expect a larger number of active model neurons in consecutive time steps following a small initial set of active nodes. The activity of netlets with parts of the curve below the steady state line always decreases. The activity of netlets with $\Theta > 1$ therefore decreases to zero network activity after small initial activations. The curves for netlets with $\Theta > 3$ never cross the steady state line and therefore always decrease to zero activity.

The activity of the netlet with $\Theta = 3$ will increase when the initial fraction of activity is larger than about $a = 0.15$. From there it will increase until the activity in the network reaches the second crossing of the steady state line. When the network activity increases further it will be pulled back in the next time step as indicated by the curve. Network activities below this steady state point will increase. This steady state point is therefore an *attractive fixpoint* of the network dynamics. In contrast, the first crossing of the curves with the steady state line is not stable due to the repulsive forces after variations of network activity around this point.

5.2.8 Netlets with inhibition

We have seen that netlets of excitatory neurons have stable fixpoints for low firing thresholds of the model neurons. These fixpoints are near to the saturation of the network with firing rates around 500 Hz. This is certainly in contradiction to cortical activity with much lower firing rates, more realistically in the range of 5–100 Hz. However, we have so far only considered networks of excitatory neurons. Inhibitory neurons, in particular inhibitory interneurons that respond to the activity in the network, can lower the steady state network activity of netlets. Anninos *et al.* also derived formulas similar to eqn 5.4 for netlets

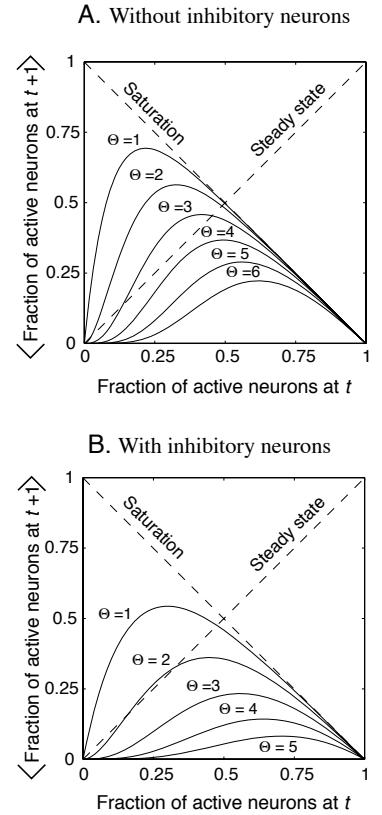


Fig. 5.10 The fraction of active nodes of netlets in the time interval $t + 1$ as a function of the fraction of active nodes in the previous time interval. The different curves correspond to different numbers of presynaptic spikes Θ that are necessary to elicit a postsynaptic spike. (A) Netlets with only excitatory neurons. (B) Netlets with the same amounts of excitatory and inhibitory connections. [Adapted from Anninos et al., *Journal of Theoretical Biology* 26: 121–8 (1970).]

that include inhibitory connections. This formula is included in the MATLAB implementations described below, which we used to plot the curves in Fig. 5.10. The behaviour of netlets with the same amounts of inhibitory and excitatory model neurons, both with the same number of synapses, $C_{\text{ex}} = C_{\text{in}} = 10$, is shown in Fig. 5.10B. The inhibition does indeed lower the netlet activity of stable steady states. However, the values still exceed typical cortical firing rates.

This discussion shows that it is not obvious that networks should have low firing rates. Even the incorporation of inhibitory model neurons did not improve the situation very much. The situation is worsened in cortical networks if we take into account that inhibitory interneurons become active only after some delay as they are driven by excitatory neurons that receive external input and thus have to fire first. Also, the inhibition does reduce the number of netlets with stable fixpoints other than zero to netlets with only low firing thresholds, as can be seen from a comparison of Fig. 5.10A and 5.10B. However, we should not forget that the netlet models we have just outlined are only very simple models with very crude approximations of real neurons. We saw in the previous chapter that real neurons have various characteristics, such as fatigue effects, that can help to achieve lower firing rates in cortical networks. Furthermore, netlet models are still network models with only random connectivity patterns. It is obvious that not much useful information processing can be achieved by purely random networks. We will therefore not consider random networks in more detail but will instead focus much more on network models with connectivity structures guided either by design or by self-organizing principles in most of the remainder of the book.

Simulation

The generalization of the formula 5.4 to a netlet with inhibitory neurons is given by

$$\langle a_{t+1} \rangle = (1 - a_t) e^{-a_t h C_{\text{in}}} \dots \\ \dots \sum_{n=0}^M \frac{(a_t h C_{\text{in}})^n}{n!} \left(1 - e^{-a_t (1-h) C_{\text{ex}}} \sum_{n=0}^{\Theta-1} \frac{(a_t (1-h) C_{\text{ex}})^n}{n!} \right), \quad (5.5)$$

where the number of excitatory and inhibitory synapses is given by C_{ex} and C_{in} , and h is the fraction of inhibitory nodes relative to excitatory nodes. The constant M is a sufficiently large number so that the value of the sum does not change much with the inclusion of higher terms.

The formula of eqn 5.5 is coded into MATLAB in program `anninos.m` shown in Table 5.2. In this example we use the same number of excitatory and inhibitory nodes, with the same number of synapses by setting the parameters `h`, `c_in`, and `c_ex` in Line 3. In Line 4, a loop over different thresholds is started, to show results for different parameters. Two temporary variables `tmp1` and `tmp2` are used to hold the values of $a_t(1-h)C_{\text{ex}}$ and $a_t h C_{\text{in}}$. The sums are calculated in Lines 9–11, and the expected fractions at t and $t+1$ are recorded in Lines 12 and 13, before being plotted in Line 15. The diagonals are added to the plot in Line 17, and Line 18 adds labels and produces the square shape of the graph.

Table 5.2 Program anninos.m

```

1 %% Plot netlet formula of Anninos et al.
2 clear; clf; hold on;
3 h=0.; c_in=10; c_ex=10;
4 for theta=1:7;
5     rec=0;
6     for a=0.:0.01:1.; %fraction of active nodes
7         rec=rec+1;
8         tmp1=(1-h)*a*c_ex; tmp2=h*a*c_in;
9         s1=0; s2=0;
10        for n=0:theta-1; s1=s1+tmp1^n/prod(1:n); end;
11        for n=0:20; s2=s2+tmp2^n/prod(1:n); end;
12        a0(rec)=a;
13        a1(rec)=(1-a)*exp(-tmp2)*s2*(1-exp(-tmp1)*s1);
14    end
15    plot(a0,a1);
16 end
17 plot([0 1],[0 1],’k’); plot([0 1],[1 0],’k’)
18 xlabel(’a(t)’); ylabel(’a(t+1)’); box on; axis square

```

5.3 More physiological spiking networks

In this final section we follow some work of *Eugene Izhikevich* combining his neuron models (see Section 3.1.5) in recurrent networks with basic physiological and anatomical constraints. The simulations shown in the following use 1000 neurons divided into a separate pool of excitatory and inhibitory neurons with a biologically plausible ratio of 4:1. The excitatory neurons are basically regular spiking (RS) neurons and inhibitory neurons are basically fast spiking (FS) neurons, but the parameters are also chosen randomly in some range around the default parameters discussed in Section 3.1.5. It is already instructive to see how such a system behaves with random connection weights similar to the networks discussed above. However, we also briefly review some simulations that include spike timing dependent plasticity (STDP) further below.

5.3.1 Random network

The first simulation considers these neurons in a fully connected random network. The excitatory weights are chosen uniformly in the range of 0–0.5 and the inhibition uniformly in the range 0–1. Each neuron also receives input, chosen at each time step of 1 ms randomly from a Gaussian distribution (excitatory from $N(5, 1)$ and inhibitory from $N(2, 1)$). An example of the spike patterns from such a simulation is shown in Fig. 5.11A. The first 800 neurons are the excitatory neurons, and neurons numbered 801–1000 are inhibitory. After a first big activation wave at the start of the simulation, the activity settles down into a more distributed pattern. However, there is some oscillatory activity visible

in the spike trains.

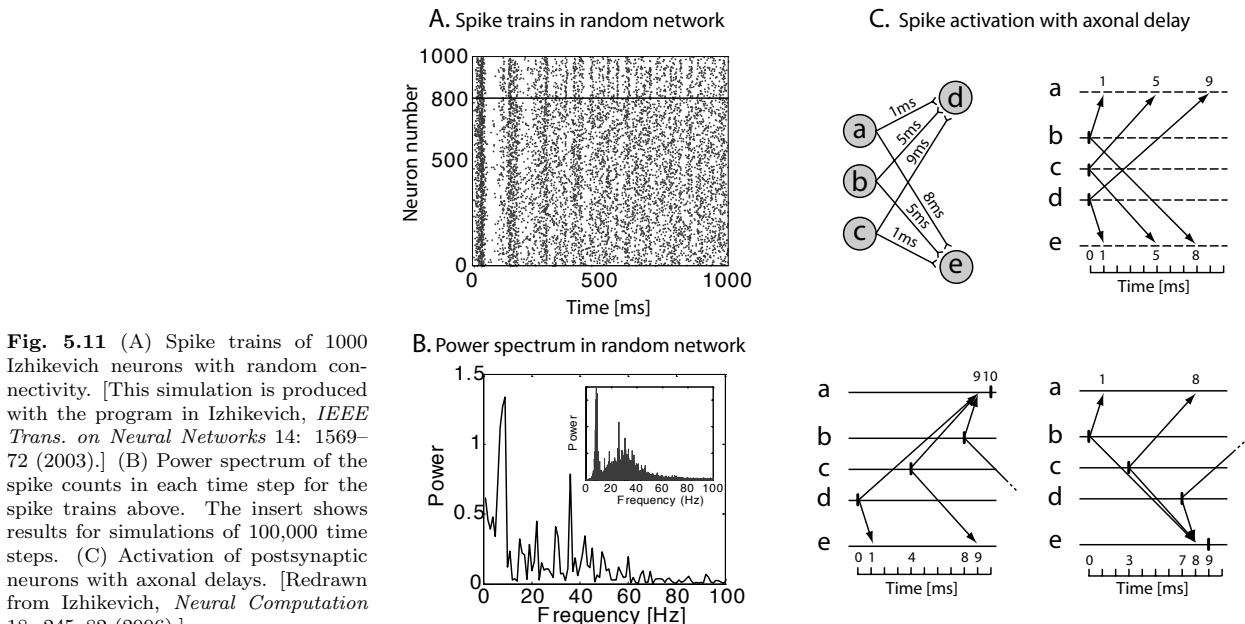


Fig. 5.11 (A) Spike trains of 1000 Izhikevich neurons with random connectivity. [This simulation is produced with the program in Izhikevich, *IEEE Trans. on Neural Networks* 14: 1569–72 (2003).] (B) Power spectrum of the spike counts in each time step for the spike trains above. The insert shows results for simulations of 100,000 time steps. (C) Activation of postsynaptic neurons with axonal delays. [Redrawn from Izhikevich, *Neural Computation* 18: 245–82 (2006).]

To analyse this more, we used the number of spikes in the network for each time step as signal for a power spectrum analysis. This is based on a *fast Fourier transformation (FFT)* which decomposes the signal into a series of sinusoidal components. The square of the amplitudes of the coefficients is called the power and is plotted in Fig. 5.11B against the frequencies of the components. Even with this short simulation it is evident that there are peaks around 10 Hz (*alpha rhythm*) and 40 Hz (*gamma rhythm*). A longer simulation with 100,000 time steps ≈ 16 min reveals that the two distinguishable rhythms are persistent, and that the second peak has some variability in the range of 20–40 Hz.

Simulation

The spike trains were generated with the program we call `IzhikevichRanNet.m` from the Izhikevitch's web site at www.izhikevich.com printed in Table 5.3. The random numbers `re` and `ri`, chosen in Line 4, are used to produce some variability in neuronal parameters set in Lines 5–8. The random weight matrix is called `S` in this program and is produced in Line 9. Initial values are set in Lines 11 and 12. Firing times and the respective number of the node that fired is kept in the matrix `firings`. The program simulates 1000 ms of network activity in time steps of 1 ms. At each time step a new random input is generated in Line 16. Firings of the neurons are defined when $v \geq 30$ mV in this time step (Line 17). The MATLAB function `find()` returns the indices of the non-zero entries in the argument array.

Table 5.3 Program IzhikevichRanNet.m

```

1  % Created by Eugene M. Izhikevich, February 25, 2003
2  % Excitatory neurons   Inhibitory neurons
3  Ne=800;           Ni=200;
4  re=rand(Ne,1);    ri=rand(Ni,1);
5  a=[0.02*ones(Ne,1); 0.02+0.08*ri];
6  b=[0.2*ones(Ne,1); 0.25-0.05*ri];
7  c=[-65+15*re.^2; -65*ones(Ni,1)];
8  d=[8-6*re.^2; 2*ones(Ni,1)];
9  S=[0.5*rand(Ne+Ni,Ne),-rand(Ne+Ni,Ni)];
10
11 v=-65*ones(Ne+Ni,1); % Initial values of v
12 u=b.*v;             % Initial values of u
13 firings=[];          % spike timings
14
15 for t=1:1000         % simulation of 1000 ms
16     I=[5*randn(Ne,1);2*randn(Ni,1)]; % thalamic input
17     fired=find(v>=30); % indices of spikes
18     if ~isempty(fired)
19         firings=[firings; t+0*fired, fired];
20         v(fired)=c(fired);
21         u(fired)=u(fired)+d(fired);
22         I=I+sum(S(:,fired),2);
23     end;
24     v=v+0.5*(0.04*v.^2+5*v+140-u+I);
25     v=v+0.5*(0.04*v.^2+5*v+140-u+I);
26     u=u+a.*(b.*v-u);
27 end;
28 plot(firings(:,1),firings(:,2),'.');

```

If some neurons fired in this time step (Line 18), then the `firings` array is appended with the time of this time step and the indices of the firing neurons in Line 19. The expression `t+0*fired` makes sure that the same time is recorded for all firing nodes in case of more than one neuron firing at this time step. Also, the potentials and recovery variable of the firing nodes are reset (Lines 20 and 21), and the input from the spiking neurons is added to the external input.

Lines 24–26 update the differential equations with a basic Euler scheme. The interesting trick used by Izhikevich is thereby to update the first equations twice with half the time step (0.5 ms) while updating the second equation with the 1 ms time step. This optimizes the running time of the program while keeping the integration time step for the more rapidly changing potential sufficiently small. Line 28 makes the raster plot shown in Fig. 5.11.

The power spectrum can be calculated with the MATLAB function `fft()` which implements a discrete Fourier transformation. The program used to

Table 5.4 Program PowerSpectrum.m

```

1 %% Power spectrum of Izhikevich data
2 for i=101:1000; y(i)=sum(firings(:,1)==i); end
3 Y = fft(y)/900;
4 N = length(Y); Y(1)=[];
5 power = abs(Y(1:N/2)).^2;
6 freq = (1:N/2)/N*1000;
7 plot(freq,power)
8 xlabel('Frequency (Hz)'); ylabel('Power')

```

produce Fig. 5.11B is shown in Table 5.4. We first count how many neurons fired at each time step in Line 2 (ignoring the first 100 time steps). The power is the absolute square of each (complex) Fourier component, which are plotted against the frequencies with this program. Note that the first component is just the sum of the data and is therefore removed in Line 4 since it is a large number.

5.3.2 Networks with STDP and polychrony

While it is already interesting to see the emergence of oscillatory activity resembling some brain dynamics as measured, for example, with field potentials, the networks above were randomly and fully connected, and the synaptic connections did not include any form of plasticity. However, the efficiency of the neuron model also allows larger simulations with additional biological constraints and mechanisms, and we discuss briefly a further study by Izhikevich quoted in the caption of Fig. 5.11C. This reference includes a MATLAB program for the simulations discussed here.

The ratio of excitatory to inhibitory neurons was kept to the previous ratio of 4:1, but neuronal parameters are now fixed to the standard RS parameters for excitatory neurons and FS parameters for inhibitory neurons, to simplify the model somewhat with details that are not crucial for the following discussion. However, Izhikevich introduced three major additional factors the advanced model:

- (1) The 1000 neurons in the network are now sparsely and more biologically connected. Each excitatory neuron projects to 100 random neurons in the network, and the inhibitory neurons project to 100 excitatory neurons only.
- (2) Synaptic strengths are modified with an STDP rule.
- (3) Each connection includes a random axonal delay of 1–20 ms.

The first addition is not only biologically more realistic, but it is also important in the sense that it restricts possible influences of the neurons to a subset of the neurons in the network. We have already discussed in Section 5.2.2 that

such a divergence rate is sensible. The STDP rule used in this model uses temporal windows of LTP and LTD similar to the curves shown in Fig. 4.7A. In addition, it restricts the weight values to a range where roughly two presynaptic events with maximal weights can elicit a postsynaptic spike. Also, the synaptic changes are only applied slowly over several seconds, and a small activity-independent increase is included to recover from silenced synapses.

The importance of axonal delays deserves some more discussion, since they are so far often ignored in modelling papers but a central feature of the proposal by Izhikevich. We use the term *axonal delay* here for wide distributions of latencies between presynaptic events and their postsynaptic consequences that have been measured in mammalian brains, sometimes ranging from fractions of milliseconds to tens of milliseconds. The effect and potential power of these delays is illustrated in Fig. 5.11C. There, three presynaptic neurons are considered, feeding with the indicated axonal delays into two postsynaptic neurons. We assume that presynaptic activities of these three neurons have to arrive in close temporal proximity (within 1 ms) at a postsynaptic neuron to elicit a postsynaptic spike. In the graph on the upper right, all presynaptic nodes fire at the same time so no spike is elicited in the either of the two postsynaptic neurons. However, if the presynaptic neurons fire in specific temporal orders, as shown in the bottom graphs, then single postsynaptic spikes are elicited in specific neurons. Thus, postsynaptic spikes become selective to specific temporal orders of presynaptic firing patterns. If axonal delays would be zero (or the same value for all presynaptic spikes) then both postsynaptic neurons would fire only if all presynaptic neurons fire synchronously. In contrast, Izhikevich coined the term *polyphony* for the situation of specific presynaptic firing patterns which are able to elicit specific postsynaptic spikes. The number of possible spike patterns coded in such networks is formally infinite if we consider changes in axonal delays, and we mentioned in Chapter 4 that changes of EPSP latencies have been observed in plasticity experiments.

When simulating such networks over long times (24 h of simulated time) with random input to single neurons every 1 ms, Izhikevich found that a large number of *polychronous groups* emerged, which are characterized by their potential for being activated by specific firing patterns in the network. Thus, these groups become active for specific polychronous spikes of input nodes of the polychronous group, and the activation of the input pattern would in turn elicit polychronous responses in specific other neurons. The analysis of these groups is not easy and was achieved by Izhikevich basically through anatomical reconstructions, but he found that the number of such groups largely exceeded the number of neurons in the network. Individual neurons can participate in several polychronous groups, and such groups also changed on an ongoing basis through continuous STDP.

The random input to these simulations does not allow learning of specific behaviourally relevant cell assemblies, but they clearly demonstrate the spontaneous generation of polychronous groups which can be the basis for the formation of behaviourally relevant cell assemblies, as envisioned by Hebb. The demonstrations by Izhikevich are also relevant for the interpretation of rate models, discussed later. If we identify population nodes with polysynchronous groups, then it suggests transiently active nodes and also that the specific in-

put driving a node can itself be a complex pattern. While we discuss later more traditional point attractor networks, specifically in Chapter 8, those networks could be interpreted as simplifications of the networks discussed here with associations between polychronous groups.

Exercises

- (5.1) In a network of 10^{12} neurons, where each neuron is connected to 1000 other neurons, what is the average number of synaptic steps to reach one neuron from another neuron?
- (5.2) If a neuron has a diameter of 40 micron (μm), and all the neurons in the brain are arranged tightly on the surface of a sphere (ball), what is the diameter of this sphere? Describe your calculation.
- (5.3) Write a program that simulates two simple neurons that are reciprocally connected. Find a condition under which such a network displays oscillatory behaviour.
- (5.4) For the simulation of Izhikevich neurons in a random network, plot the instantaneous population rate. Compare this to the average rate of the neurons and a smoothed version of the population rate.
- (5.5) Vary the thalamic input to the random network of Izhikevich neurons and plot the resulting rate of neurons (activation function). Discuss your results.

Further reading

The book by Edward White (1989) is still a valuable source for studying anatomical organization, including details regarding cortical cell types and connectivity. The book by Moshe Abeles (1991) is a classic on synfire chains. Several parts of this chapter follow his classic presentation. Also, this book includes an introduction to neocortical anatomy and neurophysiology relevant for modellers, and discusses how some esti-

mations of important parameters are extracted from experiments.

Edward L. White (1989), *Cortical circuits*, Birkhäuser.

Moshe Abeles (1991), *Corticonics: Neural circuits of the cerebral cortex*, Cambridge University Press.

Feed-forward mapping networks

6

In this chapter we explore the ability of feedforward neural networks to represent mapping functions. Mapping functions are important in many brain processes and have dominated models in cognitive science in the form of multilayer perceptrons. We start exploring the effects of choosing appropriate values for synaptic weights through learning algorithms. While feedforward networks are not enough to explain cognitive functions alone, they are an important ingredient of brain-style information processing and have contributed greatly to the development of *statistical learning theory*. This chapter includes some review about concepts of *machine learning* and more recent developments such as *support vector machines*. These less biological discussions are included for their contribution to the general understanding of learning theories, and also since such tools have become important methods for analysing data in neuroscience.

6.1 The simple perceptron

6.1.1 Optical character recognition (OCR)

To illustrate the abilities of networks introduced in this chapter, we follow an example of optical character recognition (OCR). OCR is the process of (optically) scanning an image and interpreting this digital image so that the computer ‘understands’ its meaning. For example, if we scan an image of the letter A, than we would like to store a binary number 1000001 in the computer. This binary number is a standard representation for encoding the letter A in the American Standard Code for Information Interchange (ASCII). Once the letter is represented with this code, a program can use this code to infer meaning and use this to perform specific tasks, such as displaying a representation of the letter on screen, printing the letter with different fonts on paper, or to look up words made out of such letters in a dictionary. The transformation of the digital image representation to the representation of the meaning of the image on a computer is a very challenging engineering task.

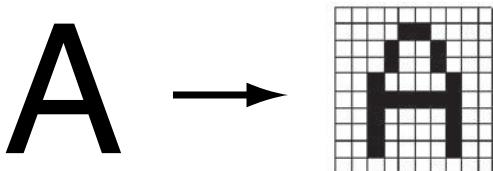
While there are now some OCR programs on the market, humans typically outperform these systems easily. Also, character recognition is a relatively easy task for humans, and it is thus interesting to compare OCR to functions in the human perceptual system. While we will stress later that the human system relies heavily on top-down expectation (Section 10.3), the feedforward systems in this chapter are an important component of information processing in the brain. To discuss the example in more detail, it is useful to distinguish two

6.1 The simple perceptron	143
6.2 The multilayer perceptron	155
6.3 Advanced MLP concepts	165
6.4 Support vector machines	173
Exercises	178
Further reading	179

processes in the *perception* of a letter, the *physical sensing* of an image of a letter, and *attaching meaning* to such an image. This is similar to the OCR example above, which includes a scanning phase, which converts the printed image into a binary representation, and an recognition phase, which relies on the transformation of a binary image into a meaningful representation. When we read a letter, each letter has to be transformed from the representation on paper to a representation in the brain which can be interpreted by the brain.

Let's discuss the example of recognizing the letter 'A' as shown on the left side of Fig. 6.1. The first step towards perception of this letter is to get the signal into the brain. This is achieved by an optical sensor called eye with photoreceptors able to transduce light falling on to the retina into signals that are transmitted to the brain. We approximate this process with a simplified digitizing model retina of only $10 \cdot 10 = 100$ photoreceptors. Each of these photoreceptors transmits a signal with value 1 if the area covered by the receptor (its *receptive field*) is covered at least partially by the image. In this way we end up with the digitized version of the image shown on the right side of Fig. 6.1.

Fig. 6.1 (Left) A printed version of the capital letter A and (right) a binary version of the same letter using a 10×10 grid.



This model is certainly a crude approximation of a human eye. The resolution in the human retina is much higher; a typical retina has around 125 million photoreceptors. Also, the receptors in a human retina are not homogeneously distributed, as in our example, but have highest density in the centre of the visual field called the *fovea*. In addition, it is well known that much more sophisticated signal processing goes on in a human eye through several neuronal layers with specialized neurons before a signal leaves the eye through the axons of ganglion cells. However, this crude model is simply intended to illustrate a general scheme, and there is no need to complicate the model with more realistic but irrelevant details for the discussions in this section.

The output of the simple model retina is a collection of values that can be collected in a vector rather than the array shown in Fig. 6.1. To do this, we give each model neuron an individual number and write the value of this node into a large column at a position corresponding to this number of the node (see Fig. 6.2). We call this vector the *sensory feature vector*, \mathbf{x} . Individual components are written with an index such as x_i , where the index i numbers the different feature values. Feature values can be binary, as in our example, real-valued, or any other symbolic representation. The number of feature values defines the *dimensionality* of the feature space, which is often very large (100 in our example; over a million for the visual feature vector generated by one eye if we use as basis the number ganglion cells in humans). It is important to keep in mind that the precise form of sensory feature vectors depends on the specific functionality of the feature sensors (the retina in our example) and the

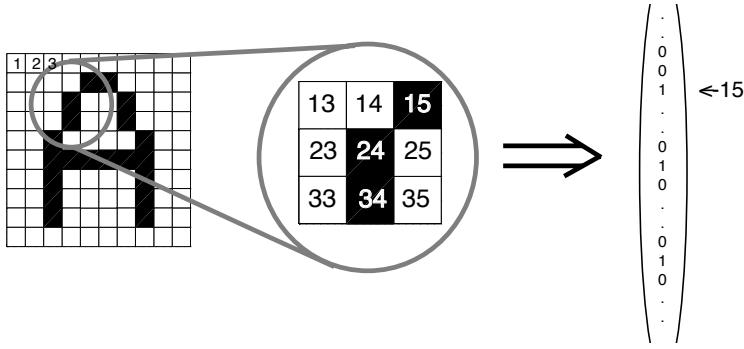


Fig. 6.2 Generation of a sensory feature vector. Each field of the model retina, which corresponds to the receptive field of a model neuron, is sequentially numbered. The firing value of each retinal node, either 0 or 1 depending on the image, represents the value of the component in the feature value corresponding to the number of the retinal node.

related encoding procedures. If the form of the feature vector is crucial, we have either to model the biological details of the sensory system or to make valid assumptions that reflect the biological system. The precise form of the sensory feature vector is not important to demonstrate the principal ideas presented in this chapter.

6.1.2 Mapping functions

A sensory feature vector is the necessary input to any object recognition system. Given such a sensory feature vector we can formulate the character recognition process as the problem of mapping this sensory feature vector on to a set of symbols representing the *internal representation* of the object. This internal representation corresponds, for example, to the ASCII code of the letter in the computer example, or to a largely unknown representation in the language area in our brain. In our example above we used the bit vector of length 7 (ASCII) for the internal representation of A, and we could use the corresponding ASCII code for other letters. Or we could use the number from 1 to 26, which corresponds to the position of the letter in the alphabet.

At this point we have reduced the recognition process to a vector function, or *mapping* for short. The OCR example is a mapping from a binary feature vector to binary vector for the computer representation of the meaning. Generally, we can define a mapping as a vector function f from a vector \mathbf{x} to another vector \mathbf{y} as

$$f : \mathbf{x} \in \mathbb{S}_1^n \rightarrow \mathbf{y} \in \mathbb{S}_2^m, \quad (6.1)$$

where n is the dimensionality of the sensory feature space ($n = 100$ in our example), and m is the dimensionality of the internal object representation space ($m = 1$ in our example). \mathbb{S}_1 and \mathbb{S}_2 are the sets of possible values for each individual component of the vectors. For example, the components of the vector can consist of binary values ($\mathbb{S} = [0, 1]$), discrete values ($\mathbb{S} = \mathbb{N}$), or real values ($\mathbb{S} = \mathbb{R}$) such as the firing rates of neurons. In the example above we mentioned a subset of the natural numbers, $\mathbb{S}_2 = [1, \dots, 26]$, to represent the meaning of the 26 letters in the alphabet.

How can we realize a mapping function? There are several possibilities, which we will discuss in turn. Our aim is to understand how the brain achieves this mapping, but it is instructive to think about some of the possibilities. The simplest solution is to construct a *look-up table*. A look-up table is a large table that lists for all possible sensory input vectors the corresponding internal representations. To illustrate this we further reduce the size of the feature vector and consider a feature vector that has only two components. Hence, the dimensionality of the sensory feature space is only 2 (not 100 as before). Let's call the first feature value x_1 and the second feature value x_2 . Each sensory feature value can only have two possible values, 0 and 1 (binary feature vector). Thus, there are four possible combinations of feature values, $\{(0\ 0), (0\ 1), (1\ 0), (1\ 1)\}$. A look-up table for a function $y(\mathbf{x})$ lists the desired output value y for each combination of the input (feature) values. An example is given in Table 6.1 representing one possible binary mapping function of two inputs, the Boolean AND function.

Table 6.1 Boolean AND function

x_1	x_2	y
0	0	1
0	1	0
1	0	0
1	1	1

With binary feature values we can have $n_c = 2^2 = 4$ different combinations of feature values. As each combination of feature values can be given an independent number of 0 and 1, we have $2^4 = 16$ possible binary mapping functions with two binary inputs. In general, the possible combinations of feature values is $n_c = b^n$, where b is the number of possible feature values ($b = 2$ for binary feature values) and n is the dimensionality of the feature space. The number of possible mapping functions is $n_f = b^{n_c}$, which can be very large. Thus, the size of the table grows very fast with the number of feature components and the number of possible feature values. For example, we need a look-up table of size $2^{100} \approx 10^{30}$ for the recognition of letters binarized by our simple model retina, and the size increases with a power law when increasing the resolution of the digitization process. Moreover, the size is infinitely large if the feature values are real-valued. There are some fixes to the problem of the large sizes needed for look-up tables, such as not list all possible combinations of the feature values in the look-up table. This would make sense in the OCR because there not all combinations correspond to a letter. We could simply ignore those combinations and have our system generate a ‘not a letter’ result if the combination is not found in the partial look-up table. However, the number of possible representations of the letters is still very large. Furthermore, we have to ask how to build such a look-up table. In our example we would have to generate all possible representations of letters, which seems unrealistic at least with respect to the development of our perceptive system.

Another possibility for realizing a mapping function, which has often been proposed as a model for human perception, is the utilization of *prototypes*. A prototype is a vector that encapsulates, on average, the features for each individual object. In our letter representation example we would need to store only 26 prototypes in a look-up table, one for each different letter. We have then to add another process that maps each sensory feature vector on to a specific prototype vector. One way to achieve this is to calculate the ‘distance’ of the sensory feature vector from each prototype and choose the prototype that is ‘closest’ to the sensory feature vector.

A remaining question in the prototype scheme is how to generate the prototype vectors. A possible scenario is to present a set of sample letters to the

system and to use the average (or another statistical measure) as a prototype for each individual letter. The interesting idea behind this scenario is that the generation of the prototypes is driven by examples. The disadvantage of the specific prototype scheme discussed here is that we have to compare each example to be recognized to each individual prototype vector. Thus, the computational requirement and time for recognition grows linearly with the number of prototypes.

6.1.3 The population node as perceptron

We demonstrate in this section that a population node, as introduced in Section 3.4, can represent certain types of vector functions. For this we set the firing rates of the input channels to

$$r_i^{\text{in}} = x_i. \quad (6.2)$$

The node in this simple perception system has to have at least n input channels, where n is the dimensionality of the feature space. For example, a simple perceptron in a two-dimensional feature space would look like the one illustrated in Fig. 6.3. The firing rate of the output represents a function

$$\tilde{y} = r^{\text{out}}. \quad (6.3)$$

To define the system completely we have to choose an activation function g for the node. We will start with the linear activation function $g(x) = x$ and discuss others below. The output of a *linear perceptron* with two inputs is

$$\tilde{y} = w_1 x_1 + w_2 x_2, \quad (6.4)$$

where w_1 and w_2 are the weight values assigned to each input channel.

More generally, we can state the *update rule* of a *single-layer mapping network* with several output nodes, also called *simple perceptron*,

$$\mathbf{r}^{\text{out}} = g(\mathbf{w}\mathbf{r}^{\text{in}}), \quad (6.5)$$

where g is an activation function. We used thereby a matrix notation (see eqn 4.20). This equation is equivalent to writing the update rule for each component,

$$r_i^{\text{out}} = g(w_{i1}x_1 + w_{i2}x_2 + \dots) = g\left(\sum_j w_{ij}r_j^{\text{in}}\right). \quad (6.6)$$

Which functions $\tilde{y} = \mathbf{r}^{\text{out}}$ can be represented by simple perceptrons? To study this we return to the simple case of a linear population node and see if it can represent a particular function, for example, the function listed partially in the look-up table Table 6.2.

The first feature vector in this table is $x^1 = (1, 2)'$. To calculate the value of the output, given the particular values of the input, we have to specify the weight values w_1 and w_2 . We have the freedom to give these parameters any value we want. Let's choose the values $w_1 = 1$ and $w_2 = -1$. The output of the node is then

$$\tilde{y}^1 = \tilde{y}(x^1) = \tilde{y}(x_1 = 1, x_2 = 2) = 1 \cdot 1 - 1 \cdot 2 = -1 = y^1. \quad (6.7)$$

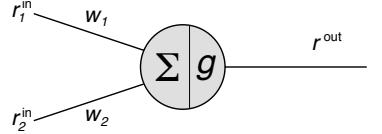


Fig. 6.3 Simple population node with two input channels as a model perceptron for a two-dimensional feature space.

Table 6.2 Look-up table for an example function

x_1	x_2	y
1	2	-1
2	1	1
3	-2	5
-1	-1	7
...

It is not surprising that we get, $\tilde{y}^1 = y^1$, an exact match between the output of the node and the function value we want to represent. The reason for this is that we choose the weight values accordingly. One could achieve the same result with other values, for example, $w_1 = -1$ and $w_2 = 0$. Indeed, there are an infinite number of solutions to represent the function value y_1 because we have only one constraining equation with two free parameters. The reason that we chose $w_1 = 1$ and $w_2 = -1$ is that we are then also able to represent the second value in the look-up table,

$$\tilde{y}^2 = 1 \cdot 2 - 1 \cdot 1 = 1 = y^2. \quad (6.8)$$

At this stage, we have used up all free parameters of the weight vector \mathbf{w} in order to represent the first two entries in the look-up table, and all the other values of the function \tilde{y} are uniquely defined. The third entry of the look-up table is also correctly represented by the perceptron, namely

$$\tilde{y}^3 = 1 \cdot 3 - 1 \cdot (-2) = 5 = y^3. \quad (6.9)$$

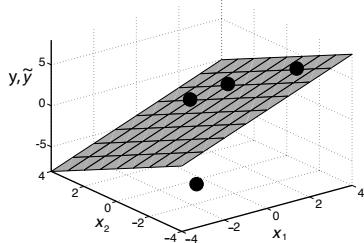


Fig. 6.4 Output manifold of population node with two input channels that is able to partially represent the mapping function listed in the look-up table (Table 6.2).

We can say that the network generalized correctly. The reason for this match is that the third point lies on the two-dimensional *sheet* defined by eqn 6.4 as illustrated in Fig. 6.4. However, the fourth point in the look-up table, y^4 , does not lie on this output sheet. Thus, the specific linear perceptron can not represent all the points of the look-up table in Table 6.2.

What about other activation functions? If we choose the activation function to be $g(x) = \sin(x)$ instead of a linear function we are able to represent all four points listed in Table 6.2 (as can be verified easily). However, what about possible additional points not listed in the look-up table? For more complex functions we have to introduce increasingly complex activation functions in order to be able to represent them with a population node. However, an entire solution to the representation of mapping functions by a single node with a complex input–output relationship seems physiologically unrealistic. A more realistic solution is to use networks.

6.1.4 Boolean functions: the threshold node

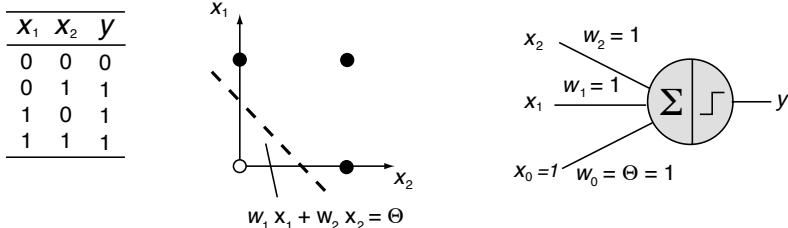
Before leaving the single population node perceptron we briefly explore an important subset of the possible vector functions, the class of *binary functions* or *Boolean functions*. These are functions that have only two possible feature values for each feature component, which we choose to be $\{0, 1\}$ in the following. This important set of functions represents only a small subset of all possible functions, but we know that we can build sophisticated (and indeed universal) learning machines from these basic functions. For Boolean functions it is natural to use a *threshold function*,

$$g(x) = \begin{cases} 1 & \text{if } x > \Theta \\ 0 & \text{elsewhere} \end{cases}, \quad (6.10)$$

as the activation function of a node since it limits the output values to the required binary values. The threshold node is equivalent to the McCulloch–Pitts neuron discussed in Section 3.1.6. A simple network of such threshold

units was originally termed ‘perceptron’ by *Frank Rosenblatt* and colleagues. We adapt here the term for more general feedforward networks with arbitrary activation functions to be consistent with later uses of this term when discussing multilayer versions.

A. Boolean OR function



B. Boolean XOR function

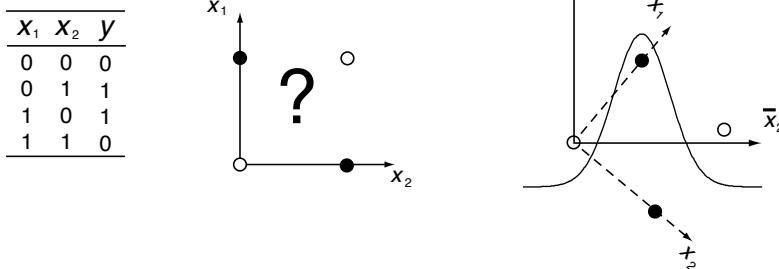


Fig. 6.5 (A) Look-up table, graphical representation, and single threshold population node with bias channel for the Boolean OR function. (B) Look-up table and graphical representation of the Boolean XOR function, which cannot be represented by a single threshold population node because this function is not linear separable. A node that can rotate the input space and has a non-monotonic activation function can, however, represent this Boolean function.

A simple threshold perceptron is able to represent a lot of Boolean functions. An example, the *Boolean OR function*, is illustrated in Fig. 6.5A. As shown in the look-up table on the left, the OR function is equal to $y = 1$ if either one of the inputs is one, and $y = 0$ otherwise. The middle graph represents this function in a graphical way by representing the binary values of y with either a white or a black dot at the locations for the possible combinations of the input values x_i . This function can be represented by the single threshold population node with the parameters shown on the right in Fig. 6.5A. We typically included the threshold parameter as weight value w_0 for a channel with constant input. Such a *bias input* allows the threshold parameter to be included in the learning algorithms outlined below.

All but two out of the 16 possible Boolean functions with two input features can be represented by the threshold population node with the appropriate choice of weight values. The activation of the population nodes with two weight values and a possible constant bias defines a linear function, and the threshold activation function sets the output to the appropriate value as long as the linear activation curve can separate the two classes of y values appropriately. We say that such classes of functions are *linear separable*. The only

Table 6.3 Number of Boolean functions in an n -dimensional feature space and the number of linear non-separable functions

n	Number of linear separable functions	Number of linear non-separable functions
2	14	2
3	104	152
4	1,882	63654
5	94,572	$\sim 4.310^9$
6	15,028,134	$\sim 1.810^{19}$

two functions that cannot be represented are the XOR (exclusive-OR) function and its corresponding reverse, non-XOR. The XOR function is defined by the look-up table in Fig. 6.5B. The graphical representation reveals the reason for the failure, it is not possible to divide the regions of the feature values with different y -values with a single line that can be implemented by the activation function of the population node. We say that this function is *not linear separable*.

It does not seem too bad that only two out of 16 functions cannot be represented. However, the number of non-linear separable functions grows rapidly with the dimension of the feature space and soon outgrows the number of linear separable functions (see Table 6.3). The problem that only linear separable Boolean functions can be represented by a single threshold perceptron, often stated as the *XOR-problem*, greatly diminished the interest in simple perceptrons for several years until multilayer perceptrons (Section 6.2) became tractable. However, it is worth realizing that we can represent the XOR function with a single population node if we employ a non-linear activation function. For example, as illustrated on the right in Fig. 6.5B, we can change the representation of the problem by rotating the axis and using a simple perceptron with a Gaussian activation function. Such non-monotonic activation functions seem physiologically not plausible when we have a single neuron in mind, since stronger input to a neuron typically elicits stronger responses. However, as we stressed in Chapter 5, nodes are primarily intended to represent a collection of neurons, and we will discuss in the next chapter that single neurons have tuning functions resembling Gaussians.

6.1.5 Learning: the delta rule

Perceptrons have enormous potential for approximating functions by properly choosing values for the weights of the input channels. How can we choose appropriate values? This seems an enormous task given the vast number of adjustable parameters in large networks and the exploding number of their combinations. The solution to this problem, one of the great achievements and attractions of neural network computing, lies in algorithms that can find appropriate values from examples. The process of changing the weight values to represent the examples is often called *learning*, *training*, or *adaptation*. We

will usually use the term *learning algorithms* for algorithms that adjust the weight values in abstract neural networks. Learning is an essential ingredient in forming memories and other cognitive functions, as discussed later in this book. If we are primarily interested in building learning machines, then we have no biological constraints and could discuss statistical learning in general terms. We will mention some general machine learning algorithms below. However, if we are interesting in understanding biological learning and memory, then we need to relate such algorithms to physical changes in the brain, such as synaptic plasticity, discussed in Chapter 4.

In this section we show how simple mapping networks can be trained. Our objective in the following is to minimize the mean difference between the output of a feedforward mapping network and a desired state provided by a teacher. We can quantify our objective with an *objective function* or *cost function*, labelled E in the following, which measures the distance between the actual output and the desired output. There are many possible definitions of a distance between two states. Formally, each definition defines the metric of the state space as long as the distance measure has certain obvious characteristics, such as that the distance is zero only if the output of the network is equal to the desired state, and the distance is positive otherwise. Some examples of distance measures are listed in Appendix A. A simple and often used cost function is the *mean square error* (MSE)

$$E = \frac{1}{2} \sum_i (r_i^{\text{out}} - y_i)^2, \quad (6.11)$$

where r_i^{out} is the actual output and y_i is the desired output of a mapping network. The factor $1/2$ is a useful convention when using the MSE as an objective function, as we will see below. If the output of each output node is equal to the desired value y_i for each node, then this distance is zero. It is larger than zero when one or more of the components do not agree.

We can minimize the error function between the desired output and the actual output of a single-layer mapping network by changing the weight values to the output layer along the negative gradient of the error function,

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \quad \text{with} \quad \Delta w_{ij} = -\epsilon \frac{\partial E}{\partial w_{ij}}. \quad (6.12)$$

This is called a *gradient descent* method. The error function is monotonically decreased by this procedure in steps proportional to the gradient along the error function. The constant ϵ is a *learning rate*. An example is shown in Fig. 6.6 for a hypothetical network with only one weight value w . By starting with some random weight value for this ‘network’ it is likely that this network shows poor performance with a large error. The weight value is then changed in the direction of the negative gradient, which is a vector pointing in the direction of the maximal slope of the objective function from this point, and has a length proportional to the slope in this direction. Changing the weight values of the network to new values in this direction therefore guarantees that the network with the new weights has a smaller error than before. When approaching a minimum the gradient of the error function is decreasing and the change in the

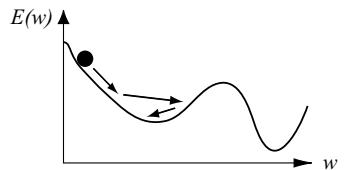


Fig. 6.6 Illustration of error minimization with a gradient descent method on a one-dimensional error surface $E(w)$.

weight values slows down. This method often results in a rapid initial decrease of the network error.

We can easily derive the particular algorithm to implement the gradient descent method for a simple perceptron when using the MSE as a measure of the distance between desired output y_i and actual output r_i^{out} of the perceptron. The gradient of the error function is given by

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} &= \frac{1}{2} \frac{\partial}{\partial w_{ij}} \sum_i (g(\sum_j w_{ij} r_j^{\text{in}}) - y_i)^2 \\ &= f'(h_i)(\sum_j w_{ij} r_j^{\text{in}}) - y_i) r_j^{\text{in}}.\end{aligned}\quad (6.13)$$

$f'(x) = \frac{\partial f(x)}{\partial x}$ is the derivative of the activation function that enters the formula by using the *chain rule* for calculating differentials

$$\frac{df(g(x))}{dx} = \frac{df(g)}{dg} \frac{dg(x)}{dx}. \quad (6.14)$$

The resulting learning rule,

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \quad \text{with} \quad \Delta w_{ij} = \epsilon g'(h_i) * (y_i - r_i^{\text{out}}) r_j^{\text{in}}, \quad (6.15)$$

is called the *delta rule* as the change of the weight values is proportional to the difference between the desired and actual output often expressed as δ -term, $\delta = y_i - r_i^{\text{out}}$. For a linear perceptron with activation function $f(x) = x$, this is simply given by $f' = 1$. The weight change for a linear perceptron is therefore

$$\Delta w_{ij} = \epsilon (y_i - r_i^{\text{out}}) r_j^{\text{in}}. \quad (6.16)$$

This rule, without the derivative of the activation function, was used previously for threshold perceptrons and is called the *perceptron learning rule*. Since the threshold function is not differentiable at the threshold, this function is formally not a gradient descent rule. However, it turns out that this rule still works for training the threshold perceptron on simple classification problems.

Also, note the similarity of the delta learning rule with Hebbian plasticity as discussed in Chapter 4 (eqn 4.10). This learning rule has now two Hebbian terms. The weights are increased with *supervision* by an amount proportional to the product of the presynaptic node (input value) r_j^{in} and the desired postsynaptic value y_i . This is a supervised Hebbian learning term. However, the weights are also decreased by the product of the input value r_j^{in} and the actual postsynaptic value r_i^{out} . This term is like unlearning the actual response of the perceptron when the perceptron does not give the right answer. Learning ceases when the actual output is equal to the desired output, since the δ -term is then zero. The algorithm of the delta rule is summarized with general activation functions in Table 6.4.

Simulation

Digitized versions of the alphabet are provided in the text (ASCII) file `pattern1` in folder `Chapter6`. The two states are coded with 0s and 1s. This file can be

Table 6.4 Summary of delta-rule algorithm

Initialize weights arbitrarily
Repeat until error is sufficiently small
Apply a sample pattern to the input nodes: $r_i^0 = r_i^{\text{in}} = \xi_i^{\text{in}}$
Calculate rate of the output nodes: $r_i^{\text{out}} = g(\sum_j w_{ij} r_j^{\text{in}})$
Compute the delta term for the output layer: $\delta_i = g'(h_i^{\text{out}})(\xi_i^{\text{out}} - r_i^{\text{out}})$
Update the weight matrix by adding the term: $\Delta w_{ij} = \epsilon \delta_i r_j^{\text{in}}$

loaded in the MATLAB editor to inspect the letters. Alternatively, individual letters can be displayed with the function `displayLetter` shown in Table 6.5. The `load` command in Line 3 reads the content of file `pattern1` into an array with the same name. The MATLAB function `reshape()` in Line 4 reshapes this 312×13 matrix to a 156×26 matrix with a column vector for each letter in the alphabet. This is useful as we can then easily select one of the letters, as done in Line 5. The letter vectors can be used as input vectors to the perceptron. Of course, for humans to recognize the letter on screen, we need to reshape the letter vector again into an array as done in Line 6, before displaying it in Line 7 with a special output formating. An example output of this function is shown in Fig. 6.7A.

Table 6.5 Function `displayLetter.m`

```

1 function displayLetter(n);
2 % Displays letter number n from file pattern1
3 load pattern1
4 letterVectors=reshape(pattern1', 12*13, 26);
5 thisLetterVector=letterVectors(:,n);
6 thisLetterArray=reshape(thisLetterVector,13, 12)';
7 format +; disp(thisLetterArray); format;
8 return

```

File `perceptronTrain.m`, shown in Table 6.6, contains a training program for a single-layer perceptron. The weight matrix `wOut` is initially set to random values with positive and negative components in Line 4. In Line 8, the `reshape()` function is used as in function `displayLetter` to create a pattern matrix `rIn` used as inputs for the perceptron. We choose a distributed representation as desired output by creating a diagonal matrix in Line 9, so that an `A` is represented by a vector with a one as first component, and zeros elsewhere, etc.

We then loop over several training steps in which all patterns are presented to the network. In Line 14, we calculate the actual output of the network. As we supply all the patterns vectors at once (input pattern matrix), the output is

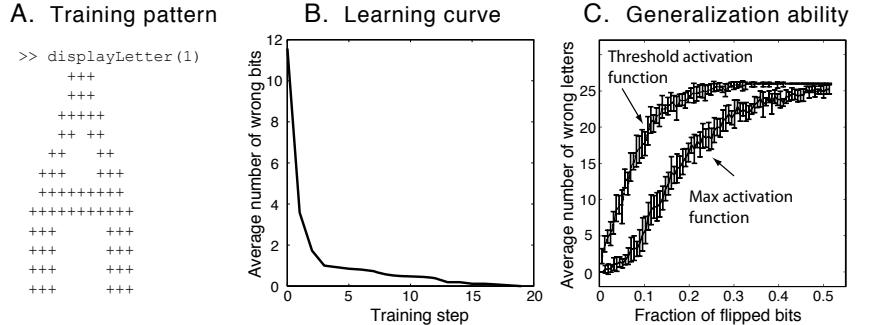


Fig. 6.7 (A) Example of using the function `displayLetter(n)` to plot the content of file `pattern1`. (B) Example of a training function during training a threshold perceptron on the letter patterns with the delta rule. (C) Performance of the trained perceptron with noisy versions of the original pattern. When interpreting the output after training, two different activation functions are shown.

also a matrix with an output vector for each input vector. The calculation of the output includes the threshold activation function. The training is done with the perceptron learning rule. The performance of the recognition is tested before each training step, using as error measure the average number of different bits (Hamming distance) between the actual and the desired output. An example training of a curve produced with the program is shown in Fig. 6.7B. The curve starts with roughly half the bits wrong in the prediction of the untrained network. This recognition error rapidly decreases and the network is fully trained after 18 training steps, when all input vectors create the desired output.

Just learning (memorizing) specific patterns is not our only goal, we also want to test the generalization ability of the trained network. We do this in program `perceptronTest` shown in Table 6.7. The training is first run in Line 2, and the instruction of the next line saves the original letter matrix in matrix `letterMatrix`. We then test the performance when flipping a specific number of bits at random positions in each letter vector. To do this, we create a matrix with function `randomFlipMatrix()` shown in Table 6.8 with zero components except in `nflip` random positions in each column vector. In this function, we use the MATLAB function `randperm(n)` to generate a random permutation of the numbers 1 to n . The first `nflip` numbers of this sequence can be used as positions at which the components in a column vector are set to one. The resulting matrix is then subtracted from the original letter matrix, which results in flipping the bits at the positions with ones in `randomFlipMatrix` when taking the absolute value of this difference.

The test program goes on to test the noisy patterns by calculating how many letters are wrongly recognized in Lines 10, 11 and 16. These results are averaged over 10 trials for each number of flipped bits since the random process of flipping can produce different results. The program also demonstrates how to create a concatenated vector of results in Lines 12 and 17, how to calculate means and standard deviations of the components in the resulting vector using MATLAB functions in Lines 18 and 19, and how to produce a plot with error

Table 6.6 Program perceptronTrain.m

```

1 %% Letter recognition with threshold perceptron
2 clear; clf;
3 nIn=12*13; nOut=26; % number of input and output channels
4 wOut=rand(nOut,nIn)-0.5; %random initial weight matrix
5
6 % training vectors
7 load pattern1;
8 rIn=reshape(pattern1', nIn, 26); % matrix of input pattern
9 rDes=diag(ones(1,26)); % matrix of desired outputs
10
11 % Updating and training network
12 for training_step=1:20;
13     % test all pattern
14     rOut=(wOut*rIn)>0; % threshold activation function
15     dist=sum(sum(abs(rDes-rOut)))/26;
16     error(training_step)=dist;
17     % training with delta rule
18     wOut=wOut+0.1*(rDes-rOut)*rIn';
19 end
20
21 plot(0:19,error)
22 xlabel('Training step')
23 ylabel('Average number of wrong bits')

```

bars in Lines 22 and 23.

The test is done in this program for two different activation functions of this trained network. The first one is the threshold function which was used for training. The results are shown in Fig. 6.7C in the upper curve. The results show some patterns can not be recalled even with small amounts of noise, while others can. Of course, the recognition does deteriorate until all letter recognition is lost at around a noise level of 20%. There are many ways to improve the recognition ability of such networks. For example, we demonstrate here the use of an activation function that activates only the node with the maximal net input. The result of this activation function is also shown in Fig. 6.7C. Now all letters are robust to some amount of noise, and the network can tolerate more noise. A biological implementation of the maximum function can be achieved with lateral inhibition where the most active node tends to suppress other nodes. We follow up this line of thought in the next chapter.

6.2 The multilayer perceptron

We have seen that the limited number of weight values in a single neuron limits the complexity of functions that we can represent using a single population

Table 6.7 Program perceptronTest.m

```

1  %% Testing generalization performance of trained perceptron
2  perceptronTrain;
3  letterMatrix=rIn;
4  for nflip = 1:80;
5      dist1=[]; dist2=[];
6      for trial=1:10;
7          rIn=abs(letterMatrix-randomFlipMatrix(nflip));
8          % Threshold output function
9          rOut1=(wOut*rIn)>0;
10         nerror=0;
11         for j=1:26; nerror=nerror+(sum(rDes(:,j)~=rOut1(:,j))>0); end
12         dist1=[dist1,nerror];
13         % Max output function
14         [v,i]=max(wOut*rIn);
15         rOut2=zeros(26); for j=1:26; rOut2(i(j),j)=1; end
16         dist2=[dist2,0.5*sum(sum(rDes~=rOut2))];
17     end
18     meanDist1(nflip)=mean(dist1); stdDist1(nflip)=std(dist1);
19     meanDist2(nflip)=mean(dist2); stdDist2(nflip)=std(dist2);
20 end
21 figure; hold on;
22 errorbar((1:80)/156,meanDist1,stdDist1,:')
23 errorbar((1:80)/156,meanDist2,stdDist2,'r')
24 xlabel('Fraction of flipped bits')
25 ylabel('Average number of wrong letters')

```

node. An obvious solution therefore is to increase the number of nodes, and thereby the number of connections with corresponding independent weight values. However, by doing so we want to keep the number of input channels, representing the feature values, and the number of output channels, representing the dimension of the internal object representation, constant. The only solution is to use *hidden* nodes. An example of such a *multilayer mapping network*, often called *multilayer perceptron*, is illustrated in Fig. 6.8. The middle layer is commonly called a *hidden layer* as the nodes in this layer have no direct input or output channel to the external world.

The number of weight values, n^w , in networks with a hidden layer grows rapidly with the number of nodes, n^h , in the hidden layer,

$$n^w = n^{\text{in}} n^h + n^h n^{\text{out}}, \quad (6.17)$$

where n^{in} is the number of input nodes and n^{out} is the number of output nodes. We have neglected the weights of the input channels because we merely assigned to the input nodes the role of distributing the input value to all of the other nodes. These nodes are hence not directly computing nodes, and we adopt the common nomenclature and call the architecture illustrated in Fig. 6.8 a

Table 6.8 Function randomFlipMatrix.m

```

1 function r=randomFlipMatrix(n);
2 % returns matrix with components 1 at n random positions of each column
3 r=zeros(156,26);
4 for i=1:26
5     x=randperm(156);
6     r(x(1:n),i)=1;
7 end

```

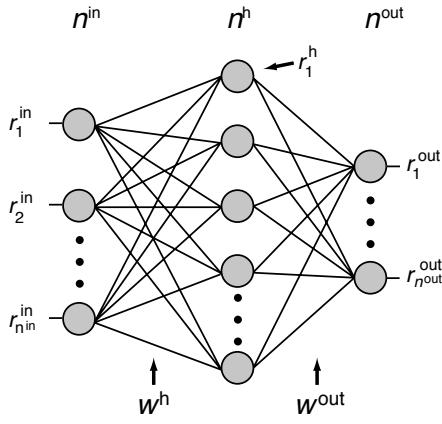


Fig. 6.8 The standard architecture of a feedforward multilayer network with one hidden layer, in which input values are distributed to all hidden nodes with weighting factors summarized in the weight matrix w^h . The output values of the nodes of the hidden layer are passed to the output layer, again scaled by the values of the connection strength as specified by the elements in the weight matrix w^{out} . The parameters shown at the top, n^{in} , n^h , and n^{out} , specify the number of nodes in each layer, respectively.

two-layer network. It is straightforward to include more layers of hidden nodes as discussed later.

6.2.1 The update rule for multilayer perceptrons

With the vector and matrix notation mentioned before, we can state the functionality of a multilayer perceptron in a compact form, directly generalizing the update rule for a single-layer mapping network (eqn 6.5). An input vector is weighted by the weights to the hidden layer, w^h , and all the inputs to one node are summed up. This corresponds to a matrix-vector multiplication,

$$\mathbf{h}^h = \mathbf{w}^h \mathbf{r}^{\text{in}}, \quad (6.18)$$

which is a compact way of writing all the equations for the individual components

$$h_i^h = \sum_j w_{ij}^h r_j^{\text{in}}. \quad (6.19)$$

The vector \mathbf{h}^h is called the *activation vector* of the hidden nodes. We next pass the activations of the hidden nodes through an activation function for

each node, which results in a vector that represents the population rates of the hidden layer,

$$\mathbf{r}^h = g^h(\mathbf{h}^h). \quad (6.20)$$

This rate vector becomes the input vector to the next layer, which can be the output layer or another hidden layer in the case of a multilayer network with more than one hidden layer. The final output vector is calculated from the outputs of the last hidden layer,

$$\mathbf{r}^{\text{out}} = g^{\text{out}}(\mathbf{w}^{\text{out}} \mathbf{r}^h). \quad (6.21)$$

We can summarize all the steps in updating the states of the multilayer feed-forward network in one equation, namely

$$\mathbf{r}^{\text{out}} = g^{\text{out}}(\mathbf{w}^{\text{out}} g^h(\mathbf{w}^h \mathbf{r}^{\text{in}})). \quad (6.22)$$

It is easy to include more hidden layers in this formula. For example, the operation rule for a four-layer network with three hidden layers and one output layer can be written as

$$\mathbf{r}^{\text{out}} = g^{\text{out}}(\mathbf{w}^{\text{out}} g^{h3}(\mathbf{w}^{h3} g^{h2}(\mathbf{w}^{h2} g^{h1}(\mathbf{w}^{h1} \mathbf{r}^{\text{in}})))). \quad (6.23)$$

This formula looks lengthy, but it is a straightforward generalization of the two-layer network and can be easily implemented with a computer program.

Let us discuss a special case of a multilayer mapping network where all the nodes in all hidden layers have linear activation functions ($g(x) = x$). Eqn 6.23 then simplifies to

$$\begin{aligned} \mathbf{r}^{\text{out}} &= g^{\text{out}}(\mathbf{w}^{\text{out}} \mathbf{w}^{h3} \mathbf{w}^{h2} \mathbf{w}^{h1} \mathbf{r}^{\text{in}}) \\ &= g^{\text{out}}(\tilde{\mathbf{w}} \mathbf{r}^{\text{in}}). \end{aligned} \quad (6.24)$$

In the last step we have used the fact that the multiplication of a series of matrices simply yields another matrix, which we labelled $\tilde{\mathbf{w}}$. Eqn 6.24 represents a single-layer network as discussed before. It is therefore essential to include non-linear activation functions, at least in the hidden layers, to make possible the advantages of hidden layers that we are about to discuss. We could also include connections between different hidden layers, not just between consecutive layers as shown in Fig. 6.8, but the basic layered structure is sufficient for the following discussions.

Which functions can be approximated by multilayer perceptrons? The answer is, in principle, any. A multilayer feedforward network is a *universal function approximator*. This means that, given enough hidden nodes, any mapping functions can be approximated with arbitrary precision by these networks. The remaining problems are to know how many hidden nodes we need, and to find the right weight values. Also, the general approximator characteristics does not tell us if it is better to use more hidden layers or just to increase the number of nodes in one hidden layer. These are important concerns for practical engineering applications of those networks. Unfortunately, an answer to these questions cannot be given in general. It depends on the function we want to approximate.

A network with threshold nodes that can represent the XOR function is shown in Fig. 6.9. How well a particular feedforward network can approximate a function depends on all network details, including the activation functions of the nodes. The most commonly used activation function in these networks is the *sigmoid*, or *logistic*, function,

$$g(x) = \frac{1}{1 + e^{-\beta(x-x_0)}} = \frac{1}{2}(\tanh(\beta(x - x_0)) + 1), \quad (6.25)$$

where the parameter β controls the slope, and x_0 is the offset value, where the function has the largest slope and is equal to $g(x_0) = 1/2$. This particular function has contributed greatly to the success of such networks in technical applications. A partial reason for this is that the sigmoid function has the useful characteristic that it changes smoothly in a somewhat confined area, and is nearly constant outside this area. By combining several such sigmoid functions we can quickly generate complex functions. We can adjust the approximation locally by adding a sigmoid function to a given approximation and subtracting a second sigmoid function with a slightly different offset. An example of approximating a sine function by the sum of a small number of sigmoid functions is illustrated in Fig. 6.10. Three dotted lines are drawn corresponding to sigmoidal activation functions of hidden nodes with the same slope and amplitude but different offsets. The sum of these functions, as can be calculated by a linear output node, is shown as a dashed line. The dashed line approximates one period of the sine function reasonably well, although the approximation is quite bad outside of this central region.

6.2.2 Generalization

An important aspect of regression models is their *generalization ability*. *Generalization* refers to the performance of the network on data that were not part of the training set. For example, to approximate the sine function in Fig. 6.10 with three sigmoidal nodes, we need only a few training examples to fix the parameters of the three sigmoidal functions to fit the sine function reasonably well. The training points are usually well approximated by the network, and points in between the training points are also fairly well approximated by the network. The generalization ability of interpolation with sigmoidal networks is often quite good when the true function does not vary widely between training points. In contrast, the generalization ability of extrapolation with such networks into unseen domains is often poor. The reason for this is that sigmoidal networks basically assume a constant dependence for such data points that might not be appropriate, as in the example shown in Fig. 6.10.

To enable good generalization abilities of networks, we need to consider carefully the degree of freedom of the function approximator, which is usually directly related to the number of free parameters that have to be determined by the learning procedure. The problem is illustrated in Fig. 6.11. We have chosen there a training set of six data points (shown as stars) derived from the 'true' function $f(x) = 1 - \tanh(x - 1)$ (shown as a solid line) to which a small random number was added, values which might represent noise in the measurements. A large network, which has many free parameters, will be able

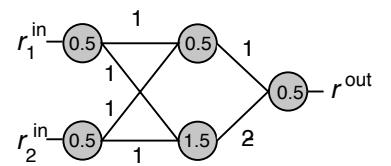


Fig. 6.9 One possible representation of the XOR function by a multilayer network with two hidden nodes. The numbers in the nodes specify the firing threshold of each node.

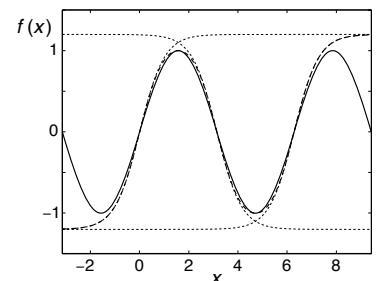
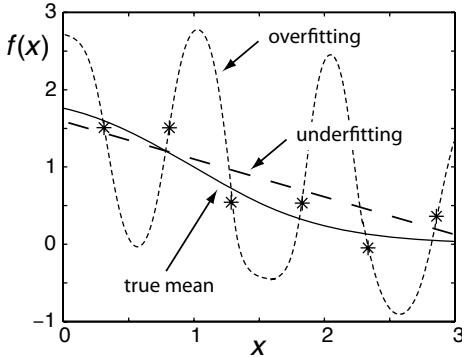


Fig. 6.10 Approximation (dashed line) of a sine function (solid line) by the sum of three sigmoid functions shown as dotted lines.

Fig. 6.11 Example of overfitting and underfitting of noisy data. Training points were generated from the ‘true’ function $f(x) = 1 - \tanh(x - 1)$, and white noise was added to these training points. A small network can represent the ‘true’ function correctly. The function represented by a large network that fits all the training points is plotted with a dotted line. The dashed line represents a linear fit.



to fit the training data with high precision but shows high *variance* between these points (dotted line). This is called *overfitting*. In contrast, a model with too few parameters, such as the straight dashed line, will have a high *bias*. This is called *underfitting*.

Several methods have been proposed to prevent over- or under-fitting. For example, high variance in a model with many parameters is often only seen late in training when the training algorithm tries to approximate the training examples with an unreasonable precision. *Early stopping* of the training has therefore been recommended to combat overfitting. Many researchers also like to restrict the number of hidden nodes to limit the number of parameters.

The problem of a *gain-variance trade-off* can be tressed more systematically with a method called *regularization*. This method is often used by introducing constraints on the number of parameters in the cost function that should be minimized by the learning rule. For example, an old idea for neural networks was to restrict the weight values by penalizing large weights. To realize this, we can add a term proportional to the length of weight vector to the MSE function used earlier,

$$E = \frac{1}{2} \sum_i (r_i^{\text{out}} - y_i)^2 - \gamma_r \frac{1}{2} \sum_i w_i^2, \quad (6.26)$$

where γ_r sets the relative importance of this regularization term relative to the estimate of the generalization error. A gradient descent on this cost function leads to a learning rule with a term resembling weight decay. We will see in the last section of this chapter that this form of regularization can be given a solid theoretical underpinning with large *margin classifiers* and *structural risk minimization*. Regularization is therefore an essential ingredient in the proper use of machine learning methods. However, the following derivation of the standard gradient descent learning rule is, for simplicity, based on MSE alone.

6.2.3 The generalized delta rules

Learning has only been discussed for single-layer networks in Section 6.1.5. The delta rule (eqn 6.15) cannot be applied directly to each layer of a multilayer mapping network since this would require a teaching signal for each node in the

network. The desired value of the output node can be supplied by a teacher, but proper values of the hidden nodes are not known *a priori*. It is, however, straightforward to generalize the delta rule in the gradient descent formalism. It is therefore surprising that this generalization was widely recognized only from the mid 1980s onward, although the algorithm was known and used for training neural networks long before. Let us illustrate the algorithm for a multilayer feedforward network with one hidden layer denoted by the superscript ‘h’. The gradient of the MSE error function with respect to the output weights is given by

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^{\text{out}}} &= \frac{1}{2} \frac{\partial}{\partial w_{ij}^{\text{out}}} \sum_i (r_i^{\text{out}} - y_i)^2 \\ &= \frac{1}{2} \frac{\partial}{\partial w_{ij}^{\text{out}}} \sum_i (g^{\text{out}}(\sum_j w_{ij}^{\text{out}} r_j^{\text{h}}) - y_i)^2 \\ &= g^{\text{out}'}(h_i^{\text{h}})(\sum_j w_{ij}^{\text{out}} r_j^{\text{h}} - y_i) r_j^{\text{h}} \\ &= \delta_i^{\text{out}} r_j^{\text{h}},\end{aligned}\quad (6.27)$$

where we have defined the delta factor

$$\begin{aligned}\delta_i^{\text{out}} &= g^{\text{out}'}(h_i^{\text{h}})(\sum_j w_{ij}^{\text{out}} r_j^{\text{h}} - y_i) \\ &= g^{\text{out}'}(h_i^{\text{h}})(r_i^{\text{out}} - y_i).\end{aligned}\quad (6.28)$$

Eqn 6.27 is just the delta rule as before because we have only considered the output layer. The calculation of the gradients with respect to the weights to the hidden layer again requires the chain rule as they are more embedded in the error function. Thus we have to calculate the derivative

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^{\text{h}}} &= \frac{1}{2} \frac{\partial}{\partial w_{ij}^{\text{h}}} \sum_i (r_i^{\text{out}} - y_i)^2 \\ &= \frac{1}{2} \frac{\partial}{\partial w_{ij}^{\text{h}}} \sum_i (g^{\text{out}}(\sum_j w_{ij}^{\text{out}} g^{\text{h}}(\sum_k w_{jk}^{\text{h}} r_k^{\text{in}})) - y_i)^2.\end{aligned}\quad (6.29)$$

After some battle with indices (which can easily be avoided with analytical calculation programs such as MAPLE or MATHEMATICA), we can write the derivative in a form similar to that of the derivative of the output layer, namely

$$\frac{\partial E}{\partial w_{ij}^{\text{h}}} = \delta_i^{\text{h}} r_j^{\text{in}},\quad (6.30)$$

when we define the delta term of the hidden term as

$$\delta_i^{\text{h}} = g^{\text{h}'}(h_i^{\text{h}}) \sum_k w_{ik}^{\text{out}} \delta_k^{\text{out}}.\quad (6.31)$$

The error term δ_i^{h} is calculated from the error term of the output layer with a formula that looks similar to the general update formula of the network, except that a signal is propagating from the output layer to the previous layer. This is the reason that the algorithm is called the *error-back-propagation algorithm*. The algorithm is summarized in Table 6.9.

Table 6.9 Summary of error-back-propagation algorithm

Initialize weights arbitrarily
Repeat until error is sufficiently small
Apply a sample pattern to the input nodes: $r_i^0 := r_i^{\text{in}} = \xi_i^{\text{in}}$
Propagate input through the network by calculating the rates of
nodes in successive layers l : $r_i^l = g(h_i^l) = g(\sum_j w_{ij}^l r_j^{l-1})$
Compute the delta term for the output layer:
$\delta_i^{\text{out}} = g'(h_i^{\text{out}})(\xi_i^{\text{out}} - r_i^{\text{out}})$
Back-propagate delta terms through the network:
$\delta_i^{l-1} = g'(h_i^{l-1}) \sum_j w_{ji}^l \delta_j^l$
Update weight matrix by adding the term: $\Delta w_{ij}^l = \epsilon \delta_i^l r_j^{l-1}$

Simulation

The implementation of the error-back-propagation algorithm is demonstrated with the program listed in Table 6.10. This program trains a sigmoidal network with two hidden nodes (`N_h=2` in Line 3) and one output node (`N_o=1`) on the XOR function discussed in Section 6.1.4, which has two input values (`N_i=2`). The weight matrices are initialized with small random values between -0.5 and 0.5 in Line 4. The training vectors are defined with the inputs in Line 7 and the desired output in Line 8. These training patterns represent the XOR function.

The training over 10,000 training examples is started with the loop in Line 11. In each of these training steps, one of the four possible patterns is selected (Line 13). This pattern is propagated through the network by calculating the activation of the hidden nodes on Line 14, and the activation of the output node on Line 15. The δ -terms (see Table 6.9) for the output layer and the hidden layer are calculated in Lines 16 and 17, and then used for the weight updates with the delta rule in Lines 18 and 19. After this, all the patterns are tested by propagating them through the network in Line 21 with the new weight matrices, and half of the summed squared distance between the actual output and the desired output is recorded in Line 22. Half of the mean square distance would be 1/4 of this value. The learning curve, which is the development of the training error, is plotted in Line 24.

A typical training curve produced with this program is shown in Fig. 6.12 (each time the program is called a different learning curve is produced because different random initial weight values are used and a different order of the training examples is presented during learning). Due to the small initial weights, the network always responds initially with values around $r^{\text{out}} \approx 0.5$ which makes the mean square error relatively small from the start. It then takes many iterations, even with the relatively large learning rate of 0.7, to finally reduce the error significantly and to reach a level of performance that is sufficient to represent the function correctly. The network responded after the 10,000 learning steps shown in the example with the values

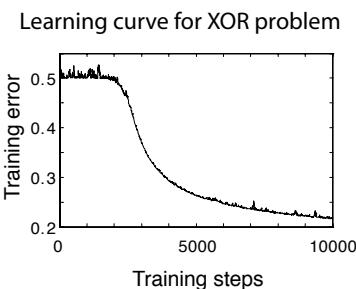


Fig. 6.12 Example of a learning function when training a MLP on the XOR problem with the error-back-propagation algorithm.

Table 6.10 Program mlp.m

```

1  %% MLP with backpropagation learning on XOR problem
2  clear; clf;
3  N_i=2; N_h=2; N_o=1;
4  w_h=rand(N_h,N_i)-0.5; w_o=rand(N_o,N_h)-0.5;
5
6  % training vectors (XOR)
7  r_i=[0 1 0 1 ; 0 0 1 1];
8  r_d=[0 1 1 0];
9
10 % Updating and training network with sigmoid activation function
11 for trial=1:10000;
12     % training randomly on one pattern
13     i=ceil(4*rand);
14     r_h=1./(1+exp(-w_h*r_i(:,i)));
15     r_o=1./(1+exp(-w_o*r_h));
16     d_o=(r_o.* (1-r_o)).*(r_d(:,i)-r_o);
17     d_h=(r_h.* (1-r_h)).*(w_o'*d_o);
18     w_o=w_o+0.7*(r_h*d_o)';
19     w_h=w_h+0.7*(r_i(:,i)*d_h)';
20     % test all pattern
21     r_o_test=1./(1+exp(-w_o*(1./(1+exp(-w_h*r_i))))));
22     d(trial)=0.5*sum((r_o_test-r_d).^2);
23 end
24 plot(d)

```

`r_o_test =`

0.0901	0.7517	0.7695	0.5596
--------	--------	--------	--------

to the four different training vectors. Such a response pattern is sufficient to represent the XOR function. Just imagine that we use a final threshold output node with threshold value 0.6.

The basic error-back-propagation is known to be very inefficient in training neural networks, and many advances have been made to improve the performance considerably, some of which we mention below. MATLAB provides a neural network toolbox that implements many such advanced algorithms, and shareware versions of different algorithms can also be found on the web. In particular, the neural network package written by Ian Nabney and Christopher Bishop called *Netlab* (see <http://www.ncrg.aston.ac.uk/netlab>) is very useful and has many advanced implementations.

6.2.4 Biological plausibility of MLPs

While mapping networks are an important ingredient in brain processing, and while MLPs have been very useful in understanding principal issues in statis-

tical learning theory, we must advise caution when applying such networks to brain theories. We mentioned above that feedforward mapping networks are universal function approximators, which necessitates caution when interpreting them in computational neuroscience. For example, in experiments we often measure some response function as a function of some parameters controlled experimentally. A multilayer mapping network, as universal approximator, will be able to approximate every such function arbitrarily well with the right choice of parameters in the network. When such networks are used to fit experimental data one might be tempted to claim that these systems represent models of the brain, on the basis that the processing nodes in the network resemble neurons. Such claims have contributed to the low acceptance of such models in the neuroscience community.

Network approximation of a particular problem depends strongly on the number of hidden nodes and the number of layers. Adding more hidden nodes can drastically change the representation of learning examples in the network. An interpretation of hidden node activities in such networks is therefore questionable when the models are aimed at a neuronal level. In connectionist models it is common to limit the number of hidden nodes drastically to enable better generalization performances of such networks. Using a small number of hidden nodes results in a relatively smooth interpolation between training data, and a better generalization can be expected for smooth problems. However, the number of hidden nodes in biological systems can be very large, which makes this kind of analysis different from building actual brain models on a neuronal level.

Training MLPs with the generalized delta rule (error-back-propagation) is particularly problematic for biological interpretations, specifically if this training is intended to model the dynamics of biological learning based on synaptic plasticity. The back-propagation of error signals, necessary for the generalized delta rule, seems difficult to realize in cortical networks. While some form of information exchange between postsynaptic and presynaptic neurons is possible, the wide use of such mechanisms for a back-propagation of errors through the whole network introduces several other problems. A major problem is the non-locality of the algorithm in which a neuron has to gather the back-propagated errors from all the other nodes to which it projects. This not only raises synchronization issues, but also has disadvantages for true parallel processing in the system. The inclusion of derivative terms in the delta signals is also problematic. The back-propagation of inaccurate derivative terms can quickly lead to inaccurate updates of the weights in the network. Finally, it has never been resolved how a forward propagating phase of signals can be separated effectively from the back-propagation phase of the error signals. Some alternative schemes closely resembling the basic scheme of error-back-propagation, while claiming to be more biologically realistic, have been proposed in the literature, although direct verifications have not yet been established.

The caution that must be used in applying mapping networks to brain modeling does not negate the importance of mapping networks in the brain. Information is frequently mapped between different representations, and cognitive abilities are supported by such mapping abilities. A better understanding of precisely how this is achieved in the brain is a major topic in computational

neuroscience, and we will later discuss how self-organization in hierarchical networks solve many of the issues. Also, feedforward networks have been useful in illustrating the difference of local versus distributed processing. Finally, it is good to recognize that even single-layer networks, which do not share most of the interpretation challenges of MLPs, can represent complicated functions, even non-linear separable functions, if we allow some more advanced processing in the single nodes compared to that accomplished by the very much simplified population node. The problem of representing non-separable functions can be reduced by *expansion recoding*, an increase of the representation dimensionality of the input vectors as discussed further in Chapter 8. Also, modular networks, which we will discuss in Chapter 10, can break the problems down in various other ways, so that simplified learning rules can be used in each subsystem.

6.3 Advanced MLP concepts ◇

6.3.1 Kernel machines and radial-basis function networks

The activity of a node in a neural network is determined by the inner product between an input vector \mathbf{r} and the weight vector of the node, \mathbf{w} . With our convention that the weight vector is a line vector and the input feature vector is column vector, we can write the net input of a node as

$$h = \mathbf{w}\mathbf{r} = \sum_i w_i r_i. \quad (6.32)$$

In the letter recognition example, we used the value of each pixel of the ‘retinal’ image, which we label here as x_i , for each component of the feature vector r_i . In machine learning it is common to call such basic features, which are given to us, the *attributes*. It has long been recognized that better performance of networks can sometimes be achieved by including combinations of attributes, such as x_1x_2 or x_i^2 , as features in the machine learning task. We can write such a transformation of the original feature space to the new feature space as

$$\mathbf{x} \rightarrow \phi(\mathbf{x}) \quad (6.33)$$

The new feature vector has, of course, a higher dimensionality, so that we also need to use more weight values. Since these weight values are anyhow parameters that we have to fix through a learning procedure, we can also introduce new parameters through the same transformation,

$$\mathbf{w} \rightarrow \phi(\mathbf{w}). \quad (6.34)$$

The node in the network for the transformed problem can then be written as

$$\tilde{h} = \phi(\mathbf{w})\phi(\mathbf{r}) = K(\mathbf{w}, \mathbf{r}). \quad (6.35)$$

We have introduced thereby a new function K , called the *kernel function*. This function calculates the scalar product of the transformed vectors. For example, let us consider a two-dimensional attribute vector $(x_1, x_2)'$ and the

transformation that considers all combinations of the components,

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \phi(\mathbf{x}) = \begin{pmatrix} x_1 x_1 \\ x_1 x_2 \\ x_2 x_1 \\ x_2 x_2 \end{pmatrix}, \quad (6.36)$$

which is a four-dimensional feature vector. The scalar product of $\phi(\mathbf{w})\phi(\mathbf{x})$ can be calculated more easily as $(\mathbf{w}\mathbf{x})^2$, as can easily be verified. We can also go the other way and propose a kernel function and then try to determine the corresponding feature space transformation. We are here particularly interested in the Gaussian kernel function

$$K(\mathbf{w}, \mathbf{x}) = e^{\frac{-(\mathbf{w}-\mathbf{x})^2}{2\sigma^2}} \quad (6.37)$$

This kernel function is interesting as it fits well the shape of some response functions of neurons, as discussed more in Chapter 7. While the dimensionality of the corresponding transformed feature space is infinite, we do not have to calculate this transformation at all for the network, we only need to replace the original network (eqn 6.32) with the kernel machine (eqn 6.35).

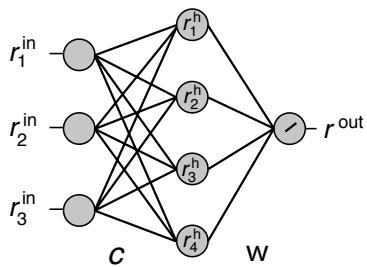


Fig. 6.13 Example of a radial-basis function (RBF) network with three input nodes, four hidden RBF nodes, and one linear output node. The weights to the RBF node, \mathbf{c} , represent the centres of the radial basis functions as shown in Fig. 6.14.

Feedforward mapping networks with activation functions that depend on the distance between an input value and weight value, most commonly in the form of the Gaussian kernel eqn 6.37, are also called *radial-basis function (RBF) networks*. A typical RBF network is shown in Fig. 6.13. This network has three input nodes and four hidden nodes with a RBF activation function as shown in Fig. 6.14. Thus, the weights to the RBF nodes, \mathbf{c} , represent the centres of the radial basis functions. We will relate such networks to cortical maps with tuning curves in Chapter 7. The example shown has one output node with a linear activation function. Such an output perceptron can decode the feature representation in the hidden layer, as discussed further in Chapter 7.

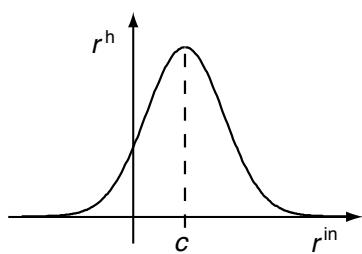


Fig. 6.14 Example of the activation function of one node in a RBF network.

6.3.2 Advanced learning

The basic error-back-propagation algorithm has many performance problems, and a lot of effort has been devoted to the improvement of the basic version of this gradient descent method. Improved techniques include the smart choice of initial conditions, different error functions, various acceleration techniques, and hybrid methods. We will only provide a short overview of some of these techniques without many details because they are often of minor biological significance and the subject of more technologically oriented publications.

In applications of the basic gradient descent method we can typically find an initial phase where the average error over the training examples is rapidly decreasing. However, this is unfortunately often followed by a phase of very slow convergence, often caused by a shallow part of the error function. Many solutions have been proposed to overcome this problem. One of the oldest is to use a *momentum* term that ‘remembers’ the change of the weights in the previous time step,

$$\Delta w_{ij}(t+1) = -\epsilon \frac{\partial E}{w_{ij}} + \alpha \Delta w_{ij}(t). \quad (6.38)$$

The momentum term has the effect of biasing the direction of the new update vector towards the previous direction (hence the name ‘momentum’), which is often a good guess for improved weight values. Another method is to increase the learning rate when the gradient becomes small. Several methods with adaptive learning rates were proposed and are used in technical applications. Acceleration of the gradient descent method is very important as the convergence to an acceptable error level can take thousands of learning steps.

Shallow areas in the error function depend on the particular choice of the error function on which the gradient descent method is based. An acceleration of the learning process can often be achieved with error functions other than the MSE. A particularly interesting choice is the *entropic error function*,

$$E = \frac{1}{2} \sum_{\mu,i} [(1 + y_i^\mu) \log \frac{1 + y_i^\mu}{1 + r_i^{\text{out}}} + (1 - y_i^\mu) \log \frac{1 - y_i^\mu}{1 - r_i^{\text{out}}}], \quad (6.39)$$

which is a measure for the information content (or entropy) of the actual output of the multilayer perceptron given the knowledge of the correct output. Although this error function looks computationally more demanding than the MSE, the application in gradient descent methods can be computationally less demanding. For example, the delta term of the output layer with an activation function $g(x) = \tanh(x)$ reduces to $\delta_i = y_i - r^{\text{out}}$. The appropriate choice of the error function depends on the statistical nature of the data. For example, it can be shown that the maximum likelihood estimate of Gaussian data correspond to minimizing the MSE error function. Modern machine learning has shed new light into this area.

The basic line search algorithm of gradient (or *steepest*) decent is known for its poor performance with shallow error functions. However, the training algorithms discussed here are based on the minimization of an error function, and we can employ many other advanced minimization techniques to achieve this goal. Several of these techniques, common in technical minimization procedures, take higher-order gradient terms into account. These can be viewed as including curvature terms describing the curvature of the error surface in the weight change calculations. Such methods typically involve the calculation of the inverse of the Hessian matrix, which can be numerically time-consuming. In the context of statistical learning theory we have to evaluate a proper distance measure such as the *Fisher information*. A superior gradient method, called *natural gradient algorithm*, has been proposed by *Shun-ichi Amari* based on such considerations.

The standard *Levenberg–Marquardt method* is also based on higher-order gradient information, and has been used to train feedforward mapping networks. This method is included in the MATLAB Neural Networks toolbox from The MathWorks, Inc. Simulations of training mapping networks with the natural gradient method and with the Levenberg–Marquardt method have demonstrated that these methods can overcome the problems of shallow minima (and even turning points in the error function), and that the convergence times for training the networks to a specified training error are many orders of magnitude less than those of other learning methods. The drastically improved convergence times can outnumber the increased simulation time due to the increased algorithmic complexity. Such algorithms should therefore be con-

sidered when applying mapping networks to technical problems. The relations of such algorithms to biological learning are, however, so far unclear.

A general limitation of pure gradient descent methods is the possibility that the network gets trapped in a local minimum of the error surface. The system is then not able to approach a global minimum of the error function as anticipated. This is illustrated in Fig. 6.6 where the global minimum on the right was actually not reached. A solution to this problem is to include some stochastic processes that enable random search. Several methods with random components are very successful in training multilayer perceptrons, most notably a method called *simulated annealing*. This method adds some noise to the weight values during the update of the weights on top of a deterministic algorithm such as gradient descent. This noise helps to escape shallow local minima. The noise level is then gradually reduced to ensure convergence.

A variety of methods utilize the rapid initial convergence of the gradient descent method and combine it with global search strategies. For example, we can employ a method in which, after the gradient descent method slows down below an acceptable level, a new starting point is chosen randomly. From this new starting point in the configuration space a few gradient descent steps are performed on the network. Only when the error value after these few gradient descent steps is lower than that at the previous level do we accept these new weight values for further gradient descent steps. Such *hybrid methods* combine the efficient local optimization capabilities of gradient descent methods with the global search abilities of stochastic processes. Genetic algorithms, discussed briefly below, use similar combinations of deterministic minimization and stochastic components.

6.3.3 Batch versus online algorithm

In technical applications of supervised learning, in which we have a set of examples of a function we want to represent with the mapping network, we can do the repetitions mentioned in the summary (Table 6.4, item 6) in various ways. A pattern can be learned very accurately with the gradient descent method if we train the network only on one pattern for a long time before switching to the next pattern. However, when a second pattern is trained consecutively in a similar way, then the network unlearns the previous pattern because the weights are adjusted entirely to represent the new pattern. This is not, of course, what we intended. In practical applications of the delta rule it is often (though not always) the best strategy to represent all patterns in a training set in a way that is known as a *batch algorithm*. In this strategy we first apply all patterns in a training set to the network and calculate the desired changes for each pattern. Only after all the patterns have been applied do we change the weights according to the mean value over all patterns before we repeat this procedure. Batch algorithms often work well in practice if there are not many redundancies in the training data, though batch algorithms are a bit more prone to get stuck in local minima.

However, batch algorithms are not plausible in biological systems, and it is more realistic to assume that the sensory information is processed directly. *Online learning algorithms* are therefore very important in computational neu-

roscience. These algorithms seem at first more restrictive and not as effective as batch algorithms as we can not repeatedly utilize past experiences. However, batch algorithms also have severe problems when applying them to many realistic applications. This becomes apparent if we think about training a network on realistic data, for example, on visual images. The data stream generated by the eyes is enormous, and even in artificial systems, such as systems that employ a digital camera with a much lower resolution than that of human eyes, the amount of data generated is still enormous. The amount of storage necessary to keep the images present for the batch algorithm will rapidly overload any reasonable system. Online algorithms are therefore not only more interesting in computational neuroscience, but also for technical applications.

6.3.4 Self-organizing network architectures and genetic algorithms

In the previous discussions we have always started with a particular architecture of feedforward networks by specifying the number of layers and the number of nodes per layer. The architecture is often crucial for the abilities of such networks. The universal-approximation theorem only tells us that mapping networks with enough hidden nodes can represent any mapping function, but this does not give us any hint as to how many nodes we need in practice, nor how these nodes should be connected. Too few nodes might prevent the mapping network from being able to represent a mapping function, and too many nodes in the design can drastically diminish the generalization abilities of the network. Several algorithms have been proposed to help with the design of networks. We call these types of learning procedures *design algorithms*.

There is often a critical dimension with regard to the number of hidden nodes necessary for a mapping network to be able to represent a particular function to a desired accuracy. An example is shown in Fig. 6.15 where the average error (normalized sum over all training examples) and the worst error (largest error of one training example) are shown. The network was trained on the task of adding two binary 3-digit numbers, and a new hidden node was created after some fixed amount of training with the specific number of hidden nodes of a mapping network, as indicated by the numbers between the dashed lines in the figure. Although the average error decreased steadily (which can be expected as the number of parameters increases), some examples were always not well represented until seven hidden nodes were reached. This is an example of a procedure called a *node creation algorithm*, which uses this effect by starting with a small network and increasing the number of nodes systematically until the required performance of the network is reached. Other procedures, called *pruning algorithms*, start with a large number and decrease the number of nodes until a satisfactory solution is found. A particularly interesting version of a pruning algorithm is *weight decay*, as already discussed in Chapter 4.

Some design algorithms, which are very relevant in biological systems, are termed *genetic algorithms* because of their attempt to simulate evolutionary processes. Evolutionary computing and genetic algorithms are active research areas where further advances can be expected; in particular, in the area of *co-evolution*. The genetic algorithms discussed here are certainly vastly abstracted

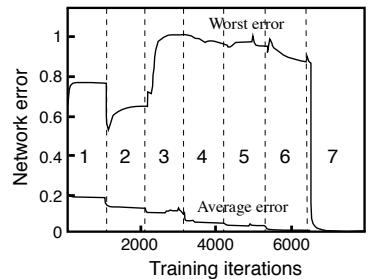


Fig. 6.15 Example of the performance of a mapping network with a hidden layer that was trained on the task of adding two 3-digit binary numbers. After a fixed amount of training a new hidden node was created (indicated by the vertical dashed line) [redrawn from Ash, *Connection Science* 1: 365 (1989)].

from evolutionary processes in nature, but they are intended to illustrate some important basic ingredients.

Genetic algorithms work commonly on large vectors, typically called *genomes*, which describe structures to be optimized. To apply such algorithms to the design of neural networks we can, for example, define a large vector with components that are either one or zero. The ones indicate the presence, and the zeros the absence of a connection between corresponding nodes in a network. Such a connectivity vector therefore describes a particular architecture, which is the analogue to an *individual* in a biological system. The principle behind genetic algorithms is to generate a large set of such genome vectors (individuals), called a *population*, and to evaluate each single vector with the help of an *objective function*, which specifies the performance of each individual. From this population we generate a new population using *genetic operators*. There are many possible choices of such operators of which we only mention some that seem particularly important. At first there should be a *survival operator* that saves only the individuals that perform well while poor performers are discarded. A new population of new individuals (*offspring*) are generated only by the good performers. Offspring can, for example, be generated by combining parts of the genome of two good performers with an operation that is called *cross-over*. In addition, it is essential to allow some random modifications to the result. We can think of this operation as *random mutation*.

The generation of a new population with individuals that are likely to perform better than previous generations in terms of the objective function is essential for the optimization capability of a genetic algorithm. In this way genomes are optimized to perform a certain task. The application of these simple algorithms shows that many generations are necessary in order to generate satisfactory individuals. This fact often prevents the use of these algorithms for the design of networks in engineering applications. The slow convergence of such simplified algorithms also suggests that additional organizing principles that enable a more efficient optimization within evolutionary strategies might be important. It is, however, widely accepted that the principles behind evolutionary algorithms have helped to develop the major structure of the central nervous system. It is important to realize that the global structures of the brain are similar in different individuals and are therefore likely to be coded genetically. The comparison of different species has shown that such structures have been evolving over time, with evolutionary forces driving new developments. In contrast to the evolutionary forces that allowed the evolution of important structures of the brain, the learning algorithms discussed throughout the book are primarily thought to fine-tune the brain within the genetically dedicated designs. Understanding the learning abilities of neural networks within the constraints of the brain organization is therefore a major concern of computational neuroscience and should not be forgotten in the discussions of neuronal networks.

6.3.5 Mapping networks with context units

The study of cognitive abilities of humans reveals that our behaviour, for example the execution of particular motor actions, often depends on the context

in which we encounter a certain situation. For example, we might encounter an equivalent situation on two consecutive days, such as seeing a person in front of a house who is apparently studying the building in some detail. On the first day we might just think that this person is interested in architecture and we will likely proceed without acting further on this encounter. In the morning of the second day we might read in the newspaper about an increase in burglaries in the area, and seeing the person of the previous day again studying a house might very well prompt us to enquire about his or her intentions.

A simple architecture that demonstrates some form of contextual processing was proposed by *Jeffrey Elman* and is sometimes called an *Elman-net*. An example is outlined in Fig. 6.16, which illustrates a mapping network with four input nodes, three hidden nodes, and four output nodes. In addition to the standard feedforward mapping components, the network has *context nodes*. The three context nodes shown in the figure contain the activation of the hidden nodes of the previous time step. Furthermore, the activations of the context units are fed back into the system as internal inputs (rather than external inputs) for the next time step. The architecture therefore includes a new class of projections that feed into cells which, in turn, can influence the sending node at a later time. The network is therefore said to be *recurrent*. This type of physical back-projection should not be confused with the information flow that is used during training the networks, such as in the error-back-propagation algorithm discussed above.

The network in Fig. 6.16 is only one example of this class of networks, where we have included context nodes that receive inputs from the hidden nodes. We can also include context units that receive inputs directly from output or input nodes. The latter remember the input of the previous time step, and such a mechanism is often termed *short-term memory*. We will discuss related but distinct forms of short-term memory as used in the physiology and psychology literature in Chapter 8.

The context units in the example of Fig. 6.16 are designed to contain the activity of the hidden nodes at the previous time step. This can formally be achieved with linear units and setting the weight values to one while assuming some delay in the projections. The network functions as follows. For each input to the network, consisting of the external input from the input nodes and from the context nodes (which memorized the previous firing rates of the hidden nodes), we calculate the activation of the hidden nodes and then the activation of the output nodes. The activation of the hidden nodes is also copied to the context nodes. All this can be thought of as a basic time step of the network. Thus, we can treat the network at each time step as a standard feedforward mapping network and can thus use the back-propagation algorithm discussed before. However, the function of the network now has inherent time dynamics in that there is a new input to the hidden nodes at the next time step (even with the same external input from the input nodes as in the previous time step). To take into account the context during training we have to train the network on whole sequences of inputs. The advantage is that the network can generate sequences of outputs, and we can thus use those models in a wider context. In particular, these networks can learn to predict the next output in a sequence of symbols, and they have been used primarily in this context.

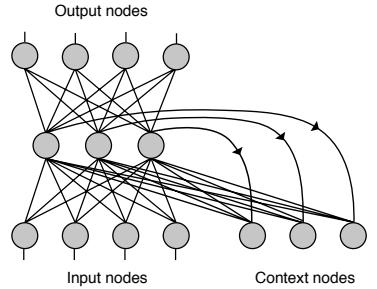


Fig. 6.16 Recurrent mapping networks as proposed by Jeffrey Elman consisting of a standard feedforward mapping network with four input nodes, three hidden nodes, and four output nodes. However, the network also receives internal input from context nodes that are efferent copies of the hidden node activities. The efferent copy is achieved through fixed one-to-one projections from the hidden nodes to the context nodes that can be implemented by fixed weights with some time delay.

6.3.6 Probabilistic mapping networks

The outputs of a mapping network can also be interpreted as probabilities that a specific input vector has certain features represented by the individual output nodes. Such networks are particularly useful for data classification. For such applications we can employ a feedforward mapping network that has n^{out} output nodes, where n^{out} is equal to the number of possible classes to which an object represented by the input vector can belong. The activity of each output node can be interpreted as the probability of membership of the object to the class represented by the node. Such mapping networks are sometime called *probabilistic feedforward networks*. Note that such probabilistic networks are different from *stochastic networks* in which the updating rule of the nodes has some probabilistic (stochastic) components (discussed further in Chapter 8).

To allow a probabilistic interpretation of the activities of the output nodes one needs to normalize the sum of all output activities to one (see Appendix C),

$$\sum_i r_i^{\text{out}} = 1. \quad (6.40)$$

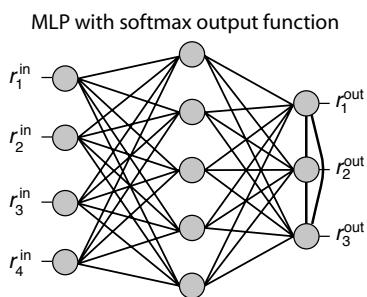


Fig. 6.17 Mapping network with collateral connections in the output layer that can implement competition in the output nodes.

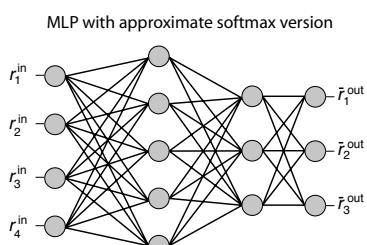


Fig. 6.18 Normalization of the output nodes with the softmax activation function can be implemented with an additional layer.

The reason for this is that the probability of the input vector belonging to any of the classes must be one if the nodes represent all possible classes. Such a condition can be achieved, for example, with an output layer that competes for the output, which can be realized with collateral connections between the output nodes as shown in Fig. 6.17. These collateral connections can be inhibitory so that the strong activity of one node inhibits the firing of other nodes. In the extreme case of very strong inhibition, so that only one node is active for each input vector, we speak of a *winner-take-all* architecture. Such competitive networks with collateral connections are discussed in Chapter 7. Here we only want to stress the probabilistic interpretation of the output of mapping networks.

A winner-take-all output layer in a mapping network can only indicate the class to which an object represented by an input vector is most likely to belong. More useful is a representation of class-membership probability, so that we can also gain some information on the confidence with which the network has classified the input vector. This can be achieved with soft competition in the output layer. In practice we can simulate this with a normalization of the output activity using the *softmax* function

$$\bar{r}_i^{\text{out}} = \frac{e^{r_i^{\text{out}}}}{\sum_j e^{r_j^{\text{out}}}}, \quad (6.41)$$

where r_j^{out} are the firing rates of the output nodes before the normalization step, and \bar{r}_i^{out} is the final output of the output nodes that can be interpreted as probabilities of class membership because $\sum_j \bar{r}_j^{\text{out}} = 1$. This can be implemented with a feedforward network as illustrated in Fig. 6.18, where the new output layer has a (non-local) softmax activation function.

Many learning algorithms are based on performance measures that represent the objective function. For the probabilistic interpretation it is appropriate to

use cross-entropy

$$E = - \sum_{\mu} \sum_i t_i^{\mu} \bar{r}_i^{\text{out}}(r_i^{\text{in}}, W; \mu) \quad (6.42)$$

as an objective function that measures the distance in the probabilistic framework. The actual output of the network, \bar{r}_i^{out} , depends on the specific input vector, for example, μ , and the weights \mathbf{w} of the network. The vector \mathbf{t}^{μ} is the desired target vector for the example μ , which consists of zeros for all but one node, the node that corresponds to the known class of the training example, which is equal to one. Training algorithms can be designed to minimize such objective functions. Often the back-propagation algorithm, which we will outline in Chapter 9, is used in such applications.

6.4 Support vector machines ◇

In this chapter we walked a fine line between biological plausibility and solving a general statistical learning problem. Perceptrons have shown us that feedforward networks of neuron-like (or population) elements have interesting capabilities and that learning in such systems, at least without hidden units, has signatures of Hebbian plasticity. More formally, we have seen that MLPs are universal approximators in the sense that correct mappings between training data can be learned given enough hidden nodes. However, there remain many problems with such machines, specifically the efficiency of learning and their generalization abilities. We encountered the inefficiency of learning by many training cycles when training an MLP on the XOR example, and larger problems often suffer from slow convergence and local minima. Also, it is not only the learning (memorization) of training data that are the important part of applications of such machines. The real challenge comes when applying such machines to unseen data. MLPs are known to be fairly good interpolators, but extrapolation of data is often weak. We have seen some pure generalization ability in the letter example in that the trained networks were not very robust against noise.

Many problems of MLPs have now been solved by advanced *machine learning* methods. In this final section, we briefly review machine learning in some more generality and introduce the basic ideas behind support vector machines which have now replaced MLPs in many applications. While this is a brief deviation of discussing brain processing, this discussion is important for two reasons. First, machine learning had historically a close connection to brain research, and many advancements of theoretical descriptions of brain processing have been derived in this scientific discipline. Second, machine learning methods are becoming essential tools in neuroscience for data analysis. For example, machine learning tools have been essential in analysing brain imaging data to build brain-computer interfaces. The material covered in this section was largely developed by the Russian scientist *Vladimir Vapnik*.

6.4.1 Large-margin classifiers

Machine learning problems can be classified into regression and classification problems. In regression problems one tries to find suitable interpolations of

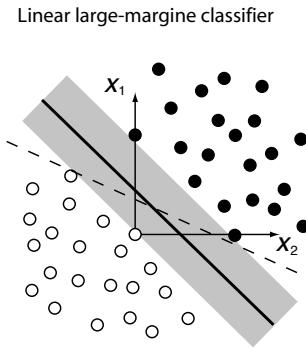


Fig. 6.19 Illustration of a large margin classifier for a linear separable problem. An example of a decision boundary as typically learned by a perceptron is included as a dashed line.

¹This linear separable classification problem is a generalization of the Boolean OR function illustrated in Fig. 6.5A.

data points, whereas in classification problems one tries to give data a label of different categories. Here we discuss primarily classification problems, although the methods mentioned here can be generalized to regression problems. We speak about binary classification if there are only two categories. Learning Boolean functions is an example of a binary classification problem. A more general binary classification problem is illustrated in Fig. 6.19. A solution that can be found by a perceptron is included as the dashed line. This line is likely to be close to some points as learning ceases after all training examples are correctly classified.

Two interesting points can be made with this example. The first point is that most of the data are not essential for classification. Only data points close to the *decision boundary* are essential¹ where the additional data points do not contribute information to the best decision boundary. The second point is that it is likely that some future data are misclassified with the perceptron solution since the training examples might not have included all the points close to this line or if noise is present in the measurements. A better solution, indeed the best solution in this situation, is to choose a line (decision boundary) that is furthest from all points. Such a solution is indicated as the shaded area. The best solution is the decision surface in the middle of this area when the width of the area is maximized. The distance from the middle line to the border is called the *margin*. Large-margin classifiers are thus typically more robust than perceptrons.

It is instructive to formalize this observation somewhat. We have already seen (formula in Fig. 6.5A) that a line in this graph can be written as

$$w_1x_1 + w_2x_2 - \theta = 0. \quad (6.43)$$

The lines on the edges of the shaded areas are similarly

$$w_1x_1 + w_2x_2 - \theta = 1 \quad (6.44)$$

$$w_1x_1 + w_2x_2 - \theta = -1, \quad (6.45)$$

²We can set the offset to an arbitrary value, m , and then divide the whole equation by this number. Eqns 6.44 and 6.45 are then recovered by changing notations with the mappings $w_1 \leftarrow w_1/m$; $w_2 \leftarrow w_2/m$ and $\theta \leftarrow \theta/m$

where we used a freedom of scale by setting the offsets of the lines to the separating line (eqn 6.43) to one.² The distance of these lines to the origin can be calculated from basic trigonometry and is given by $(\theta + 1)/|\mathbf{w}|$ and $(\theta - 1)/|\mathbf{w}|$, respectively. where $|\mathbf{w}| = \sqrt{w_1^2 + w_2^2}$ is the norm of the vector \mathbf{w} , and the distance between the lines is the size of the margin,

$$d = \frac{2}{|\mathbf{w}|}. \quad (6.46)$$

Maximizing the margin is therefore equivalent to minimizing the weights, which was already a common trick for perceptrons, as mentioned in Section 6.2.2. Of course, the maximization of the margin has to be subject to the restrictions that there are no data points in the margin. That is, all data points of a class to which we give the label $y = +1$ should be above the line defined by eqn 6.44, and the data points of the other class, to which we give label $y = -1$, should be below the line defined by eqn 6.45. These conditions can be written as inequalities in a compact way. Also, we can generalize the equations to arbitrary dimensions by using vectors for the weights (row vectors in our

standard notation) and data points (column vectors), $w_1x_1 + w_2x_2 + \dots = \mathbf{w}\mathbf{x}$. In addition, with the choice of class labels, we can multiply these equations by y so that the two eqns 6.44 and 6.45 can be summarized into one,

$$y(\mathbf{w}\mathbf{x} - \theta) - 1 \leq 0. \quad (6.47)$$

The optimization problem, minimizing the weights, or maximizing $1/2|\mathbf{w}|^2$, subject to the constraints that no points are in the margin, eqn 6.47, can be solved by using the *Lagrange formalism*. For this we define a Lagrange function, L_P , which is a form of a cost function in which the constraints are added with so-called Lagrange multipliers, α_i , for each training example i ,

$$L_P = \frac{1}{2}|\mathbf{w}|^2 + \sum_i \alpha_i y_i (\mathbf{w}\mathbf{x}_i - \theta) + \sum_i \alpha_i. \quad (6.48)$$

Minimizing L_P is a *quadratic optimization problem* which can easily and efficiently be solved with standard numerical methods. Also, minimizing the so-called *primal* Lagrangian, L_P , is equivalent to maximizing the *dual problem* defined by the dual Lagrangian

$$L_D = \sum_i \alpha_i - \frac{1}{2} \sum_{ij} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \mathbf{x}_j. \quad (6.49)$$

We will come back to this formula shortly. Note that only data points on the margins are relevant at the end, which are the only data points for which the corresponding Lagrange multipliers, α , are not zero. These data points are called *support vectors*, and the classifiers discussed in this section are called linear *support vector machines* (SVMs).

6.4.2 Soft-margin classifiers and the kernel trick

So far, we have only discussed linear separable problems in this section. The real challenge usually starts when considering non-linear problems and situations with overlapping data. Let's first tackle non-linear problems, such as the example plotted in Fig. 6.20A. There we plotted data of two classes on a line, which is hence a one-dimensional data space. These data can not be separated linearly. However, we can solve the problem by transforming the data with a vector function $\phi(x)$, by including higher moments or combinations of the original feature values in a new data vector. For example, let us consider here the simple example

$$x \rightarrow \tilde{\mathbf{x}} = \phi(x) = \begin{pmatrix} x \\ x^2 \end{pmatrix}. \quad (6.50)$$

The transformed data, $\tilde{\mathbf{x}}$, are then lying on a parabola, as shown in Fig. 6.20B, and these data can be separated linearly. Note that the data are still on a line

A. Linear not separable case



B. Linear separable case



Fig. 6.20 (A) Illustration of linear not separable classification problem that can be solved by transforming the data as illustrated in (B).

while we consider now a data space in two dimensions. The transformation does therefore appear as increasing the dimensionality of the problem.

The challenge is now to find appropriate functions $\phi(x)$. However, we note that a lot of functions would work in this example, and the precise form of $\phi(x)$ might therefore not be crucial. Furthermore, by inserting the transformed data into eqn 6.49 we see that the function only appears in the inner product of the data, $\phi(\mathbf{x}_i)\phi(\mathbf{x}_j)$. We can replace this with a kernel function of the inner product of the original data,

$$\phi(\mathbf{x}_i)\phi(\mathbf{x}_j) = K(\mathbf{x}_i, \mathbf{x}_j). \quad (6.51)$$

This has many practical advantages. For example, even if the transformation $\phi(x)$ transforms the data space formally into an infinite dimensional space, it is still possible that the kernel function provides finite values. Furthermore, with the right choice of kernel functions we still end up with a convex optimization problem that has no local minima and can be efficiently solved numerically. A common choice that fulfils these conditions is the Gaussian kernel

$$K(\mathbf{x}_i, \mathbf{x}_j) = e^{-\frac{(\mathbf{x}_i, \mathbf{x}_j)^2}{2\sigma^2}}. \quad (6.52)$$

While this kernel introduces a new parameter, σ , it has been shown through many applications that this kernel can solve many non-linear problems. *Kernel machines* have been the subject of much interest in the machine learning community. While further motivations of using kernels can be made, using kernels is strictly speaking only a trick; albeit a highly successful.

How about the problem of overfitting as mentioned for regression problems in Section 6.2.2? When solving non-linear problems with the kernel SVMs we face again a decision about distinguishing complicated boundaries from overlapping data as illustrated in Fig. 6.21. Are these data truly separated by the solid line, or do the data have a simpler decision boundary, such as the dotted line, and are only convoluted by noise? Solving the problem requires additional knowledge which is usually not present at this level of data analysis. A common approach is to assume some smoothness of the problem by including some regularization constraints. Such a constraint is included in SVMs by minimizing the weights, and it has been argued that this works well with kernel functions and implements a form of *structural risk minimization*. However, in the non-linear case, we need to allow misclassification of training data, which is done by softening the classification conditions of eqn 6.47 by allowing misclassifications with a penalty value determined by a new parameter C . This is an additional parameter that must be tuned in the standard SVM machines. Vapnik has greatly contributed to the theory of structural risk minimization that we will not discuss further. The list in ‘Further reading’ includes some references to good tutorials and books where more discussions can be found.

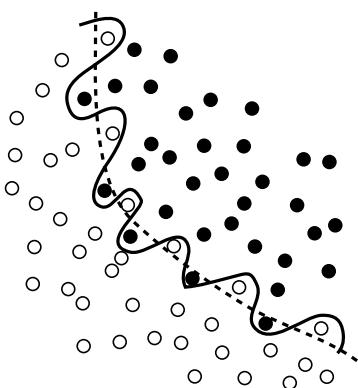


Fig. 6.21 Illustration of a non-linear classifications problem in two dimensions.

Simulation

We show here two simulations using the SVM toolbox called LIBSVM. These library functions, which provide interfaces for many programming languages including MATLAB, were written by *Chih-Chung Chang* and *Chih-Jen Lin*

Table 6.11 Program svmXOR.m

```

1  %% SVM for XOR problem
2  r_i=[0 1 0 1 ; 0 0 1 1];
3  r_d=[0 1 1 0];
4
5  model = svmtrain(r_d',r_i');
6  svmpredict(r_d',r_i',model);

```

and can be downloaded from <http://www.csie.ntu.edu.tw/~cjlin/libsvm>. We use here the default parameters of these packages, which include using a Gaussian kernel function.

The first example, shown in Table 6.11 is training the SVM on the Boolean XOR function. Training is provided in function **svmtrain** that takes a column vector of labels and a matrix of data points as inputs. The trained model is passed by the data structure **model** to the prediction routine. This function also takes labels as input, but these labels are only used to evaluate the performance. Running the program returns the line

Accuracy = 100% (4/4) (classification)

which demonstrates that the function was correctly learned by the SVM.

The second example, shown in Table 6.12, consists of overlapping data. We choose data from one class randomly with an equal distribution in a square, and draw data of the second class from a Gaussian distribution centred in the middle of the square. It is not possible to classify the data perfectly, but it is more likely that data close to the centre of the square are from the second class. The 100 data points for each class used to train the SVM are shown with different symbols for each class in Fig. 6.22. We tested the trained SVM on a grid of data to evaluate which areas are labelled by the SVM with different

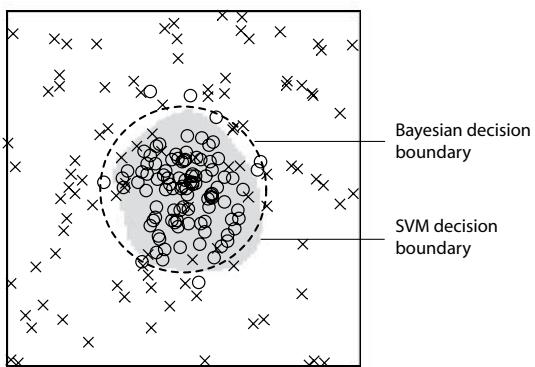


Fig. 6.22 Example of training a SVM with Gaussian kernel on overlapping data. The shaded area indicates the decision surface of the SVM that was trained on the data plotted with different symbols for each class. The dashed circle represents the optimal Bayesian decision boundary.

Table 6.12 Program svmOverlap.m

```

1  %% SVM on overlapping data
2  clear; clf; hold on
3
4  %% training vectors
5  x=[10*(rand(100,2)-0.5);randn(100,2)];
6  y=[zeros(100,1);ones(100,1)];
7  plot3(x(1:100,1),x(1:100,2),2*ones(100,1),'x');
8  plot3(x(101:200,1),x(101:200,2),2*ones(100,1),'o');
9  model = svmtrain(y,x);
10
11 %% plotting decition surface
12 x=[];
13 for x1=-5:0.1:5
14     for x2=-5:0.1:5
15         x=[x;x1 x2];
16     end
17 end
18 y=svmpredict(ones(10201,1),x,model);
19 y=reshape(y,101,101);
20 x=meshgrid(-5:0.1:5);
21 surf(x,x',y,'linestyle','none'); view(0,90);
22 cmap=[1 1 1; 0.9 0.9 0.9]; colormap(cmap);
23 axis square; box on;

```

classes. We used thereby a vector with only one label in the evaluation function as this was not needed for these experiments. The accuracy reported by the function does not, therefore, reflect the true accuracy of the SVM. The area in which the SVM predicts data points from the second class is shown as the grey area in Fig. 6.22. When evaluating the performance over 100 trials one gets, for example, an accuracy of $87.2 \pm 2.2\%$ which is very good, considering that the maximal achievable accuracy is 88.2% for this example. The last value can be calculated from the integrals over the density functions in the areas where each class is dominating.

Exercises

- (6.1) Implement a single-layer perceptron and train it to translate the digital letters given in file `pattern1` into the corresponding ASCII representation. Plot a training curve and interpret your results.
- (6.2) Implement an MLP and train it to translate the digital letters given in file `pattern1` into the corresponding ASCII representation. Plot

- a training curve and interpret your results.
- (6.3) Investigate how much noise the perceptrons can tolerate in the pattern before being unable to recognize a letter.
- (6.4) Which letter is represented in file `pattern2`?
- (6.5) Calculate the maximal achievable accuracy in the overlapping data example of the last simulationection.

Further reading

There are many books on artificial neural networks with emphasis on different aspects. One of the best books on the engineering aspects of neural networks is the book by Simon Haykin (1999), and the books by Hertz, Krogh, and Palmer (1991) and Müller, Reinhardt, and Strickland (1995) are some great introductions. A modern view of neural networks and machine learning theory is provided by Christopher Bishop (2006). A good tutorial on SVMs, which includes discussions of structural risk minimization and the VC dimension, is given by Burges (1998), and a good tutorial on SVM regression by people who have greatly contributed to this area is given by Smola and Schölkopf (2004). The lectures by Andrew Ng (2008) on machine learning are highly recommended for further studies of this area.

Neural networks had a profound influence on cognitive science in the form of connectionist models since the classic PDP books (Rumelhart *et al.*, 1986). Another example of a book that uses connectionist modelling to understand cognitive processes is McLeod *et al.* (1998). This book includes a discussion of Elman nets, which were mentioned only briefly in this chapter. This chapter only scratched the surface of perception and sensation. A thorough introduction to this area, including the psychophysics and the underlying physiology, can be found in this book by Goldstein (1999).

Simon Haykin (1999), *Neural networks: A comprehensive foundation*, MacMillan, 2nd edition.

John Hertz, Anders Krogh, and Richard G. Palmer (1991), *Introduction to the theory of neural computation*, Addison-Wesley.

Berndt Müller, Joachim Reinhardt, and Michael Thomas Strickland (1995), *Neural networks: An introduction*, Springer.

Christopher M. Bishop (2006), *Pattern recognition and machine learning*, Springer.

Laurence F. Abbott and Sacha B. Nelson (2000), *Synaptic plasticity: taming the beast*, in *Nature Neuroscience (suppl.)*, 3: 1178–83.

Christopher J. C. Burges (1998), *A tutorial on support vector machines for pattern recognition*, in *Data Mining and Knowledge Discovery* 2: 121–67.

Alex J. Smola and Bernhard Schölkopf (2004), *A tutorial on support vector regression* in *Statistics and Computing* 14: 199–222.

Andrew Ng (2008), *CS229 machine learning*, <http://www.stanford.edu/class/cs229>.

David E. Rumelhart, James L. McClelland, and the PDP research group (1986), *Parallel distributed processing: Explorations in the microstructure of cognition*, MIT Press.

Peter McLeod, Kim Plunkett, and Edmund T. Rolls (1998), *Introduction to connectionist modeling of cognitive processes*, Oxford University Press.

E. Bruce Goldstein (1999), *Sensation and perception*, Brooks/Cole Publishing Company, 5th edition.

This page intentionally left blank

Cortical feature maps and competitive population coding

7

This chapter is about *information representation* and related *competitive dynamics* in neuronal tissue. The chapter starts with a brief outline of a basic model of a hypercolumn in which neurons respond to specific sensory input with characteristic *tuning curves*. We mentioned in Chapter 5 that such feature representations are often topographically organized, and we discuss in this chapter models that show how such *topographic feature maps* can be *self-organized*. We then study the dynamics of such maps, which can be modelled nicely as *dynamic neural field theory*. Signatures of such competitive dynamics can be seen in a variety of examples in different parts of the brain, which we will discuss. This chapter also includes some more formal discussions of population coding, as well as some extensions of the basic models including dynamic updates of represented features with changing external states.

7.1	Competitive feature representations in cortical tissue	181
7.2	Self-organizing maps	183
7.3	Dynamic neural field theory	190
7.4	‘Path’ integration and the Hebbian trace rule ◇	202
7.5	Distributed representation and population coding	205
	Exercises	212
	Further reading	213

7.1 Competitive feature representations in cortical tissue

We mentioned in Chapter 3 that neurons can represent features with tuning curves, and we reviewed in Chapter 5 that feature maps are commonly organized topographically in the sense that neighbouring feature values are represented in adjacent cortical tissue. A well-known example is that of orientation tuning curves in the primary visual cortex, as discovered by Hubel and Wiesel. We now put these observations together in a basic model of a hypercolumn. This model, shown in Fig. 7.1A, consists of a line of population nodes, each responding to a specific orientation. This orientation selectivity can be achieved by combining input from spatially selective cells, as indicated for some of the nodes. This model implements a specific hypothesis of cortical organization, namely, that the input to the orientationally selective cells is focal (very selective to a specific orientation), and that the broadness of the tuning curves is primarily the result of lateral interactions indicated by arrows in the model. The details of this model will be given below, but we will first outline some basic behaviour of such a model.

Fig. 7.1C is a graph that shows the activity of nodes during a specific experiment. We used 100 nodes for this demonstration, but we will see later that this number of nodes is not essential. Each node corresponds to a certain

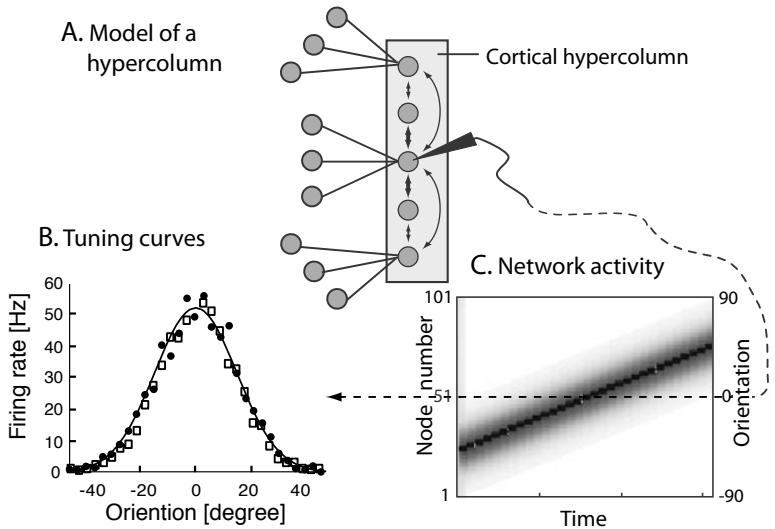


Fig. 7.1 (A) Basic model of a hypercolumn in which a lateral line of nodes is topographically organized to respond to specific orientations of line segments in the visual field. (B) The experimental data by Henry *et al.* (1974), as discussed in Section 3.3, are shown as solid circles, whereas recordings from the model are shown as open squares. (C) The evolution of the activity in the neural field during the experiment.

orientation shown with the degree scale on the right. In this experiment, the response of the nodes was probed by externally activating a very small region for a short time (in the simulations a very narrow Gaussian profile was used, essentially covering only one node). After this time, the next node was activated, probing the response to consecutive orientations during this experiment. The nodes that receive external input for a specific orientation became very active. However, neighbouring nodes are also activated through lateral interactions in the network. We call this consecutively active area an *activity packet*, although it is often simply called a *bubble*. The activation of the middle node, which responds maximally to an orientation of zero degrees, is plotted against the input orientation with open squares in Fig. 7.1B. The dots represent corresponding experimental data. The simulations also included noise proportional to the activation, not shown in Fig. 7.1C. As can be seen, the model data match the experimental data reasonably well.

Several assumptions and specific hypotheses are incorporated in this basic hypercolumn model. Specifically, we assumed in this model that orientation preference of hypercolumn nodes is systematically organized. The next section explores basic mechanisms leading to such organizations. The lateral interactions within the hypercolumn model are organized such that there is more excitation to neighbouring nodes, compared to more distant nodes, and there is inhibition between nodes that are even more remote. This lateral interaction in the model leads to dynamic properties of the model, which we study more formally in Section 7.3. Competition between features is often observed in physiological studies and has a multitude of behavioural consequences. The remainder of this chapter explores different applications and extensions of such models to demonstrate that this model captures basic brain processing mechanisms.

7.2 Self-organizing maps

Topographic maps in the brain have two major characteristics, namely that there is some order in feature space such that neighbouring areas represent neighbouring features, and that features with enhanced sensory resolution are over-represented with larger cortical space while preserving relations between feature values. The formation of such organizations is experience dependent, since the formation of such maps can be disrupted in very young animals by sensory deprivation. While the formation of these maps is often viewed as a developmental process, there is increasing evidence that cortical maps can change dramatically even in mature age when given the opportunity. As discussed in Chapter 4, the models of Hebbian plasticity in this book describe changes of synaptic weight values in general and do not distinguish between developmental and functional plasticity. The models in this subsection can be seen as general models for activity-driven structural changes in neural maps, either during development or later in the life of an organism. Such maps are called *self-organizing maps*, or *SOMs* for short.¹

7.2.1 The basic cortical map model

The model described here is based on a seminal paper by *David Willshaw* and *Christoph von der Malsburg* in 1976. We consider a two-dimensional cortical sheet, as illustrated in Fig. 7.2A. The nodes in this cortical sheet are population nodes with leaky integrator dynamics, as discussed in Chapter 3. However, we now consider these nodes in a network. To simplify the notations, we write the following equations for a one-dimensional model with N nodes, as shown in Fig. 7.2B. It is straightforward to generalize the model to two or higher dimensions by using more indices (e.g. $i \rightarrow i_1 i_2 i_3$) or treating the index i as a vector (e.g. $i \rightarrow \mathbf{i}$). The change of the internal activation, u_i , of node i is given by:

$$\tau \frac{du_i(t)}{dt} = -u_i(t) + \frac{1}{N} \sum_j w_{ij} r_j(t) + \frac{1}{M} \sum_k w_{ik}^{\text{in}} r_k^{\text{in}}(t), \quad (7.1)$$

where τ is a time constant, w_{ij} is the lateral weight from node j to node i , w_{ik}^{in} is the connection weight from input node k to cortical node i , $r_k^{\text{in}}(t)$ is the rate of the input node k , and M is the number of input nodes. The activity of input nodes represents specific feature values, whereas the rate $r_i(t)$ of the cortical node i is related to the internal activation via an activation function. In this chapter we will primarily use the *sigmoid function*, with gain parameter β and offset parameter α ,

$$r_j(t) = \frac{1}{1 + e^{\beta(u_j(t) - \alpha)}}. \quad (7.2)$$

Eqns 7.1 and 7.2 are the most fundamental equations used in many parts of this book. They were introduced in a slightly more general form by Stephen Grossberg in the late 1950s, and have since dominated many neural network models.

The basic model defined by eqns 7.1 and 7.2 distinguishes between two sets of weight values, the weight values of the input connections, w_{ik}^{in} , and the lateral weights in the cortical sheet w_{ij} . In this section we are concerned with learning

¹The models discussed here are based on changes of neuronal responses based on Hebbian plasticity and long-range competition within the cortical sheet. It is debated whether long-range topographic maps, such as the somatosensory homunculus shown in Fig. 5.5D, can be organized through synaptic mechanisms alone. It is known that cortical development is guided by extracellular chemical substances, and it is possible that ongoing topographic plasticity is guided by similar chemical processes. The models here would then apply on a more abstract level.

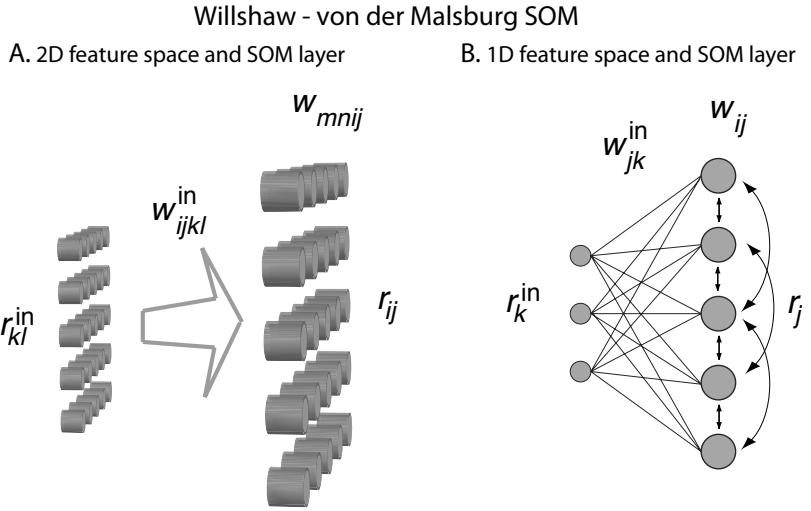


Fig. 7.2 Architecture of self-organizing maps according to Willshaw and von der Marlburg in (A) two dimensions and (B) one dimension. The left layer of nodes represents sensory units and the right layer represents a cortical sheet. The purpose of the model is to illustrate learning experience based organization of input weights, while the weights in the cortical sheet have a fixed organization of short-distance excitation and long-distance inhibition.

the topographic mapping between input and output nodes while keeping the lateral weights fixed. Learning of the lateral weights will be considered later. Here we set them to depend only on the distance between two nodes, with positive values (excitatory) for short distances and negative values (inhibitory) for large distances. Specifically, we chose a shifted Gaussian for most of the demonstrations,

$$w_{ij} = A_w \left(e^{-((i-j)*\Delta x)^2 / 2\sigma^2} - C \right), \quad (7.3)$$

with parameters A_w , σ , and C . To minimize boundary effects, we consider the model with periodic boundaries, so that the nodes form a ring (or a *torus* in higher dimensions).

The part considered next is the learning of input weights, w^{in} . We start with a random weight matrix so that no specific order is present before training. The learning follows general Hebbian philosophy. A specific feature is randomly selected, and the corresponding area around this feature value is activated in the input map. This activity triggers some response in the cortical map, likely at different places. However, an important feature of the dynamics in the cortical sheet with the lateral weights considered here is that this sheet functions as a kind of winner-takes-all (WTA) network in that it develops a dominating activity packet (as in Fig. 7.1C) through short-distant cooperation and long-distant competition. This will be discussed in more detail in Section 7.3. Thus Hebbian learning of the input rates results in an increase of weights between the activated input nodes and the winning activity packet in the cortical sheet.

7.2.2 The Kohonen model

While Willshaw and von der Malsburg have demonstrated that the model leads to topographic organizations, we will demonstrate this with a much more efficient model introduced by Teuvo Kohonen. This model makes a number of simplifications that are appropriate for the following discussions and consistent

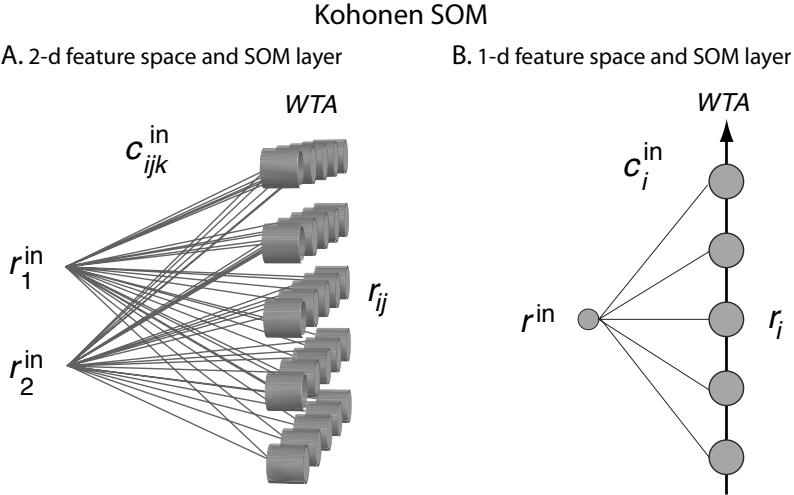


Fig. 7.3 Architecture of self-organizing maps according to Kohonen in (A) two dimensions and (B) one dimension. The WTA represents the procedure of finding the node with the maximal response for a given input feature.

with our philosophy of minimalistic modelling. The first simplification is that the representation of the input feature is changed. While a specific feature value is represented by the activation of a specific node in the previous model (e.g. by a coordinate of the activated node), the new model represents this feature value as activation value of one node in the one-dimensional model, or d input nodes in the d -dimensional case. A two-dimensional version of this model is shown in Fig. 7.3A and a one-dimensional version in Fig. 7.3B. With this feature representation, the values of the input connections should be interpreted as representing the preferred feature value (the preferred orientation in the case of V1 tuning curves) of the receiving nodes. To indicate this different interpretation of the connection values, we use the letter c for these connections (centres of the tuning curves). The activation of the cortical sheet, when activated with input r^{in} , is then given by

$$r_{ij} = e^{-\sum_k (c_{ijk} - r_k^{\text{in}})^2 / 2\sigma_r^2}, \quad (7.4)$$

where σ_r sets the width of the activated area. This activation function resembles the tuning curves discussed above, and networks with such activation functions are called radial-basis function (RBF) networks, as mentioned in Chapter 6.

The next simplification is with respect to the dynamics of the recurrent cortical sheet. We mentioned already that the dynamics of the model led to responses of the neural sheet in the form of a tuning curve as shown in Fig. 7.1, at least in the situation of single feature inputs. It was also shown that the site of this tuning curve is determined by competitive mechanisms. The strongest active area will inhibit other activities in the network. The dynamics in the cortical sheet are therefore approximated with a WTA procedure. The activation of the cortical sheet after competition is set to the Gaussian (eqn 7.4), around the *winning node*. Thus, only the active area around the winning node participates in Hebbian learning, as outlined above. The learning of the input projections is therefore intended to make the current preferred feature of the winning node closer to the training example. Since neighbouring nodes in

the cortical sheet are also activated, although less, these projections update in a similar way, but with less influence of the training example. Specifically, we use the learning rule:

$$\Delta c_{ijk} = \epsilon r_{ij}^*(r_i^{\text{in}} - c_{ijk}), \quad (7.5)$$

which makes the connections to nodes around the winning node (labelled with a ‘*’) more similar to the input example. The parameter ϵ specifies the learning rate.

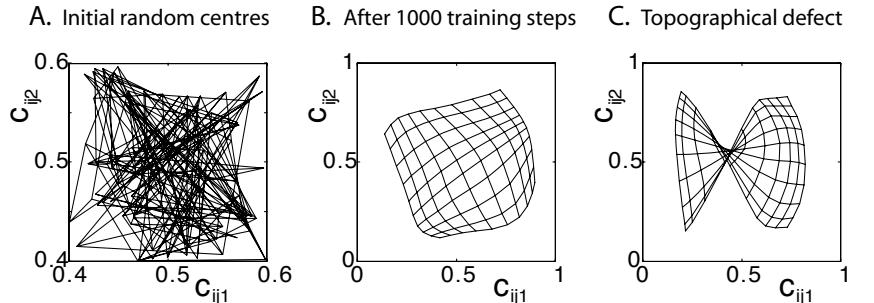


Fig. 7.4 Experiment with two-dimensional self-organizing feature maps. (A) Initial map with random weight values. (B) and (C) Two examples of the resulting feature map after 500 random training examples with different random initial conditions.

The development of the centres of the tuning curves, c_{ijk} , in such a network with a cortical layer of 10×10 nodes is demonstrated in Fig. 7.4. In these figures, the locations of centres in feature space are plotted for each node as an intersection of lines, where the lines connect neighbouring nodes in the cortical sheet. The simulations were started with random values, as shown in Fig. 7.4A, so that no orderly relationship was present. The network was then exposed to equally distributed random examples within a unit square. After several training examples, we get a relatively homogeneous representation of the feature space as shown in Fig. 7.4B. The network has learned to represent the feature values of the training set in a topographic map in which neighbouring nodes represent neighbouring feature values. The form of the maps can differ dramatically depending on the initial conditions and the precise sequence of examples. In Fig. 7.4C, we show another example of the same simulations with a different initial state. A twist, or fold, in the representation occurred as different parts of the grid started organizing independently. Such results are not uncommon in simulations. They are sometimes called topographic defects and are stable in the sense that it is nearly impossible to unfold these twists. However, these can be avoided by, for example, using initially large σ values.

Simulation

The program that was used to produce the simulation plots in Fig. 7.4 is included in folder **Chapter7** and shown in Table 7.1. As usual, we define some constants at the beginning. The constant **sig2** is a shorthand notation of the denominator in the exponential of eqn 7.4. It is wise to define such reoccurring calculations with constants at the beginning of the program to save valuable computer time later. We also define constant matrices **X** and **Y** with the help of the MATLAB function **meshgrid()**. **X** is a matrix that contains the ‘i’ index

of a node, and Y is a matrix that contains the ‘j’ index of a node. We need this later in the program to plot the mesh.

After choosing random centres in Lines 6 and 7, we set up an infinite loop. So the program needs to be terminated externally, for example by closing the figure window. Before training, and after every 100 training examples, programmed as a condition in Line 11 using the MATLAB `mod()` function, we plot the mesh by connecting neighbouring nodes. The MATLAB function `INT2Str()` converts an integer number into a string value which is used as the figure title in Line 14. In Line 15 we use the MATLAB command `waitForbuttonpress` to stop the program until a button (or mouse) is clicked. In this way, we can view intermediate results. Note that the `cursoheufiger r` must be in the figure window for this function to work. In Line 18 we calculate the activation of the cortical map. We did not include the variance term in this line as the activation calculated there is only used to find the winner in the next line. The activation of the winning node is then calculated correctly in Line 20, which is then used to update the centres according to eqn 7.5.

Table 7.1 Program som.m

```

1 %% Two dimensional self-organizing feature map al la Kohonen
2 clear; nn=10; lambda=0.2; sig=2; sig2=1/(2*sig^2);
3 [X,Y]=meshgrid(1:nn,1:nn); ntrial=0;
4
5 % Initial centres of preferred features:
6 c1=0.5-.1*(2*rand(nn)-1);
7 c2=0.5-.1*(2*rand(nn)-1);
8
9 %% training session
10 while(true)
11     if(mod(ntrial,100)==0) % Plot grid of feature centres
12         clf; hold on; axis square; axis([0 1 0 1]);
13         plot(c1,c2,'k'); plot(c1',c2','k');
14         tstring=[int2str(ntrial) ' examples']; title(tstring);
15         waitForbuttonpress;
16     end
17     r_in=[rand;rand];
18     r=exp(-(c1-r_in(1)).^2-(c2-r_in(2)).^2);
19     [rmax,x_winner]=max(max(r)); [rmax,y_winner]=max(max(r'));
20     r=exp(-((X-x_winner).^2+(Y-y_winner).^2)*sig2);
21     c1=c1+lambda*r.*((r_in(1)-c1);
22     c2=c2+lambda*r.*((r_in(2)-c2);
23     ntrial=ntrial+1;
24 end

```

7.2.3 Ongoing refinements of cortical maps

Every new training example changes the map, and one is often faced with what Grossberg coined the *plasticity–stability dilemma*, in that we want to stabilize a map while still being able to develop the map. In many technical applications, one starts the training with a large learning rate and gradually reduces this to small values. This is sometimes compared to developmental processes in mammals with critical periods in early postnatal periods. However, the brain also has the ability to adapt at high ages when given the opportunity. To demonstrate this we show another simulation in Fig. 7.5.

We started this network with perfectly organized centres (see graph for $t = 0$) to avoid topographic defects, since we are not interested here in the initial organization, but rather in the development of the maps in response to new exposures. The organized initial centres, as shown in the figure, were produced with:

```

for i=1:n_out;
  for j=1:n_out;
    c1(i,j)=i/n_out; c2(i,j)=j/n_out;
  end;
end;

```

The network is first exposed to random training vectors in the unit square, as before. The representation did not change much after 1000 updates; only some small distortions were introduced caused by the updates of the weight matrix with the training examples. After 1000 training examples we changed the training vectors presented to the network. The new training examples consisted of random values similar to the previous ones, and, in addition, training examples with components $1 \leq r_i^{\text{in}} < 2$, with random examples from each of the two quadrants. The simulation results after 100 and 1000 training episodes with these new training examples are shown in the last two graphs in Fig. 7.5. The topographic map branched out to reach representations of the additional feature values, although the representation is not as good as the representation of the initial training set even after some considerable time.

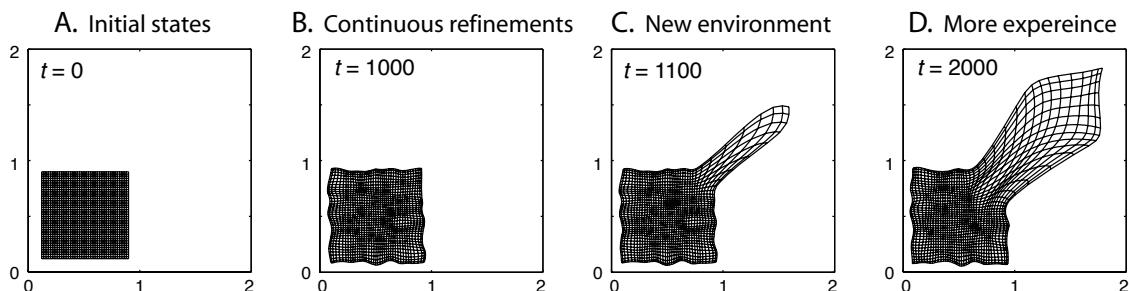


Fig. 7.5 Another example of a two-dimensional self-organizing feature map. In this example we trained the network on 1000 random training examples from the lower left quadrant. The next training examples were chosen randomly from the lower-left and upper-right quadrants. The parameter t specifies how many training examples have been presented to the network.

The simulations above demonstrate that SOM networks can learn to represent new domains of feature values, although the representation seems less fine-grained compared to the initial feature domain. The simulation results lead to some interesting conclusions if we speculate that the formation of cortical representation is driven by similar mechanisms. For example, it seems best to be exposed early in life to training examples from a broad feature space, including examples from all the features for which a fine-grained representation is desirable. This suggests that it is important to be exposed to different languages early in life to be able to achieve some high-level of sound discrimination ability in different languages. Another example is face discrimination from people with different ethnic features. This is often difficult until one is exposed much more frequently to people with such different ethnic backgrounds, such as moving to a different part of the world.

Based on the above model we can suggest some training strategies that might be useful when exploring new domains. It seems possible to create a ‘smart’ training set that can help to adapt to new representations much better than a random training set. It might thereby be advantageous to include at first only training examples that are close to the feature domain originally learned and then slowly to increase the difference of the feature values. The model suggests that we can move and expand the whole feature representation in this way much more smoothly compared to the case in which we adapt to some extreme cases first and have then only a limited number of nodes available to represent the new feature domain. Also, constraining examples from highly trained areas is beneficial, and maybe even essential, in efficient learning. It is possible that we can develop better therapeutical treatments in rehabilitation based on such ideas.

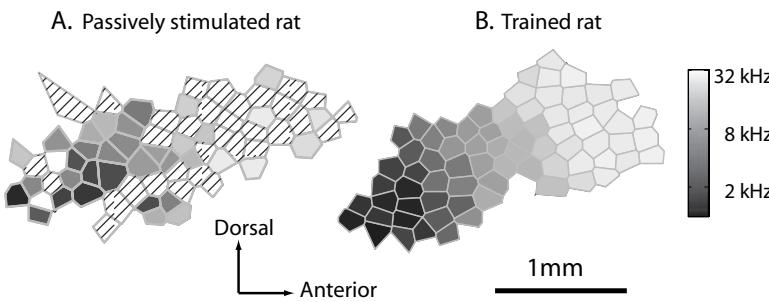


Fig. 7.6 (A) Frequency map in A1 of a rat that was developmentally degraded by being raised in noisy environments. The diagonally dashed areas contain neurons with poor frequency tuning. (B) Tonotopic map of an adult rat that recovered with training on a frequency discrimination task. [Data courtesy of Zhou and Merzenich, Proceedings of the National Academy of Science, USA 104: 15935-40 (2007).]

A nice demonstration of representational plasticity in adult mammals is shown in Fig. 7.6 from experiments by *Xiaoming Zhou* and *Michael Merzenich*. They raised rat pups in a noisy environment that severely impaired the development of tonotopy (orderly representations of tones) in the primary auditory cortex (A1), which lasted into adulthood. An example of such a map in an adult rat is shown in Fig. 7.6A. The diagonally dashed areas represent areas with neurons that showed pure frequency tuning. These rats were not able to recover normal tonotopic representation in A1 even though they were stimulated in adulthood with sounds of different frequencies. However, when the same sound patterns were used to train the rats in a discrimination task, which

they had to solve to get a food reward, the developmentally degraded rats were able to recover a normal tonotopic maps, as shown in Fig. 7.6B. This example demonstrates that some goal-directed learning must take place in the animal. Also, traditional SOM models, as discussed in this section, are entirely driven by input data in a bottom-up way, and it is unlikely that such models are sufficient to explain these experimental results. Thus, top-down processing, such as guidance of the development of early sensory maps by goals and attention, seems important, and we will return to the discussion of top-down processing in later chapters.

7.3 Dynamic neural field theory

We now turn to the other issue of cortical (and as we will see some subcortical) maps mentioned above, that of the dynamics of the cortical sheet for a given topographic organization. The continuous time dynamic equation has already been introduced for a discrete set of nodes in eqn 7.1. We stated in the example of orientation tuning curves that the number of nodes does not really matter as long as there is a sufficient number of nodes to represent the feature space with sufficient granularity. Also, the feature of ‘orientation’ is really a continuous variable, and many concepts that the brain has to learn are of continuous nature. Indeed, cognitive processes are mostly of a continuous nature in space and time, and the following model is very successful in explaining many behavioural findings. It is therefore natural, and mathematically very convenient, to write eqn 7.1 in a spatially continuous form,

$$\tau \frac{\partial \mathbf{u}(\mathbf{x}, t)}{\partial t} = -\mathbf{u}(\mathbf{x}, t) + \int_{\mathbf{y}} \mathbf{w}(\mathbf{x}, \mathbf{y}) \mathbf{r}(\mathbf{y}, t) d\mathbf{y} + I^{\text{ext}}(\mathbf{x}, t) \quad (7.6)$$

$$\mathbf{r}(\mathbf{x}, t) = g(\mathbf{u}(\mathbf{x}, t)), \quad (7.7)$$

with a general activation function $g(\mathbf{u})$. We replaced the external input to this map with a general form $I^{\text{ext}}(\mathbf{x}, t)$, as we will mainly study the dynamics of the *neural field* as topographically organized input. We used the term ‘neural field’ since the quantities $u(\mathbf{x}, t)$ and $\mathbf{r}(\mathbf{x}, t)$ depend on continuous spatial coordinates and are called *fields* in physics. We wrote the equations for arbitrary spatial dimensions, although we will use mostly a one-dimensional version of this model to simplify the discussions. In a one-dimensional model we can replace the vector quantities, $\mathbf{x}, \mathbf{y}, \mathbf{u}, \mathbf{r}$, with real valued variables, x, y, u, r . We also use periodic boundary conditions to minimize boundary effects. Thus the feature space is formally a ring in one dimension, and we frequently use feature values in $(0, 2\pi]$ for demonstration purposes. When implementing the model for computer simulations we need to discretize the model with:

$$x \rightarrow i\Delta x \quad (7.8)$$

$$\int dx \rightarrow \Delta x \sum. \quad (7.9)$$

We recover the discrete model used above, eqn 7.1, when we identify the correspondences $u(i\Delta x, t) = u_i(t)$ and $\Delta x = 1/N$. Notice the scale factor in the last equation, which can easily be forgotten when replacing the integral with a sum.

7.3.1 The centre-surround interaction kernel

We studied the training of input weights for a fixed interaction kernel, \mathbf{w} , in the previous section. We now discuss the formation of w with fixed topographic input. We will use a one-dimensional example in which we call the feature that is represented by the cortical sheet the ‘direction’. Each location of the cortical sheet represents a specific direction, but it also responds, to some extent, to neighbouring directions in the form of a Gaussian,

$$r(x - x^P) = e^{-(x-x^P)^2/2\sigma_r^2}, \quad (7.10)$$

where x^P is the *preferred direction* of a location in the cortical sheet. With periodic boundaries we should replace the distance between x and x^P with,

$$|x - x^P| \rightarrow \min(|x - x^P|, 2\pi - |x - x^P|). \quad (7.11)$$

We now use the continuous version of the basic Hebbian learning (eqn 4.10) to train an initial zero weight kernel on one example of every possible direction,

$$\mathbf{w}^E(x, y) = \int_0^{2\pi} r(x - x^P)r(y - x^P)dx^P. \quad (7.12)$$

This convolution integral is itself a Gaussian,

$$\mathbf{w}^E(|x - y|) = A_w e^{-(x-y)^2/4\sigma_r^2}, \quad (7.13)$$

with a width that is a factor $\sqrt{2}$ larger than the width of the training patterns, and a scaling factor, A_w , that depends on the strength of the training pattern and a learning rate. This weight kernel is only a function of the distance between nodes and is always positive (excitatory), hence the label E. However, we also need inhibition in the network. We therefore consider a pool of inhibitory nodes which are activated by excitatory activations in the network and inhibit in turn the the neural field globally. This can be incorporated in the network by shifting the excitatory kernel by an amount $A_w C$, where the constant C describes the relative amount of inhibition. The final weight kernel can be written as continuous version of eqn 7.3,

$$\mathbf{w}(|x - y|) = A_w \left(e^{-(x-y)^2/4\sigma_r^2} - C \right). \quad (7.14)$$

In this example we have derived the Gaussian weight kernel from training a recurrent network systematically on training examples with Gaussian shape. However, most of the behaviour discussed in the following holds also for models that are trained on different pattern shapes, as long as the resulting weight functions depend only on the distance (shift invariance), and have short-distance excitation and long-distance inhibition. For example, the commonly used *Mexican-hat* function, which can be calculated from the difference of two Gaussians, is illustrated in Fig. 7.7. This function is very similar to the shifted Gaussian above when the extent of the Mexican hat is on the order of the periodic cortical maps discussed here.

Various physiological examples suggest an effective interaction structure, with short-distance excitation and long-distance inhibition, in the brain. Such

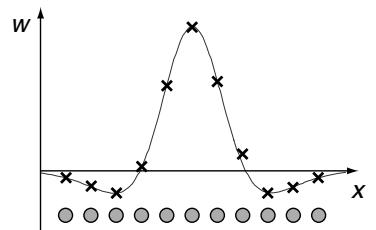


Fig. 7.7 Illustration of a Mexican-hat function. The crosses indicates the values that are used in a discrete version of a dynamic neural field model with nodes as indicated at the bottom.

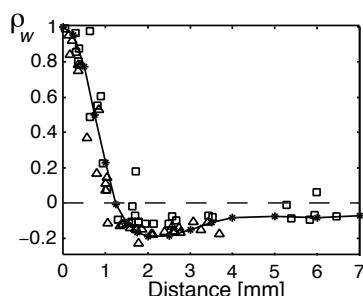


Fig. 7.8 Data from cell recordings in the superior colliculus in a monkey, which indicate the interaction strength ρ_w between cells in this midbrain structure. The solid line displays the corresponding measurement from simulations of a CANN model of this brain structure. [From Trappenberg *et al.*, *J. Cogn. Neurosci.* 13: 256–71 (2001).]

evidence influenced, for example, Cowan and Wilson to study neocortical models with such interaction structures in 1973. Another example was found in the intermediate layer of the superior colliculus, a midbrain area that is an important integration stage for many cortical and subcortical pathways that guide the direction of gaze. An example of the interaction structures within the superior colliculus from cell recordings in monkeys is shown in Fig. 7.8. In this figure we plotted the results for two sample neurons. The figure shows the influence of activity in other parts of the colliculus on the activity of each neuron. This influence has the characteristics of short-distance excitation and long-distance inhibition, and a model of the superior colliculus based on these characteristics was able to reproduce many behavioural findings for the variations in the time required to initiate a fast eye movement as a function of various experimental conditions.

7.3.2 Asymptotic states and the dynamics of neural fields

The cortical sheet is formally a dynamic system (recurrent network), and we can study the trajectories and asymptotic states in the state space of the network, as will be done more formally below. Here, we summarize some principal regimes of such models with local *cooperation* and global *competition*, which are important in the following discussions. These different regimes depend on the level of inhibition C , and can be categorized as follows.

- (1) **Growing activity:** When the inhibition is weak compared to the excitation between nearby nodes, so that the model is dominated by excitation, the dynamics of the model are governed by positive feedback. The whole map will eventually become active in this regime, which is therefore undesirable for brain processes.
- (2) **Decaying activity:** When the inhibition is strong compared to the excitation, then the dynamics of the model are dominated by negative feedback. The neurons will respond to sufficiently strong input, but the activity of the map decays after removal of external input. Nevertheless, this regime is interesting from an information processing point of view since it can facilitate competition between external inputs.
- (3) **Memory activity:** In an intermediate range of inhibition, which we will specify further below, an active area can be stable in the map, even when an external input is removed. This regime can therefore represent memories of feature values through ongoing activity in the network.

The regimes can easily be explored numerically with the program detailed below. An example of such a simulation is shown in Fig. 7.9 where we plotted the firing rates of the nodes in a network of 100 nodes on a grey-scale during the evolution of the system in time. All nodes are initialized to have medium firing rates, and a strong external stimulus to nodes number 40–50 was applied at time $t = 0$. The stimulated nodes, as well as some neighbouring nodes to the stimulated site, respond with an increase in their firing rate due to the excitatory weights to nearby nodes. Nodes far from the stimulated site decrease their firing rates due to inhibition. As mentioned above, we call the collection

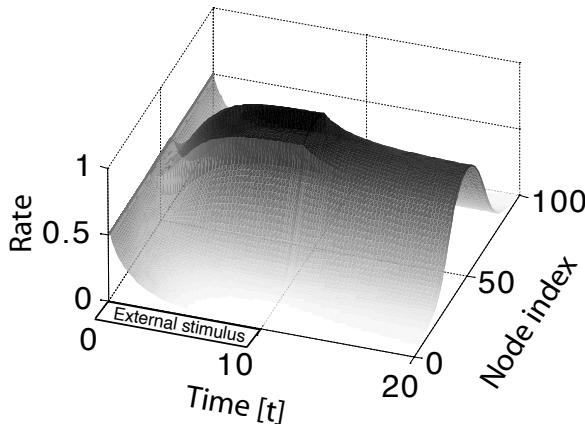


Fig. 7.9 Time evolution of the firing rates in a dynamic neural field model with 100 nodes. Equal external inputs to nodes 40–60 were applied at $t = 0\tau$. This external input was removed at $t = 10\tau$ (parameters of the models as in listing Table 7.2).

of nodes that are active compared to their background rate an activity packet, but in some of the literature it is also called a *bubble*, or *bump*.

The external stimulus was removed at $t = 10\tau$. The overall firing rates in the network decreased slightly following this removal of the external stimulus, and the activity packet became lower and a bit broader. However, the most interesting effect is that a group of neighbouring nodes with the same centre as the external stimulus, stayed active asymptotically. The dynamics of the cortical sheet is therefore able to memorize a feature with ongoing activity in the cortical map. We will argue below that this corresponds to a form of *working memory* as seen in physiological data. The activity packet can be stabilized at any location in the network depending on an initial external stimulus. The attractor is therefore a continuous manifold in the neural field limit, and the *dynamic neural field* (DNF) model is sometimes called a *continuous attractor neural network* (CANN). This is a special case of more general *attractor neural networks* (ANNs) discussed in Chapter 8. A new activity packet can be established at a new location by applying another external stimulus centred around a different node. The externally activated nodes, however, still receive inhibitory activity from the other activity packet in the network, and the external input has to be of sufficient strength and duration to ensure that the new activity packet becomes dominant in the network.

The rate profile at time $t = 20\tau$ is shown as a solid line in Fig. 7.10. In these simulations we used an inhibition constant of $C = 0.5$ and a weight strength of $A_w = 4$. Using a larger weight strength, $A_w = 10$, results in a more rigid activity packet with sharper boundaries and rates of the nodes deep inside the activity packet nearly reaching the rate limit. Similar effects can be seen when using larger gains in the sigmoidal activation function or when using less inhibition. Using threshold functions as activation functions results in a box-shaped rate profile. We will utilize this finding in the following analysis. The active area decays with large inhibition constants, such as with $C = 0.7$ in the example. However, the decay process can take some time, so that a trace of the evoked area can still be seen at $t = 20\tau$, as shown with the dotted line in Fig. 7.10.

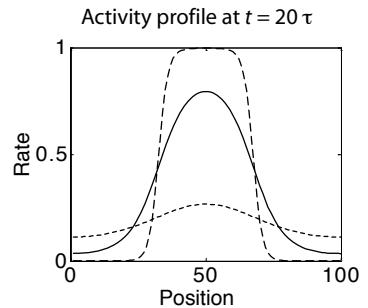


Fig. 7.10 The solid line represents the firing rate profile of the simulation shown in Fig. 7.9 at $t = 20\tau$. The dashed line shows similar results with increased weights $A_w = 10$. The dotted line shows results for a decaying activity packet with increased inhibition ($C = 0.7$).

Simulation

An implementation of the DNF model, used for the simulations shown in Figs 7.9 and 7.10, is given in Table 7.2. After the usual clearing of the workspace and figure windows in Line 2, some parameters are defined in Line 3. This includes the number of nodes, which is set to `nn=100` in this example. It is important that our results do not depend on this number, and it is good to check whether simulation results are unchanged when using a different number (please try). The number of nodes will only change the resolution in feature space, `dx`. The variable `sig` influences the standard deviation of the weight matrix, and the constant `C` sets the inhibition.

Table 7.2 Program dnf.m

```

1 %% Dynamic Neural Field Model (1D)
2 clear; clf; hold on;
3 nn = 100; dx=2*pi/nn; sig = 2*pi/10; C=0.5;
4
5 %% Training weight matrix
6 for loc=1:nn;
7     i=(1:nn)'; dis= min(abs(i-loc),nn-abs(i-loc));
8     pat(:,loc)=exp(-(dis*dx).^2/(2*sig^2));
9 end
10 w=pat*pat'; w=w/w(1,1); w=4*(w-C);
11 %% Update with localised input
12 tall = []; rall = [];
13 I_ext=zeros(nn,1); I_ext(nn/2-floor(nn/10):nn/2+floor(nn/10))=1;
14 [t,u]=ode45('rnn_ode',[0 10],zeros(1,nn),[],nn,dx,w,I_ext);
15 r=1./(1+exp(-u)); tall=[tall;t]; rall=[rall;r];
16 %% Update without input
17 I_ext=zeros(nn,1);
18 [t,u]=ode45('rnn_ode',[10 20],u(size(u,1),:),[],nn,dx,w,I_ext);
19 r=1./(1+exp(-u)); tall=[tall;t]; rall=[rall;r];
20 %% Plotting results
21 surf(tall',1:nn,rall','linestyle','none'); view(0,90);

```

It is common in the application of the DNF model to set the components explicitly with a function such as the difference of two Gaussians. However, in this example we chose to train the excitatory part of the weight matrix with Hebbian training on Gaussian patterns. The pattern matrix `pat` contains `nn` patterns, each is a Gaussian centred around one of the nodes in the network (Lines 6–9). In Line 10 includes the Hebbian learning rule in matrix form (see eqn 4.21 and its following comments) and is normalized to the maximal element (each diagonal element is equal and maximal). The resulting value is then shifted by the inhibition constant and again scales up with a factor of 4. Such a scaling factor can be related to the learning rate or other forms of normalization.

Table 7.3 Program rnn_ode.m

```

1 function udot=rnn_ode(t,u,flag,nn,dx,w,I_ext)
2 % odefile for recurrent network
3 tau_inv = 1.;          % inverse of membrane time constant
4 r=1./(1+exp(-u));
5 sum=w*r*dx;
6 udot=tau_inv*(-u+sum+I_ext);
7 return

```

In Line 12 we start the experiment by defining two empty arrays to record the resulting time values and corresponding rate values of the nodes. In Line 13, we define an external input with non-zero elements for the 21 nodes around the central node. The update of the neural field until $t = 10$ is done with the MATLAB function `ode45()`, which implements a higher-order, adaptive, Runge–Kutta algorithm. The file specifying the dynamical equations is shown in Table 7.3. This function returns an array with the time at which the functions is evaluated and the corresponding internal state values of the nodes. The rate values are then calculated with a sigmoid activation function and recorded in Line 15. The external input is then set to zero in Line 17, and the neural field is updated with this zero input until $t = 20$. The results of the simulations are shown in a surface plot using the MATLAB function `surf()`. We removed the grid lines to improve visibility of the colour plot better visible. For plotting the grey-scale image in Fig. 7.9 we used the command `colormap(1-gray)` after running the simulation.

7.3.3 Examples of competitive representations in the brain

The tuning curves discussed in Section 7.1 only show maximal responses of a neuron during each stimulus presentation. We turn now to some other examples from different brain areas to demonstrate that such mechanisms capture fundamental aspects of brain processing.

Higher cortical areas represent increasingly complex features beyond orientation selectivity of V1 neurons, but we can still think of these representations in terms of neural fields. The only difference is that the represented feature value (orientation in V1) should be replaced by another feature value to which the other cortical area responds.² For example, the inferior-temporal (IT) cortex in monkeys is quite selective in responding to specific complex objects. Some data from recordings in this area by Chelazzi and colleagues are shown Fig. 7.11A. In their experiment they showed different objects to monkeys and selected objects to which the recorded IT cell responded strongly (*good* objects) and weakly (*bad* objects). In the illustration of Fig. 7.11A, the average firing rate of an IT cell to a good stimulus is shown as solid line. The period when the cue stimulus was presented is indicated by the grey bar in this figure.

²We do not want to leave the impression that specific cortical areas process only the specified features. Indeed, these features are somewhat biased by experimental conditions. Even neurons in V1 respond to other features. The ‘true features’ processed in a cortical area might be a complicated mix of features defined by the experimenter. However, the arguments discussed in this section do not crucially depend on the feature choice as long as the neurons show significant modulations with these feature values.

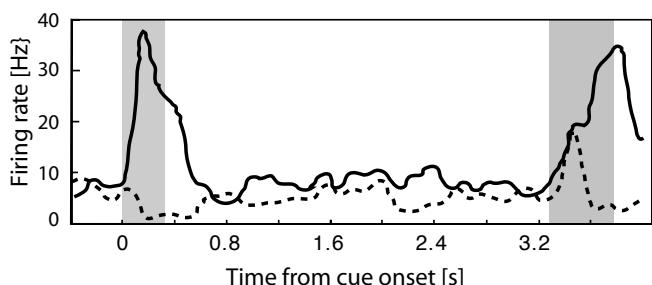
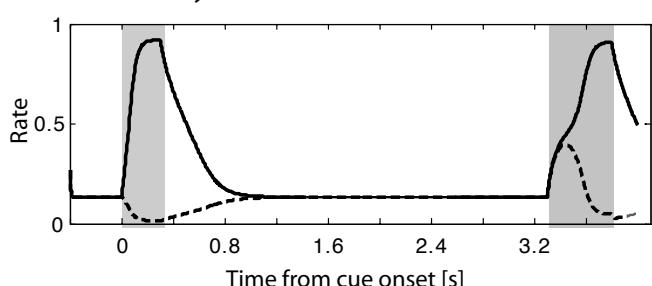
A. Experimental data from IT recordings

Fig. 7.11 (A) Example of responses of IT neurons from experiments by Chelazzi, Miller, Duncan, and Desimone, *Nature* 363: 345–7 (1993). Response to a ‘good’ object is shown as solid line, the response to a ‘bad’ object as a dotted line. (B) Simulations of the DNF model with moderately-strong inhibition. The solid line is a recording from the central ‘good’ location, the dotted line from the ‘bad’ location [for details see Trappenberg, in *Computational modelling in behavioural neuroscience*, Heinke and Mavritsaki (eds), Psychology Press (2009)].

B. Dynamic neural field simulations

The response to a ‘bad’ object is illustrated in the figure with a dashed line. Instead of increasing the rate during the cue presentation, this neuron seems to respond with a firing rate below the usual background rate. At a later time, the monkey was shown both objects, and asked to select the object that was used for cueing. The IT neuron responds initially in both conditions, but the response is quite different at later stages.

Several aspects of the experimental data are captured by simulations of the DNF model as shown in Fig. 7.11B. The solid line represents the activity of a node within the response bubble of the neural field. In these simulations, a fairly large inhibition constant was used so that the activity declines after the external stimulus is removed. The activity of the dashed line corresponds to the activity of a node outside the activity bubble. The activity of this node is weakened as a result of lateral inhibition during the stimulus presentations. Also, when both objects are used as input, activity at both locations will initially rise. Only later will the slightly larger activity package, such as when modelled by additional support from working memory, dominate the dynamics and actively inhibit the other activity packet.

Another important issue is that of demonstrating physiological working memory by ongoing firing in the brain. The maintenance of neural activity over a delay period has been observed experimentally. An example of such an experiment by Funahashi, Bruce, and Goldman-Rakic is shown in Fig. 7.12. These researchers trained a monkey to maintain its eyes on a central fixation spot until a ‘go’ signal, such as a tone, indicated that it should move the eyes and focus on one of several possible targets peripheral to the fixation spot. The

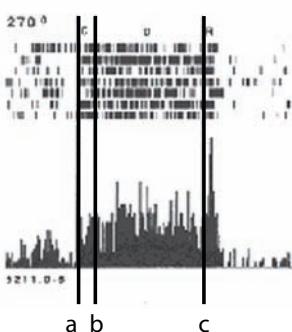


Fig. 7.12 Demonstration of physiological working memory through the maintenance of delay activity in physiological experiments. A target was presented between times *a* and *b*, and the location of the target had to be remembered until an action was required at time *c* [detail from Funahashi, Bruce, and Goldman-Rakic, *Journal of Neurophysiology* 61: 331–49 (1989)].

choice of the target to which the eyes should be moved in each trial was indicated by a short flash, between the first two vertical bars in the figure, but the subject was not allowed to move its eyes until the ‘go’ signal indicated by the third vertical bar in the figure. Thus, the target location for each trial had to be remembered during the delay period. The experimenters recorded from neurons in the dorsolateral prefrontal cortex (area 46) and found neurons that were active during the delay period. These neurons were sensitive to the particular target direction (the shown neuron only responded when the target was at 270 degrees), and this neuron could therefore indicate by its delayed activity to which target the eye should be directed after the delay period. It is possible that such ongoing activity of neurons is supported by intracellular mechanisms such as activation of ion channels with high reversal potentials. However, a more common explanation is that such working memory activity is sustained through lateral reverberating neural activity as captured by the DNF model. This model is attractive for several reasons, such as the ability to simulate interactions with other memories, as discussed later.

The final example illustrates the representations of space in the *archicortex*. Some neurons in the *hippocampus* of rats fire in relation to specific locations within a maze in which the rat is freely moving. In some experiments, the activity of many neurons has been recorded while a rat could freely run in a maze. When the firing rates of the different neurons are plotted on a map reflecting their physical location in the hippocampus, the resulting firing pattern looks like a randomly distributed code. A specific topography of neurons within the hippocampal tissue with respect to their maximal response to a particular place has not been found. However, if the plot is rearranged so that neurons that fire maximally in response to adjacent locations are plotted adjacent to each other, then a firing profile like the one shown in Fig. 7.13 can be seen. This is an example of a two-dimensional activity packet.

Since hippocampal place cells do not display a regularity in the physical space of the brain tissue, a hardwired (for example, genetically coded) connectivity pattern is likely not employed in this structure. However, we saw the regularities in the above models when organizing the nodes according to the feature values they represented. Discovering this organization in the functional map would have been difficult otherwise. This is demonstrated in Fig. 7.14 with a one-dimensional example, where we have labelled the nodes and indicated the relative strengths of weights between the different nodes with lines of different widths, corresponding to the relative strengths of the connections. Before learning, illustrated in Fig. 7.14A, we can assume that all nodes have equal weights relative to each other. The dimensionality of this model can be regarded as high when we take the number of neighbours, the number of nodes to which each node is relatively strongly connected, as a measure. Unlike the previous model, we have assigned each node randomly to a preferred direction where it fires maximally. After training we therefore get weights in the ‘physical’ space of the nodes, as indicated in Fig. 7.14B, that look rather random. The order in the connectivity only becomes apparent when we finally reorder the nodes so that strongly connected nodes are adjacent to each other. After doing so (Fig. 7.14C), we see that it has a one-dimensional structure which reflects the one-dimensional structure of the feature space that was used for

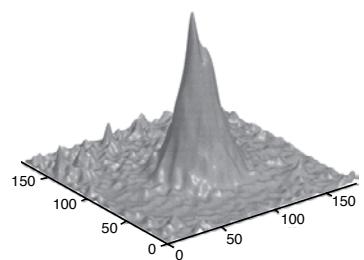


Fig. 7.13 Neuronal response from many hippocampal neurons in a rodent that responded to the subject’s location (places) in a maze. The figure shows the firing rates of the neurons in response to a particular place, whereby the neurons were placed in the figure so that neurons with similar response properties were placed adjacent to each other [Samsonowich and McNaughton, *Journal of Neuroscience* 17: 5900–20 (1997)].

training this network.

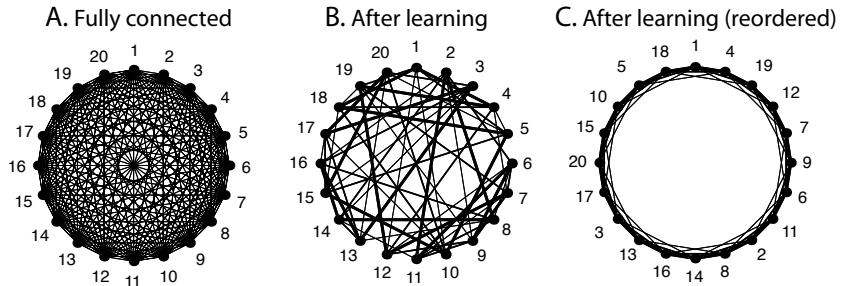


Fig. 7.14 A recurrent associative attractor network model, where the nodes have been arbitrarily placed in the physical space on a circle. The relative connection strength between the nodes is indicated by the thickness of the lines between the nodes. Each node responds during learning with a Gaussian firing profile around the stimulus that excites the node maximally. Each node is assigned a centre of the receptive field randomly from a pool of centres covering the periodic training domain. (A) Before training, all nodes have the same relative weights between them. (B) After training, the relative weight structure has changed, with a few strong connections and some weaker connections. (C) The regularities of the interactions can be revealed by reordering the nodes so that nodes with the strongest connections are adjacent to each other.

We reduced the dimensionality of the initial network to a one-dimensional connectivity pattern. If we had trained the initial network with examples from a two-dimensional feature space we would have produced a two-dimensional structure in the weight matrix, although this might only be visible after rearranging the nodes accordingly. The rearrangement of nodes we used is equivalent to the rearrangement of hippocampal neurons discussed above and explains why this had to be done to see the ‘regularity’ in the firing pattern. From this we can see that the network self-organizes to reflect the dimensionality of the feature space. The network ‘discovers’ the dimensionality of the underlying problem from activity dependent co-activation of neurons, which is the basic feature of Hebbian learning, as discussed in Chapter 4.

7.3.4 Formal analysis of attractor states ◇

The dynamic neural field model in the form presented here was introduced by *Sun-ichi Amari* in 1977, when he also studied the solutions of such models in detail. We follow here some of the basic ideas in his analysis. For simplicity, we assume a threshold activation function, $g(x) = \theta(x)$, which makes the firing rates within the activity packet equal to one while setting the firing rates outside the activity packet to zero. As we will see, this choice is convenient. The results with a threshold activation function are a good approximation for networks with smoother activation functions, because the activity packets can be sharp even with a smooth sigmoidal activation function. The threshold activation function is useful because the stationary state ($\partial u / \partial t = 0$) of the dynamic eqn

7.6, without external input ($I_{\text{ext}} = 0$), can then be written as:

$$u(x) = \int_{x_1}^{x_2} w(x, y) dy, \quad (7.15)$$

where x_1 and x_2 are the positions of the boundaries of the activity packet. This must be true for all x , including the boundaries for which $u(x_1) = u(x_2) = 0$, so that the following equation must also hold,

$$\int_{x_1}^{x_2} w(x_1, y) dy = 0. \quad (7.16)$$

For the Gaussian weight kernel, eqn 7.14, the above equation can be solved with the error function (see eqn C.13),

$$\sqrt{\pi} \sigma_r \text{erf}\left(\frac{x_2 - x_1}{2\sigma_r}\right) = C(x_2 - x_1). \quad (7.17)$$

This equation can be solved numerically and is illustrated in Fig. 7.15 as a dotted line. Corresponding simulations of the DNF model are shown in Fig. 7.15 as a solid line. The analytical solution describes the stable regime quite well, but deviates somewhat from numerical solutions in the transition range between the different regimes.

What about the stability of the activity packet with respect to movements? To answer this question we calculate the velocity of the boundaries. A movement of the activity packet without external input can only result from forces generated by the shape of the activity packet itself. The force is then proportional to the gradient of the activity packet, and the velocity is

$$\frac{dx}{dt} = -\Delta x \frac{du}{dt}, \quad (7.18)$$

where we took a possible explicit time dependence of the activity packet into account. To get the velocity of the boundaries we have to substitute $x = x_1$ or $x = x_2$ into this equation. We can then substitute eqn 7.6 into this equation to get a formula for the velocity of the boundary, or similarly for the centre of the activity packet,

$$x_c(t) = \frac{1}{2}(x_1(t) + x_2(t)), \quad (7.19)$$

given by

$$\frac{dx_c}{dt} = -\frac{1}{2\tau\Delta x_1} \int_{x_1}^{x_2} w(x_1, y) dy - \frac{1}{2\tau\Delta x_2} \int_{x_1}^{x_2} w(x_2, y) dy. \quad (7.20)$$

The expression on the right-hand side of the equation is zero when the weighting function is symmetrical and shift-invariant, and the gradients of the activity packet at the boundaries are the same except for their sign. These conditions are illustrated in Fig. 7.16. For the Gaussian weighting function, which is symmetrical and shift-invariant, we see that the integral from x_1 to x_2 of the Gaussian centred around x_1 is the same as that for the Gaussian centred around x_2 . Also, once we have a symmetrical activity packet, it is clear that the gradients at the boundaries are equal except for their sign, as illustrated in

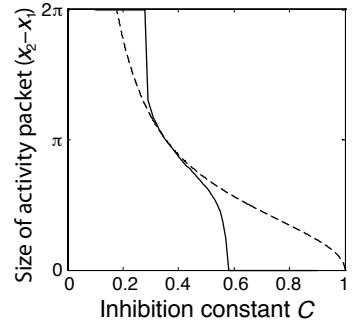


Fig. 7.15 Size of the activity package as function of the inhibition constant C . The dotted line corresponds to the solutions of eqn 7.17, whereas the solid line shows results from simulations.

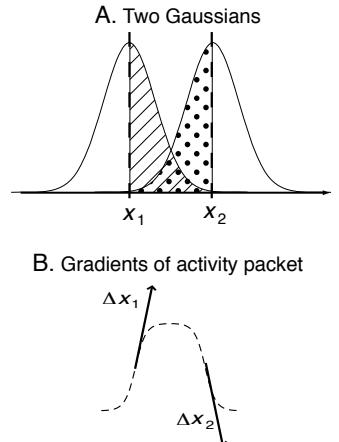


Fig. 7.16 (A) Two Gaussian bell curves centred around two different values x_1 and x_2 . The size of the striped and dotted areas are the same due to the symmetry of the bell curve. The integrals from x_1 to x_2 over the two different curves are therefore the same. This is not true if the two curves are not symmetrical and have different shapes. (B) The dashed line outlines the shape of an activity packet from a simulation. The symmetry of this activity packet makes the gradients of boundaries equal except for a sign.

Fig. 7.16B. The velocity of the centre of the activity packet for a symmetrical weighting function is therefore zero ($dx_c/dt = 0$), and the activity packet stays centred around the location where it was initialized.

We have formally derived that the activity packet in models with shift-invariant and symmetrical weight matrices is stable. Eqn 7.20 also tells us when the activity packet is not stable and can hence drift. For example, the velocity of the centre of the activity packet is not equal to zero when the shift invariance of the weight matrix is broken. The weight matrix generated by Hebbian learning on random patterns, as will be discussed in Chapter 8, is not shift-invariant. Such associative memory networks therefore drift away from initial conditions toward a point attractor, as will be discussed in the next chapter.

Another important factor is noise in the system, which we have always to take into account when discussing brain mechanisms. Noise breaks the symmetry as well as the shift invariance of the weighting functions when the noise is independent for each component of the weight. The activity packet in such networks therefore drifts to some points where the shifting forces compensate each other. This leads to a *clustering* of end states. An example is shown in Fig. 7.17A. Each curve in the plot corresponds to the time evolution of the centre of gravity of the activity packet after the network was initialized with an activity packet centred around a different node in the network. This drift caused by noise has some interesting consequences for operations in the brain. For example, the drift makes an accurate representation of locations over a long time impossible. Indeed, experiments show that the sense of direction in the dark diminishes after some period of time. Note that the drift of the activity packet due to noise decreases with the size of the system because the noise components can average out when the weighting function is the result of the interaction of many noisy nodes.

Another source of asymmetries in the weighting function is irregular or partial training of the network. In the previous examples, we always trained the networks for the same amount of time, with activity packets centred at each node in the network. This not only requires long training sessions, but is also unrealistic in biological terms as the subject would have to explore a new environment in a very regular manner. The effects of partial training are illustrated with the simulations shown in Fig. 7.17B–D. There, we have trained the network with activity packets centred around only 10 different nodes in the network. A partial view of the resulting weight matrix is shown in Fig. 7.17B. The values of the weight matrix are largest at the locations that were used for training. The centre of the activity packet with the resulting weight matrix, not including any noise, also shows a drift for initial states around the trained locations (Fig. 7.17C).³ This is a point attractor network, discussed more in the next chapter.

The drift in the activity packet can be stabilized by a small increase in the excitability of neurons once they have been recently activated. This corresponds to a voltage-dependent non-linearity in the postsynaptic neuron that can help to maintain the firing in a recurrent network. Such voltage-dependent non-linearities could, for example, be implemented by NMDA receptors. We have already mentioned in Chapter 2 that NMDA receptors are blocked by

³The locations precisely in-between the trained locations are also stable, but only marginally so. A small deviation would result in a drift to the trained locations.

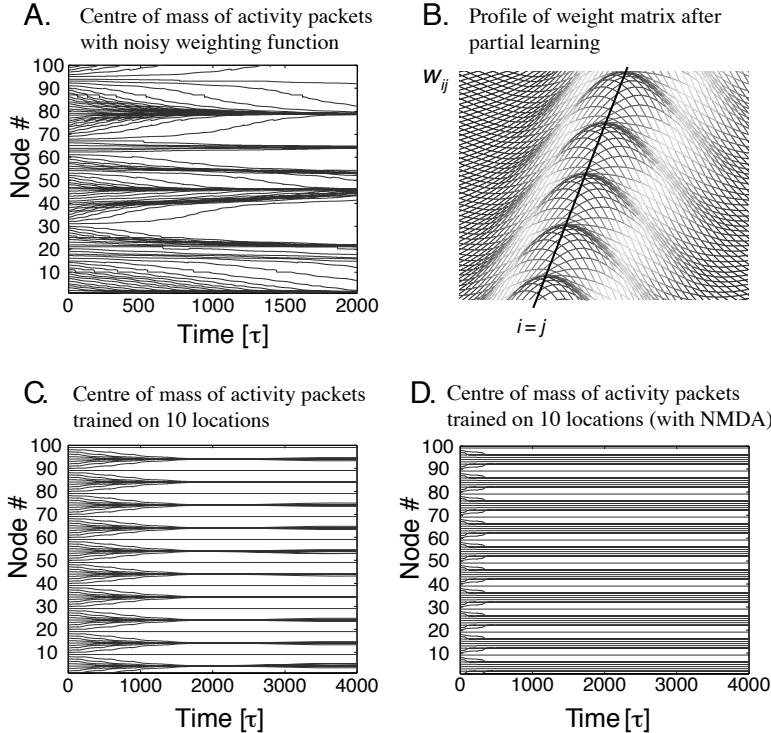


Fig. 7.17 (A) Simulations with noisy weight matrix: Time evolution of the centre of gravity of activity packets in a DNF model with 100 nodes. The model was trained with activity packets on all possible locations. Each component of the resulting weight matrix was then convoluted with some noise. (B) Irregular or partial learning: Partial view of the weight matrix resulting from training the network with activity packets on only a few locations. (C) Time evolution of the centre of gravity of activity packets in DNF model with 100 nodes after training the network on only 10 different locations. (D) ‘NMDA’-stabilization. The network trained on the 10 locations was augmented with a stabilization mechanism that reduces the firing threshold of active neurons.

magnesium ions when neurons are at rest. This blockade is removed after an increase in the membrane potential. Thus, we can excite the neuron the next time much more easily, if the time necessary to block the channel again is long relative to the time of the next incoming spike.

We can simulate such a voltage-dependent non-linearity with a relatively long time constant by altering the threshold in the activation function of the model neurons. In the simulations shown in Fig. 7.17D we have changed the threshold value α from the value $\alpha = 0$ used in all previous simulations to a lower value of $\alpha = 10$ for neurons that exceeded 50% of their maximal firing rates. The value was reset to $\alpha = 0$ when the neurons fell below this 50% threshold. This change in the simulations was sufficient to increase the number of attractors (see Fig. 7.17D), with most of the states close to the trained locations being stable. Only states far from the trained location drifted initially to the closest attractor state. An increase of the voltage-dependent non-linearity would make more states stable. However, we do not want to make this mechanism too dominant, as we would otherwise lose the competitive nature of the network, which is relevant for most applications of these network models. There is, so far, no direct experimental verification of this proposal, but a direct verification might be possible with well-directed blocking of NMDA receptors. The model predicts that drift in the activity packets should increase, corresponding to a quick confusion in the sense of direction.

7.4 ‘Path’ integration and the Hebbian trace rule ◇

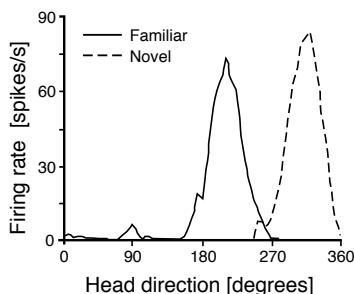


Fig. 7.18 Experimental response of a neuron in the subiculum of a rodent when the rodent is heading in different directions [redrawn from Golob and Taube, *Journal of Neuroscience* 19: 7198–211 (1999)].

Humans usually have to a certain degree a sense of direction. This suggests that we must have some form of spatial representation in our brain. The sense of direction can be tested with a simple experiment. If a subject is placed in a rotating chair with closed eyes while someone rotates the subject a certain amount, it can be shown that the subject’s guess of the new direction is quite accurate. This demonstrates two important issues: first, we have to have a representation of body, or head, direction and, second, we have to have a mechanism to update this information without visual cues.

We already saw that special information is represented with place fields in the hippocampus of rats, and other spacial information, such as *head directions*, can also be found there and in other proximate areas. For example, in the subiculum of rodents it was found that firing of neurons represents the direction in which the rodent was heading. An example is shown in Fig. 7.18 from recording activities of a cell when the rodent was rotated in different directions. The solid line represents the response property of this neuron in a familiar maze. The neuron fires maximally for one particular direction and fires with lower firing rates to directions around the preferred direction of this neuron. The shown curve is a tuning curve of *head direction* representations in the subiculum. The dashed line represents the new head properties of the same neuron when the rodent is placed in a new, unfamiliar, maze. The new response properties will normally be similar to the previous one, that is, head direction cells try to maintain approximately their response properties to specific head directions. However, the results shown were produced in experiments with a rodent that had cortical lesions that weakened the ability to maintain the response properties after the rodent was transferred into a new environment. It was shown that the head-direction neurons continue to fire in the dark, which is another example of physiological working memory through ongoing neural activity after removal of the stimulus. We discuss, in this section, how the representation of the current head direction can be updated in DNF models.

7.4.1 Path integration with asymmetrical weight kernels

One way of updating the state represented by a DNF layer is to apply an external stimulus to a new location. This is, of course, only possible if we know the absolute value for the new location, which should be represented explicitly so that we can apply an external stimulus at the corresponding location in the network. However, a subject might not have such an absolute value available, for example, when rotating a subject with closed eyes. This is like driving a blindfolded person around in a city and, after some time, asking where we are. To solve this problem we have to ‘calculate’ the new position from the old position and the changes we made (velocity information including rotation and forward speed) over this time period. This ‘calculation’ is called *path integration*, and we will adopt this terminology for the generic situation of calculating a new state representation from an initial state representation plus signals that indicate the change of the state.

We saw in the last section that asymmetries in the weight kernel in DNF models lead to a movement of the activity packet. A proposal for solving the path integration problem involves using such asymmetries in a systematic way. To do this we have to find a way to relate the strength of the asymmetry to the velocity of the movement. A velocity signal can be generated by the subject itself, and we call such information *idiothetic cues*, where ‘idiothetic’ means self-generated. Examples are inputs from the *vestibular system* in mammals, which can generate signals indicating the rotation of the head and *proprioceptive* feedback from muscles that signal the change in their position. Such signals will be the input to our following models, where we will again concentrate on head direction as an example.

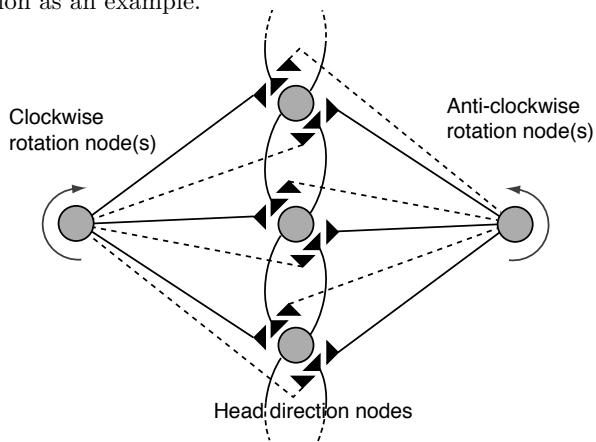


Fig. 7.19 Model for path integration which shows a few head direction cells with collateral connections. The rotation nodes represent collections of neurons that signal rotation velocities proportional to their activity. The afferents of the rotation cells can modulate the collateral connections within the head direction network, which is symbolized with synapses close to the synapses of the collateral connections. Each rotation cell can synapse on to each synapse in the head direction network. The separation of the connections, as indicated by the solid and dashed lines, is self-organized during learning.

A proposal as to how idiothetic velocity signals can be used in DNF models of head direction representations to update the system is shown in Fig. 7.19. For simplicity, we have only shown three nodes of the recurrent network representing the head direction of a subject, and have only included collateral connections to the neighbouring nodes. In addition to these head direction nodes we included two other nodes (which can also represent a collection of neurons), which we call *rotation nodes*. We assume for simplicity that the firing rate of these nodes is directly proportional to the velocity of the head movement. The principal idea behind the model is that these rotation nodes can *modulate* the strength of the collateral connections between DNF nodes. This modulatory influence makes the effective weight kernel within the attractor network in one direction stronger than in the other direction, thus enabling the activity packet to move in a particular direction with a speed that is determined by the firing rate of the rotation nodes.

The effect of rotation node activity on the attractor network has to be modulatory (that is, multiplicative) as opposed to additive because the latter case would produce only an equal external input to all nodes that could not shift the activity packet. The modulatory effect can, for example, be implemented with sigma-pi nodes as introduced in Section 3.5. This, in turn, has several possible implementations on the neuronal level. For the following discussion, it is sufficient to think about two physically close synaptic terminals that can interact to produce such non-linear modulation affects.

7.4.2 Self-organization of a rotation network

The major problem we have to solve to make this model biologically realistic is to find a way to self-organize the network. If we simply assume that rotation nodes modulate all synapses equally we cannot move the activity packet. Instead, the network has to learn that the firing in a rotation node that indicates, for example, clockwise rotations should modulate only the appropriate ‘clockwise synapses’ in the network. Thus, the network has to learn that synapses have strong weights only in response to the appropriate weights in the recurrent network, as indicated by solid lines in Fig. 7.19. The influence of the opposite synapses, indicated by dashed lines, has to at least be weaker. To achieve this, we need a learning rule that can associate the recent movement of the activity packet with the firing of the appropriate rotation node. We therefore need to have a trace, or ‘short-term memory’, in the nodes, which is related to the recent movement of the activity packet. An example of such a *trace term* (indicated by a bar over the firing rate) is,

$$\bar{r}_i(t+1) = (1 - \eta)\bar{r}_i(t) + \eta r_i(t). \quad (7.21)$$

This is a discrete version of a leaky integrator and can also be written in a differential form. This trace term represents the sliding average of recent firing, with an exponential sliding window of width characterized by the parameter η . The precise form of this trace term is not essential for the following mechanisms, and other trace terms can be used. With this trace in the firing of the nodes in the recurrent network, we can associate the co-firing of rotation cells with the movement of the activity packet in the recurrent network. The weights between rotation nodes (which have a superscript ‘rot’) and the synapses in the recurrent network (which have no superscript) can be formed with a Hebbian rule,

$$\delta w_{ijk}^{\text{rot}} = \epsilon r_i \bar{r}_j r_k^{\text{rot}}. \quad (7.22)$$

The parameter ϵ is, as usual, a learning rate. The rule strengthens the weights between the rotation node and the appropriate synapses in the recurrent network. As before, we can form these weights during the learning phase where the firing of the nodes is determined by the firing of external input. This learning phase corresponds to the exploration of an environment by a subject using visual cues and is *hetero-associative* in the sense that it associates consecutive states during learning.

7.4.3 Updating the network after learning

After the weights have been learned, we can update head directions of a subject without external input. The dynamic of the model is specified with

$$\tau \frac{\partial h(x, t)}{\partial t} = -h(x, t) + \int_y w^{\text{eff}}(x, y, r^{\text{rot}}) r(y, t) r_i^{\text{rot}}(t) dy, \quad (7.23)$$

where the index i labels the group of nodes of either clockwise or anti-clockwise rotation cells. The weight matrix w^{eff} depends on the activity of the rotation nodes, and describes the effective weight kernel within the recurrent network

from the collateral connections and the modulatory influence of the idiothetic cues. The modulatory nature of these influences can, for example, be expressed by

$$w_{ij}^{\text{eff}} = (w_{ij} - c)(1 + w_{ijk}^{\text{rot}}r_k^{\text{rot}}), \quad (7.24)$$

though other forms of modulatory functions are possible and generally lead to results similar to the ones outlined below.

The behaviour of the model when trained on examples of one clockwise and one anti-clockwise rotation, with only one rotation speed, is demonstrated in Fig. 7.20. An external position stimulus was applied initially for 10τ to initiate an activity packet, and this activity packet is stable after the removal of the external stimulus when the rotation nodes are inactive. Between $20\tau \leq t \leq 40\tau$ we applied a clockwise rotation activity corresponding to the activity used during learning. The activity packet then moved in the clockwise direction linearly within this time. The movement stops immediately after the rotation cell firing is abolished at $t = 40\tau$. During $50\tau \leq t \leq 70\tau$ we applied an anti-clockwise rotation activity corresponding to the activity used during learning. The activity packet moved at nearly twice the speed in the anti-clockwise direction, demonstrating that the network can generalize to other rotation speeds.

Examples of the weighting functions after learning are shown in Fig. 7.21. The solid line represents symmetrical collateral weighting values $w_{50,i}$ between node 50 and the other nodes in the network. The clockwise rotation weights $w_{50,i,1}^{\text{rot}}$ are shown as a dashed line in the figure. Their functional form is not symmetrical as expected. The corresponding anti-clockwise rotation weights $w_{50,i,2}^{\text{rot}}$, not shown in the figure, are a mirror-image of the dashed line. The resulting effective weighting function (eqn 7.24) is shown as a dotted line in the figure. All the examples are within a feature space with an intrinsic one-dimensional topography. However, path integration can be achieved in analogous ways with feature spaces in higher dimensions as the network self-organizes.

7.5 Distributed representation and population coding

We have discussed in this chapter how information is represented in cortical and some subcortical maps, and we will end this chapter with a more general discussion of distributed coding. We consider here that a stimulus is represented by a vector with components that have values given by the responses (such as firing rates) of neurons in a brain area. We then want to know how many components are actively used to represent a stimulus in the brain. We can distinguish roughly three classes of representation:

- (1) **Local representation:** In a local representation only one node represents a stimulus in that only one node is active when a particular stimulus is presented. A single node (or neuron) would be sufficient to indicate that a particular stimulus was present. Neurons with such characteristics have been termed *cardinal cells*, *pontifical cells*, or *grandmother cells*. A

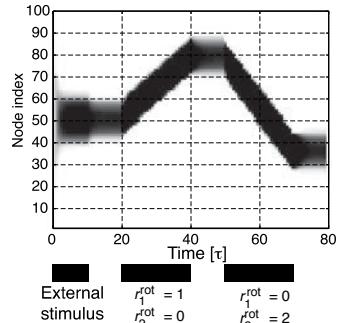


Fig. 7.20 Simulation of a DNF model with idiothetic updating mechanisms. The activity packet can be moved with idiothetic inputs in either clockwise or anti-clockwise directions, depending on the firing rates of the corresponding rotation cells. The experimental stimuli are indicated at the bottom.

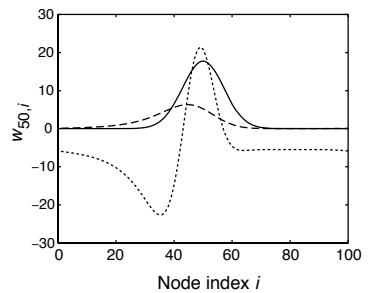


Fig. 7.21 The different weighting functions from node 50 to the other nodes in the network after learning. w , solid line; w^{rot} , dashed line; w^{eff} , dotted line.

vector of length N could represent N different features with such a local representation. The number of stimuli that can be encoded by such a vector increases linearly with the number of components (nodes). Reading out which stimulus is represented is very easy with a local representation. In the case of representations with binary nodes that are either ‘on’ or ‘off’, which we will consider frequently later, only one node would be ‘on’ for each possible stimulus.

- (2) **Fully distributed representation:** The fully distributed representation can be seen as the other extreme, compared to the local representation. A stimulus is encoded by the combination of the activities of all the components in the vector representing a stimulus. In the case of binary nodes we can think of the case where the probability of each node being *on* or *off* is equal (that is, 50%), so that the number of active nodes is 50% on average. The number of different stimuli that can be encoded with such vectors scales exponentially with the number of nodes and is therefore much larger than in the case of local representations. The information stored in a node vector is, however, the same, as they always specify a stimulus uniquely. An advantage of such a representation is that we can define similarities of stimuli by counting how many components of the vector have similar values. This is important for building associations.
- (3) **Sparingly distributed representation:** Somewhat of a compromise between the above two representations is a sparingly distributed representation. In this scheme only a fraction of the components (nodes) of a vector are involved in representing a certain stimulus. The number of stimuli that can be represented by a vector of length N is then somewhere in-between the cases above. Note that the information content has to be the same in all cases because the information cannot change with the internal representation as long as the representation is information-preserving; only the information content of each node will be different. In the case of binary vectors, we would have a larger probability for a node to be ‘off’ compared to the probability for the nodes to be ‘on’.

7.5.1 Sparseness

To specify more quantitatively how many (or what percentage of) neurons are involved in the neural processing of individual stimuli, we define here a measure of *sparseness* of a representation. A definition is obvious in the case of binary nodes. If we consider neurons (or population nodes) that are either active ($r = 1$) or not ($r = 0$), then the average number of nodes that are active for a set of stimuli is

$$a = \frac{1}{S} \sum_s \frac{1}{N} \sum_i r_i^s, \quad (7.25)$$

where S is the number of stimuli over which the sparseness is evaluated, and N is the number of neurons in the considered population. In other words, if we consider relative firing rates r that are either 0 if the neuron is not responding or 1 if it is responding, then the sparseness of the representation with binary

nodes is defined by the average relative firing rate,

$$a = \langle r_i^s \rangle_{i,s}, \quad (7.26)$$

where the average is taken over the number of neurons in the population and the number of stimuli in the test set. A fully distributed binary representation has in this definition a sparseness of $a = 0.5$, and the sparseness of sparsely distributed representations is less if we restrict ourselves to the case where the number of nodes that are *on* is less than $N/2$. Note that the other case of having more *on*-nodes than *off*-nodes can be mapped to the previous case by redefining the values for the representation of *on* and *off*.

The definition of sparseness in the case of vectors with continuous (real-valued) components is not that obvious. We have then to decide how much weight we put on the contribution of small versus large firing rates. What we would like to do is to take the information in the firing rate as a weighting factor, which means that we should take the firing rate relative to the variance of the firing rate into account. We thus define the sparseness of a representation, again defined as average over a set of stimuli, as

$$a = \frac{\langle r_i^s \rangle_{i,s}^2}{\langle (r_i^s)^2 \rangle_{i,s}}. \quad (7.27)$$

For example, we can measure the firing rate of N neurons in response to a set of S stimuli and estimate the sparseness as

$$a = \frac{\left(\frac{1}{S} \sum_s \frac{1}{N} \sum_i r_i^s \right)^2}{\frac{1}{S} \sum_s \frac{1}{N} \sum_i (r_i^s)^2}. \quad (7.28)$$

The definitions 7.25 and 7.27 are equivalent in the case of binary vectors with components of value 0 and 1. The sparseness of coding with tuning curves, as modelled by DNF models, can be calculated from the size of the activity packet. For example, the sparseness of the end states in the simulation of Fig. 7.9 is $a = 0.56$. Thus, orientation tuning curves in V1 indicate a strongly distributed representation in a hypercolumn. However, we will see that some brain areas, in particular the hippocampus, which is discussed more in the next chapter, are though to have much sparser representations.

7.5.2 Probabilistic population coding

We expect that information about the external world and the processing of information is distributed in the brain, but this representation can be very noisy given the stochastic nature of neuronal activities, as discussed in Chapter 3. Here we discuss probabilistic *encoding*, the specific way in which a stimulus is coded in a neuron or a population of neurons, and *decoding*, how a message can be read from the responses of a neuron or a population of neurons. We can express the encoding of a stimulus pattern in terms of the response probability of neurons in a population with a joined probability distribution of neuronal responses conditional on a stimulus s ,

$$P(\mathbf{r}|s) = P(r_1^s, r_2^s, r_3^s, \dots | s), \quad (7.29)$$

where r_i^s is the stimulus-specific response of neuron i in the population. We will here mainly consider stimulus-specific firing rates as the response, although other response quantities such as the latencies of firings can be treated in the same framework. We are using the symbol P as either meaning the probability for discrete random variables or the probability densities in the case of continuous random variables.

Decoding is the inverse of encoding, that is, we want to deduce what stimulus was presented from the neuronal responses of a neuron or a population of neurons. Decoding brain activity has recently become of much interest in brain-computer interfaces, and the brain may have to perform such computations at some stages. The probability that a stimulus was present, given a certain response pattern of the neurons in the population, is expressed by the conditional probability

$$P(s|\mathbf{r}) = P(s|r_1^s, r_2^s, r_3^s, \dots). \quad (7.30)$$

If we know this conditional probability, we can say which stimulus was most likely present given a certain response of the neuron population. It is obvious to choose the most likely stimulus as an answer to the decoding problem,

$$\hat{s} = \arg \max_s P(s|\mathbf{r}), \quad (7.31)$$

where \hat{s} is our estimation of the stimulus and $\arg \max f(x)$ is a function that selects the argument x that maximizes the expression $f(x)$. We can estimate the conditional probabilities $P(\mathbf{r}|s)$ from recordings of cells as stated above, but we need the conditional probability $P(s|\mathbf{r})$. However, these two conditional probabilities are related through the identity called *Bayes's theorem*⁴

$$P(s|\mathbf{r}) = \frac{P(\mathbf{r}|s)P(s)}{P(\mathbf{r})}. \quad (7.32)$$

This theorem is important because it tells us how to combine *prior* knowledge, such as the expected distribution of stimuli, $P(s)$, with some evidence as measured by $P(\mathbf{r}|s)$, to get the *posterior* distribution $P(s|\mathbf{r})$ from which the optimal estimate (in a statistical sense) of the stimulus can be calculated with eqn 7.31.

$P(r)$ is the proper normalization so that the left-hand side is again a probability. To use the theorem we need to have an estimate of the prior $P(s)$ and the evidence $P(\mathbf{r}|s)$. The normalization is not necessary if we are only interested in finding the most likely stimulus. Let us consider the case of equally likely stimuli. We then need to estimate the probability $P(\mathbf{r}|s)$. However, this is what is given by the tuning curves. Indeed, we can view this as a function of the stimulus. When treating the probability function as a function of the stimulus, which is usually done by building a concrete model (hypothesis) of the response, this function is then called the *likelihood function*. Thus, we can ask which sensory stimulus provides the *maximum likelihood* of the data,

$$\hat{s}_{\text{ML}} = \arg \max_s P(\mathbf{r}|s), \quad (7.33)$$

which is called the *maximum likelihood estimate*. In practice we mostly maximize the logarithm of the likelihood because a joined probability of independent

⁴In the literature there are sometimes discussions and critiques on the *Bayesian view*. This refers to a more philosophical discussion on the use of probability theory to describe experiments with underlying mechanisms that are thought to be deterministic in nature. In contrast, Bayes's theorem is an identity within probability (or set) theory and is not questioned by statisticians.

variables can be expressed as the product of their marginal probabilities. The logarithm of a product is the sum of the logarithm of the individual terms. Such a sum is often more easy to calculate. Also, note that when we have to take the prior into account we need to use Bayes' theorem to calculate the maximum posterior distribution (MAP; eqn 7.31) to estimate the most likely stimulus causing the neural response.

The importance of the maximum likelihood estimate in statistics is that it is an *unbiased estimate*, which is an estimate for which the expectation value (mean) of the estimate is equal to the correct parameter, $E(\hat{s}_{\text{ML}} = s)$. Furthermore, the variance of this estimate is optimal in the sense that it approaches the minimum possible variance given by the *Cramér–Rao bound*,

$$E((\hat{s}_{\text{ML}} - s)^2) = \frac{1}{I_F}, \quad (7.34)$$

where I_F is the Fisher information

$$I_F = - \int p(\mathbf{r}|s) \left(\frac{d}{ds} \ln p(\mathbf{r}|s) \right)^2 ds. \quad (7.35)$$

The maximum likelihood estimate can therefore provide some reasonable estimates with limited data sets.

7.5.3 Optimal decoding with tuning curves

Tuning curves are commonly deduced from average responses of neurons to many different stimuli, and these curves provide some estimate of the likelihood function. We use again Gaussian tuning curves $f_i(s)$,

$$r_i = f_i(s) = e^{-(s-s_i^{\text{pref}})^2/2\sigma_{tc}^2}, \quad (7.36)$$

with width σ_{tc} , and where s_i^{pref} is the preferred stimulus of this node. An example is provided in Fig. 7.22A. Note that we cannot determine the feature value of the stimulus from the firing rate of one neuron unambiguously because a certain firing rate can be the result of two different stimuli. This ambiguity is, however, removed in a population code. A second neuron with a shifted tuning curve, illustrated in Fig. 7.22B, can resolve the ambiguity. In the illustrated example the firing rate of the neuron is higher in the case of the true stimulus corresponding to the right solution compared to the alternative. Note also that the accuracy of the decoding can be different for different values of the stimulus feature value s . A good resolution can be achieved where the tuning curve changes most with respect to the stimulus feature value s , around the largest slope of the tuning curve, $\max(f'(s))$, not at the maximum. However, with noisy data we also have to take the signal to noise ratio into account, which might be highest at the centre of the tuning curves.

In the following we will assume that the response fluctuations of the neurons around this average response profile are independent, so that the conditional probability $P(\mathbf{r}|s)$ can be written as the product of the conditional probabilities of the individual neurons,

$$P(\mathbf{r}|s) = \prod_i P(r_i|s). \quad (7.37)$$

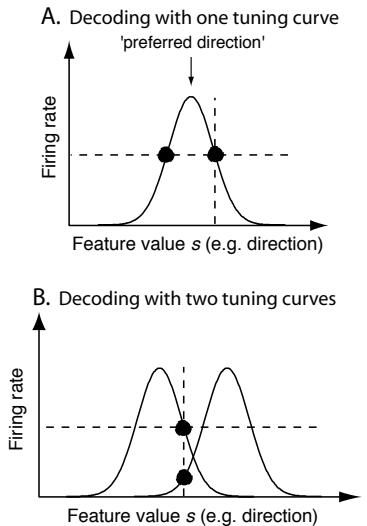


Fig. 7.22 Gaussian tuning curves representing the firing rate of a neuron as a function of a stimulus feature. (A) A single neuron cannot unambiguously decode the stimulus feature from the firing rate. (B) A second neuron with shifted tuning curve can resolve the ambiguity.

This is called a naive Bayes assumption. We still do not know the individual probability densities and have to guess this. Since we estimated the tuning curves from average responses, and since the average of random numbers tends to be Gaussian distributed (according to the central limit theorem), we use

$$P(r_i|s) = \frac{1}{\sqrt{2\pi}\sigma_i} e^{-(r_i - f_i(s))^2/2\sigma_i^2}, \quad (7.38)$$

we can calculate the *log-likelihood function* by taking the logarithm of this expression and viewing it as a function of s . The maximum likelihood estimator can thus be extracted in this situation by minimizing the expression,

$$\hat{s} = \operatorname{argmin}_i \sum \left(\frac{r_i - f_i(s)}{\sigma_i} \right)^2, \quad (7.39)$$

which is a least square fit of the data points r_i (measured firing rates) to the expected firing rates $f_i(s)$ of stimulus s .

7.5.4 Implementations of decoding mechanisms

We can now outline some practical methods for population decoding, which have been suggested as implementations of decoding mechanisms in the brain. The most commonly used method is simply called *population vector decoding*, which we will use to demonstrate some general properties of decoding with different widths of tuning curves. For this demonstration, we consider a set of neurons with Gaussian tuning curves, as illustrated in Fig. 7.23 for eight neurons. These nodes have equally distributed preferred directions with centres s_i^{pref} every 45 degrees. We will compare a system with two different widths of the receptive fields, $\sigma_t = 10$ degrees shown in Fig. 7.23A, and $\sigma_t = 20$ degrees shown in Fig. 7.23B. We might intuitively think that sharper tuning curves lead to more accurate decoding. This is, however, not the case. The principal reason for this can be seen in the firing rate pattern of the eight nodes in response to a specific stimulus pattern shown in the second row of Fig. 7.23. There, we plotted the noiseless response of the neurons to a stimulus at 130 degrees, indicated by the vertical dashed line. The neurons with preferred direction close to this stimulus value respond heavily. However, we also see a response from some of the neurons in the population with wide receptive fields, which helps in the decoding process.

To decode the stimulus value for the firing pattern of the population we multiply the firing rate of each neuron by its preferred direction and sum the contributions of all the neurons,

$$\hat{s}_{\text{dir}} = \sum_i r_i s_i^{\text{pref}}. \quad (7.40)$$

For a stimulus that coincides with a preferred direction for one neuron, and with small sizes of the receptive fields so that the firing rates of the other neurons can be neglected, we get an estimate that is r_i times the preferred direction of the node that is firing. We will therefore have to apply some further normalization.

In our example, the stimulus values are real values for the direction of the moving object. However, we can apply this method to higher-dimensional

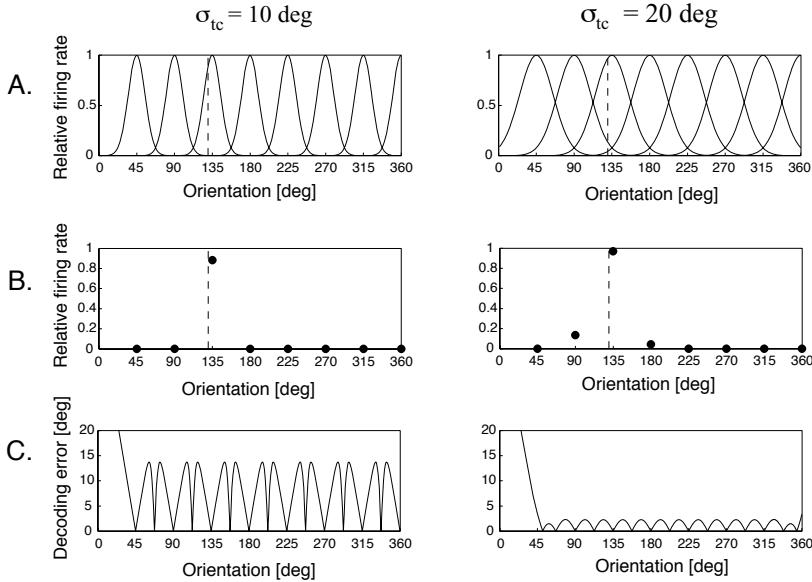


Fig. 7.23 (A) Gaussian tuning curves representing the firing rate of a neuron as a function of a stimulus feature. (B) Example of the firing rates of the eight neurons in response to a stimulus with direction 130 degrees. (C) Decoding error when the stimulus is estimated with a population code.

stimulus presentations, so that the preferred stimulus of a neuron is a feature vector s_i^{pref} . The estimate (eqn 7.40) is then a vector that is an estimate of the direction of the feature vector, but has a length that depends on the sum of the firing rates. If we are interested in the precise values of the stimulus features, and the nodes have different dynamical ranges $r_i^{\min}-r_i^{\max}$, we have to normalize the firing rates to the relative values, and the sum in eqn 7.40 to the total firing rates,

$$\hat{r}_i = \frac{r_i - r_i^{\min}}{r_i^{\max}} \quad (7.41)$$

$$\hat{s}_{\text{pop}} = \sum_i \frac{\hat{r}_i}{\sum_j \hat{r}_j} s_i^{\text{pref}}. \quad (7.42)$$

This is the *normalized population vector* that can be used as an estimate of the stimulus. The absolute error of decoding orientation stimuli with this scheme in our example of eight neurons is shown in the last row of Fig. 7.23. The decoding error is very large for small orientations because this part of the feature space is not well covered by the population of neurons. Reasonable estimates are, however, achieved for the areas of the feature space that are covered reasonably well by the receptive fields of the neurons. The error is not uniform and depends on the feature values. The average error is much less for the larger receptive fields compared to the average errors of the population of neurons with smaller receptive fields.

We have only shown results for the noiseless case. The real test is the performance of estimates with noisy data, which we leave as an exercise. It is well known that other estimation methods can easily outperform the population vector decoding-procedure outlined here. In particular, DNF models can be

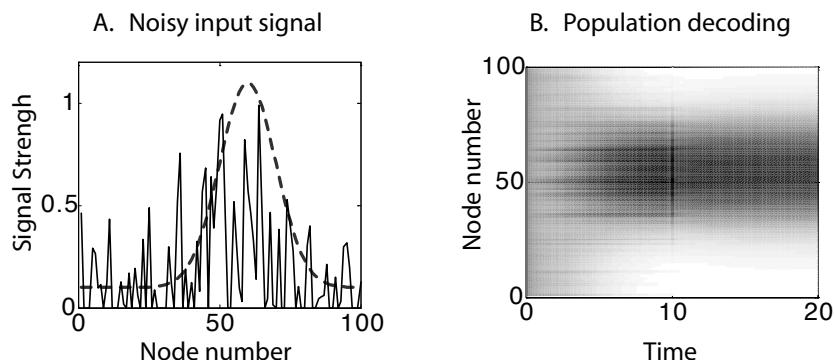


Fig. 7.24 (A) Noisy signal (solid line) derived from a Gaussian signal (dashed line) centered around node 60. (B) Population decoding with dynamical neural fields in which the noisy signal was used as an input vector to the model until $t=10$.

used for this purpose. These have the advantage of being computationally efficient and biologically plausible. To use DNFs for noisy population decoding we can simply apply a noisy population vector as input to the model. An example is shown in Fig. 7.24. In this example we used a very noisy signal, shown as a solid line in Fig. 7.24A. This was derived from the noiseless Gaussian signal around node 60, shown as a dashed line in the same graph. The noisy input is then applied as input to the DNF model with the program used before (Table 7.2). The time evolution for this simulation is shown in Fig. 7.24B. The competition within the model cleans up the signal and finds a winning node that corresponds to the direction of the noiseless signal. There is already some advantage in decoding before the signal is removed at time $t = 10$. This example demonstrates that simple decoding using the maximal value would produce large errors with the noisy signal. However, the maximum decoding can easily be applied to the clean signals after some updates in the DNF model.

Exercises

- (7.1) What is a tuning curve, and what is a receptive field of a neuron?
- (7.2) Modify program `som.m` to measure the distance between the centres of the receptive fields in the SOM map and the centres of a perfect map with grid points at `c1p=x/nn;` `c2p=y/nn;`. Plot the development of this distance measure over time and interpret your results.
- (7.3) Modify the `dnf.m` program to simulate two simultaneous inputs with the same strength at $1/4 \text{ nn}$ and $3/4 \text{ nn}$ and observe the network activity. Change the strength of one input and discuss the results. Return to the same strength of input, but choose different locations. Discuss your results.
- (7.4) Given the noisy population signal in file `noisyPopulationSignal`, estimate the feature value encoded with this signal and explain your choice.

Further reading

The computational framework of associative and self-organizing networks has been emphasized by Teuvo Kohonen since the late 1960s (Kohonen, 1989). The Kohonen network is a computational efficient model of self-organization, although, as argued in this chapter, it should be seen as a more abstract and constrained model of the biologically more direct model of Willshaw and von der Malsburg (1976). The latter model follows some earlier work by von der Malsburg (1973), and this paper includes discussions of connectivity patterns of excitatory and inhibitory pools, similar to the paper by Wilson and Cowan (1973), which followed their 1972 paper mentioned in Chapter 3. The dynamic equations in Willshaw and von der Malsburg (1976) were formally introduced as dynamic neural field theory by Shun-ichi Amari (1977) as an abstraction of the model by Wilson and Cowan (1973). Also, Takeuchi and Amari (1979) later analysed the Willshaw–von der Malsburg model in more detail. A nice example of dynamic neural field models as applied to head direction cells is given by Kechen Zhang (1999), and corresponding self-organization and path integration is discussed in Stringer *et al.* (2002). Population decoding is discussed nicely in Pouget *et al.* (2000).

Teuvo Kohonen (1989), *Self-organization and associative memory*, Springer Verlag, 3rd edition.

David J. Willshaw and Christoph von der Malsburg (1976), *How patterned neural connexions can be set up by self-organisation*, in *Proceedings of the Royal Society B* 194: 431–45.

Christoph von der Malsburg (1973), *Self-organization of orientation sensitive cells in the striate cortex*, in *Kybernetik* 14: 85–100.

Huge R. Wilson and Jack D. Cowan (1973), *A mathematical theory of the functional dynamics of cortical and thalamic nervous tissue*, in *Kybernetik* 13: 55–80.

Shun-ichi Amari (1977), *Dynamic pattern formation in lateral-inhibition type neural fields*, in *Biological Cybernetics* 27: 77–87.

Akikazu Takeuchi and Shun-ichi Amari (1979), *Formation of topographic maps and columnar microstructures in nerve fields*, in *Biological Cybernetics* 35: 63–72.

Kechen Zhang (1996), *Representation of spatial orientation by the intrinsic dynamics of the head-direction cell ensemble: A theory*, in *Journal of Neuroscience* 16: 2112–26.

Simon M. Stringer, Thomas P. Trappenberg, Edmund T. Rolls, and Ivan E.T. de Araujo (2002), *Self-organizing continuous attractor networks and path integration I: one-dimensional models of head direction cells*, in *Network: Computation in Neural Systems* 13: 217–42.

Alexandre Pouget, Richard S. Zemel, and Peter Dayan (2000), *Information processing with population codes*, in *Nature Review Neuroscience* 1: 125–32.

This page intentionally left blank

Recurrent associative networks and episodic memory

8

This chapter continues the discussion on the dynamics of *recurrent networks*. However, instead of using the characteristic recurrent weight profile of the last chapter, we now train the networks on random patterns. We show that such networks can function as *auto-associative* memories, which can rapidly store items and are able to recall stored items from partial information. In contrast to DNF models, which have a continuous manifold of point attractors (*continuous attractors* for short), the models here have isolated *point attractors* of their dynamics. We discuss the storage capacity of point attractor networks and their extraordinary robustness to noise and lesions. But memories can also break down rapidly when overloading the network or when lesions become to severe, and we will study the physics of such sharp transitions to amnesic phases.

8.1 The auto-associative network and the hippocampus

8.1.1 Different memory types

The words ‘learning’ and ‘memory’ can be applied to many different domains. We have already discussed several types of learning and memory, and it is time conceptualize the different types in some form. In the last chapter, we encountered a type of short-term memory, often called working memory by physiologists, in which representations of concepts seem to be held active by sustained firing of neurons. This type of short-term storage is essential for many mental abilities and thus relates to what psychologists call working memory and which will be discussed further in Chapter 9. We also discussed, in the last chapter, the formation of cortical maps, which is a form of learning and memory since it influences brain processes from environmental factors. This type of memory is long term, since these learned organizations can be stable for a long time. However, there are other aspects of learning and memory we would like to comprehend, such as learning concepts, remembering specific events, or learning motor skills.

Fig. 8.1 shows a classification of memory adopted from a review by *Larry Squire*. In this scheme, memory is distinguished by either being *declarative* or

8.1 The auto-associative network and the hippocampus	215
8.2 Point-attractor neural networks (ANN)	219
8.3 Sparse attractor networks and correlated patterns	233
8.4 Chaotic networks: a dynamic systems view ◇	238
Exercises	246
Further reading	247

non-declarative. Declarative memories are explicit memories such as recalling specific events or facts. Recalling specific events, such as remembering what you did this morning, is called *episodic memory*. Remembering facts, such as the name of the capital of Italy, is called *semantic memory*. Non-declarative memory is more implicit, which includes concepts such as *procedural* and *perceptual* learning and memory, classical conditioning, and *non-associative* learning. Procedural learning includes learning motor skills, such as how to ride a bike, or social skills, like learning to function in a group. Perceptual learning includes the formation of cortical maps, which is related to priming in the psychology literature. Classical conditioning, such as responding to a bell with an eye blink when the bell was previously combined with an air puff, is another widespread adaptation mechanism in the brain. We will discuss this topic together with reward learning in Chapter 9. All these forms of memory can be related to Hebbian type associative mechanisms. The last category includes non-associative memories, such as reflexes.

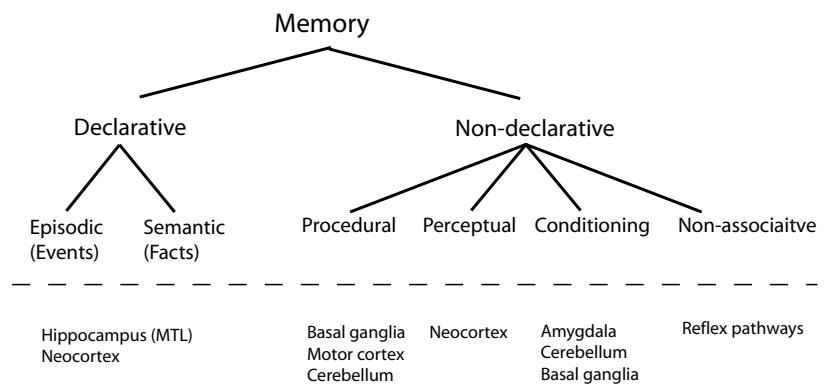


Fig. 8.1 Outline of a memory classification scheme adapted from Squire, *Neurobiology of Learning and Memory* 82: 171–7 (2004).

Fig. 8.1 includes, at the bottom, some examples of brain areas in the nervous system that have been associated with such memory concepts. Conditioning has been associated with several brain areas. For example, implicit emotional responses, such as fear have been associated with the amygdala, and the cerebellum has been shown to be involved in eye blink conditioning. We will specifically discuss reward learning in the basal ganglia in the next chapter. The formation or reorganization of cortical maps is an example of perceptual learning, and procedural learning has been associated with the motor cortex, the basal ganglia, and the cerebellum, to name a few. Interestingly, a very specific area called the *hippocampus*, together with adjacent areas in the medial temporal lobe (MTL), are frequently associated with declarative memory. This area is of particular interest to the discussion in this chapter. While we will discuss the hippocampus as an example, it is important to keep in mind that the models in this chapter are more general and that this area alone is certainly not the only brain area involved in declarative memory. Declarative memory relies heavily on cortical processes, and the precise contribution of the hippocampus in various forms of declarative memory is still under considerable debate.

Before getting more specifically into models of declarative memories, we will contrast this with Chapter 6, where we discussed statistical learning. Networks in that chapter were aimed at learning to extract *central tendencies* to allow generalizations to unseen data. The memorization of single instances was distractive, since they cause overfitting. While such types of learning are important in the brain and are likely present in some form of semantic memory, we concentrate here on the learning of specific instances and associative mechanisms, as discussed in Chapter 4.

The network considered in this chapter, shown in Fig. 8.2, is based on the same type of recurrent network as the ones in the last chapter. The only difference between the auto-associative memory discussed in this chapter and the DNF model discussed in Chapter 7 is that the network is trained on a different pattern set. While the DNF models were systematically trained on all possible features to be represented in the map, the networks in this chapter will be trained on random patterns. We will see that this leads to well-separated point-attractors in these networks, whereas the DNF model had a continuous manifold of point attractors. The models here are therefore called *point attractor neural networks* (PANNs), or simply *attractor neural networks* (ANNs), in contrast to the continuous attractor neural networks (CANNs) of Chapter 7. The model in Fig. 8.2 is similar to the associator network shown Fig. 4.2, except that the input of each node is fed back to all of the other nodes in the network. In this way we can associate a pattern with itself. Such networks are therefore also called *auto-associator*, although it is possible to realize an auto-associative memory as feedforward architecture, often called *auto-encoders*. In this chapter, we are specifically interested in recurrent networks, and we will use the auto-associative network synonymously with the specific form of a recurrent network.

The back-projections in this associative network introduce attractive features into the model. As has already been seen, associators are able to perform some form of pattern completion. After an external input pattern is presented to the network, the network will respond with an output pattern more similar to a trained pattern when Hebbian learning is applied. This response is now fed back as input to the same network, so that the network will respond with a pattern that is again made closer to a learned pattern. We therefore expect that the cycling in a recurrent network can enhance the pattern completion ability of simple one-step associative nodes. The dynamic process of changing responses in the recurrent network will stop when a learned output pattern is reached.

The network, as used Chapter 7, keeps an asymptotic state active until another pattern is applied to the network. This is analogous to the sustained activity in the last chapter. However, working memory will not be the main focus of the discussions here. Furthermore, while we will talk about asymptotic and stationary states in this chapter, this should not be taken as an argument to dismiss such models since this is only a convenient way describe the system mathematically. Such states are meant to describe the systems in some way and should be thought of as describing the trends of a system with ongoing iterations. We will see that some systems can come close to asymptotic states within a few iterations.

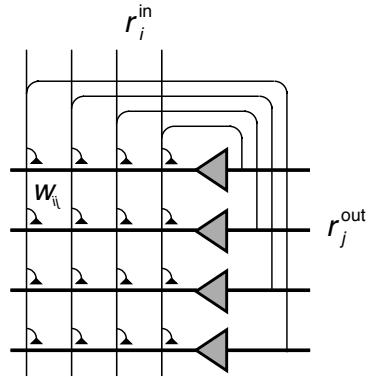


Fig. 8.2 An auto-associative network which consists of associative nodes that not only receive external input from other neural layers but, in addition, have many recurrent collateral connections between the nodes in the neural layer.

8.1.2 The hippocampus and episodic memory

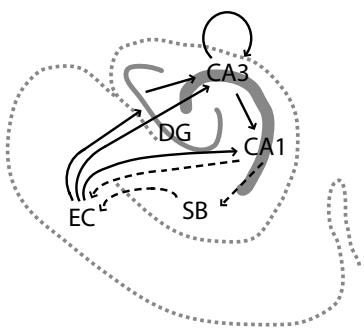


Fig. 8.3 A schematic outline of the medial temporal lobe with some connections mentioned in the text. Some areas are indicated by acronyms including the entorhinal cortex (EC), dentate gyrus (DG), hippocampal subfield cornus ammonis (CA), and subiculum (SB).

The models discussed in this chapter have been implicated with hippocampal functions since a theory of the archicortex was proposed by *David Marr* in 1976. This phylogenetically older part of the cortex is part of the medial temporal lobe and is illustrated in Fig. 8.3 with a coronal section. This structure is called *hippocampus* for its shape resembling a seahorse, and some of its subfields are labelled CA for *cornus ammonis* after the greek ram god. This is the same structure in which *Bliss* and *Lomø* discovered LTP and which is a primary area examined in plasticity research as mentioned in Chapter 4. Also, we mentioned place fields in Chapter 7, which are often recorded in the hippocampus of rodents, as well as head direction cells, which are often recorded from the neighbouring subiculum.

The hippocampus has been implicated with the acquisition of *episodic memories* since the famous case of patient H.M. In this patient, large parts of both medial temporal lobes were removed to treat his epileptic condition. Subsequently, he suffered from a form of amnesia marked by the inability to form new long-term memories of episodic events. In contrast, long-term memory that was acquired before the removal of this structure was not impaired. However, he still could learn new motor skills and even acquire some new semantic memories. This condition is called *retrograde amnesia*. The precise involvement of the hippocampus in memory acquisition and recall is still under intense debate, but it seems that the hippocampus can rapidly store memories of events which may later be consolidated with neocortical information storage.

The hippocampus seems extraordinarily adapted to the rapid storage of complex events. The input to the hippocampus comes primarily from the entorhinal cortex (EC), which receives input from many cortical areas. The coding within these areas, in particular in the dentate gyrus (DG), is very sparse, and we will discuss later how this helps to minimize interference with other memories. The DG is also an area where *neurogenesis*, the creation of new neuronal cells throughout the lifetime of an organism, has now been established. There are two major pathways to subfield CA3, one that first projects to DG, and another which passes through DG, called the *perforant pathway*. The CA3 region has many collateral connections which contact other cells in CA3 and CA1. These abandoned collaterals, in particular within CA3, have largely inspired the study of recurrent networks of the form shown in Fig. 8.2. CA1 projects back to the EC, but EC also has direct projections to CA1. It is possible that some of the pathways are preferentially active in either a storage or retrieval phase, as discussed further below.

The basic models discussed further in this section only capture some aspects of hippocampal processing, most notably the storage abilities of specific events in auto-associative networks. While such models are capture an important ability of networks, that of forming associative memory in recurrent networks, there are now much more detailed proposals of hippocampal functions with respect to sequence processing, novelty detection, memory consolidation, and other functions. But the basic model was an important milestone in the computational neuroscience.

8.1.3 Learning and retrieval phase

Before we outline the memory abilities of recurrent associative networks in more detail, we must mention a difficulty that occurs when combining associative Hebbian mechanisms with recurrences in the networks. Associative learning depends crucially on relating presynaptic activity to postsynaptic activity that is imposed by an unconditioned stimulus. The recurrent network will, however, drive this postsynaptic activity rapidly away from the activity pattern we want to imprint if the dynamic of the recurrent network is dominant. A solution to the problem, which we will follow in the discussion below, is to divide the operation of the networks into two phases, a *training phase* and a *retrieval phase*. These phases can be interleaved. There is no inherent difficulty in assuming such different phases as we rarely want to store and retrieve information at the same time.

There are several proposals as to how the switching between the learning and retrieval phase could be accomplished in the hippocampus. For example, *mossy fibres* from *granule cells* in the DG provide strong inputs to CA3 neurons with the largest synapses found in the mammalian brain, which David Marr termed *detonator synapses*. Thus, CA3 firing patterns could be dominated by this pathway during a learning phase. In contrast, the perforant pathway could stimulate the CA3 neurons in the retrieval phase, where the CA3 collateral and CA3–CA1 projections could help to complete patterns from partial inputs. This ability could also be supported by different types of synapses and their proximity to the cell body.

Another proposal, investigated by *Michael Hasselmo*, is that chemical agents, such as *acetylcholine* (ACh) and *noradrenaline* (also called *norepinephrine*), could modulate learning and thereby enable the switching between a retrieval and learning phase. It has been shown that such *neuromodulators* facilitate synaptic plasticity and that their presence enhances the firing of the neurons. At the same time, neuromodulators suppress excitatory synaptic transmission, which can then suppress the effects of the recurrent collaterals. The neurons are therefore mainly responsive to external input to the system, which mirrors the proposal of the learning phase that we will generally apply in the following models.

Even if ACh is not directly responsible for switching between learning and retrieval phases in the hippocampus, it is well known that this chemical is important as a modulator of plasticity throughout the cortex. The switching between learning and retrieval phases might also be necessary for the transfer of intermediate-term memories stored in the hippocampus to long-term memory stores in other cortical areas. This might happen during sleep, and fluctuations of ACh during sleep have been detected. There are many further interesting questions, such as the control of the neuromodulation and the time scale of the switching between learning and retrieval, which need further investigation.

8.2 Point-attractor neural networks (ANN)

Many researchers have speculated about the importance of feedback connections within brain networks.¹ Much of the following is based on work by Amit,

¹Grossberg was among the first to propose, analyse and popularize such networks in connection with brain processes, since the late 1950s. Other important pioneers include *Caianiello*, *Longuet-Higgins*, *Willshaw*, *Edelman*, *Amit*, and *emphKohonen*, among others. For a nice collection of early papers see *Shaw and Palm*, *Brain theory*, preprint edition, World Scientific (1988). *John Hopfield* popularized these networks among physicists in the early 1980s.

Gutfreund and Sompolinsky. We will now discuss *attractor states*, since they dominate the time evolution of the dynamic systems discussed here, at least asymptotically. Since it may take a considerably long time for the system to come close to such states, the relevance of such states for brain processes has been questioned. However, we will see that attractor states are reached fairly rapidly in most of the situations discussed here. Also, there is no need for the brain to settle into attractor states. Rather, the rapid path towards attractor states may be sufficient to use them as associative memories. While this suits the real-time processing demands of brain functions much better, we discuss attractor states to analyse the dynamic regimes of the networks.

8.2.1 Network dynamics and training

The dynamic rule of ANNs is the same as the one covered in Chapter 7, although a description with individual nodes is more appropriate to use here. We therefore use the discrete version (eqn 7.1) with external inputs as in eqn 7.6,

$$\tau \frac{du_i(t)}{dt} = -u_i(t) + \frac{1}{N} \sum_j w_{ij} r_j(t) + I_i^{\text{ext}}(t) \quad (8.1)$$

$$r_i = g(u_i), \quad (8.2)$$

and the Hebbian covariance rule for learning N_p patterns with components r_i^μ for pattern μ . When starting this rule with an initial weight matrix of zeros, and using a learning rate of ϵ (see also Section 4.4.2, eqn 4.23), we can write the resulting weight matrix as

$$w_{ij} = \epsilon \sum_{\mu=1}^{N_p} (r_i^\mu - \langle r_i \rangle)(r_j^\mu - \langle r_j \rangle) - c_i. \quad (8.3)$$

We included thereby an inhibition constant, c_i for each receiving node, analogous to the additional inhibition used in Chapter 7. The angular brackets stand for the expected values as used before. This chapter primarily considers training binary patterns. To keep notations consistent with the last chapter, we start with representations $r_i^\mu \in \{0, 1\}$, for which a threshold activation function,

$$r_i = \Theta(u_i; \theta) = \begin{cases} 1 & \text{if } u_i > \theta \\ 0 & \text{otherwise} \end{cases}, \quad (8.4)$$

is appropriate. Note that pattern values, in contrast to state values of the network, are denoted with a superscript for the different pattern numbers.

The literature on ANNs often uses patterns with representation $s_i^\mu \in \{-1, 1\}$, where the letter s is used to allude to an analogy with spin models, discussed more below. This representation is used for most of the discussions in this chapter. Of course, the arguments derived here should not depend on the representation, so some care must be given to the correct translation between the notations. To translate rates $r \in \{0, 1\}$ to spins $s \in \{-1, 1\}$ we can simply multiply the rate values by 2 and subtract a 1, $s = 2r - 1$. Hence, to transform

eqns 8.1–8.3 to the s -representation, one can simply use the substitutions

$$r_i = \frac{1}{2}(s_i + 1), u_i = \frac{1}{2}(u_i^s + 1). \quad (8.5)$$

Substituting the rate variables with the spin variables yields

$$\tau \frac{du_i^s(t)}{dt} = -u_i^s(t) + \frac{1}{N} \sum_j w_{ij} s_j + \sum_j w_{ij} + 2I_i^{\text{ext}}(t) - 1, \quad (8.6)$$

$$s_i = g^s(u_i^s) = \text{sign}(u_i^s - 2\theta + 1), \quad (8.7)$$

where the activation function is now appropriately the sign function with some adjusted threshold. The weight values, when expressed with patterns in the s -representations, are

$$w_{ij} = \epsilon \frac{1}{4} \sum_{\mu=1}^{N_p} (s_i^\mu - \langle s_i \rangle)(s_j^\mu - \langle s_j \rangle) - c_i, \quad (8.8)$$

which has the same form as in the r -representation, but with an adjustment in the learning rate.

In this chapter we are primarily interested in *stationary states*, which are the states that do not change under the dynamics of the system and which are hence marked by $du_i^s/dt = 0$. The dynamic equations are simple in the case of $\sum_j w_{ij} + 2I_i^{\text{ext}}(t) = 1$ using a threshold of $\theta = 0.5$. The stationary states are then *fixpoints* of the discrete system,

$$s_i(t+1) = \text{sign} \left(\sum_j w_{ij} s_j(t) \right) \quad (8.9)$$

where we have applied the activation function to both sides of the stationary state equation. We first discuss the case $c_i = 0$. Both, the *continuous time model* (eqns 8.6 and 8.7), and the discrete *fixpoint model* (eqn 8.9) can illustrate the asymptotic features of the system with weight matrix eqn 8.8. It is only when we are interested in the *transient response dynamics* of the model, for example when modeling the reaction times of the system, that we have to use the continuous equations.

The attractor models can be considered with noise, either with stochastic background input, noisy weights, or probabilistic transmissions. A common noise model used for results shown later is a probabilistic updating rule, which replaces the deterministic activation function (eqn 8.9)

$$s_i(t) = \text{sign} \left(\sum_j w_{ij} s_j(t-1) \right) \quad (8.10)$$

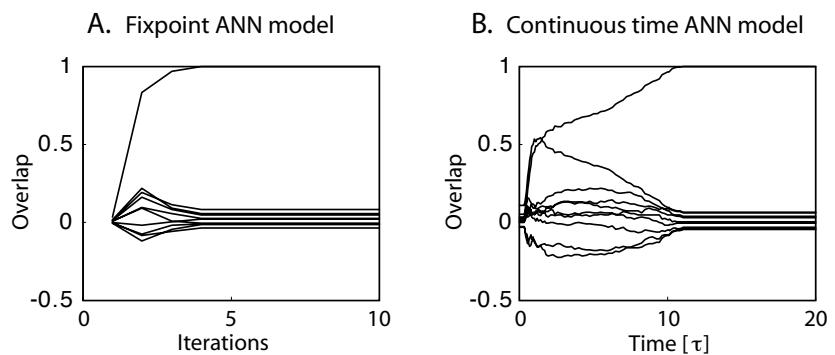
with a probabilistic version

$$P(s_i(t) = +1) = \frac{1}{1 + \exp(-2 \sum_j w_{ij} s_j(t-1)/T)}. \quad (8.11)$$

This noise model corresponds to the Boltzmann statistics in thermodynamic systems, and the noise parameter T is therefore sometimes called temperature. The sign function is recovered in the limit $T \rightarrow 0$ (see exercise).

The main conclusion of this chapter is that dynamic networks, as introduced in this section, can function as an auto-associative memory devise. This is demonstrated for the fixpoint model and the continuous time model in Fig. 8.4. Both networks were first trained on 10 random patterns. The fixpoint model

Fig. 8.4 Examples of results from simulations of ANN models. (A) Simulation of the fixpoint model with program `ann_fixpoint.m`. The overlap here is the normalized dot product of the network states during an update with all of the 10 patterns that were imprinted with Hebbian learning into the network. The network was initialized randomly, and one of the stored patterns was retrieved. (B) Simulation of the continuous time version of an attractor network with program `ann_cont.m`. A noisy version of one stored pattern was applied as external input until $t = 10\tau$.



(Fig. 8.4A) was then initialized with a random pattern, and a noisy version of one pattern was used as external input until $t = 10\tau$ for the continuous time model. Both plots show the normalized dot product, called *overlap* in these plots, between the network states with each stored pattern while updating the networks. The normalized dot product measures the cosine of the angle between two vectors and is hence equal to one if the vectors are pointing in the same direction and zero when they are perpendicular. In the figure, one of the lines becomes one, which shows that one pattern was retrieved. The simulation with the continuous model demonstrates recovery of a (noisy) memory since we used such a noisy state as input pattern until $t = 10\tau$. We show in this figure a particularly interesting case in which two patterns competed for some time before the correct stored pattern was retrieved. The simulations in the discrete case were started with a random pattern and demonstrate that one of the stored patterns was retrieved. Both simulations also demonstrate *working memory with sustained firing* after removal of the external input.

Simulation

A minimal simulation program for the fixpoint ANN model of an auto-associative neural network includes the following five steps:

- (1) Generate the pattern to be stored.
- (2) Train the network with Hebbian learning.
- (3) Initialize the network.
- (4) Update the network.
- (5) Plot the results.

Each step can be programmed with simple instructions, and the example program shown in Table 8.1, which uses a single line for each step above.

Table 8.1 Program ann_fixpoint.m

```

1 pat=2*floor(2*rand(500,10))-1;           % Random binary pattern
2 w=pat*pat';                            % Hebbian learning
3 s=rand(500,1)-0.5;                      % Initialize network
4 for t=2:10; s(:,t)=sign(w*s(:,t-1)); end % Update network
5 plot(s'*pat/500)                         % plot overlaps

```

Line 1 generates 10 patterns (number of columns) for a network with $N = 500$ nodes (number of rows). The required random numbers with values of either -1 or 1 are generated with the MATLAB `rand()` function, which returns uniformly distributed random variables between 0 and 1 . This number is multiplied by 2 before using the `floor()` function, which returns the lower integer part of this number, either 0 or 1 in this case. This number is multiplied by 2 and 1 is subtracted to get the required binary values.

Line 2 is the standard Hebbian rule, as described Section 4.4.2. The difference between this code and the one used in program `weightDistribution.m` (Table 4.1) is that matrix `a` and `b` are now the same since the connections are between nodes in the same layer.

Line 3 sets the initial state of the network to a random value in the range of $-0.5 \leq s < 0.5$. One can also try other initial values, such as using a noisy version of one of the trained patterns.

Line 4 updates the network for nine time steps with a loop. We used the MATLAB `sign()` function, but this can also be programmed with an explicit inequality as $2*(w*s(:,t-1)>0)-1$.

Line 5 plots the results as normalized dot products between the network states at each time step and each of the 10 stored patterns, which produces a plot with 10 lines. Each of these lines illustrates the difference between the current network state and one of the stored patterns.

Each run of the program produces different results, due to the random nature of the stored patterns and the random initial conditions of the network. In the example shown in Fig. 8.4A, the network state recovered one of the imprinted patterns. Sometimes one finds a normalized dot product of -1 , corresponding to the inverse of one of the learned patterns. At other times the networks settles into a state which is not one of the trained patterns. These are spurious states that will be discussed below.

For a simulation of the continuous time model, shown in Fig. 8.4B, we used the same implementation of the dynamic equations as used for the simulations of DNF models in Chapter 7 (the ODE file listed in Table 7.3), to stress the equivalence of these models. The sigmoid function is used as activation function in the ODE file, and the step function is only used in the main program `ann_cont.m` (see Table 8.2) in Lines 12 and 16 to display the overlaps similar

Table 8.2 Program ann_cont.m

```

1  %% Continuous time ANN
2  clear; clf; hold on;
3  nn = 500; dx=1/nn; C=0;
4
5  %% Training weight matrix
6  pat=floor(2*rand(nn,10))-0.5;
7  w=pat*pat'; w=w/w(1,1); w=100*(w-C);
8  %% Update with localised input
9  tall = [] ; rall = [];
10 I_ext=pat(:,1)+0.5; I_ext(1:10)=1-I_ext(1:10);
11 [t,u]=ode45('rnn_ode',[0 10],zeros(1,nn),[],nn,dx,w,I_ext);
12 r=u>0.; tall=[tall;t]; rall=[rall;r];
13 %% Update without input
14 I_ext=zeros(nn,1);
15 [t,u]=ode45('rnn_ode',[10 20],u(size(u,1),:),[],nn,dx,w,I_ext);
16 r=u>0.; tall=[tall;t]; rall=[rall;r];
17 %% Plotting results
18 plot(tall,4*(rall-0.5)*pat/nn)

```

to the previous simulations. This program contains only small modifications compared to the DNF program in Table 7.2. In the simulations here we use 500 nodes as in the discrete model simulations, but the DNF simulations in the previous chapter produce similar results when changing the number of nodes. A scale factor of $\Delta x = 1/N$ is now used, although we compensate somewhat for this by including an amplitude factor of 100 in Line 7. The main difference between the DNF model of the previous chapter and the ANN model discussed here is the set of training patterns. Here, we use random patterns with components $\in \{0, 1\}$, and we subtract the average to use the covariance rule. This produces inhibition so that we set the inhibition constant to zero. The relationship between the covariance learning rule, the inhibition constant, and values of firing thresholds will be discussed later in more detail.

In Line 10, we use a noisy version of the first training pattern as external input until $t = 10\tau$, in which the first 10 bits are reversed. As mentioned above, we changed the activation function in Lines 12 and 13 compared to the DNF model in Chapter 7, although internally the sigmoid function is used as stated above. Finally, we change the states representation to components $\in \{-1, 1\}$ in the argument of the plot function in Line 18 to plot the results similarly to the simulation with the fixpoint model.

8.2.2 Signal-to-noise analysis ◇

It is possible to study the recall abilities of fixpoint networks more formally. The state of the network at each consecutive time step is given by the discrete

dynamics, eqn 8.9, in which a Hebbian-trained weight matrix can be inserted,

$$s_i(t+1) = \text{sign}\left[\frac{1}{N} \sum_j \sum_{\mu} s_i^{\mu} s_j^{\mu} s_j(t)\right]. \quad (8.12)$$

Let us test the network when initialized with one of the trained patterns. Without loss of generality, we can choose $\mu = 1$ for the demonstration. It is then useful to split the terms in the sum over μ into a term for the first training pattern and a second term with the rest of the training patterns,

$$s_i(t+1) = \text{sign}\left[\frac{1}{N} s_i^1 \sum_j s_j^1 s_j(t) + \frac{1}{N} \sum_j \sum_{\mu=2}^{N^{\text{pat}}} s_i^{\mu} s_j^{\mu} s_j(t)\right]. \quad (8.13)$$

The expression $s_j^1 s_j(t)$ in the first term simplifies for the initial condition $s_j(t) = s_j^1$. This product is always one with this choice of training patterns (either 1^2 or $(-1)^2$), and the sum of these ones just cancels the normalization factor N . We then get the expression

$$s_i(t+1) = \text{sign}[s_i(t) + \frac{1}{N} \sum_j \sum_{\mu=2}^{N^{\text{pat}}} s_i^{\mu} s_j^{\mu} s_j(t)]. \quad (8.14)$$

We want the network to be stationary for the trained pattern, and the first term does indeed point in the right direction. We call this term the ‘signal’ part, since it is this part that we want to recover after the updates of the network. The term $\frac{1}{N} \sum_j \sum_{\mu=2}^{N^{\text{pat}}} s_i^{\mu} s_j^{\mu} s_j^1$ describes the influence of the other stored patterns on the state of the network, and is called the *cross-talk term*. This cross-talk term is thought to be analogous to interference between similar memories in a biological memory system. The cross-talk in our formal analysis is a random variable because we used independent random variables s_i^{μ} as training patterns (see Appendix C). The cross-talk term can thus be considered as ‘noise’. The activity of the node i remains unchanged in the next time step as long as the cross-talk term is larger than -1 for $s_i^1 = 1$ or smaller than 1 for $s_i^1 = -1$. The probability of the node changing sign depends on the relative strength of that signal and noise, hence the name of the analysis.

The special case of a network with only one imprinted pattern ($N^{\text{pat}} = 1$) is particularly easy to analyse. In this case, we do not have a cross-talk term so the network stays in the initial state when started with the imprinted pattern. The imprinted pattern is thus a fixpoint of the dynamics of this network,

$$s_i(t+1) = \text{sign}[s_i] = s_i(t). \quad (8.15)$$

What happens if we do not start the network with the trained pattern, but instead start the network simulation with a noisy version of this pattern where some of the components are randomly flipped? In this case, we have to go back to eqn 8.13 and to notice that the sum $\sum_j s_j^{\mu} s_j(t)$ is always positive as long as we change fewer than half of the signs of the initial pattern. We therefore retrieve the learned pattern even when we initialize the network with a moderately noisy version of the trained pattern. The retrieval is also very fast, as it takes only one time step. The pattern will remain stable for all following

time steps. The trained pattern is therefore a *point attractor* of the network dynamics, because initial states close to the trained pattern are attracted by this point in the state space of the network. As an interesting side note, if the number of flipped states is more than half of the number of nodes in the network, then the inverse pattern $s_i = -s_i^1$ is retrieved. Thus, the inverse of a trained pattern is also an attractor of such networks.

Now let us turn to the situation when we train the network on more than one pattern, ($N^{\text{pat}} > 1$). Let us further discuss the case of a pattern component $s_i^1 = 1$. The state of the corresponding node is preserved when the network is initialized with the first pattern, only if the contribution from the cross-talk (noise) term is larger than -1 . The cross-talk term is a random variable because we used random variables s_i^μ as training patterns, as mentioned before. The mean of the random term is zero because the individual components are independent and the mean of the individual components is zero. Thus, we can expect some cases in which some of the many trained patterns are stable. However, the probability of the cross-talk term reversing the state of the node depends on the variance of the noise term. We therefore have to estimate the variance of the cross-talk term.

In our example, we used a training pattern with components that were equally distributed between values -1 and 1 . Each part of the sum in the cross-talk term is therefore an equally distributed number with values -1 or 1 . The sum of such binary random numbers over the training patterns (except the pattern that we used as the initial condition for the network) is a binomially distributed random number with mean zero and standard deviation $\sqrt{N^{\text{pat}} - 1}$. The large sum of such random numbers over the number of nodes in the network is then well approximated by a Gaussian distribution with mean zero and standard deviation $\sqrt{(N^{\text{pat}} - 1)N}$ (see Appendix C). We finally have to take the normalization factor, N , in the cross-talk term into account, and the standard deviation of this ‘noise’ term is therefore,

$$\sigma = \sqrt{\frac{(N^{\text{pat}} - 1)}{N}} \approx \sqrt{\frac{N^{\text{pat}}}{N}} = \sqrt{\alpha}. \quad (8.16)$$

In this equation we have introduced the *load parameter*, $\alpha = N^{\text{pat}}/N$, which specifies the number of trained patterns, relative to the number of nodes in the network.

With knowledge of the probability distribution of the cross-talk term we can specify the probabilities of the cross-talk term changing the activity value of the node. The probability of the cross-talk term producing values less than -1 is illustrated in Fig. 8.5. Such an integral of a Gaussian is given by the *error function* (see eqn C.13),

$$P_{\text{error}} = \frac{1}{2}[1 - \text{erf}\left(\frac{S}{\sqrt{2}\sigma}\right)], \quad (8.17)$$

where S is the strength of the signal (which is equal to one in our case), and σ is the variance of the cross-talk term, as estimated above. To ensure that the probability of the cross-talk term changing sign is less than a certain value,

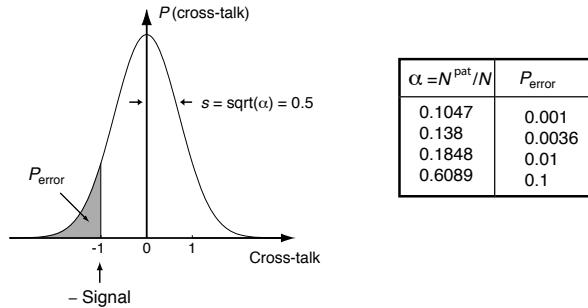


Fig. 8.5 The probability distribution of the cross-talk term is well approximated by a Gaussian with mean zero and variance $\sigma = \sqrt{\alpha}$. The value of the shaded area marked P_{error} is the probability that the cross-talk term changes the state of the node. The table lists examples of this probability for different values of the load parameter α .

that is, $P_{\text{error}} < P_{\text{bound}}$, we need a load parameter less than

$$\alpha < \frac{1}{2[\text{erf}^{-1}(1 - 2P_{\text{bound}})]^2}. \quad (8.18)$$

The expression for the cross-talk term distribution tells us that the probability of the component flipping sign is small when the load parameter of the network is small ($\alpha \ll 1$), that is, if the number of patterns in the training set is much smaller than the number of connections per node in the network ($P \ll N$). In particular, if we train the network on a number of patterns that is equal to 10% of the number of nodes in the network, that is, $P = 0.1 * N$, then the probability that the component will flip signs is less than 0.001. Some other examples are listed in the table in Fig. 8.5.

8.2.3 The phase diagram

The signal-to-noise analysis shows that it is likely that trained patterns are still fixpoints of the network dynamics for a moderate number of trained patterns. On the other hand, an increasing number of training patterns will increase the probability of flipping signs in the activities of nodes. The flipped states can cause further nodes to flip signs in the next time step, and even cause an avalanche effect which leads the network state away from the trained pattern. However, the network is robust for moderate load parameters. Quantitative values for the parameters at which the memory breaks down can only be derived from a more detailed analysis of the dynamics. One way to do this is to use simulations as outlined in Fig. 8.4.

The pattern completion ability of the associative nodes makes the trained patterns *point attractors* of the network dynamics in networks with small load parameters, as demonstrated in Fig. 8.4. In Fig. 8.6 we studied this memory breakdown more systematically. In these simulations, we used a larger network ($N = 1000$) of a continuous time ANN with time constant $\tau = 10$ ms and a larger weight amplitude to allow faster convergence. We monitored the state of the network by calculating the distance, defined by

$$d(\mathbf{a}, \mathbf{b}) = \frac{1}{2} \left(1 - \frac{\mathbf{a}' \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|} \right), \quad (8.19)$$

between the training vector $\mathbf{a} = \mathbf{s}^1$ and the state of the system $\mathbf{b} = \mathbf{s}$. In contrast to the overlap measure used in Fig. 8.4, we here normalized the dot

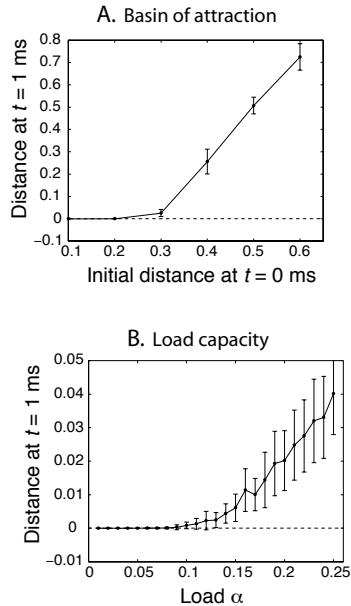


Fig. 8.6 Simulation results for an auto-associative network with continuous time, leaky-integrator dynamics, $N = 1000$ nodes, and a time constant of $\tau = 10$ ms. (A) Robustness to noise pattern recall. Average distance between network state and memory state at $t = 1$ ms as a function of the distance at time $t = 0$ ms, for a fixed number of training patterns ($N_p = 100$). (B) Average distance with different loads for a fixed distance of initial states ($d_0 = 0.01$).

product so that the value is equal to the percentage of changed signs as compared to the training vectors.

In Fig. 8.6A, the network was started with a specific number of reversed components (flipped bits) of one training vector, and the distance of the network to this training vector was measured at $t = 1$ ms (0.1τ). The results demonstrate that the network converges, on average, to a trained pattern if the initial distance is less than a certain value around $d_{BA} \approx 0.3$. The trained pattern is therefore a point attractor under the dynamics of the network with a *basin of attraction* of size d_{BA} in these settings.

In Fig. 8.6B we show the results where the number of training patterns was changed and the network was initialized with a fixed small number (1%) of flipped bits of one training pattern. An analysis of networks with sparse connectivity shows that the relevant load parameter is the number of training patterns relative to the number of connections per node, C ,

$$\alpha = \frac{N^{\text{pat}}}{C}. \quad (8.20)$$

Of course, the number of connections per node is equal to the number of nodes in a fully connected network. The relatively sharp transitions between the domain in which the network can restore a noisy version of a training pattern to its original state, and the domain where the network is not able to retrieve the pattern has the signature of a *phase transition*, which is a transition between domains of a system with different properties. The transition point when the memory breaks down as the load is increased is called the *load capacity*, α_c , of the network.

The load capacity of auto-associative networks can be analysed in much more depth by realizing a useful correspondence of ANN models to so-called *spin models* developed in statistical physics. The binary states of the nodes are interpreted as spins, or little magnets, that can have two orientations, up or down. These ‘magnets’ interact with all the other magnets in the network. Two neighbouring magnets try to align each other in the same direction if the interaction between two magnets is positive (positive weight value). However, *thermal noise* is another force which tends to randomize the direction of the magnets (spins of the nodes). This randomizing force gets stronger with increasing temperature, T . The competition between the magnetic force, which tends to align the magnets, and the thermal force, which tends to randomize the directions, results in a sharp transition between a *paramagnetic* phase, in which there is no dominant direction of the magnets, and a *ferromagnetic* phase, in which there is a dominating direction of the elementary magnets. These phases have very different physical properties, and the transition is therefore called a *phase transition*.

Powerful analytical methods have been developed to describe systems of interacting spins. The situation in auto-associative networks is, however, further complicated by the fact that the force between the nodes is not consistently positive, but is instead somewhat random with a Gaussian distribution with positive and negative weights. Those conflicting forces can result in complicated spin states of the system. Such systems are known as *frustrated systems* or *spin glasses*; the latter alludes to the correspondence with glasses that have

similar properties. Spin glasses are complicated systems and only partially tractable with standard methods of statistical mechanics, such as *mean field theory*. However, a breakthrough mathematical trick called the *replica method* was able to generalize the methods so that these can be applied to spin glasses. Physicists such as *Daniel Amit* have considerably advanced our understanding of attractor neural networks using such methods.

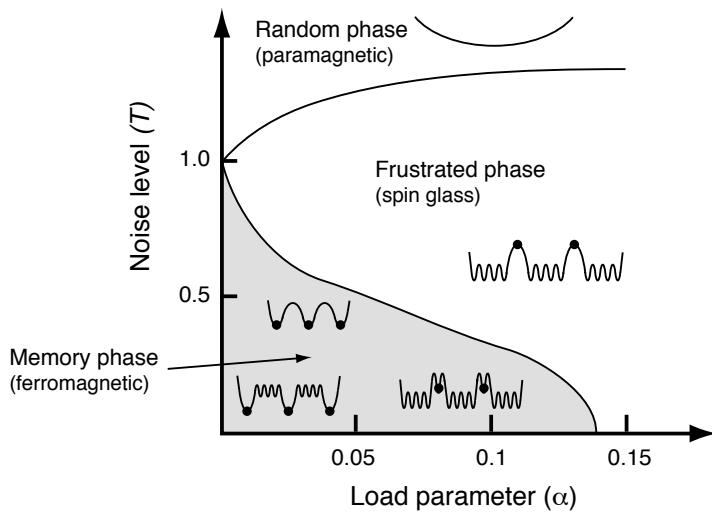


Fig. 8.7 Phase diagram of the attractor network trained on a binary pattern with Hebbian imprinting. The abscissa represents the values of the load parameter $\alpha = N^{\text{pat}}/C$, where N^{pat} is the number of trained patterns and C is the number of connections per node. The ordinate represents the amount of noise in the system. The shaded region is where point attractors proportional to the trained pattern exist. The behaviour of the different phases is indicated with various cartoons of the energy landscape, where the states of training patterns is indicated with dots (see text). [Adapted from Amit, Guttmann and Sompolinsky, *Physical Review Letters* 55: 1530–3 (1995), and Hertz, Krogh, and Palmer, *Introduction to the theory of neural computation*, Addison-Wesley (1991).]

The phase diagram of the ANN model is summarized schematically in Fig. 8.7. This phase diagram outlines the phase boundaries as a function of two parameters, the load parameter, α , on the abscissa and the temperature, T (specifying the noise in the network), on the ordinate. A detailed analysis of such noisy network models shows that the shaded region in the phase diagram is where point attractors exist that correspond to trained patterns. The network in this phase is therefore useful as an associative memory. For vanishing noise, $T = 0$, a transition point to another phase occurs at around $\alpha_c(T = 0) \approx 0.138$. For load parameters larger than this value, the network is in a *frustrated phase* (spin glass phase), in which point attractors of trained memories become unstable. This frustrated phase is reached for smaller values of the load parameter if we include noise in the network. For strong noise the behaviour of the system is mainly random. Simulations have confirmed the validity of the analytical results.

The formal analysis of the network in this section describes the average behaviour of infinitely large networks. This is a good assumption in the sense that networks in the nervous system are comprised of many thousands of highly interconnected neurons. The transitions are less sharp in finite systems of networks with a finite number of nodes. Also, the behaviour of a particular network depends strongly on the specific realization of the training pattern. The phase diagram is therefore specific to the choice of training pattern, which we chose to be random binary numbers. Nevertheless, the example outlines the

general picture that attractor states can be expected in recurrent networks, and that phase transitions to a phase in which the memory system breaks down (*amnesic phase*) is possible under various circumstances. Breakdowns occur only after the network is brought to its limits. A load capacity of $\alpha_c \approx 0.138$ means that over 1000 memories can be stored in a system with nodes receiving 10,000 inputs, as is typical for neurons in the brain. This provides plenty of states that a small patch of cortex could store.

8.2.4 Spurious states and the advantage of noise ◊

It seems that noise would only be destructive for the memory abilities of the associative network. However, this is not entirely the case. To see how noise can help the memory performance of ANN, let us study how a mixture of some trained patterns behaves under the dynamics of the network. For example, let us start the network with a pattern that has the sign of the majority of the first three patterns,

$$s_i(t=0) = s_i^{mix} = \text{sign}(s_i^1 + s_i^2 + s_i^3). \quad (8.21)$$

The state of the node after one update of this node with the discrete dynamics (eqn 7.1) is then

$$\begin{aligned} s_i(t=1) &= \text{sign}\left(\frac{1}{N} \sum_j \sum_{\mu=1}^{N^{\text{pat}}} s_i^\mu s_j^\mu \text{sign}(s_j^1 + s_j^2 + s_j^3)\right) \\ &= \text{sign}\left(\frac{1}{N} \sum_j (s_i^1 s_j^1 + s_i^2 s_j^2 + s_i^3 s_j^3) \text{sign}(s_j^1 + s_j^2 + s_j^3)\right. \\ &\quad \left. + \frac{1}{N} \sum_j \sum_{\mu=4}^{N^{\text{pat}}} s_i^\mu s_j^\mu \text{sign}(s_j^1 + s_j^2 + s_j^3)\right) \end{aligned} \quad (8.22)$$

The last term is again a cross-talk term, which can be shown to be of the same magnitude as the cross-talk term in the case of a trained pattern. The magnitude of the signal term itself can have different values. For example, if the components s_i^1 , s_i^2 , and s_i^3 all have the same value, which happens with a probability of 1/4, then we can pull out this value from the sum in the signal term,

$$\begin{aligned} &\frac{1}{N} \sum_j (s_i^1 s_j^1 + s_i^2 s_j^2 + s_i^3 s_j^3) \text{sign}(s_j^1 + s_j^2 + s_j^3) \\ &= s_i^1 \frac{1}{N} \sum_j (s_j^1 + s_j^2 + s_j^3) \text{sign}(s_j^1 + s_j^2 + s_j^3). \end{aligned} \quad (8.23)$$

The components in the remaining sum of the signal term can have various values as can be seen from the possible combinations of components as listed in Table 8.3. We get a contribution of 3 if the signs of all three nodes s_j^1 , s_j^2 , and s_j^3 are the same. This happens again with probability 1/4. The remaining 3/4 of the time we get a contribution of 1. On average we therefore get a signal that is 3/2 of the signal of a trained pattern. However, the signal term

Table 8.3 The possible states of three binary nodes and their summed value

s^1	s^2	s^3	$s^1 + s^2 + s^3$
1	1	1	3
1	1	-1	1
1	-1	1	1
1	-1	-1	-1
-1	1	1	1
-1	1	-1	-1
-1	-1	1	-1
-1	-1	-1	-3

is different when one sign of the nodes s_i^1 , s_i^2 , or s_i^3 is different from that of the other ones, which happens with probability 3/4. For example, if s_i^3 has a different sign from s_i^1 and s_i^2 , we can write the signal term as

$$\begin{aligned} \frac{1}{N} \sum_j (s_i^1 s_j^1 + s_i^2 s_j^2 + s_i^3 s_j^3) \text{sign}(s_j^1 + s_j^2 + s_j^3) \\ = s_i^1 \frac{1}{N} \sum_j (s_j^1 + s_j^2 - s_j^3) \text{sign}(s_j^1 + s_j^2 + s_j^3). \end{aligned} \quad (8.24)$$

The average values of the remaining sum can be evaluated as before, showing that this is equal to 1/2. We conclude that we have, on average, a signal that has the strength of

$$\frac{1}{4} * \frac{3}{2} + \frac{3}{4} * \frac{1}{2} = \frac{3}{4}$$

times the signal when updating a trained pattern. This indicates that there can be mixture states of the trained patterns that are also attractors in the system. These are an example of what is called a *spurious state* in the network; attractors that are different from the pattern used to train the network. Those strange memory states have been termed *schizophrenic* states to allude to apparent memory recalls with strange content. The analysis here shows the possibility that under certain conditions some memory recalls are contaminated by other memory states, even more than through the usual cross-talk term analysed above.

We found that the average strength of the signal for the spurious states is less than the average strength for trained patterns. This makes the spurious states, under normal conditions, less stable than attractors related to trained patterns. This can be used to our advantage by forcing the system out of the spurious states should it be attracted by an initial pattern close to them. We can do this by introducing noise into the system. This can be done in various ways, as we discussed in Chapter 3. With an appropriate level of noise we can kick the system out of the basin of attraction of some spurious states and into the basin of attraction of another attractor. It is likely that the system will

then end up in a basin of attraction belonging to a trained pattern as these basins are often larger for moderate load capacities of the network. Noise can thus help to destabilize undesired memory states.

8.2.5 Noisy weights and diluted attractor networks

Fig. 8.6 shows that ANN memories are remarkably robust to noise, both in terms of recall of noisy patterns (Fig. 8.6A), or to noise (cross-talk) produced by an additional pattern (Fig. 8.6B). The curves show that there is perfect recall for a wide range of noise. However, there is a sudden and extensive breakdown when the noise level passes a critical value. Such sharp transitions are a consequence of the attractor dynamics in such networks.

We can also ask how robust the network is to noise in the weight matrix, deletion of synapses, or the loss of whole nodes. Results of corresponding simulations are shown in Fig. 8.8. In these simulations of the fixpoint model, we used 1000 nodes, and the memory of a stored pattern was tested when the ANN was initialized with pattern that had 10 components flipped compared to a stored pattern. The graphs show the mean and standard deviation over 100 runs.

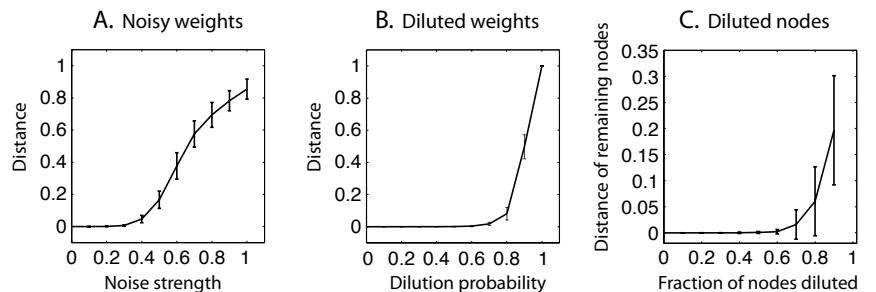


Fig. 8.8 Simulation results for a fixpoint ANN with 1000 nodes, which was trained on 50 patterns and tested on initial states of a stored pattern with 10 flipped bits. Error bars show standard deviations. (A) Mean distance between network state and stored pattern after 10 updates with different levels of static noise in the weight matrix. (B) Mean distance with diluted weight matrices. The abscissa gives the probability that a weight value was set to zero. (C) Mean distance with a fraction of nodes set to zero.

Fig. 8.8a shows the robustness and breakdown of the memory model when adding static Gaussian noise to the weight matrix. The abscissa denotes the strength of this noise and is in units of the variance of the weight matrix. *Haim Sompolinsky* showed that this model has a similar phase diagram to the one shown in Fig. 8.7 with the temperature replaced by the noise strength. Fig. 8.8b shows how robust the system is to deleting synapses. Indeed, a very high percentage of synapses have to be destroyed before the system breaks down, although this breakdown is then rapid. Of course, the brain is not fully connected as the model used here as baseline. These simulations demonstrate that attractor networks are also possible in much less connected networks. Finally, the large robustness and sudden breakdown also holds true when deleting

whole nodes. Of course, the nodes which are removed can not be compared to the memory states, so the graph depicts the percentage error in the remaining nodes. Again, a very high percentage of nodes have to be removed before the attractor breaks down.

8.3 Sparse attractor networks and correlated patterns

The load capacity for the noiseless ANN model with standard Hebbian learning on random binary patterns is $\alpha_c \approx 0.138$. These training patterns are, on average, uncorrelated. The sensory signals driving the learning in our brains are, on the contrary, often correlated in some way. For example, the visual patterns on the retina are correlated across space, and specific images (for example, a fish) are commonly seen in conjunction with other images (for example, water). Correlations between the training patterns worsens the performance of the network, since the cross-talk term can yield high values in this case. A solution to this problem is to use a preprocessing step in which the representations of the training pattern get modified to yield orthogonal patterns. Orthogonal patterns have the property that the dot product between them is zero,

$$\mathbf{s}^\mu \mathbf{s}^\nu = \delta^{\mu\nu}. \quad (8.25)$$

We used thereby the *Kronecker symbol* defined by

$$\delta^{\mu\nu} = \begin{cases} 1 & \text{if } \mu = \nu \\ 0 & \text{otherwise} \end{cases}. \quad (8.26)$$

The cross-talk term for such patterns is exactly zero, so the network can store up to C patterns, that is, $\alpha_c=1$.

One can use this fact to maximize the storage capacity by minimizing the average overlap between the patterns. A typical engineering approach would be to calculate a matrix with elements representing the dot product between the patterns in the training set,

$$Q_{\mu\nu} = \sum_i s_i^\mu s_i^\nu. \quad (8.27)$$

For linearly independent training vectors, we can invert this matrix and use it in the learning rule to orthogonalize the patterns. This learning rule is known as the *pseudo-inverse* method,

$$w_{ij} = \frac{1}{N} \sum_{\mu\nu} s_i^\mu (\mathbf{Q}^{-1})_{\mu\nu} s_j^\nu, \quad (8.28)$$

which results in a storage capacity of $\alpha_c = 1$. However, calculating the inverse of such a matrix is biologically not very plausible, although we can still use this learning rule in simulations as long as we can argue that the orthogonalization can be implemented by biological means.

Reaching the optimal storage capacity is not necessarily what is needed in biological systems. Using a random remapping of correlated patterns is thus a

possible mechanism to handle correlated patterns in attractor networks. This can be combined with remapping to a sparser representation which will further reduce overlaps between patterns.

8.3.1 Sparse patterns and expansion recoding

Decreasing the cross-talk between stored patterns, such as by using sparse patterns, increases the storage capacity of associative networks. Indeed, the brain seems to use this strategy with the help of *expansion recoding*. An example illustrating the principle of expansion recoding using a single-layer perceptron is shown in Fig. 8.9. The system has two input channels that can have four different combinations of input values if we restrict ourselves to binary patterns, namely $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$. The perceptron has four output nodes, and we have included a variable firing threshold with a bias in the activation function implemented by an additional input node with constant activation value (see Chapter 6). In the figure, we specified an example of the weight values for which a network with threshold output nodes transforms the initial pattern representation into an orthogonal representation. All four patterns can therefore be stored in a subsequent auto-associative memory network with only four nodes.

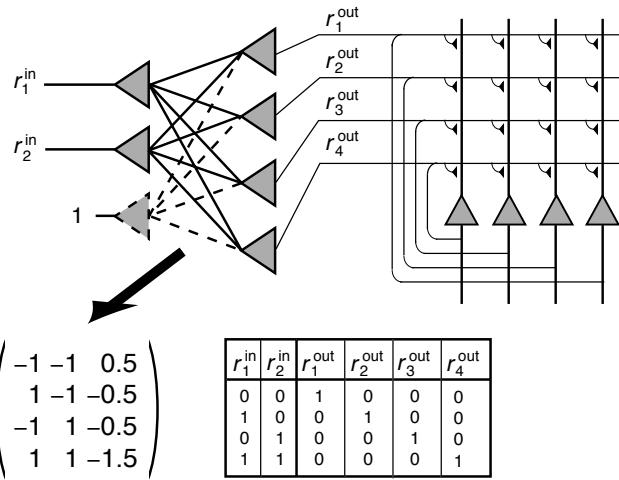


Fig. 8.9 Example of expansion recoding that can orthogonalize a pattern representation with a single-layer perceptron. The nodes in the perceptron are threshold units, and we have included a bias as a separate node with constant input. The orthogonal output can be fed into a recurrent attractor network where all inputs are fixpoints of the attractor dynamics.

Expansion recoding can also be realized with competitive networks. Common to the expansion schemes is that the number of nodes representing a pattern is expanded, while at the same time the representation is made more sparse. We mentioned already that sparse codes are thought to be used in the hippocampus, and these codes might be produced by expansion recoding in the dentate gyrus which has a large number of granule cells. Expansion coding might also be at work in the cerebellum, where mossy fibres contact a large number of granule cells, which exceed, by far, the number of mossy fibres.

Expansion recoding indicates that the load capacities of attractor networks can be larger for patterns with sparse representations. Alessandro Treves and

Edmund Rolls analysed the storage capacity of attractor networks with such sparsely distributed random patterns and found, in the limit of small a and large networks, a storage capacity of

$$\alpha_c \approx \frac{k}{a \ln(1/a)}, \quad (8.29)$$

where k is a constant that depends weakly on some details of the network and the pattern representations, but is roughly on the order of 0.2–0.3. The number of patterns in a pattern set with sparseness $a = 0.1$ that can be stored in a attractor network with nodes that have 10,000 synapses can therefore exceed 20,000, more than enough for many memory requirements in small brain areas. Note, however, that the information content does not change. The enhanced storage capacity of the network has to be compared with the reduction of the amount of information that can be stored in a sparse representation compared to that in a representation with more active components. The information is proportional to $a \ln(1/a)$, the denominator in eqn 8.29. The amount of information that can be stored in the network stays approximately constant.

We have only analysed weight matrices produced with basic Hebbian correlation rules. There are many other possible learning algorithms which can produce different weight matrices. It is interesting to ask what would be the best possible solution for a weight matrix. To be more precise, if we have a certain training set, we can ask what the load capacity of the network is, with a weight matrix that was produced with the optimal learning rule. To answer this question, we have to try out all the possible weight matrices, which is, of course, a daunting task. However, methods from statistical physics can help us to get the answer to this question, as shown by *Elizabeth Gardner*. She found that the maximal storage capacity of auto-associative networks with binary patterns and sparse representations is:

$$\alpha_c = \frac{1}{a \ln(1/a)}. \quad (8.30)$$

Comparing this result with sparse Hebbian networks (eqn 8.29) shows that the simplest Hebbian rule comes close to giving the maximum value.

8.3.2 Control of sparseness in attractor networks

The investigation of the properties of attractor networks with sparse patterns is important when studying brain functions. Indeed, the ANN models used so far will not work with sparse patterns since the network dynamics try to make half of the nodes active, as shown below. The important question is how one can ensure that the sparseness of retrieved states, a^{ret} , has the sparseness of training patterns, a . There are several possible solutions to the problem of achieving the right retrieval sparseness. One is to adjust the firing thresholds of the nodes appropriately so that only a nodes can fire in the retrieval process. However, adjusting the thresholds of neurons with respect to the firing of all the nodes in the networks is a non-local operation that has to be carefully implemented to be biologically plausible.

Another possibility, already illustrated in Chapter 7, is to include additional inhibition, on top of that produced by the Hebbian covariance rule, to control

Table 8.4 The contributions of the four possible firing patterns of pre- and postsynaptic firing rates to the Hebbian covariance matrix, and the probability of the occurrence of these patterns for training sets with patterns of sparseness a

r_i	r_j	\tilde{w}	$P(\tilde{w})$
0	0	a^2	$(1-a)^2$
0	1	$-a(1-a)$	$a(1-a)$
1	0	$-a(1-a)$	$a(1-a)$
1	1	$(1-a)^2$	a^2

the overall activity in the network. Let us demonstrate this with binary patterns that have components 0 and 1, where 0 indicates no firing and 1 indicates firing of the node. The sparseness of the pattern is then given by the number of nodes that are active. The patterns are imprinted with a Hebbian rule

$$w_{ij} = \frac{1}{\sqrt{N^{\text{pat}}}} \sum_{\mu} (r_i^{\mu} - a)(r_j^{\mu} - a) - C. \quad (8.31)$$

Each pattern contributes four possible combinations of pre- and postsynaptic firing to the weight values. To shorten the presentation, let us use the short form $\tilde{w} = (r_i^{\mu} - a)(r_j^{\mu} - a)$. The four possible values for \tilde{w} are listed in Table 8.4, together with their respective probabilities. The mean of each contribution is

$$\langle \tilde{w} \rangle = \sum \tilde{w} P(\tilde{w}) \quad (8.32)$$

$$= a^2 - (1-a)^2 - 2a^2(1-a)^2 + a^2(1-a)^2 = 0, \quad (8.33)$$

and the variance is

$$\langle \tilde{w}^2 \rangle = \sum \tilde{w}^2 P(\tilde{w}) = a^2(1-a)^2. \quad (8.34)$$

The mean and the variance of the weight distribution after imprinting a large number N^{pat} of patterns, using the central limit theorem and taking the normalization and inhibition into account, is

$$\langle w \rangle = -C \quad (8.35)$$

$$\langle w^2 \rangle = a^2(1-a)^2. \quad (8.36)$$

If this weight matrix is used with an iterative rule for updating the states of the system,

$$r_i = \Theta(h_i) = \Theta\left(\sum_j w_{ij} r_j - \theta\right), \quad (8.37)$$

where Θ is the step gain function and θ is the firing threshold, the stationary state must obey the self-consistent condition,

$$\frac{P(h_i > 0)}{P(h_i < 0)} = \frac{a^{\text{ret}}}{1 - a^{\text{ret}}}. \quad (8.38)$$

Since a^{ret} nodes are active, the probability density of the net input, $P(h)$ is a Gaussian with mean $-Ca^{\text{ret}}$ and variance $\sigma^2 = a^2(1-a)^2a^{\text{ret}}$, as illustrated in Fig. 8.10. Thus,

$$P(h_i > 0) = \frac{1}{2} - \text{erf}\left(\frac{Ca^{\text{ret}} + \theta}{\sqrt{2}\sigma}\right), \quad (8.39)$$

$$P(h_i < 0) = \frac{1}{2} + \text{erf}\left(\frac{Ca^{\text{ret}} + \theta}{\sqrt{2}\sigma}\right), \quad (8.40)$$

and the appropriate inhibition constant, or threshold, can be calculated from

$$a^{\text{ret}} = \frac{1}{2} - \text{erf}\left(\frac{Ca^{\text{ret}} + \theta}{\sqrt{2}\sigma}\right). \quad (8.41)$$

The equation seems to indicate that a fine tuning of parameters is necessary. However, we have not taken the attractor dynamics of the stored patterns into account. These will bias the networks to the desired sparseness once the inhibition (or threshold) brings the network activity into the right range. Simulation results for a network of 500 nodes, trained on 40 patterns with sparseness $a = 0.1$, are shown in Fig. 8.11. The two curves show the average retrieval sparseness and the average Hamming distance, respectively, and standard deviations within 10 runs are indicated with error-bars. The threshold was thereby set to $\theta = 0$, and the curves show results for different values of the inhibition constant C .

Simulation

The curves in Fig. 8.11 were produced with the program listed in Table 8.5. Some network parameters are set in Line 3, including the number of nodes (`nn`), the sparseness (`a`), and the number of patterns to be learned (`npat`). Results are averaged over 10 runs with the loop that starts at Line 4. The sparse patterns are produced by setting, at first, all pattern values to zero in Line 5. In Line 6 we produce, for each pattern vector, a random permutation of the indices to the components, and set the components for the first `a*nn` random components to one. Line 7 is the standard Hebbian covariance rule with a standard normalization.

The experiments with different inhibition constants `c` start at Line 8. The index `ic` is used to store the results for runs with different inhibition constants. The network is started with a noisy version of the first pattern produced in Line 9, where the first 10 components have been flipped. The network is then updated 10 times in Line 10. Line 12 calculates the retrieval sparseness. Since node activities are either 0 or 1, this can be done by summing the components and dividing by the number of nodes. Line 12 calculates the normalized Hamming distance, which is the number of different bits between the network state and the stored pattern relative to the total number of bits in each pattern. The results are plotted in Lines 15 and 16 with error bars.

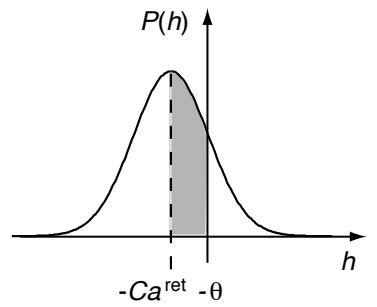


Fig. 8.10 A Gaussian function centred at a value $-Ca^{\text{ret}}$. Such a curve describes the distribution of Hebbian weight values trained on random patterns and includes some global inhibition with strength value C . The shaded area is given by the Gaussian error function described in Appendix C.

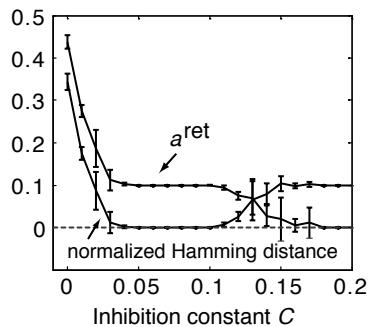


Fig. 8.11 Simulation of fixpoint ANN for pattern with sparseness $a = 0.1$.

Table 8.5 Program ann_sparse.m

```

1  %% Sparse auto-associative network
2  clear; clf; hold on;
3  nn=500; a=0.1; npat=40;
4  for irun=1:10;
5      pat=zeros(nn,npat);
6      for i=1:npat; idx=randperm(nn); pat(idx(1:a*nn),i)=1; end
7      w=(pat-a)*(pat-a)'; w=w/sqrt(npat); ic=0;
8      for c=0:0.01:0.2; ic=ic+1; cc(ic)=c;
9          s=pat(:,1); s(1:10)=1-s(1:10);
10         for t=1:10; s=((w-c)*s)>0; end    % Update network
11         aret(ic,irun)=sum(s)/nn;
12         hd(ic,irun)=((1-s)'*pat(:,1)+s'*(1-pat(:,1)))/nn;
13     end
14 end
15 errorbar(cc,mean(aret'),std(aret'));
16 errorbar(cc,mean(hd'),std(hd'));

```

8.4 Chaotic networks: a dynamic systems view ◇

This section contains a short excursion into the theory of dynamic systems which will show us why we have called the memory states of auto-associative memories ‘point attractors’, and which will give us further insight into the possible behaviour of recurrent networks. In particular, we will outline the conditions under which a recurrent network has point attractors. We will also show that recurrent networks with biologically more plausible, non-symmetric weight matrices, in comparison to the symmetric weight matrices resulting from simplified Hebbian learning, frequently have properties similar to those of the Hebbian counterpart.

We have already stressed the dynamic nature of the models when recurrences are involved. In general, models with continuous dynamics are described in dynamic systems theory by a set of coupled, ordinary, first-order differential equations, the so-called *equations of motion*,²

$$\frac{dx}{dt} = f(x). \quad (8.42)$$

²Higher-order ordinary differentials can be cast into a set of coupled first-order differential equations.

This is a coupled set of equations that only appears to be a single equation since we have used a compact vector notation. The dynamics of a recurrent network with continuous dynamics (eqn 8.1) is a special form of eqn 8.42, and an auto-associative network is therefore formally a dynamic system. Recurrent networks with discrete dynamics, for example the systems specified by eqn 7.1, are still dynamic systems, but with discrete dynamics.

It is useful to know some of the basic terminology of dynamic systems theory.

The number of equations, that is, the number of nodes in the network, define the *dimensionality* of the systems, and recurrent neural networks must therefore be considered as high-dimensional dynamic systems. The vector \mathbf{x} in eqn 8.42 is called a *state vector*, and a set of values for all components is called a *state*. A state describes a point within the high-dimensional *state space*, the space of all possible state values. The evolution of the state, defined by the equations of motion, describes a *trajectory*, a path in state space.

8.4.1 Attractors

Dynamic systems can display a variety of different dynamic behaviour. We have already seen some examples of recurrent networks in which the networks converged to a fixpoint of the dynamic equations. This is a point in the state space, and it is for this reason that we called this point a *point attractor*. Other forms of attractors are also possible in dynamic systems. For example, the attractor can be a loop within the state space, a so-called *limit cycle*, in which the system cycles through a continuous set of points. Such movements in state space would appear in the components as oscillations, and oscillations in the brain may well be described by such types of attractors of neural networks. We can also define the dimensionality of an attractor. For example, a line attractor has a dimensionality of one, and a point in the state space has a dimensionality of zero. Higher-dimensional attractors are also possible in dynamic systems, although it is often difficult to find corresponding regularities in the movements of the system.

So far, we have only considered regular movements of dynamic systems. However, we know of examples of dynamic systems that display movements that are not completely regular, but yet are also not completely stochastic (like noise). For example, a system can be attracted by two points in the phase space with irregular domination of these two attractor points. A popular example of a chaotic system is the *Lorenz system* defined by the equations of motion

$$\frac{dx_1}{dt} = a(x_2 - x_1) \quad (8.43)$$

$$\frac{dx_2}{dt} = x_1(b - x_3) - x_2 \quad (8.44)$$

$$\frac{dx_3}{dt} = x_1x_2 - cx_3. \quad (8.45)$$

Note that we can write this system as a recurrent network of three nodes with sigma and sigma-pi couplings such as

$$\frac{dx_i}{dt} = \sum_j w_{ij}^1 x_i + \sum_{jk} w_{ijk}^2 x_j x_k \quad (8.46)$$

with

$$\mathbf{w}^1 = \begin{pmatrix} -1 & a & 0 \\ b & -1 & 0 \\ 0 & 0 & -c \end{pmatrix} \quad \text{and} \quad \mathbf{w}^2 = \begin{cases} w_{213}^2 = -1 \\ w_{312}^2 = -1 \\ 0 \text{ otherwise} \end{cases}. \quad (8.47)$$

An example of a trajectory of the Lorenz system, showing the famous Lorenz attractor, is plotted in Fig. 8.12. With the above-mentioned definition of the

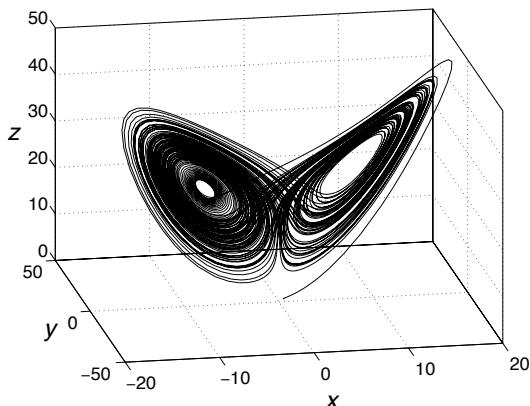


Fig. 8.12 Example of a trajectory of the Lorenz system from a numerical integration within the time interval $0 \leq t \leq 100$. The parameters used were $a = 10$, $b = 28$, and $c = 8/3$.

dimensionality of an attractor, we must consider two points to have a dimensionality larger than zero and less than one. It then becomes a question of how to define such fractional dimensionalities, although we will not go into these discussions here. For our purposes it is enough to realize that there can be fractional dimensional attractors, from which the term *fractals*, often mentioned in the dynamic systems literature, is derived.

The movement around fractal attractors is very irregular, yet there is still some order. Such movements are therefore very different from random movements, as such movements are dominated by noise. We call such behaviour *deterministic chaos*. It is deterministic because there is no noise in the equations and the future states of the system are uniquely defined by the initial conditions. In contrast, *stochastic* systems include noise, so that systems that start with identical initial conditions can evolve in different ways.

Simulation

Fig. 8.12 was produced with the program given in Tables 8.6 and 8.7. The numerical integration is done with the higher-order Runge–Kutta algorithm. The only new function in this program is only the use of the MATLAB function `plot3()`, which can be used to draw lines in three-dimensional space (or, to be precise, two-dimensional projections of the curves in three-dimensional space).

8.4.2 Lyapunov functions

³ While chaotic fluctuations are not desirable within the basic ANN models discussed here, chaotic networks may have useful properties that are employed by the brain. For example, EEG measurements of normal brain activity indicate a chaotic brain dynamic on a system level, while synchronized modes seem to occur in epileptic seizures.

In this chapter, we have outlined that point attractors of recurrent networks are useful as memories, and chaotic fluctuations in such systems are not normally desirable.³ It is of considerable interest to examine under which conditions recurrent networks have point attractors, and under which conditions dynamic, chaotic behaviour is expected.

From dynamic systems theory, we know that a system has a point attractor if a *Lyapunov function* exists. The idea behind this statement can be illustrated with the help of Fig. 8.13. We illustrate there a ‘landscape’ in which a ball,

Table 8.6 Program lorenz.m

```

1 %% Plot trajectory of Lorenz system
2 clear; clf;
3 a=10; b=28; c=8/3;
4 u0 = zeros(3,1)+0.5; param=0; tspan=[0,100];
5 [t,u]=ode45('lorenz_ode',tspan,u0,[],a,b,c);
6 plot3(u(:,1),u(:,2),u(:,3))

```

Table 8.7 Function lorenz_ode.m

```

1 function udot=lorenz_ode(t,u,flag,a,b,c)
2 % odefile for lorenz system
3 udot(1)=a*(u(2)-u(1));
4 udot(2)=u(1)*(b-u(3))-u(2);
5 udot(3)=u(1)*u(2)-c*u(3);
6 udot=udot';
7 return

```

driven by gravity and influenced by friction, can roll down a hill into a valley. The ball will ultimately come to a halt at the minimum (the valley) of the function describing the landscape. More formally, if there is a function $V(\mathbf{x})$ that never increases under the dynamics of the system,

$$\frac{dV(\mathbf{x})}{dt} \leq 0, \quad (8.48)$$

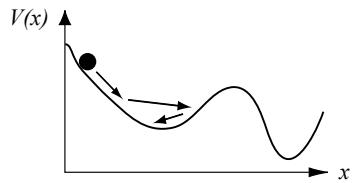
where the \mathbf{x} is governed by the dynamic equations of the system (eqn 8.42), then there has to be a point attractor in the system (as long as the state space is bounded), corresponding to the minimum of the function V . If such a function exists with the required properties, then this function is called a *Lyapunov function* in dynamic systems theory and an *energy function* in physics.

Can we find such a function for the recurrent networks? The answer is that we know of a function that fulfils the conditions under certain circumstances. Let us illustrate this for the special case of the discrete dynamics, eqn 7.1,

$$h_i(t+1) = \sum_j w_{ij} r_j(t) + I_i^{\text{ext}}(t). \quad (8.49)$$

For this system, we propose to study the following function,

$$V(r_1, \dots, r_N) = -\frac{1}{2} \sum_i \sum_j w_{ij} r_i r_j - \sum_i I_i^{\text{ext}} r_i. \quad (8.50)$$

**Fig. 8.13** A ball in an ‘energy’ landscape.

The change of this function in one time step is given by

$$\begin{aligned}\Delta V &= V(t+1) - V(t) \\ &= -\frac{1}{2} \sum_k \sum_j w_{kj} r_k(t+1) r_j(t+1) + \frac{1}{2} \sum_k \sum_j w_{kj} r_k(t) r_j(t) \\ &\quad - \sum_k I^{\text{ext}}[r_k(t+1) - r_k(t)].\end{aligned}\quad (8.51)$$

It is easiest to consider this model with sequential updates. In this case, when the i th node is updated, the other nodes stay constant, that is, $r_k(t+1) = r_k(t)$ for $k \neq i$. Only terms from node i contribute to the change of the function V , and the change of this function at this time is given by

$$\begin{aligned}\Delta V &= -\frac{1}{2} r_i(t+1) \sum_{j \neq i} w_{ij} r_j(t) - \frac{1}{2} r_i(t+1) \sum_{k \neq i} w_{ki} r_k(t) \\ &\quad + \frac{1}{2} r_i(t) \sum_{j \neq i} w_{ij} r_j(t) + \frac{1}{2} r_i(t) \sum_{k \neq i} w_{ki} r_k(t) - I_i^{\text{ext}}[r_i(t+1) - r_i(t)] \\ &= -[r_i(t+1) - r_i(t)][\sum_{j \neq i} \left\{ \frac{1}{2} (w_{ij} + w_{ji}) r_j(t) \right\} + I_i^{\text{ext}}].\end{aligned}\quad (8.52)$$

This difference is zero if $r_i(t+1) = r_i(t)$, but this would mean that we have already reached a stationary state. The most interesting case occurs when $r_i(t+1) \neq r_i(t)$. Then we have to inspect the second term more carefully, and we notice that the result depends on the weight matrix. Let us first examine the case of Hebbian learning that results in a symmetrical weight matrix, $w_{ij} = w_{ji}$. In this case, we have $(w_{ij} + w_{ji})/2 = w_{ij}$, and the second factor in the last equation equals $h_i(t)$ according to eqn 8.49. We can therefore write eqn 8.52 as

$$\Delta V = -[r_i(t+1) - r_i(t)]h_i(t). \quad (8.53)$$

Let us consider again a system with binary states. If $r_i(t) = 1$, then it must be true that $h_i(t) < 0$ in order to have $r_i(t+1) = -1$. If $r_i(t) = -1$ then it must be true that $h_i(t) > 0$ in order to have $r_i(t+1) = 1$. In both cases we have $\Delta V < 0$ and hence a Lyapunov function. The system therefore always converges to a stable state, each of which corresponds to the trained patterns, as we saw in the last section.

8.4.3 The Cohen–Grossberg theorem

Michael Cohen and *Stephen Grossberg* studied more general systems with continuous dynamics of the form

$$\frac{dx_i}{dt} = -a_i(x_i) \left(b_i(x_i) - \sum_{j=1}^N (w_{ij} g_j(x_j)) \right) \quad (8.54)$$

with functions a_i and b_i . This dynamic equation corresponds to the leaky integrator dynamics (eqn 8.1) with generalizations such that the time constants and the activation functions can be different for each individual node in the

network. Cohen and Grossberg found a Lyapunov function under the conditions that

- (1) **Positivity** $a_i \geq 0$: The dynamics must be a leaky integrator rather than an amplifying integrator.
- (2) **Symmetry** $w_{ij} = w_{ji}$: The influence of one node on another has to be the same as the reverse influence.
- (3) **Monotonicity** $\text{sign}(dg(x)/dx) = \text{const}$: The activation function has to be a monotonic function.

The statement that under these conditions a Lyapunov function exists has come to be known as the *Cohen–Grossberg theorem*. This theorem proves the existence of point attractors of the noiseless recurrent networks under the conditions just outlined.

8.4.4 Asymmetrical networks

Synaptic weights between neurons in the nervous system cannot be expected to fulfil the condition of weight matrix symmetry required to guarantee stable attractors in these networks. Also, we have studied so far weight matrices that have arbitrary negative and positive values. Positive weight values represent excitatory synapses, while negative values represent negative synapses. Neurons receive a mixture of input from excitatory and inhibitory presynaptic neurons, but each class of neurons often makes only one type of synaptic contact,⁴ which defines the neuron type as either excitatory or inhibitory. This corresponds to a weight matrix in which the signs within each column have to be consistent (all the elements in a column, corresponding to the synaptic efficiencies of one presynaptic neuron, are either positive or negative). This violates the symmetry condition of the weight matrix in the Cohen–Grossberg theorem, since an inhibitory node could then only receive inhibitory connections and vice versa. However, the Cohen–Grossberg theorem only describes the special case in which we can prove that networks have point attractors, while it is still possible that networks have point attractors, even if the conditions for the Cohen–Grossberg theorem are not fulfilled. We will now test how networks that violate some of the conditions of the Cohen–Grossberg theorem behave.

Let us start by studying a particularly simple case of non-symmetrical weight matrices. Each matrix can be decomposed into a symmetrical and an antisymmetrical part,

$$\mathbf{w} = g^s \mathbf{w}^s + g^a \mathbf{w}^a, \quad (8.55)$$

where

$$w_{ij}^s = w_{ji}^s \quad (8.56)$$

$$w_{ij}^a = -w_{ji}^a, \quad (8.57)$$

and g^s and g^a are parameters. These parameters allow us to study network conditions while varying the strength of the symmetrical and antisymmetrical

⁴This is called Dale’s principle, which is not always observed. For example, there are two types of dopamine receptors, D1 and D2, in spiny neurons of the caudate nucleus, which have opposite effects on EPSPs.

parts of the weight matrix. A simple example of symmetrical and antisymmetrical matrices is to set the magnitude of all components of \mathbf{w}^s and \mathbf{w}^a equal to one,

$$w_{ij} = \begin{cases} g^s + g^a & \text{for } i > j \\ 0 & \text{for } i = j \\ g^s - g^a & \text{for } i < j \end{cases} \quad (8.58)$$

Let us study by simulations how networks with such matrices behave. We will use a network with dynamics specified by eqn 7.1 using a sigmoidal activation function. To get an indication as to whether the networks have reached a steady state (point attractor) we measure the square difference of the states of two consecutive time steps of the network,

$$d(t) = (|\mathbf{r}(t)| - |\mathbf{r}(t-1)|)^2. \quad (8.59)$$

This distance is zero if the network has converged to a point attractor, and positive otherwise. The values of this convergence indicator are plotted on a grey scale for different values of strength values, g^s and g^a , in Fig. 8.14A. The value is zero for symmetrical weight matrices, as predicted by the Cohen–Grossberg theorem (horizontal line at $g^a = 0$). What is especially interesting is that a stationary state is reached as long as the strength of the asymmetrical part of the weight matrix is smaller than the strength of the symmetrical part of the weight matrix. This indicates that only strong asymmetries destroy point attractors in the network.

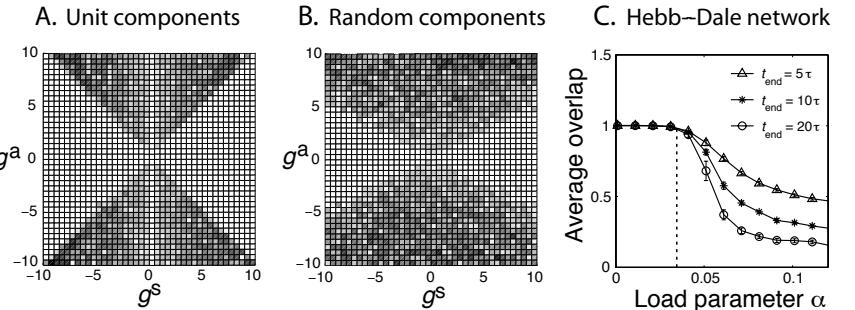


Fig. 8.14 (A) Convergence indicator for networks with asymmetrical weight matrices where the individual components of the symmetrical and antisymmetrical matrix are of unit strength. (B) Similar to (A) except that the individual components of the weight matrix are chosen from a Gaussian distribution. (C) Overlap of the network state with a trained pattern in a Hebbian auto-associative network that satisfies Dale’s principle.

Of course, weight matrices with components of unit strength are a drastic simplification of biological systems. In Fig. 8.14B we show another example where the components in the symmetrical and antisymmetrical matrices were chosen from a Gaussian distribution, which is typical for weight matrices trained with Hebbian rules (see Section 4.4.2). The results are similar to the previous experiment in that the asymmetry has to be quite strong before the network displays signs of the onset of chaos. Strong asymmetries are only achieved if an excitatory connection from one neuron to another is countered by strong inhibition of the other neuron, or vice versa.

The next experiment was performed to test networks with separate excitatory and inhibitory nodes, which comply with Dale’s principle and still utilize basic Hebbian imprinting. We simulated networks with weight matrices chosen by the following procedure. We first generated a symmetrical weight matrix through

Table 8.8 Program ann_chaos.m

```

1 %% Assymetric network model
2 clear; nodes=251; dt=.1;
3 %wa=ones(nodes); ws=ones(nodes);      %unit
4 wa=randn(nodes); ws=randn(nodes);    %random
5 for i=2:nodes; for j=1:i; wa(i,j)=-wa(j,i); end; end
6 for i=2:nodes; for j=1:i; ws(i,j)=ws(j,i); end; end
7 for i=1:nodes; wa(i,i)=0; ws(i,i)=0; end
8
9 for a=1:41; gs=(a-21)*.5;
10 for b=1:41; ga=(b-21)*.5; disp([gs,ga]);
11 w=gs*ws+ga*wa;
12 u=2*rand(nodes,1)-1;
13 for t=0:dt:100;
14     s=tanh(u); normu_prev=norm(u);
15     u=(1-dt)*u+dt*w*s;
16 end %loop over time steps
17 xgs(a)=gs; xga(b)=ga;
18 u_dif(a,b)=abs(norm(u)-normu_prev);
19 end %gs
20 end %ga
21 surf(xgs,xga,u_dif'); view(0,90)

```

Hebbian learning. We also generated another matrix of the same size, in which we randomly assigned excitatory and inhibitory nodes. We then deleted (set to zero) all the entries in the weight matrix that were either inconsistent with the nature of the node (to be inhibitory or excitatory), or which would result in a direct feedback between the nodes. This procedure generated a highly diluted weight matrix that is consistent with biological constraints (while still being Hebbian). The overlap of the network state with a stored pattern after different durations of network cycles is shown in Fig. 8.14C. The results indicate that the storage capacity of this network coincides with the storage capacity of the ANN network with equivalent dilution, indicated by the vertical dashed line. This demonstrates that the theoretical results about the behaviour of recurrent networks that we discussed in the first part of this chapter and that were obtained in highly idealized networks are likely to carry over to networks with more biologically realistic constraints.

Simulation

In Table 8.8 we list the program that was used to generate Fig. 8.14A and B. Lines 3 and 4 create weight matrices with components that are either all 1 (Line 3) or all random following a normal distribution (Line 4). The version chosen in this example is the random weight matrices as used in Fig. 8.14B since Line 3 is commented out with the comment symbol (%). While we choose values

for all components, half of them are overwritten in Lines 5 and 6 to enforce the symmetry and antisymmetry by appropriate copying of the values from the lower triangular part of the matrix into the upper part. Self-connections are set to zero in Line 7, which is commonly done in attractor networks.

Lines 9 and 10 start the loops over different contributions of the symmetric and antisymmetric contributions of the weight components stored in `ws` and `wa`. The resulting weight matrix is produced in Line 11. The network is then initialized with a random state in Line 12, and uses the Euler method to update the quasi-continuous network in Lines 13–16. We store the norm of the network states `u` of the previous time step in the variable `normu_prev` in Line 14 to later calculate the difference between the last time step and the previous one in Line 18. Also, we store in Line 17 the values of the strength parameters used in this run in vectors `xgs` and `xga`, so that the surface plot can be produced with the right labels for the values in Line 21. Note that this program runs for a relatively long time compared to the other programs in this book.

8.4.5 Non-monotonic networks

We have so far only violated the condition of the Cohen–Grossberg theorem regarding the symmetry of the weight matrix. However, the theorem indicates that networks can also behave chaotically when violating other constraints. *Masahiko Morita* and others have studied models of Hebbian trained networks with non-monotonic activation functions. They found that point attractors still exist in such networks and demonstrated the profoundly enhanced storage capacities of those networks. Their results also indicate that the point attractors in these networks have basins of attraction that seem to be surrounded by chaotic regimes. The basins of attraction are like islands in a chaotic ocean. These chaotic regimes can have important functions. For example, they can indicate when a pattern is not recognized because it is too far from any trained pattern in the network. Non-monotonic activation functions seem, on first inspection, biologically unrealistic. However, we have to keep in mind that nodes in these networks can represent collections of nodes, and a combination of neurons can produce non-monotonic responses. An effective non-monotonicity can also be the result of appropriate activation of excitatory and inhibitory connections, although such possibilities are largely unexplored.

Exercises

- (8.1) Explain how the deterministic update rule of the fixedpoint version of ANN, eqn 8.10, is recovered in the noiseless limit, $T \rightarrow 0$, from the probabilistic update rule eqn 8.11.
- (8.2) Derive the equivalent ANN model of the model specified by eqns 8.1–8.4 with the binary representation of $x=10$ for the state with low neuronal activity and $x=01$ for high neuronal activity.
- (8.3) Write an ANN to memorize the letter patterns in file `pattern1`, which we discussed in Chapter 6. How many letters can be memorized by the network? How robust is this network to noise?

Further reading

Statistical physicists made huge contributions to the auto-associative memory literature during the 1980s. This contribution is probably best described in the classic book by Daniel Amit (1989). A shorter summary is given in Hertz, Krogh, and Palmer (1991), which is still one of the best introductions to the theory of artificial neural networks. The book by Rolls and Treves (1998) discusses the relations of attractor networks and other type of neural networks to brain functions, and includes discussions on sparse attractor networks. Eduardo R. Caianello (1961) was among the first to point out the abilities of dynamic recurrent memories with Hebbian learning as brain models, and many other pioneers, like Grossberg, Amari and Edelman have made important contributions before John Hopfield (1982) popularized these networks with physicists. The Cohen–Grossberg theorem was discussed in Cohen and Grossberg (1983), and some updates are discussed by Lu and Chen (2003). The discussion of non-monotonic networks is taken from Morita (1993). Some discussion on the roles of acetylcholine and noradrenaline in modulating learning can be found in Hasselmo and Linster (1999), which contains some issues around neuromodulation in general. Further studies of this area should consider memory consolidation, which should include the classic model of Alvarez and Squire (1991).

Daniel J. Amit (1989), *Modeling brain function: The world of attractor neural networks*, Cambridge University Press.

John Hertz, Anders Krogh, and Richard G. Palmer (1991), *Introduction to the theory of neural computation*, Addison-Wesley.

Edmund T. Rolls and Alessandro Treves (1998), *Neural networks and brain function*, Oxford University Press.

University Press.

Eduardo R. Caianello (1961), *Outline of a theory of thought-process and thinking machines*, in *Journal of Theoretical Biology* 2: 204–35.

John J. Hopfield (1982), *Neural networks and physical systems with emergent collective computational abilities*, in *Proceedings of the National Academy of Science, USA* 79: 2554–8.

Michael A. Cohen and Steven Grossberg (1983), *Absolute stability of global pattern formation and parallel memory storage by competitive neural networks*, in *IEEE Transactions on Systems, Man and Cybernetics*, SMC-13: 815–26.

Wenlian Lu and Tianping Chen, *New conditions on global stability of Cohen–Grossberg neural networks*, in *Neural Computation*, 15: 1173–89.

Masahiko Morita (1993), *Associative memory with nonmonotone dynamics*, in *Neural Networks* 6: 115–26.

Michael E. Hasselmo and Christiane Linster (1999), *Neuromodulation and memory function*, in *Beyond neurotransmission: neuromodulation and its importance for information processing*, Paul S. Katz (ed.), Oxford University Press.

Pablo Alvarez and Larry R. Squire (1991), *Memory consolidation and the medial temporal lobe: a simple network model*, in *Proceedings of the National Academy of Science, USA* 15: 7041–5.

This page intentionally left blank

Part III

System-level models

This page intentionally left blank

Modular networks, motor control, and reinforcement learning

9

The last two chapters are concerned with system-level models. This chapter explores combinations of basic networks, starting with a discussion of *modular mapping networks* that can improve performance of mapping networks through the strategy of *divide-and-conquer*. We then discuss *modular recurrent networks* and show that the number of modules in coupled attractor networks is limited to a small number if we demand a useful balance between the independence of the modules and the cooperation between them. A system-level example of interacting networks is *complementary memory systems*. We then discuss more specific *control architectures*, specifically in relation to *motor control* for the accurate and purposeful guidance of limb movements. These systems have to learn from experience and are generally guided by supervision with simple *reward* contingencies. This chapter closes with a discussion of the important subject of *reinforcement learning*.

9.1 Modular mapping networks	251
9.2 Coupled attractor networks	257
9.3 Sequence learning	262
9.4 Complementary memory systems	264
9.5 Motor learning and control	270
9.6 Reinforcement learning	274
Further reading	285

9.1 Modular mapping networks

Several ideas about modular architectures and the way in which modules can interact and organize themselves have been discussed in the literature. For example, *Marvin Minsky* discussed some of his thoughts in his book, *The society of mind*. Another example is the *committee machine* first described by *N.J. Nilson* in 1965, and the *pandemonium* suggested by *O.G. Selfridge* in 1958. To get a deeper understanding of such information-processing architectures within the neural network framework we have to study how neural networks can be combined and interact. We will review some properties of *modular networks* that are comprised of the basic networks we discussed in previous chapters. The whole system can still be called a neural network, but the fact that we can distinguish subsystems in those networks makes them modular for our purposes.

It is possible to view modular networks as large-scale networks with constraints. An example of a purely physical constraint is the following. Consider a completely interconnected network with a number of elements (nodes) of the order of the neurons in the brain (say conservatively 10^{11}) and with individual interconnecting axons of $0.1 \mu\text{m}$ radius. Placing the nodes on the surface of a sphere and allowing the interior of the sphere to be densely packed with the axons would result in a sphere with a diameter of more than 20 km. The

constraints of the physical extent of interconnecting ‘wires’ therefore demands some more economic solutions. We will see in this chapter that there are many important advantages of modular structures within large systems.

In light of our earlier discussion of completely connected neural networks as universal function approximators (see Chapter 6), it is obvious to ask why modular specialization is used in the brain. Some suggestions of the functional significance of modular specialization in visual processing have been outlined by the British psychologist *Alan Covey*. He speculated that, if the cortex uses inhibition to sharpen various visual attributes, such as colour, edges, or orientations, then it is possible to use local inhibition, which can be implemented with inhibitory interneurons within retinotopic maps, as long as the different attributes are kept separate. Such a specialization would also allow the separate attentional amplifications of separate features. Other advantages of task decompositions and modular architectures include learning speed, generalization abilities, representation capabilities and task realizations within hardware limitations.

9.1.1 Mixture of experts

There are many possible architectures for modular networks, and we will concentrate here only on some fundamental examples that can give us a feeling for specific properties of modular networks. In this section we start by combining feedforward mapping networks in various ways. An example of such an architecture, called the *mixture of experts*, is illustrated in Fig. 9.1. In this architecture we have a column of parallel working modules or *experts*. Each of these experts receives potentially the same input. Another module, called the *gating network*, also receives some input from the general input stream. The purpose of the gating network is to weight the outputs of the expert networks appropriately to solve a specific task. The weighted outputs of the expert networks are then combined by the *integration network*. In the first examples below, this combination is done by summing the weighted outputs of the experts. Since the weighting values are supplied by the outputs of the gating network, the integration network can be viewed as a network with sigma–pi nodes mentioned in Section 3.5.

What are the benefits of using such an approach rather than using a single large mapping network, especially as we know that a large mapping network

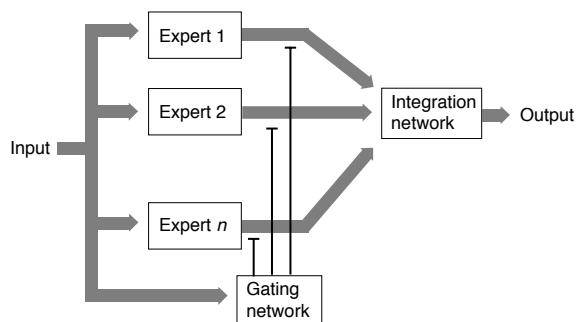


Fig. 9.1 An example of a type of modular mapping network called mixture of experts. Each expert, the gating network, and the integration network are usually mapping networks. The input layer of the integration network is composed of sigma–pi nodes, as the output of the gating network weights (modulates) the output of the expert networks to form the inputs of the integration network.

is a universal function approximator and can thus solve any mapping task (in principle). An example can illustrate some of the benefits of such a modular architecture. In Fig. 9.2A we plotted the absolute function $f(x) = |x|$. It is difficult to approximate this piecewise linear function with a single mapping network due to the sharp discontinuity at $x = 0$. However, a single linear node can represent each branch of the function. By dividing the problem into these subproblems we can employ a simple solution for the subproblems.

This modular network is an example of the *divide-and-conquer* strategy often used in technical applications. Also, this seems to be a strategy used frequently in our daily lives in order to solve complex problems. By dividing a problem into subproblems it is possible that each subproblem can be solved by simple means. For example, functions that can only be represented by a mapping network with hidden nodes can always be decomposed into linear separable subfunctions. We then still have to solve the problem to combine these subfunctions in the appropriate way, which is sometimes not trivial. In the case of the absolute function illustrated in Fig. 9.2A we can solve this by a simple network of two threshold nodes, one that is active for positive input values and one that is active for negative values, as illustrated in Fig. 9.2B. The outputs of the expert networks are then gated by the outputs of the gating network by simply multiplying the outputs of the experts by the gating values.

Training such networks is a major concern when we want those systems to be able to solve specific tasks in a flexible manner. Training the experts alone now has two components: one is to assign the experts to particular tasks, and the other is to train each expert on the designated task. However, even with trained experts we still have to train the gating network, which is a form of a *credit-assignment problem*. Note that the division of the training into a task assignment phase and an expert training phase may be useful in solving the training problem, but in biological systems we might not be able to divide the learning into these separate steps. We will see below an example where the different steps are combined. Several training methods have been proposed in the neural network and machine learning literature. We will not discuss these here in detail as their relevance for brain processes is still not clear. Instead we will follow an instructive example.

9.1.2 The ‘what-and-where’ task

Several studies have shown that the brain has two partly distinguished visual pathways: the *ventral visual pathway*, which is mainly concerned with the recognition of objects; and the *dorsal visual pathway*, which is particularly well adapted to selecting objects for action for which the spatial coordinates are important. The two pathways are sometimes simply termed the ‘*what*’ and the ‘*where*’ pathways, respectively, although this is certainly a drastic simplification of the processing within the brain. Performing object recognition and determining the location of objects in a single network are difficult tasks, and we therefore use this example to illustrate the benefits of a modular network with separate experts for translation-invariant object recognition and object-invariant location recognition.

Robert Jacob, Michael Jordan, and Andrew Barto demonstrated the principal

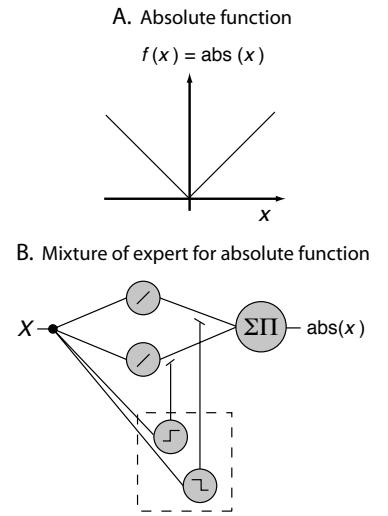
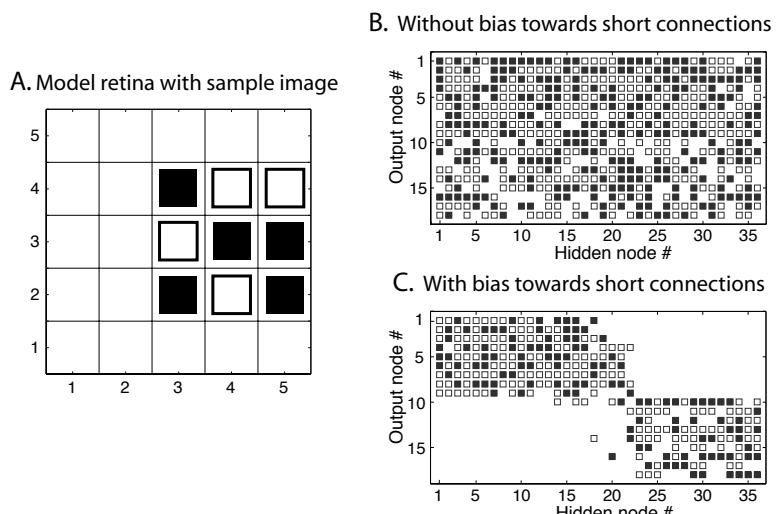


Fig. 9.2 (A) Illustration of the absolute function $f(x) = |x|$ that is difficult to approximate with a single mapping network. (B) A modular mapping network in the form of a mixture of experts that can represent the absolute function accurately.

Fig. 9.3 Example of the ‘what-and-where’ tasks. (A) 5×5 model retina with 3×3 image of an object as an example. (B) Connection weights between hidden and output nodes in a single mapping network. Positive weights are shown by solid squares, while open squares symbolize negative values. (C) Connection weights between hidden and output nodes in a single mapping network when trained with a bias towards short connections. [Adapted from Jacobs and Jordan, *Journal of Cognitive Neuroscience* 4: 323–36 (1992).]



ideas in 1991. They used a simple model retina of 5×5 cells on which different objects represented by different 3×3 patterns can be placed in nine different locations (see the example in Fig. 9.3A). A network that can solve this ‘what-and-where’ task can be designed to have 26 input channels, consisting of 25 for the retinal input and one for the task specification. The desired output can be represented with 18 nodes, the first nine specify nine objects and the second nine specify the locations. This representation is local in each subtask since only one node for the location and object identity is active at any time. Jacobs and co-workers have demonstrated that a single network with a hidden layer of 36 nodes can solve the task after training the multilayer mapping network with back-propagation learning. However, the authors also demonstrated improved performance with modular networks, for reasons summarized next. Note that the use of back-propagation learning is acceptable in this circumstance since the algorithm is only used to find an example of a network that can solve the problem and no claim of biological plausibility for this step is necessary for the questions asked in this study.

A general problem that diminishes learning in single networks is conflicting training information. Conflicting training information can cause different problems. For example, the network will quickly adapt to reasonable performances of the ‘what’ task if we train the network first entirely on this task. The representations of the hidden layers will, however, change in a subsequent learning period on the ‘where’ task, which is likely to conflict with the representation necessary for the ‘what’ task. This *temporal cross-talk* is generally a problem in training sets with conflicting training patterns. In addition, there can be conflicting situations within one training example due to the distributed nature of the representations. This is called *spatial cross-talk*. The division of the tasks into separate networks can abolish both problematic cross-talk conflicts.

The above discussion suggests that modular networks have important merits, though we haven’t yet solved the problem of establishing a modular network

that can solve the what-and-where task. An important contribution of *Jacobs, Jordan, and Barto* was to demonstrate that modular networks can learn task decomposition. For this they used a gating network that increased the strength to the expert network that significantly improved the output of the system. An increased dedication of such an expert ensured the increased learning of this expert in response to subsequent training examples of the same tasks because the back-propagated error signal is modulated by the gating weights, and the module that contributed most to the answer will also adapt most to the new example. Another expert can then take on the representation of another task since a training example of another task would probably diminish the performance of the already specialized expert.

It is interesting to note that this example provides some insight into the *nature–nurture* debate. In the above example it is obvious that architectural constraints can influence the specific task decomposition found through learning. The ‘where’ task is linear separable, so that a single-layer network can represent this task and is easier to train compared to a more complicated network. A simple expert without hidden layer therefore tends to be used for the ‘where’ task. In contrast, the ‘what’ task requires hidden nodes to achieve shift-invariant object recognition. A simple perceptron-like expert is likely to produce insufficient improvements, so that the gating network would not assign this task to this expert. Thus, architectural constraints influence the acquisition of specific abilities, but these inherent abilities are not necessarily fixed and can still be changed by specific learning.

In the above example, the experts were chosen *ad hoc*. Another question is how modularity could evolve through learning. An example of this was outlined by *Jacobs, and Jordan* in a subsequent study. In this study they considered a physical location of the nodes in a single mapping network and used a distance-dependent term in the objective function, which leads to a weight decay favouring short connections. The multilayer network was trained on the objective (or error) function

$$E = \frac{1}{2} \sum_i (r_i^{\text{out}} - y_i)^2 + \lambda \sum_{ij} \frac{d_{ij} w_{ij}^2}{1 + w_{ij}^2}. \quad (9.1)$$

The first term is the common mean square error between the actual and desired output of the network (as in eqn 6.11), which enforces the correct functioning of the network. The second term is yet another weight decay term (see Section 4.3.2) that makes solutions with large weight values between distant nodes unfavourable. Fig. 9.3B and 9.3C indicate the connections between the hidden nodes and the output nodes trained on the what-and-where task, as outlined above, with a single 25–36–18 feedforward mapping network. In Fig. 9.3B the network was trained entirely on the MSE objective function (that is, $\lambda = 0$), whereas Fig. 9.3C shows an example of the results when trained with the bias term that enforces short connections. As can be seen, the connections from the hidden layers to the output nodes, and therefore the hidden nodes themselves, were separated in relation to the two separate tasks. More hidden nodes were thereby allocated to the more difficult ‘what’ task. A learning procedure with such physical constraints therefore leads to a modularization of the network.

9.1.3 Product of experts

We mentioned in Chapter 6 that the normalized output of a feedforward mapping network with a competitive output layer can be interpreted as the probability that an input vector has a certain feature or belongs to a certain class symbolized by the output node. The previously discussed modular networks can easily be generalized to allow similar interpretations if the summed outputs of the expert networks are normalized. We can thus view the mixture of experts as a collection of experts whose weighted opinion is averaged (added and re-normalized) to determine the probability of the feature value. However, if we are considering probabilities, it is much more natural to consider products of probabilities that would determine the joined probability of independent events. This was pointed out by *Geoffrey Hinton*, and he proposed an architecture that he named *product of experts*. He not only stressed some of the advantages of such networks but also alluded to their biological plausibility when compared to empirical data.

To illustrate this further, consider 10 experts that are asked to state their opinion of what the likely increase in sea levels is due to global warming. The answers, expressed as (Gaussian) probability distributions is shown in Fig. 9.4 with dotted lines. If we combine these answers as a sum, and normalizing the sum to get again a probability density function, the answer would be as indicated with the dashed line in the figure. Summing density functions results in wide distribution that do not provide precise answers. The sum of experts only indicates that there is a wide range of answers. However, if we give all experts the same weight (ignoring that the first expert on the left denies global warming), we should also take into account how consistent experts area and how much support the final answer has from the distribution of expert opinions. This is much better described by the product of experts, shown with a solid line in the figure.

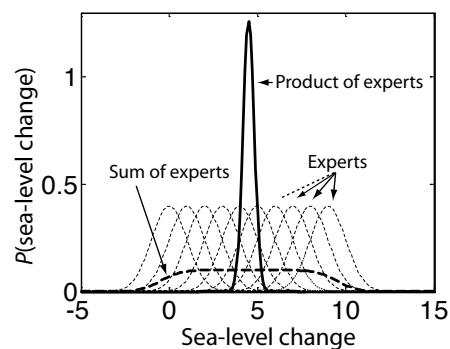


Fig. 9.4 Demonstration of combining expert opinions in different ways. The expert opinions, shown as normal distributions with different centres (dotted lines), are summed and re-normalized in the dashed line. The solid line represents the normalized product of the experts.

A product of experts can produce much sharper probability functions. The reason for this is that even a large probability assigned by one expert can be largely suppressed by low probabilities assigned by other experts. A large probability is only assigned to events for which there is some agreement between the experts. This helps with another problem in density estimation, that of estimating densities with low probabilities. Areas with low densities are more

difficult to estimate well, since events in these areas have low probabilities of occurring. Errors in the estimation are therefore larger than areas in which many events can be used to estimate the density functions. An overestimation of rare events is thus common in statistical analysis. Products of experts are less prone to such errors. Or, in other words, opinions of experts outside their domain of expertise have less of an effect when other experts confirm that these areas are of less relevance.

How to train products of experts is not obvious, and some research is currently being pursued in this area. Hinton and colleagues have already proposed generalized gradient descent methods for training such networks. However, the general ideas outlined here are now also integrated with *deep belief networks* discussed Chapter 10.

9.2 Coupled attractor networks

In contrast to the combination of feedforward networks in the previous section, this section discusses the combination of basic recurrent networks. Such modular recurrent networks can also be viewed as a subdivided recurrent network. An example is illustrated in Fig. 9.5A where an overall recurrent network is divided into two groups of nodes. It is useful to distinguish between connections (or weights) between nodes in the same group (intra-modular connections) and connections (or weights) between nodes of different groups (inter-modular connections). The whole system is, of course, still a recurrent network and thus basically still one large dynamic system. However, by distinguishing the different type of connections and neuronal groups, we can study more explicitly systems with strongly coupled subsystems and weak interaction between such subsystems. In general, neither the nodes within the groups, nor the nodes between groups, have to be completely connected. However, we will simplify the discussion by considering fully connected attractor networks of the types discussed in Chapter 8.

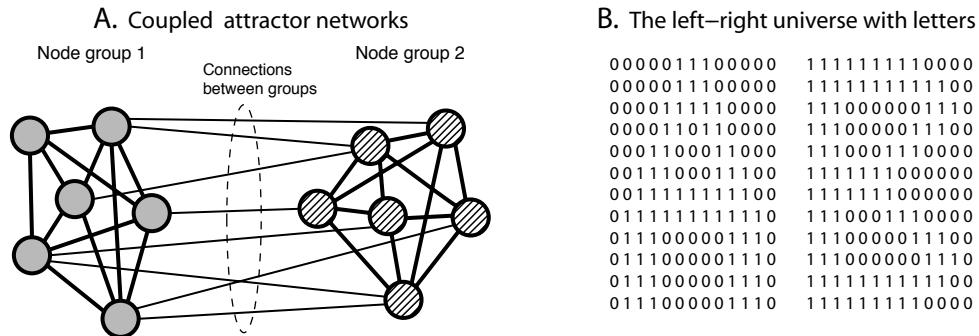


Fig. 9.5 (A) Coupled (or subdivided) recurrent neural networks. The nodes in this example are divided into two groups (the nodes of each group are indicated with different shadings). There are connections within the nodes of each group and between nodes of different groups. (B) The left-right universe that is made up of words with two letters.

9.2.1 Imprinted and composite patterns

Similar to the discussions in the last chapter, we now discuss the associative storage abilities of such modular attractor networks, following largely some discussions by *Yaneer Bar-Yam*. The main purpose is to show that there is a sensitive balance between the strength of connection between subnetworks in order to influence processes in other subnetworks, and the ability of the subnetworks to function semi-autonomously.

Let us first compare the extremes of coupled attractor networks, one single point attractor network as discussed in Chapter 8 versus two single point attractor networks. As an example we want to imprint into this system all objects that can be described by two independent feature vectors representing, for example, shape and colour. We consider therefore three different shapes *{circle, square, triangle}* and three different colours *{red, blue, green}*. The possible training set with this example consists thus of nine possible combinations *{red circle, blue circle, ...}*. Another example, a world with letter images on the left and right visual fields, respectively, is shown in Fig. 9.5B. In general, with m possible feature values (objects), one can build m^2 possible (combined) objects as long as there are no correlations between the features that would make certain combinations more likely than others.

Let us now consider a separate attractor network for each feature. From the discussion in Chapter 8 we know that a network with 1000 nodes can store around 138 patterns (feature values) when random representations of the feature values are trained with the Hebbian rule. A network with two such independent subnetworks, each carrying the information of one feature, could store

$$P = (\alpha_c N/m)^m = 138^2 = 19,044 \quad (9.2)$$

objects represented by all possible feature combinations, where m is the number of features (or the number of modules) with an equal number of nodes N/m . The number of patterns that can be stored in a single network is only

$$P = \alpha_c N. \quad (9.3)$$

To store all the 19,044 possible objects in one attractor network, we need a network with at least $N = 19,044/\alpha_c = 138,000$ nodes. The saving of resources of the modular network in terms of nodes is very impressive (2000 versus 138,000), and the savings are even more impressive if we consider the number of weights we have to use ($1000^2 + 1000^2 = 2 \times 10^6$ versus $138,000^2 \approx 2 \times 10^{10}$).

Why, then, would we ever want to use large single networks? The answer is that we have to use them if we want to represent correlations of features and remember specific objects. For example, if we want to remember a *green square* and a *blue triangle*, then we have to imprint these two objects with the specific combination of the features, and not the other possible combinations. If we do this we can recall the objects from partial information; for example, *green* can trigger the memory of *green square*. In the case of separate networks the input of *green* cannot trigger any memory in the ‘shape’ network, and we would potentially recall all possible combinations of features even if they were not part of the set of objects we used for training.

9.2.2 Signal-to-noise analysis

The previous discussion suggests that a certain intermediate level of coupling between attractor neural networks is likely to be necessary for more complex (and realistic) processing in the brain. We can get some further insight into the behaviour of coupled attractor networks using a signal-to-noise analysis along the lines of that outlined in Chapter 8. We consider an overall network of N nodes that can be divided into

$$m = \frac{N}{N'} \quad (9.4)$$

modules, each having the same number of nodes N' . The weights are trained with the Hebbian rule

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^P \xi_i^\mu \xi_j^\mu. \quad (9.5)$$

However, we modulate the weight values between the modules with a factor g . Formally, we can define a new weight matrix with components

$$\tilde{w}_{ij} = g_{ij} w_{ij}. \quad (9.6)$$

The components of a modulation matrix \mathbf{g} are given by

$$g_{ij} = \begin{cases} 1 & \text{if nodes } i, j \text{ are within the same module} \\ g & \text{otherwise} \end{cases}. \quad (9.7)$$

To shorten the statement ‘if nodes i, j are within the same module’, we define a set $m(i)$ that lists all the node numbers that are in the same module as node number i .

The next step in the signal-to-noise analysis is to evaluate the stability of the imprinted pattern. First we will evaluate an imprinted pattern that is made up of the subvectors of each module that were imprinted together. Without loss of generality we can evaluate again the first pattern $\mu = 1$, that is, $s_i(t) = \xi_i^1$. We can then separate the signal terms from the noise terms in the updating rule as outlined in Chapter 8. This gives

$$\begin{aligned} s_i(t+1) = & \text{sign}\left(\frac{N'}{N} \xi_i^1 + g \frac{N-N'}{N} \xi_i^1 + \dots\right. \\ & \left. \dots + \frac{1}{N} \sum_{j \in m(i)} \sum_{\mu \neq 1} \xi_i^\mu \xi_j^\mu \xi_j^1 + \frac{g}{N} \sum_{j \notin m(i)} \sum_{\mu \neq 1} \xi_i^\mu \xi_j^\mu \xi_j^1\right). \end{aligned} \quad (9.8)$$

From this we can identify the signal and noise terms. The strength S of the signal term is the factor in front of the desired signal ξ_i^1 , and the variance of the noise term is determined by the remaining part analogously to the procedure described in Chapter 8. In summary we get

$$\begin{aligned} \text{signal: } S &= \frac{N'}{N} + g \frac{N-N'}{N} = \frac{1}{m} + g\left(1 - \frac{1}{m}\right) \\ \text{noise: } \sigma &= \sqrt{\frac{P-1}{N}} \sqrt{\frac{1}{m} + g^2\left(1 - \frac{1}{m}\right)} \\ &\approx \sqrt{\frac{P}{N}} \sqrt{\frac{1}{m} + g^2\left(1 - \frac{1}{m}\right)} = \sqrt{\alpha z_1}. \end{aligned} \quad (9.9)$$

We introduced in the last line a shorthand notation z_1 that encapsulates the difference between the modular networks ($m > 1$) and the standard single network ($m = 1$) for which $z_1 = 1$. The parameter α is again the load parameter defined in eqn 8.16. Further discussions on the load capacity depend again on the error bound we impose on the network (see eqn 8.17). To simplify the formulas we introduce the shorthand notation,

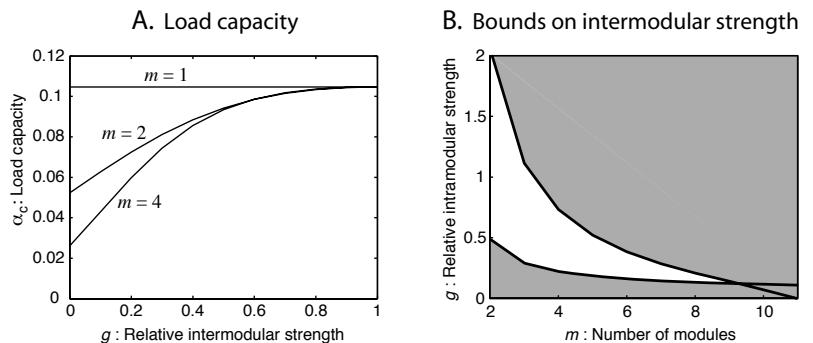
$$z_2 = [\text{erf}^{-1}(1 - 2P_{\text{error}})]^2. \quad (9.10)$$

With these terms we can write the capacity bound (eqn 8.18), generalized to modular networks started in a trained pattern, as

$$\alpha < \frac{S^2}{2z_1 z_2}. \quad (9.11)$$

We called the limiting case the critical load parameter or the storage capacity, α_c , which is plotted for $P_{\text{error}} = 0.001$ and different values of g and m in Fig. 9.6A. With only one module, we recover the results of Chapter 8. However, with more than one module ($m > 1$) the results depend on the relative strength of the intramodular weights relative to the intermodular weights, which we parametrized with g . With values of $g < 1$ we get less support from the signals of the other modules, so that the load capacity of the networks diminishes.

Fig. 9.6 Coupled attractor neural networks: results from signal-to-noise analysis. (A) Dependence of the load capacity of the imprinted pattern on relative intermodule coupling strength g for different numbers of modules m . (B) Bounds on relative intermodule coupling strength g . For g values greater than the upper curve the imprinted patterns are stable. For g less than the lower curve the composite patterns are stable. In the narrow band in-between we can adjust the system to have several composite and some imprinted patterns stable. This band gets narrower as the number of modules m increases and vanishes for networks with many modules.



In contrast to the previous case we can start the network from states that correspond to different subpatterns in the different modules. We can consider all kinds of combinations of patterns in different modules, but for the following discussion there are two most interesting and limiting cases. The first one occurs when all the subpatterns in the modules are chosen to be the first training pattern except for the module to which the node under consideration belongs. For this module we choose a random initial state and ask how strong the intermodular coupling strength has to be in order for the other modules to trigger the corresponding state in the module with the random initial state. We thus study the network with the starting state

$$s_j(t) = \begin{cases} \eta_j & \text{for } j \in m(i) \\ \xi_j^1 & \text{otherwise} \end{cases}, \quad (9.12)$$

where the variable η_j is a random binary number. The state of the network after one update is then

$$\begin{aligned} s_i(t+1) = & \text{sign}\left(g \frac{N - N'}{N} \xi_i^1 + \dots\right. \\ & \left. \dots - \frac{1}{N} \xi_i^i \sum_{j \in m(i)} \xi_j^1 \eta_j + \text{other noise-terms for } \mu \neq 1\right). \end{aligned} \quad (9.13)$$

The corresponding strength of the signal is determined by the first factor. The second term is an additional noise term that has to be added to the noise term of the previous case (eqn 9.9). However, this noise term contributes only with $1/(mN)$ and, since we consider large networks, we can ignore this small contribution. In summary we get

$$\begin{aligned} \text{signal: } S &= \frac{N - N'}{N} g = \left(1 - \frac{1}{m}\right) g \\ \text{noise: } \sigma &\approx \sqrt{\alpha z_1} \end{aligned} \quad (9.14)$$

We can insert these expressions into eqn 9.11 from which we can derive a lower bound on the relative intracoupling strength g (for a given number of modules m and load parameter α) that we have to impose if we want to be able to trigger the first pattern in the module that we started in a random state. This is given by

$$g^2 > \frac{2\alpha z_2}{(m-1)\left(1 - \frac{1}{m} - 2\alpha z_2\right)}. \quad (9.15)$$

This lower limit on g is plotted in Fig. 9.6B as the line restricting the white area from below (for $P_{\text{error}} = 0.001$ and $\alpha = 0.01$). The curve confirms the intuitive conjecture that the g -factor can be reasonably small if the number of modules increases because an increasing number of modules results in an increasing relative number of nodes that are pointing in the desired direction.

We can also study the reverse case with

$$s_j(t) = \begin{cases} \xi_j^1 & \text{for } j \in m(i) \\ \eta_j & \text{otherwise} \end{cases} \quad (9.16)$$

as starting state. With this starting state we can ask how low the g -factor has to be in order for the substate ξ^1 in one module to be stable while all the other modules are pulling in other directions. The state of the network after one update with this starting state is

$$s_i(t+1) = \text{sign}\left(\frac{N'}{N} \xi_i^1 + \frac{g}{N} \xi_i^i \sum_{j \notin m(i)} \xi_j^1 \eta_j + \text{other noise-terms for } \mu \neq 1\right). \quad (9.17)$$

We can again neglect the additional noise term and use

$$\begin{aligned} \text{signal: } S &= \frac{N'}{N} = \frac{1}{m} \\ \text{noise: } \sigma &\approx \sqrt{\alpha z_1} \end{aligned} \quad (9.18)$$

in the signal-to-noise analysis. Thus, if we want to be able to keep a state of a module stable (free from interferences from the other modules) the g -factor has to obey

$$g^2 < \frac{1 - 2\alpha z_2 m}{2\alpha z_2 m(m-1)}. \quad (9.19)$$

The upper limit on g from these considerations is plotted in Fig. 9.6B as the line restricting the white area from above. With g -factors somewhere in the white area we could find points where patterns in different modules are relatively stable while still responding to some influence from the other patterns.

The analysis discussed here is only intended to allude to the possible interaction between subnetworks in modular networks, and many simplifying assumptions have been made. For example, we have assumed that intermodular weights have been trained with all patterns. In more realistic circumstances we would train intermodular connections for only specific subpattern combinations and could thus allow the white area to extend to higher values of m . Also, we have seen that the values from the signal-to-noise analysis do not accurately describe the full dynamics in such networks. Nevertheless, what is striking from our analysis is that there is a limit on the number of modules in the system when we want to have a system with a sensible balance of modular independence and modular interactions. This restricts the modularity that can be utilized in complex systems at each level. To be able to utilize many modules to encapsulate certain functionalities we have to combine these many modules within a hierarchical structure where on each level we have a restricted number of branches. We conjecture that the sensible balance of cooperation versus independence between modules is a driving force behind possible structures of complex systems such as the brain.

9.3 Sequence learning

While we have mainly analysed point attractor states to highlight the pattern completion ability of associative networks, the brain is much more dynamic in that some memories can trigger other memories, and the importance of sequential aspects in brain processing is increasingly realized. In this section we briefly discuss *sequence learning* and show that sequence learning in associative networks can benefit from modular architectures.

The principal idea behind the application of recurrent networks to sequence learning has already been discussed in Section 7.4, where we have seen that *hetero-associations* between consecutive states can move the system between attractor states. It is useful to distinguish between auto-associative weights

$$w_{ij}^A = \frac{1}{N} \sum_{\mu} s_i^{\mu} s_j^{\mu}, \quad (9.20)$$

which associate patterns with themselves, and hetero-associative weights

$$w_{ij}^H = \frac{1}{N} \sum_{\mu} s_i^{\mu+1} s_j^{\mu}, \quad (9.21)$$

which are associations between different, in this case consecutive, patterns. Both auto-associative and hetero-associative connections are necessary to achieve robust sequence memory. The principal idea is that the hetero-associations drive the system to a new, possibly noisy version, of a consecutive memory state, and that the auto-association helps to clean up the noisy version of

the new state. For example, Hopfield considered a model with asymmetrical weights

$$w_{ij} = w_{ij}^A + \lambda w_{ij}^H \quad (9.22)$$

in his 1982 paper. However, he also found that this basic model only works for very short sequences. Many variations of this basic model have subsequently been proposed, including refined choices of hetero-associative weights and temporally varying parameters λ . However, here we focus on a solution within a modular network.

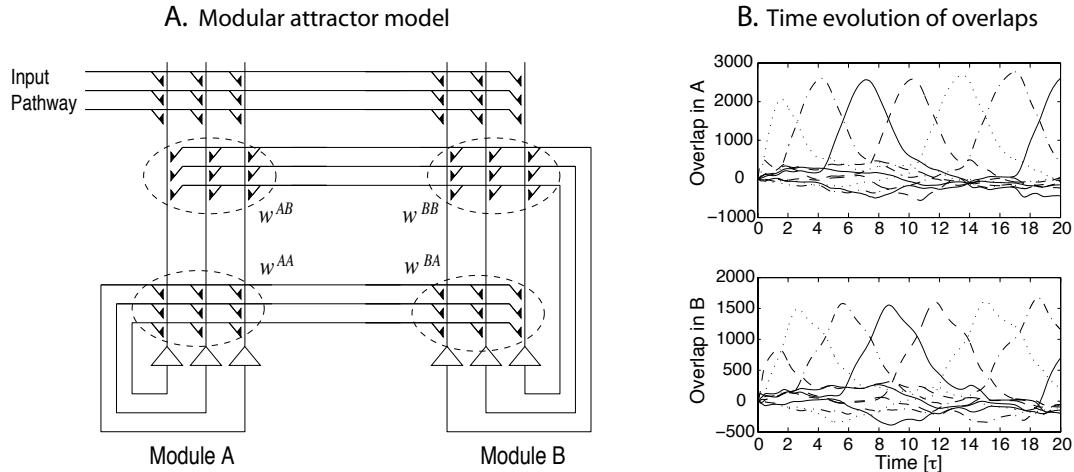


Fig. 9.7 (A) A pair of connected recurrent modules A and B. (B) Overlap of the net activation $\mathbf{u}(t)$ of nodes in each module with the stored patterns in a sequence of length 10 during recall. [Adapted from Lawrence, Trappenberg and Fine, *Neurocomputing* 69: 634–41 (2006); see also Sommer and Wennekers, *Neurocomputing* 65–66: 449–54 (2005).]

We consider a modular network as shown in Fig. 9.7A. This network consists of two auto-associative models labelled A and B, which are coupled through a hetero-associative coupling \mathbf{w}^{AB} . The reverse intermodular coupling is auto-associative. Results of the simulations are shown in Fig. 9.7B as overlaps between the internal activation $\mathbf{u}(t)$ of the nodes and the stored patterns. As the activation overlap rises for a particular pattern in module B (Fig. 9.7B, bottom), module A begins moving toward the next pattern in the sequence (Fig. 9.7B, top) due to the B→A hetero-associations. As the activation overlap rises for a particular pattern in A, B follows shortly behind from the A→B auto-associations. Note that only four different line styles were used in Fig. 9.7B, so that some patterns share the same line style.

Further analysis of this network shows that long sequences can be learned by this architecture. Similar architectures have been proposed for hippocampal functions, with suggestions that the CA3 subfield represents one auto-associative subnetwork, and that recurrencies in the hippocampus through the dentate gyrus provide the anatomical substrate for the other auto-associator. Thus architectures could then not only support the rapid learning of sequences within the hippocampus, but these learned sequences could also be used to

anticipate sensory states and thus guide behaviour. Modular attractor networks, with auto-associative and/or hetero-associative components, have also been used to model behavioural data such as the influence of mood on memory recall. The characterization of interacting recurrent networks is thus an important area of research within computational neuroscience.

9.4 Complementary memory systems

We will now start to outline some more specific examples of system-level models of brain functions, still with a particular emphasis on modular networks. We start with a specific hypothesis on the implementation of *working memory* in the brain. We reserve the term ‘working memory’ for a concept as used by cognitive scientists, although the term is frequently used by physiologists to refer to an apparent memory through ongoing neuronal activity, as discussed in Chapter 7. We use the term *short-term memory* for this latter concept and discuss here their relation.

Since working memory is a concept derived from behavioural studies, it is not easy to define this construct precisely, and many different views of this concept in terms of definition and functional roles exist in the literature. The reason that it is useful in cognitive science is that it is a very appealing concept when describing possible mechanisms of complex cognitive processes that seem to rely on some form of workspace, which provides the necessary buffer to hold information to solve complex tasks. Examples of such complex cognitive tasks include language comprehension, mental arithmetic, and all forms of reasoning that are necessary for problem-solving and decision-making. Working memory is different from the specific form of short-term memory (STM) discussed in Chapter 7 because many complex cognitive tasks also rely on other forms of memory. For example, episodic memory, which is possibly mediated by the hippocampus, and semantic memory, as supported by functions of the neocortex, can be necessary to directly solve complex tasks faced by humans.

9.4.1 Distributed model of working memory

Several models of working memory have been advanced in the literature, and the models by Alan Baddeley have been particularly influential. However, here we outline a conceptual model of working memory proposed by *Randall O'Reilly, Todd Braver, and Jonathan Cohen*, which is based on a modular structure as summarized in Fig. 9.8. Three modules are included in this model, each having some characteristic functionalities.

It is often difficult to distribute specific functionalities in a distributed system as there are interactions within the system and the system properties frequently rely on the whole system. Nevertheless, what we have in mind here is to assign to each module a different memory functionality in terms of short-, intermediate-, and long-term memory.

The module labelled PFC has many (mainly) independent recurrent subsystems represented as single recurrent nodes. Each such node could hold a specific, previously presented, item of information over a short period of time through reverberating neural activity. This module thus mimics basic opera-

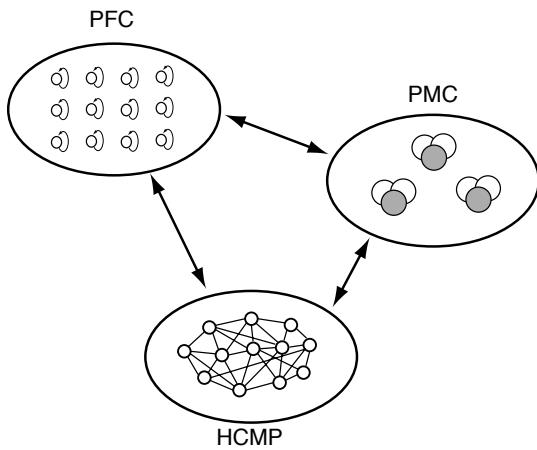


Fig. 9.8 Modular system with short-, intermediate-, and long-term memory, which are associated with functionalities of the prefrontal cortex (PFC), the hippocampus and related areas (HCMP), and the perceptual and motor cortex (PMC), respectively [adapted from O'Reilly, Braver, and Cohen, in *Models of working memory*, Miyake and Shah (eds), Cambridge University Press (1999)].

tions sometimes attributed to the *prefrontal cortex* (see Chapter 7). The structure labelled HCMP represents structures such as the *hippocampus* that we identified as suitable for rapid learning of associations as required for episodic memory (see Chapter 8). This form of memory is based on rapid synaptic plasticity, in contrast to the predefined weights of short-term memory in cortical maps. The module labelled PMC, which stands for *perceptual and motor cortex*, is aimed at describing more slowly learned concepts such as semantic memory or action-repertoires. The interesting scientific question is to study how the different memory models interact and how the brain utilizes the kind of information that is necessary to perform complex mental tasks. O'Reilly and collaborators discussed such complementary systems in more depth, which is highly recommend as further reading. Here we will only focus on a specific issue of working memory which is repeatedly discussed in the literature.

9.4.2 Limited capacity of working memory

It is easy to remember very quickly a small list of numbers such as '31, 27, 4, 18'. However, the ability to recall a list of numbers (without repeated learning) breaks down drastically when the list has a few more items. Try '62, 97, 12, 73, 27, 54, 8'. The memory required to perform such a task depends strongly on short-term memory mechanisms. However, in light of the discussion above we think of them more as limitations of working memory. The limited capacity of working memory was described in 1928 by H.S. Oberley. G. Miller coined the phrase 'magical number 7 ± 2 ' for this limit. It is now clear that the precise limit depends strongly on the specific task that has to be performed, and the apparent limit can be modified by other processes. For example, a waiter is often highly skilled in quickly remembering long lists of orders. This is made possible by utilizing 'mental tricks' such as grouping items together. However, when more carefully ruling out such metal tricks in experiments, it seems that a limit around four item is more common.

The very limited capacity of working memory is puzzling if we remember that

we can easily build systems that can rapidly store many thousand of items. It is also an important issue since there are some indications that the individual capacity of working memory is a good predictor of success in completing courses, indeed some argue even more than classical measurements of IQ. Thus, a larger storage capacity should make us fitter to survive, and we can ask why evolution was not able to increase the working memory capacity considerably. The search for the reasons behind the limited capacity of working memory is therefore prominent in cognitive neuroscience.

There are many hypotheses about the reasons behind the limited capacity of working memory. *Donald Broadbent* was among the first to point out that this hints to a bottleneck in the information processing capabilities of the brain. *Nelson Cowan* advanced this further by investigating more specifically how limits in the attentional system, which is often an essential part of working memory models, could account for the data. Another suggestion is that short-term memory due to reverberating neural activity, which is also a essential ingredient in most working memory models, could hold the key. For example, we outlined in Chapters 7 and 8 how such short-term memory could be realized in an attractor network. Such networks are limited to represent a single item at each time. However, *Lisman* and *Idiart* pointed out that different objects could be encoded in different high-frequency subcycles of low-frequency oscillations. In the following we will discuss two further suggestions, mainly as an exercise in applying previous discussions to the quantification of these suggestions. The first example is based on a spike-train representation of objects, whereas the second example brings us back to interacting recurrent modules.

9.4.3 The spurious synchronization hypothesis

Experimental results of human performance in a typical working memory experiment are shown in Fig. 9.9. In these experiments, *Steven Luck* and *Ed Vogel* showed images with N^{obj} simple objects, such as objects with different shapes and/or colours, to a subject. After a delay of 900 ms, the subject was presented with a second image, and the subject had to judge if the two images were identical or not. The performance of the subjects, as measured by the average percentage of correct responses, is plotted in Fig. 9.9A with solid squares as a function of the number of objects (squares) in the images. A sharp breakdown of the performance with an increasing number of objects in the image can be seen. With similar experiments the authors demonstrated in addition that: (1) the capacity limit does not depend on the number of features of objects relevant for making the decision; (2) the visual working memory is independent of the load in the verbal working memory; (3) the capacity limit is not due to an increase of the number of decisions that have to be made for an increasing sample set.

Luck and *Vogel* also suggested a possible reason for the limited capacity of working memory. Their idea is based on the assumption that a feature is represented by a particular spike train, as illustrated in Fig. 9.9B. We assume that a feature is coded by a spike train with an arbitrary code (that is, a random spike train), and that the decoding of the spike trains is limited by a finite resolution of the spike times. The different features of different objects

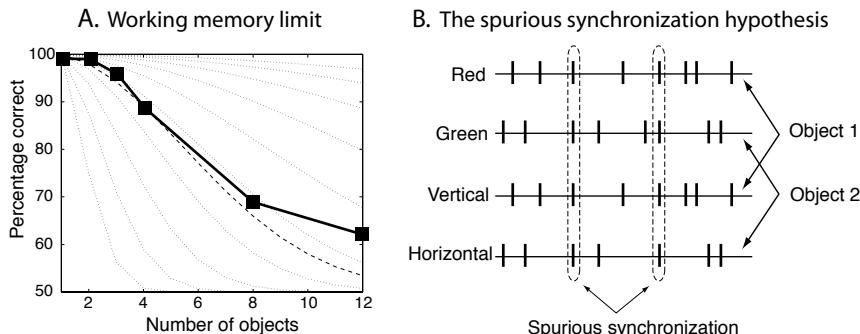


Fig. 9.9 (A) The percentage of correct responses (recall ability) in human subjects in a sequential comparison task of two images with different numbers of objects N^{obj} (solid squares). The dotted lines illustrate examples of the functional form as suggested by the synchronization hypothesis. The dashed line corresponds to the results with $P^{\text{SS}}(2) = 0.04$, where P^{SS} is the probability of spurious synchronization. (B) Illustration of the spurious synchronization hypothesis. The features of the object are represented by different spike trains, so that the number of synchronous spikes within a certain resolution increases with an increasing number of objects. [Adapted from Luck and Vogel, *Trends in Cognitive Science* 2: 78–80 (1998).]

could be represented by *synchronized spike trains*. This is a popular way of solving the *binding problem*, the problem of relating the different features of one object when those features are represented in different brain areas. Of course, we have to assume that different objects are represented by different spike trains. However, it is possible that some spikes in the different spike trains for the different objects fall within the same time window, as illustrated in Fig. 9.9B. As this is based purely on random coincidences, we call this type of synchronization *spurious synchronization*. The number of coincident spikes between some objects does increase if we increase the number of objects in a set. At some point the level of spurious synchronization would become very large, possibly large enough to destroy the ability to distinguish between the objects. This, as argued by Luck and Vogel, could explain the breakdown of performance in their experiments.

The idea of spurious synchronization as the reason behind the capacity limit of working memory is very appealing. It would, for example, predict that the capacity limit does not depend on the number of features that an object has as found experimentally. The explanation for this within the spurious synchronization hypothesis is that each additional feature of one object would in any case be synchronized with the other features of the object so that only the number of objects with different spike trains would matter.

Computational neuroscience offers some way of testing this hypothesis further, specifically by quantifying the hypothesis so that it can be compared more directly to experimental data. This can be done for the above hypothesis by using only basic combinatorics. The number of pairs N^{PAIRS} in a set of N^{obj} objects is given by

$$N^{\text{pairs}} = \binom{N^{\text{obj}}}{2} = \frac{N^{\text{obj}}!}{2! (N^{\text{obj}} - 2)!}. \quad (9.23)$$

With a given neural code, such as that specified by a given probability distribution of the spikes, there will be a given probability of spurious synchronization between two spike trains, which we symbolize as $P^{\text{ss}}(2)$. The probability of not having spurious synchronization between two spike trains is then $1 - P^{\text{ss}}(2)$. The probability of spurious synchronization between at least two spike trains in a set of N^{obj} spike trains (pattern) is then given by

$$P^{\text{SS}}(N^{\text{obj}}) = 1 - (1 - P^{\text{ss}}(2))^{N^{\text{pairs}}}, \quad (9.24)$$

because the spike trains are independent which means that the probabilities simply multiply. An increasing probability of spurious synchronization decreases the performance in the object comparison task. We can thus quantify the hypothesis of Luck and Vogel with a functional expectation of the percentage of correct recall given by

$$P^{\text{correct}} = (1 - P^{\text{SS}}(N^{\text{obj}})) + (P^{\text{SS}}(N^{\text{obj}})) 0.5. \quad (9.25)$$

The second term on the right-hand side is added because a random answer can be expected if objects cannot be correctly separated. The recall performances from eqn 9.25 for various values of the probabilities of spurious synchronization between two spike trains are shown as dotted lines in Fig. 9.9A. The dashed line corresponds to the spurious synchronization hypothesis with $P^{\text{ss}}(2) = 0.04$. We see that we can use the quantification of the hypothesis to extract some parameters that can be further compared to experimental data. For example, the probability of spurious synchronization between two spike trains can be compared to models of spike trains that take realistic values for spike distributions, firing rates, and time precision into account.

The comparison of experimental and theoretical data also indicates some possible discrepancies that should be analysed further. For example, the experimentally observed transition seems a little bit sharper than the one derived from the spurious synchronization hypothesis. Also, there seems to be a strong deviation for large numbers of objects that could indicate a separate effect on working memory. We will not follow this line of research further as the discussion was only intended to illustrate that the quantification of hypotheses is often important and should enable us to develop better models in the future.

9.4.4 The interacting-reverberating-memory hypothesis

The previous discussion on interacting memory modules offers another possible reason behind the limited capacity of working memory. We have seen that we can build separate memory modules in which to store separate items, and storing more items would only require more memory modules. However, we also discussed that interactions in the brain are necessary to solve complex tasks, and we have seen already that there is a sensible balance between the number modules and the interaction strength between them to allow for some flexibility in the operation of the system.

In this section we will quantify this observation in terms of dynamic neural field models, which, as argued in Chapter 7, model some aspects of representations in cortical maps. For example, we can think about reverberating neural

activity in the frontal cortex that holds information of high-level concepts and activates lower-level details that make up specific objects. So while the implementation in the brain is likely based on distributed activity in the brain, we will simplify the discussion here by showing the concept with a single feature dimension.

Neural field models with appropriate balance of centre-surround inhibition and local excitation have the ability to store features with ongoing neural activity, as outlined in Section 7.3.2. An example is shown in Fig. 9.10A, which demonstrates that an external stimulus applied until $t = 10\tau$ was memorized with ongoing neural activity after the external stimulus was removed. This short-term memory ability is what we are using in the following simulations. The program used for these simulations is that given in Table 7.2 with slightly modified parameters as noted in the figure caption.

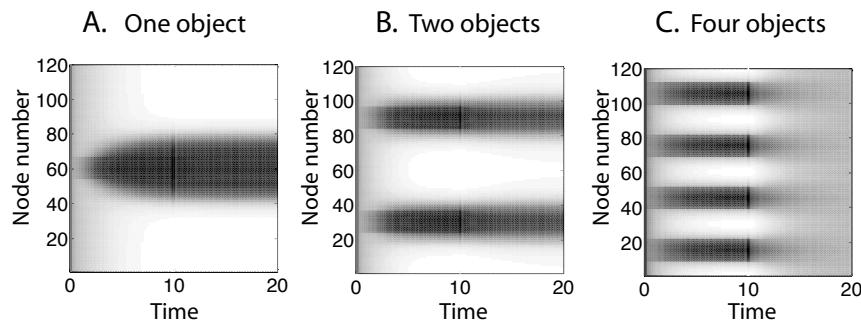


Fig. 9.10 Simulations of the DNF model based on the program of Table 7.2 in which some parameters have been changed. In these simulations 120 nodes are used ($\text{nn} = 120$), the width of the patterns ($\text{sig} = 2\pi/30$) and the inhibition constant ($C=0.2$) have been reduced, the scaling factor of the weights has been enlarged ($w=10*(w-C)$), and the width of external simulations have been halved. All these changes are only made to demonstrate better the coexistence of activity packets. The major change to the earlier simulations are the number of areas that are stimulated externally, with (A) one, (B) two, or (C) four areas of external stimulation until time $t = 10\tau$.

For convenience in the discussion, we interpret each memory state in the DNF model in this case as representing an object. In the case of many separate DNF networks, we could thus represent many different objects in the system. However, we are more interested in exploring the interacting case and discuss this in the case of the other extreme, in having a single, large DNF network. In this limit we can have only one activity packet at a time due to the winner-takes-all nature of the network. However, there are several factors in practice which can support the coexistence of several activity packets, at least for a considerable time during a short-term memory phase. These stabilizing mechanisms in DNF models have been discussed in Section 7.3.4. For example, if we make it easier for neurons that are already active to fire again, such as NMDAR activation in recurrent networks, then we can stabilize more than one activity packet. There are other means to stabilize some packets, at least for some sufficient time, and we can even simulate this easily by using a spatially discrete implementation of a DNF model. The discrete nature of

the node representations in the computer provide some energetic hurdle for two activity packets within a certain proximity to stay instead of moving toward each other or merge in the continuous case. An example is shown in Fig. 9.10B. However, when presenting four objects to the network, as shown in Fig. 9.10C, the network activity breaks down rapidly.

This simulation demonstrates some interesting properties of competitive networks. When moving from one to two objects in the simulation, the representation of the ‘objects’ become sharper. This sharpening was already noted by Wilson and Cowan, and is often argued to explain effects such as *edge enhancement*. Furthermore, while localized activity leads to a strong memory response in the network, diffuse activity can shut off the network activity. It is this principle that could underlie the limited capacity of working memory. Of course, the simulations shown here are very idealized in many ways, such as using equidistant objects which corresponds to a situation of maximal stability. In the brain one needs to consider more distributed representations. For example, neurons can participate in different maps at the same time. When such maps are uncorrelated, then there is little inference between the representations in the maps, since an activity packet in one map would appear as random background in the other map. This idea has been formalized by *Alexei Samsonovich* as the concept of *charts* in neural field models. Such charts could, for example, explain how many different place field maps could be accommodated in the hippocampus. In terms of working memory, however, it is possible that overlap in higher-order concepts of the brain will limit the number of charts that can be activated at the same time. Studying the interaction of coupled subnetworks is thus an important area of research.

9.5 Motor learning and control

In Chapter 8, we discussed a specific type of learning, that of forming specific memories through fast associative learning. In contrast, learning of motor skills, such as catching a ball or riding a bicycle, requires some time. This type of learning is often concerned with discovering regularities in training data in a statistical sense and typically acts on a large amount of training data without the intention of storing all the specific examples. We discussed some examples of such learning in Chapter 6. In addition, biological motor systems must function in noisy and uncertain environments. For example, sensory information or task instructions can be noisy, the physical system executing the command can be noisy and constantly changing, and the environment itself can be usually not a simple repetition of the same tasks. It is necessary for the survival of a species to achieve some precision in limb movements, and motor control is therefore an important task for our nervous system.

In this section we discuss control systems and learning with reward, using motor systems as the main example. Reinforcement learning is different to supervised learning in that only very general feedback, such as reward or punishment, is given by a teacher. We will see that such learning systems can be used to build better control systems. Control systems are necessary to guide specific actions in an uncertain environment. For example, it might be possi-

ble to pick up a cup in front of us without looking at it, but visual guidance certainly helps to achieve this task much more reliably; hence the brain must be able to direct the control system for arm and hand movements with visual signals. It is a fascinating experience when we perturb this system, for example, by using prism glasses that shift the visual appearance of a target systematically away from its actual location, or by changing the dynamics of an arm with a computer-controlled guidance system. Reaching for a target with altered parameters of the controlled system we will typically fail within the first few trials. However, we are commonly able to adapt to the changed environment within only a few additional trials.

9.5.1 Feedback controller

The control of mechanical or electrical devices is a major challenge faced regularly by engineers. It is thus not surprising that engineers have contributed a lot of ideas on how limb movements could be controlled by the nervous system. A typical example of a control system is shown in Fig. 9.11.

In the example of this section we designate a signal specifying the desired state as input to the system. This input signal is then converted by a specific module into a motor command that drives the device we want to control. We call this module *motor command generator*. The generated motor command could, for example, be the signal to activate specific motor neurons in the brainstem that in turn stimulate muscle fibres. The motor command generator can be viewed as an *inverse model* of the dynamics of the controlled object as it takes a state signal and should produce the right motor command so that the controlled object ends up in the desired state. The motor command thus causes a new state of the object that we have labelled ‘actual state’ in Fig. 9.11.

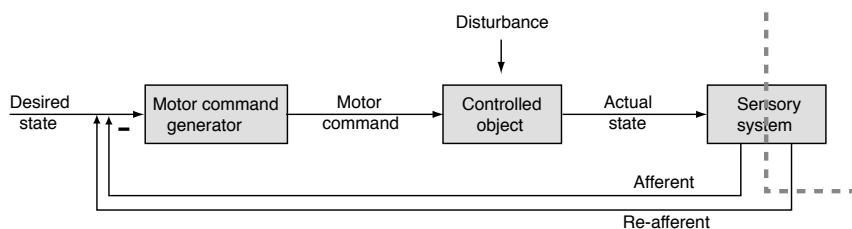


Fig. 9.11 Negative feedback control and the elements of a standard control system.

It is a major question how to find and implement an appropriate and accurate motor command generator. In principle we can utilize a feedforward mapping network that is trained on examples of the motor actions. Whatever method we use, if the motor command generator is a perfect inverse model of the controlled object and can produce the right motor commands, and the controlled object responds perfectly, then we have reached the goal. However, in reality this almost never happens as many circumstances can prevent such idealized motor movements. We have indicated this in the figure by a disturbance signal to the controlled object, which can include internal noise, external influences such as different loads on the system, ageing of the system, different temperatures altering physical properties, etc. In such cases we might not reach the desired state and we have to initiate a new movement to compensate for the

discrepancy between the desired and actual state of the controlled object. It is then necessary to estimate this discrepancy, which demands a sensory system. For arm movements we could, for example, use visual feedback as mentioned above. The sensory feedback is then generated outside of our body, and we include a dotted line running through the sensory system box to indicate sensory cues originating inside and outside of our body. The visual input about the position can be converted into a feedback signal that we call *re-afferent*. Sensory feedback can also originate from internal sensors in the body. For example, neural signals are generated by the contraction of muscles that are fed back to the central nervous system, so-called *proprioceptive feedback*. We labelled this type of sensory feedback *afferent* in Fig. 9.11.

The sensory feedback can be used in various ways to regulate the system. Fig. 9.11 illustrates the simplest example, that of a *negative feedback controller*. The sensory signal that indicates the actual state of the system is therein subtracted from the signal that specifies the desired state. This produces a new desired state, which is sometimes called a *motor-error* signal. This motor-error signal for the correction movement, generated with the help of the feedback signal, generates a new movement that will probably be closer to the desired state. Of course, the sensory feedback has to be converted into the right reference frame to be used by the motor control system (not included in the figure). The basic feedback control systems often work well and are commonly used in mechanical and electrical devices. However, there are several factors that make this simple scheme insufficient for some biological systems. For example, the sensory feedback is usually too slow for many control tasks, and the necessary accuracy is also not without problems. Proprioceptive feedback can be faster but might not be accurate. We therefore turn to more advanced systems that we think are at work in the brain.

9.5.2 Forward and inverse model controller

Two refined schemes for motor control with slow sensory feedback are illustrated in Figs. 9.12. The first one employs some subsystems that mimic the dynamic of the controlled object and the behaviour of the sensory system. These subsystems are called *forward models*. If these models are good approximations of the real systems they are modelling, then we can use the output of these systems, instead of the slow sensory response, as feedback signal. The models have, of course, to be gauged against the real system during ongoing learning. Thus, the sensory feedback is used to change the behaviour of the forward model. The forward model, which is divided into a *dynamic* and *output* component in the figure, influences the sensory feedback to improve performance in the main control loop. This scheme works as long as the changes in the systems that they mimic are much slower than the time scale of the movement that is controlled, which is often the case in biological motor control situations.

The second scheme, shown in Fig. 9.12B, employs an inverse model instead of the forward model in the previous scheme and is therefore called an *inverse model controller*. The inverse model, which is incorporated as a side-loop to the standard feedback controller, learns to correct the computation of the motor

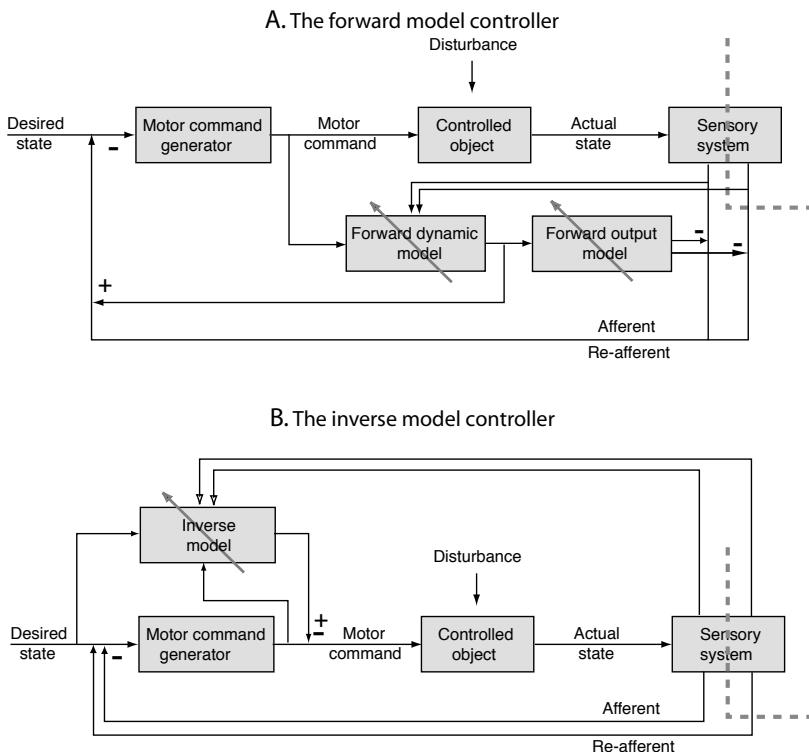


Fig. 9.12 Two advanced control systems which incorporate models for corrective adjustments in the system. These models are trained with sensory feedback, which is indicated by the arrows going through these components.

command generator. The reason that this scheme works is similar to that for the previous example. The sensory feedback can be used to make the inverse model controller more accurate while it provides the necessary correction signals in time to be incorporated into the motor command. Both control systems, the forward model controller and the inverse model controller, are robust controllers and could well be implemented in the brain. For example, such models have been discussed as models for the cerebellum, as outlined below. Later we discuss another control system, the adaptive critic, which has been proposed as a model of the basal ganglia.

9.5.3 The cerebellum and motor control

Are these control schemes used in the brain? There is some evidence that adaptive controllers are realized in the brain and are vital for our survival. The above-mentioned adaptation to the changed physics of the controlled system (for example, arm stiffness) or the variation of the sensory system (prism glasses) hints at an adapting motor control system. Both control schemes, the forward model and inverse model controller, can be realized by mapping networks, and it is a question of identifying characteristic components and signals in order to show which control scheme is realized in the brain.

A brain area that has long been associated with motor control is the *cerebellar*

cortex or cerebellum. The anatomy of the cerebellum displays great regularity and is summarized schematically in Fig. 9.13. Inputs from different sources enter the cerebellum through *mossy fibres* that contact *granule cells* and *Golgi cells*. Granule cells are probably the most numerous cells in the brain, estimated to be in the order of 10^{10} – 10^{11} . Granule cells exceed the number of mossy fibres by a few hundred times, which makes this architecture a candidate for expansion recoding, as discussed in Chapter 8. The granule cells, and some other intermediate neurons, provide a major input to *Purkinje cells* through *parallel fibres*, each Purkinje cell receiving as many as 80,000 inputs from different granule cells. Purkinje cells in turn provide the (inhibitory) output of the cerebellum.

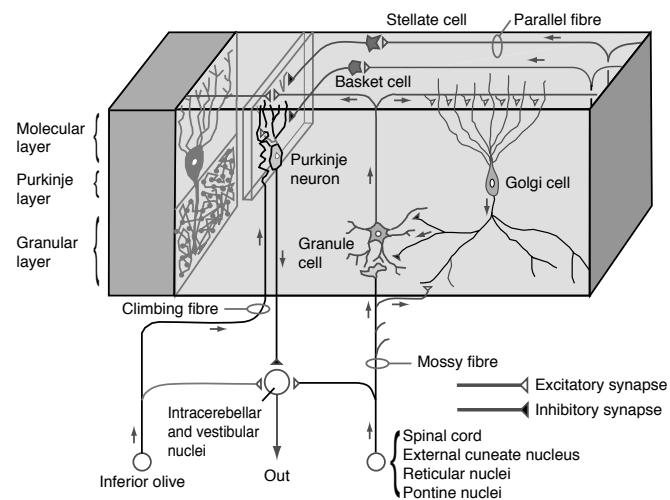


Fig. 9.13 Schematic illustration of some connectivity patterns in the cerebellum. Note that the output of the cerebellum is provided by the Purkinje neurons that make inhibitory synapses. Climbing fibres are specific for each Purkinje neuron and are tightly interwoven with their dendritic tree. [Adapted from Albus, *Mathematical Bioscience* 10: 25–61 (1971).]

The Purkinje cells also receive input from so-called *climbing fibres* from the *inferior olive*, each cell with its own climbing fibre. This one-to-one architecture is unique in the brain, and the climbing fibres have long been speculated to provide a teaching signal to the Purkinje cells. The cerebellum can be viewed as a mapping network, and such networks can learn to predict specific output from input patterns, as discussed in Chapter 6. It is thus possible that the cerebellum implements a forward or inverse model, or another adaptive model that can be used by the motor system in the midbrain or brainstem to guide motor functions.

9.6 Reinforcement learning

In supervised learning as outlined in Chapter 6, we assumed that a teacher supplies the exact desired state of each output node in the network. Such a teacher has to supply very detailed information, which seems unrealistic in common learning situations. More realistic are teaching procedures with only limited feedback. For example, the feedback for a student's performance might

only be binary such as ‘correct’ or ‘incorrect’, or ‘good’ and ‘bad’. In animal experiments, such a training signal following a correct response is often supplied with a food reward. An absence of a food reward can indicate a ‘false response’ to the animal. Furthermore, feedback is often only provided at the end many actions, and we need to solve how to reward components of actions that have contributed to successful outcomes. This section discusses such learning systems.

9.6.1 Classical conditioning and the reinforcement learning problem

Learning with reward signals has been studied by psychologists for many years under the term *conditioning*. An example of classical conditioning is shown in Fig. 9.14. In the illustrated experiment, we place a rodent in a box that has three chambers, two doors, and two buttons. The rodent is placed in the chamber with the two buttons, and the doors to the other two chambers, which contain food, are initially closed. A press of a button can open a door to the corresponding food chamber. The system is prepared in such a way that a press of the button only opens a door within a small time interval after the time indicated by an auditory signal such as the ringing of a bell. The rodent can run around freely in the button chamber, but initially does not know the conditions under which the door will open. However, it may happen that the rodent will press by chance the left button in the required time after the bell rings. The door then opens and the rodent gets some small food reward. The rodent will then quickly associate the ringing of the bell with the motor action it has to take, pushing the button, to get a reward. This implies that the system must be able to associate the ringing of the bell with the occurrence of the reward later in time. This condition has been termed the *temporal credit assignment problem*, the question of how to assign or divide credit for a reward to a series of previous actions. Neural mechanisms have to be able to solve this problem.

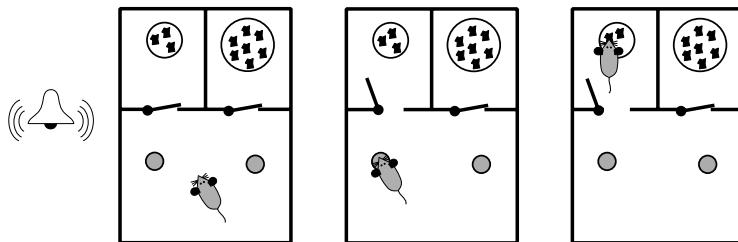


Fig. 9.14 Classical conditioning and temporal credit assignment problem. A subject is required to associate the ringing of a bell with the pressing of a button that will open the door to a chamber with some food reward. In the example the subject has learned to press the left button after the ringing of the bell. This is an example of a temporal credit assignment problem. It is difficult to devise a system that is still open to possible other solutions such as a bigger reward hidden in the right chamber.

The hypothetical experiment shown in Fig. 9.14 includes another chamber with a larger food reward, which the rodent would certainly prefer. The rodent was conditioned in this example by chance to open the left door after the ringing of the bell. If the rodent always stuck to the initial conditioned situation it

would never learn about the existence of the larger food reward. However, if the rodent is running around randomly in the button chamber before the bell rings, it could still happen that it presses the right button before running to the left button. The opening of the right door and the subsequent larger food reward then changes the association of the auditory signal to a new motor action, that of pressing the right button. This is an example of *stochastic escape* that can balance *habit* versus *novelty*. The hypothetic experiment discussed here thus alludes to several issues which must be solved by an agent to efficiently learn from reward. This includes the evaluation of environmental conditions and possible actions in terms of their value for the agent, which can be a learning problem by itself. The agent must further be able to learn to associate between environmental conditions and possible actions with future reward, and the agent must learn to select appropriate actions while still being able to explore novel actions.

The reinforcement learning problem can be formulated more generally by introducing *policies* and *value functions*. In the following we denote the primary reward given to an agent when taking an action a from state s with $r(s, a)$. The agent's *goal* is to maximize future reward, R . Future reward can be related to the primary reward in various ways. For example, future reward at some time t , $R(t)$ could be the summed (or averaged) reward in some time window before time t . We will specify this later. The agent's *policy*, $\pi(s, a)$, is the probability of taking action a when being in the state s . An example of a policy is the ϵ -greedy procedure, in which the agent takes most of the time action a which maximizes his expected reward, and takes different actions only in a small number of instances to facilitate stochastic escape.

In the control problem of reinforcement learning, we want to change the policy of the agent to maximize future reward. In order to achieve this goal, the agent must learn to predict future reward when the agent is in state s and follows policy π . This prediction of future rewards is called the *state value function*, $V^\pi(s)$. This value function is appropriate if we evaluate a specific policy which selects a specific action. In some cases, it might be more appropriate to evaluate different actions from a specific state. Such an evaluation is summarized in an *action value function*, $Q^\pi(s, a)$. We will first concentrate on the *evaluation problem* in reinforcement learning, that of estimating future reward for a given policy, or, in other words, to calculate value functions from experiences of the agent that is acting under a specific policy. We will specifically introduce a method called *temporal difference (TD) learning* to calculate value functions.

Using reinforcement learning for control systems consists of an iterative method of evaluating and changing policies to maximize future reward. There are different ways to realize such systems. For example, the agent could follow a certain policy while using prediction form the value function to evaluate future reward from alternative policies. Such algorithms are said to be *off-policy*. In contrast, we discuss here mainly an *on-policy* control algorithm, where the agent follows specific policies and evaluates them while using the evaluation also to change the policy on-line.

9.6.2 Temporal delta rule

In order to introduce implementations of systems which learn from reward within neural architectures, we discuss in the following a basic example. In this example, the action of the agent is fixed. The purpose of the system introduced now is to predict future reward with a fixed policy, or in other words, to evaluate the state value function $V^\pi(s)$, from episodes in which some states are rewarded, possibly at a later time. Let's assume that this state value function, relating states to reward, is simple enough to be learned by a linear population node as discussed in Chapter 6 and illustrated in Fig. 9.15A. The input of the node represents a state at time t , $s(t)$, which is conveyed by a specific pattern of rates, $r_i^{\text{in}}(s)$ in the input channels. The output of the linear node is

$$V(s) = \sum_i w_i(t) r_i^{\text{in}}(s). \quad (9.26)$$

The rate values of the perceptron are, of course, time dependent since the input states s are time dependent. We also indicated explicitly the time dependence of the weight values, $w_i(t)$, since they will change over time as a function of reinforcement signals.

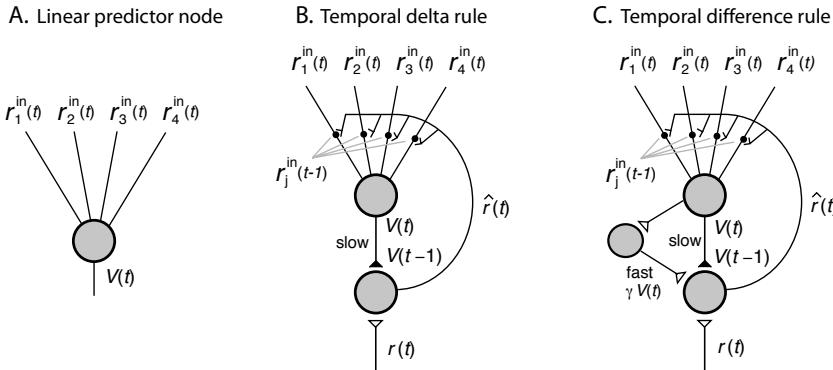


Fig. 9.15 Neural implementation of reward learning algorithms. (A) Linear predictor node. (B) Temporal delta rule with primary reward r . The output of the predictor node is indicated at time $t - 1$ since the connection is considered to be slow. (C) Temporal difference rule with a fast side loop.

The agent is taking an action in response to the state according to the fixed policy, π , and this action results in a reward at the next time step, $r(t + 1)$, and we want to update the weights at this time of reward to reflect the new knowledge. We first consider the case in which the system should predict the reward for each state, which corresponds to a value function that treats the reward following each state independently. In this case, the actual reward can be used as the desired output, and we can use the delta rule discussed in Chapter 6 to train the linear perceptron,

$$w_i^{t+1} = w_i^t + \alpha * \hat{r}(t+1) r_i^{\text{in}}(s). \quad (9.27)$$

The parameter α is a learning rate, and we introduce in this equation the *effective reinforcement signal*

$$\hat{r}(t+1) = r(t+1) - V(t). \quad (9.28)$$

The learning rule, eqn 9.27, is sometimes called a temporal delta rule because of the temporal difference between predicted and actual reward, but the standard delta rule, discussed in Section 6.1.5, can be applied to this example when supplying the reward at the same time the occupancy of the corresponding state. However, we keep this time difference to stress some necessary memory in the neural implementation. Thus, the system needs to remember the value $V(s^t)$ for one time step. We include this memory in the model shown in Fig. 9.15B as slow output channel, or axonal delay. In addition, the system must remember the input values ($r_i^{\text{in}}(s)$) for one time step because we attribute the cause for the reward to taking the fixed action from the previous state. A more general case would be to attribute the reward to different state-action pairs in the past, for which an *eligibility trace* is needed. The eligibility trace for the example discussed here is simply a short-term memory which can easily be implemented in neural tissue. The eligibility trace is shown in Fig. 9.15B as short-term memory at the synapses from the node that supplies the training signal.

The eligibility trace and the axonal delay solves here the temporal credit assignment problem. The weights of the input channels to the node that learns the state value function depend on its delayed output and the primary reward (reinforcement) signal r . The primary reinforcement signal is assumed to have an excitatory effect on a node that mediates learning, whereas the prediction node has an inhibitory effect on the node which mediates training. This *training node* therefore calculates the effective reinforcement signal of equation 9.28. This effective reinforcement signal has to be conveyed to all the synapses of the prediction node, where it has to be correlated with an eligibility trace to produce the appropriate weight change proposed in eqn 9.27. This model of reward learning is equivalent to the *Rescorla-Wagner theory* in classical conditioning. The model can produce one-step ahead predictions of a reward signal and corresponds to a particular choice of the state value function. The effective reinforcement signal is zero when the activity of the prediction node matches the primary reinforcement signal.

9.6.3 Temporal difference learning

Learning in the previous model is restricted to the prediction of reward in the next time step, or, in other words, to an agent that values only immediate reward. Much more important for the survival of an individual is the ability to predict future reward at different time steps or even whole series of rewards. A more appropriate state value function could have the form

$$V(t) = \alpha_1 r(t+1) + \alpha_2 r(t+2) + \alpha_3 r(t+3) + \dots, \quad (9.29)$$

where we included parameters α_i to specify the weights for rewards at different times. For example, we can still give an immediate reward a large weight, but might also value the state, often somewhat less, for its potential to produce additional reward in the future. The prediction of this reinforcement value seems to be a hopeless task since we do not know the future rewards and have to wait a long time before receiving it. In the case of the temporal delta rule, the system had to remember the events in the previous time steps to solve this

problem, and it seems that the system would need now an infinite memory of all previous events to learn the above state value function.

However, for specific values of weights α_i , the problem becomes tractable. We consider here a simple realization of such a model with $\alpha_i = \gamma^{i-1}$, where the *discount factor* is $0 \leq \gamma < 1$. The value function is then

$$V^\pi(t) = r(t+1) + \gamma r(t+2) + \gamma^2 r(t+3) + \dots \quad (9.30)$$

The Rescorla–Wagner model corresponds to the case $\gamma = 0$. To derive the learning rule for this value function, let us assume that we already have a system that can predict the reinforcement value, eqn 9.30, at time t correctly. Lets denote this perfect prediction as V^* . This implies that we were able to predict the correct reinforcement value at the previous time step, that is,

$$\begin{aligned} V^*(t-1) &= r(t) + \gamma r(t+1) + \gamma^2 r(t+2) + \dots \\ &= r(t) + \gamma[r(t+1) + \gamma r(t+2) + \dots]. \end{aligned} \quad (9.31)$$

The expression in square brackets of eqn 9.31 is equal to the expression on the right-hand side of eqn 9.30 so that we can replace this with $V^*(t)$. We therefore have the condition

$$r(t) + \gamma V^*(t) - V^*(t-1) = 0, \quad (9.32)$$

which must be true if we have a perfect prediction. If we do not have a perfect prediction, then the equation does not hold. *Richard Sutton* and *Andrew Barto*, who developed much of the reinforcement learning theory, proposed to minimize the *temporal difference error*

$$\hat{r}(t) = r(t) + \gamma V^\pi(t) - V^\pi(t-1). \quad (9.33)$$

This temporal difference should be used in the learning rule eqn 9.27 as effective reinforcement signal to update the weights. Compared with the temporal delta rule ($\gamma = 0$), we only need the additional value $V^\pi(t)$. Fig. 9.15C shows a possible neuronal implementation of a temporal difference learning, which includes a fast side-loop with a decay factor that conveys the value $\gamma V^\pi(t)$ to the node that calculates the primary reinforcement signal.

The learning of a state value functions are illustrated in Fig. 9.16. In this example, fice input patterns are chosen randomly, and always supplied in a fixed sequence, called an episode. The reward following pattern 2 is $r(\text{pattern}_2) = 1$, the reward following pattern 4 is $r(\text{pattern}_4) = 0.5$, and the reward for the other patterns is zero. In the example shown on the left-hand side, Fig. 9.16A, the state value function is that of treating reward for each pattern independently, ($\gamma = 0$). The example shown on the right-hand side, Fig. 9.16B, uses the state value function from eqn 9.30 with $\gamma = 0.8$ with the same training patterns used in Fig. 9.16A. The upper curves show the development of the temporal difference error,

$$\text{TDerror} = \sum_i |\hat{r}(\text{pattern}_i)|. \quad (9.34)$$

This error becomes very small, which demonstrate that the system can learn the value functions. The learned state value function after 100 episodes, shown in the bottom graphs of Fig. 9.16B, has a similar shape as the in the case of

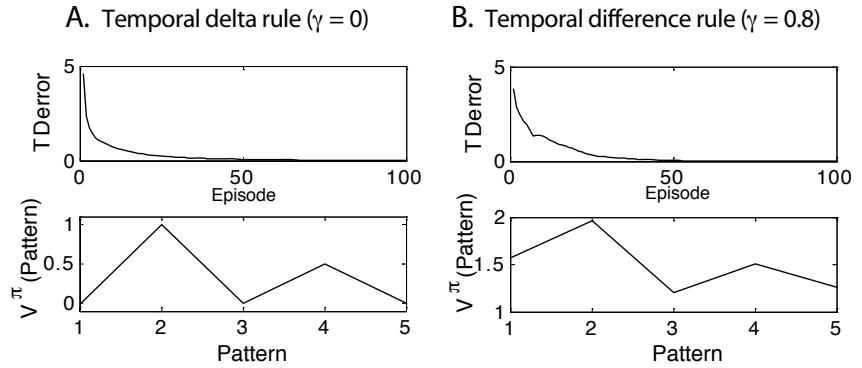


Fig. 9.16 Example of learning value functions of five random patterns, where patterns 2 and 5 receive primary reward $r(2) = 1$ and $r(4) = 0.5$. (A) Value function which considers only primary reward (eqn 9.30 with $\gamma = 0$), which can be learned by with a delta rule. (B) Value function where primary reward is assigned to all patterns in a sequence with decaying weights (eqn 9.30 with $\gamma = 0.8$).

$\gamma = 0$. However, patterns other than 2 and 4 are now also valued since some of the primary rewards from pattern 2 and 4 are now associated with them.

It is straightforward to apply the temporal difference method to learning action value functions, $Q(s, a)$. One needs only to keep scores for each action taken from individual states. The temporal difference method for evaluating the action value function, for a given policy, is summarized in Table 9.1. The states s' are thereby the states following states s by taking action a , and a' is the action following a . The previously discussed evaluation of a state value function $V^\pi(s)$ corresponds to the algorithm where $Q(s, a)$ is replaced with $V^\pi(s)$ in Table 9.1 (Line 7)

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha[r + \gamma V^\pi(s') - V^\pi(s)]. \quad (9.35)$$

The action chosen in Line 6 does not influence the update of the value function and is only needed in the next step of the episode. The algorithm in Table 9.1 is termed *on-policy*, since the value function is updated for the specific actions taken in each step of an episode. However, this is not the only possible choice. A very powerful method was invented by *Watkins* and is called *Q-learning*. In this method, all possible actions are evaluated for each reached state during an episode, and the value function is updated based on the action which estimates

Table 9.1 Summary of on-policy TD learning of action value function

Initialize value function $Q(s, a)$ arbitrary
For each episode
Initialize state s , choose action a from s using policy π
Repeat for each step of episode
Take action a , observe reward r and next state s'
Choose action a' from s' using policy derived from Q
$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
$s \leftarrow s', a \leftarrow a'$

Table 9.2 Program TDlearning.m

```

1 %% TD learning
2 clear; clf; hold on;
3 gamma=0.8; reward=[0 1 0 0.5 0]; pattern_vector=rand(10,5);
4 w=rand(1,10); V_mem=0; previous_state=5;
5
6 for episode=1:100; TDerror(episode)=0;
7   for pattern=1:5
8     V=w*pattern_vector(:,pattern);
9     rhat=reward(previous_state)+gamma*V-V_mem;
10    w=w+0.2*rhat*pattern_vector(:,previous_state)';
11    TDerror(episode)=TDerror(episode)+abs(rhat);
12    previous_state=pattern; V_mem=V;
13  end
14 end
15
16 subplot(2,1,1);
17 plot(TDerror); xlabel('Episode'); ylabel('TD error')
18 subplot(2,1,2);
19 plot(w*pattern_vector); xlabel('Pattern'); ylabel('V(Pattern)')

```

the largest return,

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_a Q(s', a) - Q(s, a)]. \quad (9.36)$$

This method is termed *off-policy* since the learning is not based on the specific action taken during an episode.

Simulation

Fig. 9.16 was produced with the program `TDlearning.m` listed in Table 9.2. In Line 3, the discount factor and the reward amount are specified, and random patterns for the five states are chosen. Each state is thereby represented by a random vector with 10 components. In Line 4, random initial weights are chosen, and the short-term memories are initialized. The initial values for the memory states are not critical as the memory of these states will disappear after one iteration. In Line 6, we start the loop over episodes, where in each episode we present all pattern in a fixed sequence (loop in Line 7). For each pattern, we calculate the value function from the current approximation (Line 8), calculate the learning signal \hat{r} from eqn 9.33 (Line 9), and update the weights with a learning rate $\alpha = 0.2$. We update the short-term memories in Line 12.

To plot the results we use the MATLAB construct of `subplot`. The first parameter specifies how many rows of subplots should be produced, and the second parameter specifies the number of columns. The third parameter makes

the n -th subplot active. The first subplot therefore contains the temporal difference error, whereas the second subplot shows the output of the perceptron, which is the function $V^\pi(s)$ for each of the five states after learning on 100 episodes.

9.6.4 The actor–critic scheme and the basal ganglia

In the previous section we have discussed methods to estimate value functions, which can then be used to evaluate policies. When using reinforcement learning for control, we want to change policies to maximize the return for an agent. The full methods do then consist of interleaved procedure, in which policies are changed according to estimated value functions, and then the new policy is evaluated by estimating the value function for this policy. Interestingly, it is not necessary to estimate the value function perfectly for a given policy or to find directly the best policy for a given value function. The system can start out with a weakly functional system, and reinforcement learning improves the system incrementally. It can be shown that such a bootstrapping system converges to optimal solutions as long as the system is ergodic, which means that each possible state of the environment is reached in finite time. However, rather than following here the many advanced engineering applications in this area, we will discuss some examples that have been implicated with brain processing.

Sutton and Barto have incorporated temporal difference learning into a powerful control method called the *actor–critic scheme*, illustrated in Fig. 9.17. Note the similarities of this control scheme to the inverse model controller outlined in Fig. 9.12B. The critic is, however, designed to predict the correct motor command for accurate future actions and can thus supervise the motor command generator. The motor command generator is often called *actor* within this framework. The *adaptive critic* estimates value functions and uses them to guide the actions of the agent. This scheme has proven to be very useful in engineering applications such as controlling elevators or adjusting the parameters of a petroleum refinery's operation. Neural implementations have been suggested, and it has been noted that there are some similarities with specific architectural features in the *basal ganglia*.

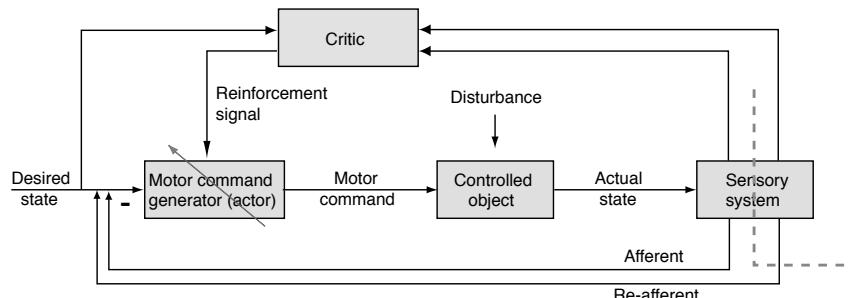


Fig. 9.17 Adaptive critic controller.

The basal ganglia, which are known to be instrumental in the initiation of motor commands, have many of the structural components required to implement an adaptive critic and to supervise actors that can control the initiation of motor movements. The basal ganglia are a collection of five subcortical nuclei as illustrated in Fig. 9.18. They receive cortical and thalamic input mainly through the *putamen* and the *caudate nucleus*, together called the *striatum* which comprises the input layer of the basal ganglia. The information stream then runs through the *globus pallidus* (with an internal and external segment) to the major output layer, the *substantia nigra pars reticulata*. The internal side-loop from the *globus pallidus* via the *subthalamic nucleus* is also important for our next discussion. In addition, note that the *substantia nigra pars compacta* projects back to the striatum with neurons that use *dopamine* as neurotransmitter.

The information streams within the basal ganglia are thought to be segregated (to a certain extent) within interleaved modules that are called matrix modules and striosomal modules, respectively. A suggested implementation of the actor–critic scheme in the basal ganglia is shown in Fig. 9.19. The input layer of the basal ganglia is rich in *spiny neurons* (SP), which receive massive cortical (C) connections. The spiny neurons in the striosomal module (SPs) also receive projections from dopaminergic neurons (DA) in the substantia nigra pars compacta (SNC) which synapse on to spines of spiny neurons in the caudate and putamen. It is possible that dopaminergic input is able to alter the efficiencies of specific cortical inputs that are marked with an eligibility trace. The neurons shown in the basal ganglia are also inhibitory so that the dopaminergic neurons in the SNC are inhibited by SPs activity. The subthalamic side-loop, in contrast, disinhibits the DA, which can result in some excitation of DA neurons proportional to the inputs from this side-loop and a primary reinforcement signal. If, in addition, the side-loop is faster than the direct SPs–DA influence, then it is possible that the striosomal module implements the critic that minimizes the temporal difference, as discussed above (compare Fig. 9.15C).

Several experiments show signals of neural activities in the basal ganglia that can be related to reinforcement learning. For example, work by *Wolfram Schultz* has shown how activities of dopaminergic neurons in SNC relate well to the temporal difference error in reinforcement learning. An example is shown in Fig. 9.20. These neurons respond to an unexpected reward, but do not respond to a reward after the agent has learned to associate the reward with a stimulus. The activity of the dopaminergic neurons in SNC is also transferred when introducing an earlier predictive stimulus. Furthermore, these neurons respond with decreased activity when an expected reward is not given. This behaviour corresponds well to the effective reinforcement signal in the schemes discussed above and simulated below. An example is also shown in Fig. 9.20 where reward was only given at the time of presenting Pattern 3. Before learning, the neuron which conveys the effective reinforcement signal responds only at the time following the reward (the time Pattern 4 is presented), but the response is transferred to the time of presenting Pattern 3 after learning. At episode 50, a new pattern (Pattern 2) is introduced at the time step preceding Pattern 3, and the neurons transfers the response to this pattern. The removal

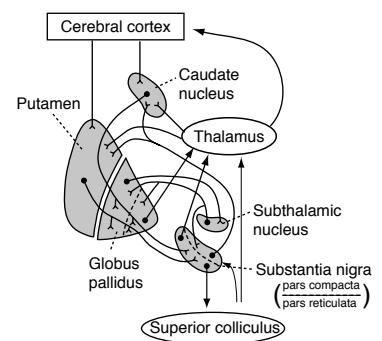


Fig. 9.18 Anatomical overview of the connections within the basal ganglia and the major projections comprising the input and output of the basal ganglia. [Adapted from Kandel, Schwarz, and Jessell, *Principles of neural science*, McGraw-Hill, 4th edition (2000).]

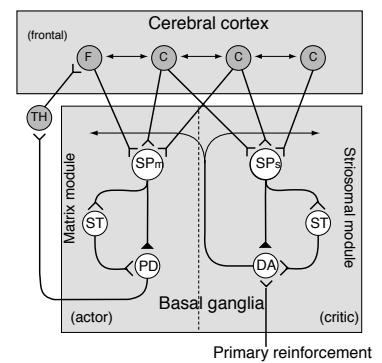


Fig. 9.19 Actor–critic model of the basal ganglia with cerebral cortex (C), frontal lobe (F), thalamus (TH), subthalamic nucleus (ST), pallidus (PD), spiny neurons in the matrix module (SP_m), spiny neurons in the striosomal module (SP_s), and dopaminergic neurons (DA). [Adapted from Houk, Adams, and Barto, in *Models of information processing in the basal ganglia*, Houk, Davies, and Beiser (eds), MIT Press (1995).]

of an expected reward leads to a negative value of the effective reinforcement learning signal, not shown in the simulation.

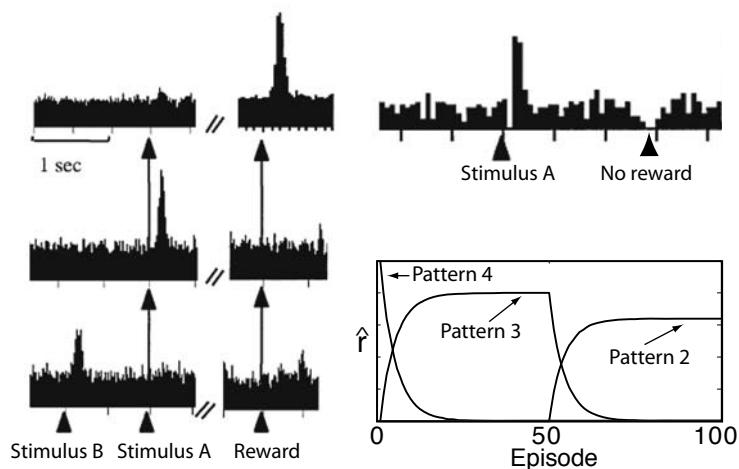


Fig. 9.20 Activities of dopaminergic neurons from Schultz and colleagues [adapted Suri, *Neural Networks* 15: 523–533 (2002)] and results from simulations discussed below.

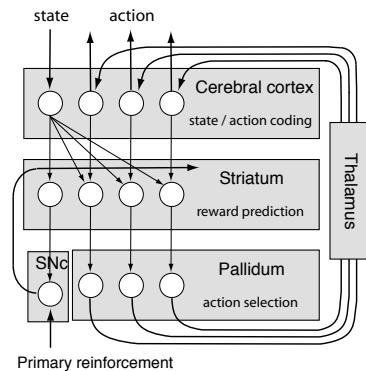


Fig. 9.21 Q-learning scheme of the basal ganglia [adapted from Doya, *HFSP Journal* 1: 30–40 (2007)].

The similarities between reward responses of dopaminergic neurons in the SNC and the effective reinforcement signal in temporal difference learning have contributed largely to the hypothesis of reinforcement learning theories of the basal ganglia. This part of the reinforcement system corresponds largely to the critic in the control system. Specific implementations of actors in the basal ganglia have been discussed much less in the literature. One of the earliest suggestions was that the matrix module could implement the actor. Dopaminergic neurons project to these spiny neurons (SPm) and could therefore alter the C-SPm weights in a fashion similar to that in the C-SPs connections. The only difference is that these neurons project to the internal segment of the pallidus and the substantia nigra pars reticulata (SNr), which are thought to be the major output layers of the basal ganglia. The output of the basal ganglia can, in such a way, control the initiation of specific motor actions that are associated with reward.

The classic proposal of an actor-critic implementation in the basal ganglia discussed above leaves many open questions and some questionable assumptions. While there are still many open questions regarding the specific implementation of reinforcement learning in the basal ganglia and the more refined role of the basal ganglia in brain processing, there is a wide consensus that the basal ganglia do contribute to some form of reinforcement learning. More recently, Q-learning has also been mapped on to basal ganglion functions in the work of leading researchers such as *Kenji Doya* and *Peter Dayan*. An example is illustrated in Fig. 9.21 where the striatum is hypothesized to calculate a state-action value function, and the dopaminergic neurons in SNC calculate the TD error that is used to update cortico-striatal connections. The state-action value function is then used in the pallidum and thalamus for stochastic action selection. Other researchers, such as *Peter Redgrave* and *Kevin Gurney* also pointed out that the basal ganglia might have a large role in discovering novelty

Table 9.3 Program TDlearning2.m

```

1 %% TD learning
2 clear; clf; hold on;
3 gamma=0.8; reward=[0 0 1 0 0]; pattern_vector=zeros(10,5);
4 pattern_vector(:,3)=[1;0;0;0;0;0;0;0;0];
5 w=zeros(1,10); V_mem=0; previous_state=5;
6
7 for episode=1:100; TDerror(episode)=0;
8     if episode==50; pattern_vector(:,2)=[0;0;0;1;0;0;0;0;0;0]; end;
9     for pattern=1:5
10         V=w*pattern_vector(:,pattern);
11         rhat(episode,pattern)=reward(previous_state)+gamma*V-V_mem;
12         w=w+0.2*rhat(episode,pattern)*pattern_vector(:,previous_state)';
13         TDerror(episode)=TDerror(episode)+abs(rhat(episode,pattern));
14         previous_state=pattern; V_mem=V;
15     end
16 end
17
18 plot(rhat); xlabel('Episode'); ylabel('rhat');

```

actions, which are also essential in establishing task-relevant behaviour. This area is a nice example where theoretical and experimental research strongly benefit from each other.

While we concentrated our discussion of reinforcement learning in the brain on the basal ganglia, it should not be forgotten that there are other areas in the brain that have been associated with reinforcement learning, or at least association of reward contingencies with specific motor actions. Some brain areas that have been implicated with making reward associations are, for example, the prefrontal cortex and the subcortical area called *amygdala*. Both areas receive projections from many senses and are therefore placed strategically to form associations between different modalities and reward contingencies. Bilateral damage of the amygdala is known to considerably impair such associations. Furthermore, it has been shown that neurons in the *orbitofrontal cortex* adapt their response to stimuli after changing their reward associations. Dopaminergic neurons also project into the frontal cortex so that reward mechanisms could originate predominantly in the frontal cortex as opposed to the hypothesis of the basal ganglia discussed above. Finally, there are also other neurotransmitters and specific reward systems that might be important in reinforcement learning. For example, serotonin is a neurotransmitter that has been associated with the modulation of such things as mood, appetite, sexuality, and aversive signals, which are all factors that can influence reward values and action selection. Also, human planning, as well as abilities to evaluate long-term goals, are not yet covered with the simple reward associations used in the experiments discussed here.

Simulation

The simulation in Fig. 9.20 was produced with program `TDlearning2.m` listed in Table 9.3. This program includes only a few changes compared to program `TDlearning.m`. We specified in Line 2 that reward is received only at step 3 of an episode, and we used fixed pattern vectors in these simulations. The weight values are initially set to zero for clarity. The only other changes are that we now record the values \hat{r} for each time step in each episode (`rhat=rhat(episode,pattern)`), and that a new pattern at step 2 is added from Episode 50 in Line 8.

Further reading

The influential paper by Jacobs *et al.* (1991) popularized modular architectures, and Hinton (1999) introduced the product of experts. The presentation of modular recurrent networks follows largely Yaneer Bar-Yam (1997); while this book is on complex systems, neural networks are as a prominent example. The two chapters on neural networks (chapters 2 and 3) include a nice discussion of modular networks as well as some speculation about the role of sleep in training such structures. The principal idea of the restrictions on the number of modules in modular architectures is similar to the arguments outlined in this chapter, although some details in the derivation are different. A nice example of modelling interacting recurrent networks in some brain processes is given by Rolls and Stringer (2001). Some classical work on adaptive models of the mind are Nilsson (1965), Selfridge (1958), and Minsky (1986). A good discussion on models of working memory is given in Miyake and Shah (1999).

A good review of the control schemes and the cerebellum by leading researchers in this area is Wolpert *et al.* (1998). Chapter 9 of Rolls and Treves (1998) includes a discussion of the involvement of the cerebellum and the basal ganglia in motor control, and Chapter 7 discusses of reward associations in the amygdala and the orbitofrontal cortex. The book by Houk *et al.* (1995) includes a highly readable introduction to the actor-critic scheme and temporal difference learning, as well as speculations about the relations of these schemes to the basal ganglia. Temporal difference learning and much of the theory of reinforcement learning was invented by Richard Sutton and Andrew Barto, and their book (Sutton and Barto, 1998) is

a well-written treatment of this subject in a general machine learning setting. Chapter 9 of Dayan and Abbott (2001) is highly recommended for a more in-depth discussion of reinforcement learning related to brain functions.

Robert A. Jacobs, Michael I. Jordan, and Andrew G. Barto (1991), *Task decomposition through competition in a modular connectionist architecture: the what and where tasks*, in *Cognitive Science* 15: 219–50.

Geoffrey Hinton (1999), *Products of experts*, in *Proceedings of the Ninth International Conference on Artificial Neural Networks*, ICANN '99, 1: 1–6.

Yaneer Bar-Yam (1997), *Dynamics of complex systems*, Addison-Wesley.

Edmund T. Rolls and Simon M. Stringer (1999), *A model of the interaction between mood and memory*, in *Networks: Computation in Neural Systems* 12: 89–109.

N. J. Nilsson (1965), *Learning machines: foundations of trainable pattern-classifying systems*, McGraw-Hill.

O. G. Selfridge (1958), *Pandemonium: a paradigm of learning, in the mechanization of thought processes*, in *Proceedings of a Symposium Held at the National Physical Laboratory*, November 1958, 511–27, London HMSO.

Marvin Minsky (1986), *The society of mind*, Simon & Schuster.

Akira Miyake and Priti Shah (eds.) (1999), *Mod-*

- els of working memory, Cambridge University Press.
- Daniel M. Wolpert, R. Chris Miall, and Mitsuo Kawato (1998), *Internal models in the cerebellum*, in *Trends in Cognitive Science* 2: 338–47.
- Edmund T. Rolls and Alessandro Treves (1998), *Neural networks and brain function*, Oxford University Press.
- James C. Houk, Joel L. Davis, and David G. Beiser (eds) (1995), *Models of information processing in the basal ganglia*, MIT Press.
- Richard S. Sutton and Andrew G. Barto (1998), *Reinforcement learning: An introduction*, MIT Press.
- Peter Dayan and Laurence F. Abbott (2001), *Theoretical neuroscience*, MIT Press.

This page intentionally left blank

10

The cognitive brain

This chapter is a continuation of discussing system-level models of the brain. We will take a further step to deriving more specific models of the cortex by dividing the brain representation into more layers, one layer that represents fairly low-level representations of sensory and action states, and other layers that represent increasingly abstract representational layers in the brain. We then discuss how invariant object recognition is solved in such architectures and how visual attention guides recognition and search. We then discuss a workspace hypothesis that shows how the brain is able to produce novel solutions to new tasks. After this we turn to a more general discussion of the brain theory in bidirectional layered models of cortex where anticipation will be a central feature. This section includes some outlines of specific models such as Bayesian formulations, the *Boltzmann machine*, and *adaptive resonance theory* (ART).

10.1 Hierarchical maps and attentive vision

Humans are able to recognize objects even though they vary in form, size, location and viewing angle, etc. For example, take a cup and move it in front of your eyes, rotate it, or view it from different distances. Recognizing the cup does not crucially depend on the viewing conditions. While we have seen that neural systems can learn to recognize objects through supervised learning in mapping networks, such recognition processes are quite sensitive against changes in the input vector, and the changes in the viewing condition of the cup result in very different activation patterns of the retina. Invariant object recognition has hence puzzled neuroscientists and engineers alike for a long time.

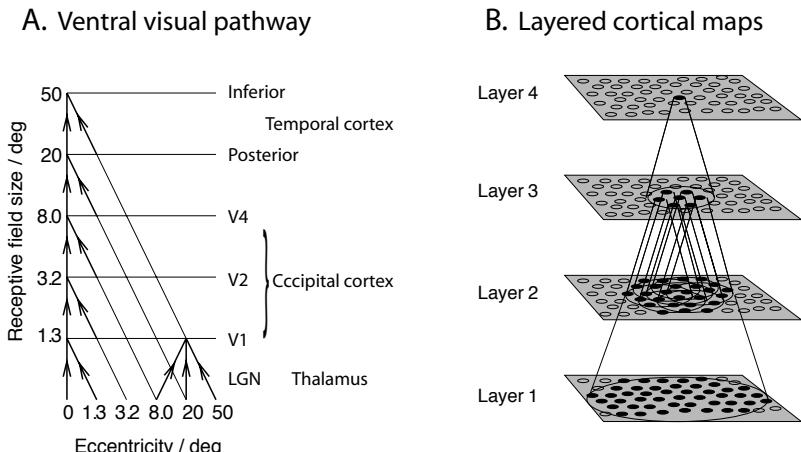
Solving invariant object recognition can be achieved in hierarchical networks. We discuss this issue in this section together with related questions of attention in the visual system, although there is no reason why similar processes should not apply to other sensory modalities.

10.1.1 Invariant object recognition

Most of the human brain is involved in visual processing to some degree, and vision research is a strong domain within neuroscience. Here we outline briefly a hypothesis of how invariant visual object recognition is achieved within the ventral visual pathway of the brain (see discussion of the what-and-where pathways in Section 9.1.2). We follow here the suggestions that have been advanced for several years by *Edmund Rolls*, *Simon Stringer* and other collaborators in an architecture they call *VisNet*.

10.1 Hierarchical maps and attentive vision	289
10.2 An interconnecting workspace hypothesis	295
10.3 The anticipating brain	298
10.4 Adaptive resonance theory	313
10.5 Where to go from here	319
Further reading	321

Fig. 10.1 (A) Illustration of the increase of receptive field sizes in the ventral visual pathway. (B) Layered model called VisNet of the ventral visual pathway, which consists of competitive layers where each node in a layer is connected to a restricted area of nodes in the previous layer. [Adapted from Stringer and Rolls, *Neural Networks* 21: 888–903 (2008), and references therein.]



The hierarchical levels in the ventral visual pathway are outlined in Fig. 10.1A. After some preprocessing of visual patterns in the retina, where important functions are implemented, such as adaptation to brightness, the patterns are mapped into the *lateral geniculate nucleus* (LGN) of the thalamus, which is the major target of the optic nerves from the eyes. Then, the signal reaches the primary visual cortex V1, and is then passed to higher cortical areas. In VisNet, these areas are typically associated with V2, and then areas in the posterior and inferior-temporal cortex. VisNet is therefore represented with four cortical layers, as shown in Fig. 10.1B. A crucial component of the model is that each node in a layer is connected to a spatially restricted area of nodes in the layer below. As a consequence, the receptive fields of the nodes, which is the area of the visual fields to which they can respond, increases with each level. The receptive fields in the inferior-temporal cortex are large and effectively cover the entire visual field.

Each layer in the model is a competitive map, where competition is implemented through adjustment of the firing threshold of nodes until a predefined sparseness is reached in each layer, as discussed in Section 7.5.1. The model is trained on sequences of patterns from movements of objects in the visual field, such as translation or rotations of objects. Weights between the layers are adjusted with Hebbian learning proportional to the activation of pre- and postsynaptic activity. Early versions of the model used a form of Hebbian learning with a trace rule, where some memory in the neural activity (trace) was used for Hebbian weight changes. Such rules use temporal associations between moving objects. However, recent simulations also showed that similar results can be achieved without a trace when objects in consecutive time steps have some overlap with previous neural representations. These rules thus use spatial associations. Simulations have shown that VisNet is able to learn invariant object recognition, including invariance to translation, rotation, and size. Recent studies also showed that multiple objects can be trained simultaneously. VisNet clearly demonstrates how a hierarchical, competitive network with increasing sizes of receptive field and corresponding increasing abstraction

representations, from an early image-based representation to an later object-based representation, can achieve invariant object recognition. Since this is a common architecture of cortical processing, the results should also hold for other sensory modalities.

10.1.2 Attentive vision

VisNet includes interactions within each layer to stabilize the sparseness, though the information flow between the layers is strictly feedforward. The next step is to consider top-down information flow in cortical models. Top-down information is crucial in cognitive processes. For example, when asking subjects to search for specific items in a visual scene, such as the top of the Eiffel tower in the example shown in Fig. 10.2. Such a *visual search* demands the top-down

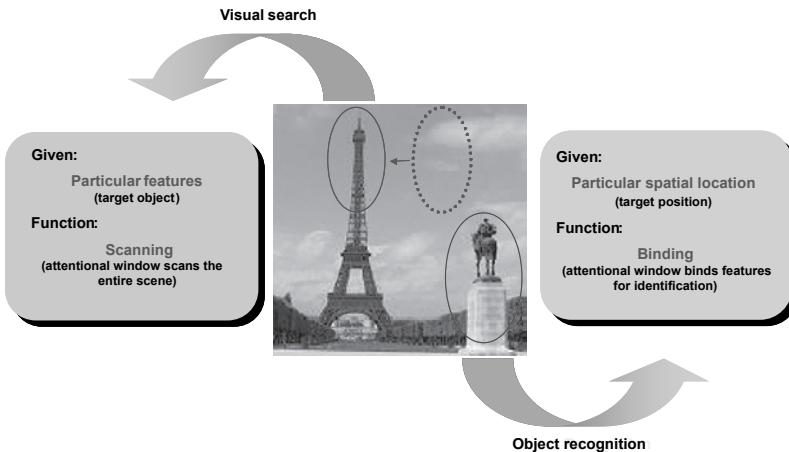


Fig. 10.2 Illustration of a visual search and an object recognition task. Each task demands a different strategy in exploring the visual scene [figure courtesy of Gustavo Deco].

influence of an object bias that specifies what to look for. In contrast, we can ask the subject to identify an object at a particular location, such as identifying the object in the lower right-hand corner of the picture in Fig. 10.2. This demands a spatial attention. *Gustavo Deco* and collaborators have developed a model that can shed some light on the processes involved in visual search and object recognition. The overall scheme of their model is outlined in Fig. 10.3. The model basically has three important parts: one that is labelled ‘V1–V4’, one that is labelled ‘IT’, and one that is labelled ‘PP’. We will outline the specific architectures and roles of these parts separately in the following. It is, however, good to keep in mind right from the beginning that the different parts do not work in isolation. Here we are particularly interested in how the different parts influence each other. Many of the features discussed here are generalized in Section 10.4.

Bottom-up input to the model came from visual scenes. In the model this is simulated by taking input images and decomposing them with *Gabor functions*. This corresponds to representations in the LGN since turning curves in the LGN can be parametrized with *Gabor functions*. Early parts of the cortical processing are represented with a module labelled ‘V1–V4’ in the model. In

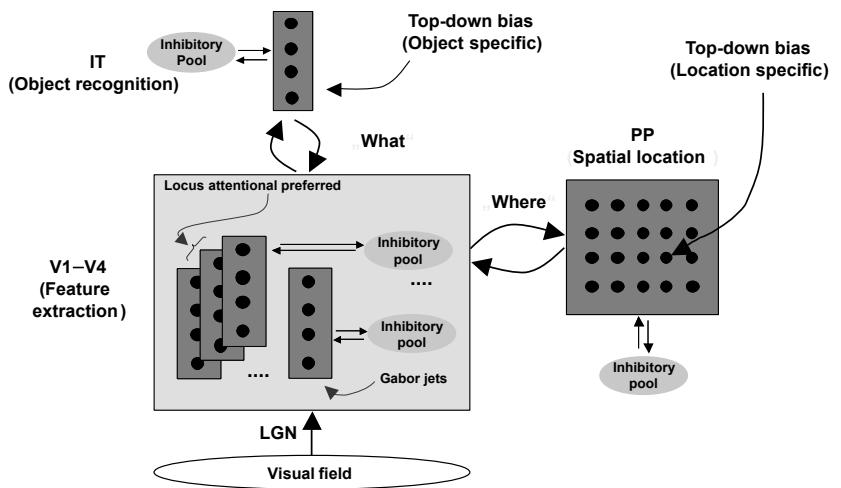


Fig. 10.3 Outline of a model of the visual processing in primates to simulate visual search and object recognition. The main parts of the model are inspired by structural features of the cortical areas thought to be central in these processes. These include early visual areas (labelled 'V1–V4') that represent the content of the visual field topographically with basic features, the inferior-temporal cortex (labelled 'IT') that is known to be central for object recognition, and the posterior parietal cortex (labelled 'PP') that is strongly associated with spatial attention [figure courtesy of Gustavo Deco].

the specific model implementation discussed here, this is only a single layer. However, hierarchical implementation of this model has also been done but is not essential for the following discussion. The principal role of this part of the model is the decomposition of the visual scene into features. This is known to occur at this early stage of visual processing in primates. For example, V1 neurons respond mainly to simple features such as the orientation of edges, and later areas are often specialized to represent other features such as colour, motion, or combinations of basic features.

From the modelling point of view it is not essential that we precisely rebuild all the details of the visual field representation in the brain, as we are primarily interested in other aspects of the visual processing. It is only important here that the feature representation in this part of the model is topographic in that features are represented in sections (shown as boxes in the 'V1–V4' module) which correspond to the location of the object in the visual field, as discussed in Section 5.1.5. Each section represents a feature of a part of the visual field as a vector of node activities. There is no global competition between the modules; only within each section is there an inhibitory pool that keeps the sparseness of the activity in each section roughly constant as discussed in Section 7.5.1.

The representation of the visual field, as decomposed in 'V1–V4', feeds into the module labelled 'IT' in the model. This part is aimed at modelling processes in the inferior-temporal cortex, which is known to be involved in object recognition. This recognition process is modelled as associative memory as outlined in Chapter 8. Point attractors of specific objects are then formed through the Hebbian-trained collateral connections within this structure. The connections between the 'V1–V4' and the 'IT' can also be trained with Hebbian learning. By placing, in turn, an object at all locations in the visual field that are covered by the sections in 'V1–V4', weights between the sections in 'V1–V4' and nodes in 'IT' are basically the same for all sections. This enables the point attractor network to 'recognize' trained objects in test trials at all locations in the visual field, which thus simulates translation-invariant object recognition. However,

the attractor network also contributes to the translation invariance of the object recognition since the attractor network in ‘IT’ completes a pattern input from ‘V1–V4’ even if the set of weights between a section in ‘V1–V4’ and the nodes in ‘IT’ is weaker than others.

The contribution of the attractor network in ‘IT’ to translation-invariant object recognition explains some recent experimental findings that showed that the size of the receptive field of inferior-temporal neurons depends on the content of the visual field and the specifics of the task. For example, if a single object is shown on a screen with a blank background, then it was found that the receptive field of a neuron that responds to this object can be very large (as much as 30 degrees or more). An example of the firing rate of one inferior-temporal neuron in a non-human primate engaged in a visual search task is shown by a solid line in Fig. 10.4A. Interestingly, if two objects are presented simultaneously, or if the target object is shown on top of a complex background (which can be viewed as a scene with many objects), then the size of the receptive field shrinks markedly (see dashed line in Fig. 10.4A for the case of an object on a complex background).

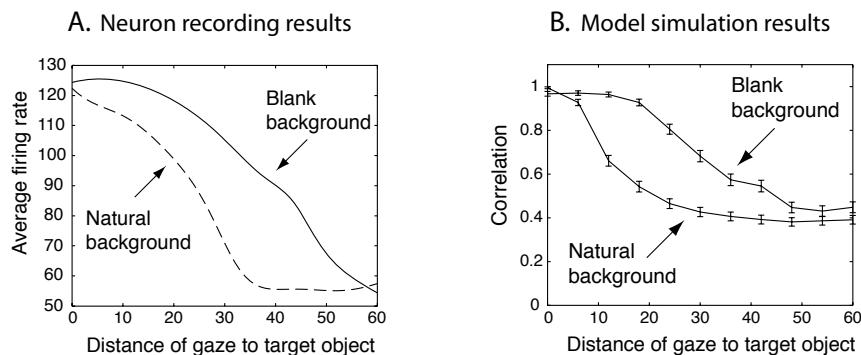


Fig. 10.4 (A) Example of the average firing rate from recordings of a neuron in the inferior-temporal cortex of a monkey in response to an effective stimulus that is located at various degrees away from the direction of gaze [experimental data courtesy of Edmund Rolls]. (B) Simulation results of a model with the essential components of the model shown in Fig. 10.3. The correlation is thereby a measure of overlap between the activity of ‘IT’ nodes with the representation of the target object that was used during training. [Adapted from Trappenberg, Rolls, and Stringer, *Advances in Neural Processing Systems* 14, (2002).]

A possible explanation of the experimental findings is given by the model discussed in this section, which is based on the attractor dynamics of the auto-associator network in ‘IT’. If only one object is shown, then this object would trigger the right point attractor and thus the recall of the object regardless of its location, which corresponds to large receptive fields. If, however, two or more trained objects are shown, then it is likely that the final state of the attractor network is mainly dominated by the object closest to the fovea, which gets the most weight due to cortical magnification. Simulation results, shown in Fig. 10.4B, confirm this hypothesis.

10.1.3 Attentional bias in visual search and object recognition

Visual search can be simulated within the model shown in Fig. 10.3 by supplying an object bias input to the attractor network in ‘IT’ that tells the system what to look for. Such top-down information is thought to originate in the frontal areas of the brain. Simulations have shown that the additional input of an object bias to ‘IT’ can speed up the recognition process in ‘IT’. The object bias also supports the recognition ability of the input from ‘V1–V4’ that corresponds to the target object in visual search. Correspondingly, it was found, in simulations as well as in primate experiments, that the receptive fields of target objects are larger than receptive fields of non-target objects.

Parallel conclusions can be drawn in an object recognition task in which top-down input to a specific location in module ‘PP’ is given. The label of this module suggests processing in the posterior parietal cortex, which is part of the dorsal visual processing pathway (‘where’ pathway), and hence specifically adapted in spatial representations. This is modelled with a spatially organized neural sheet, which is connected to the corresponding section in ‘V1–V4’. This activity in ‘PP’ can thus enhance the neural activity in ‘V1–V4’ for the features of the object that are located at the corresponding location in the visual field. Consequently, it will be easier (faster) for the ‘IT’ network to complete the input patterns for objects, and hence to ‘recognize’ objects, at the corresponding location.

The attentional biases have different origins in the model shown in Fig. 10.3. The attentional bias used in visual search originates from top-down input to ‘IT’ and is object-based. The attentional bias in the object recognition task acts on ‘PP’ and is location-based. However, it may be difficult to separate the different forms of attention in experiments because all parts of the model are bidirectional, in agreement with anatomical findings. For example, an object bias to ‘IT’ in the visual search task will ultimately trigger some activity in ‘PP’ that corresponds to the location where the object is, and this activity itself will help the recognition process of that object.

The time elapsed until the activity of a node in ‘PP’ reaches a certain threshold can be taken as an indication of the reaction time needed to find a specific object in visual search. Deco and colleagues simulated a visual search task in which the visual scene had a target (the letter ‘E’) and a number of distractors. The distractors (the letter ‘X’) in one experiment were visually very different from the target pattern. The simulated reaction time was in this case independent of the number of objects in the visual scene as indicated in Fig. 10.5A. In contrast, the reaction time increased linearly with the number of distractors when the distractor objects (the letter ‘F’) were visually similar to the target (see Fig. 10.5B). A linear increase of the reaction time with the number of objects in the visual scene is commonly attributed to a serial search mechanism in contrast to the lack of dependence of the reaction time on the number of objects that indicates parallel search.

Similar findings in psychophysical experiments with humans have often been interpreted as evidence for different search strategies thought to be implemented by different neural mechanisms. However, the simulation results demon-

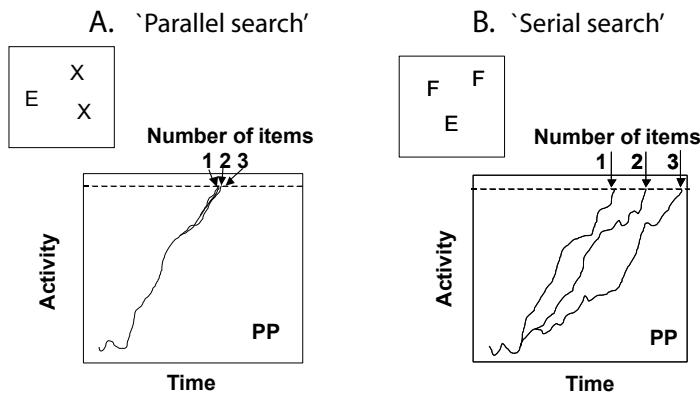


Fig. 10.5 Numerical experiments in which the model simulated a visual search task of a target object (the letter 'E') in a visual scene with visual distractors. (A) In one experiment the distractors consist of the letters 'X', which are visually very different from the target letter. The activity of a 'PP' node that corresponds to the target location increases in these experiments independently of the number of distractors, implying parallel search. (B) The second experiment was done with distractors (letter 'F') that were visually similar to the target letter. The reaction times, as measured from 'PP' nodes, depend linearly on the number of objects, a feature that is also characteristic of serial search. Both modes are, however, present in the same 'parallel architecture'. [Adapted from Gustavo Deco, personal communication.]

strate that such psychophysical results are consistent with a single parallel neural machinery, and that the apparent serial search is only due to the more intense conflict-resolution demand in the recognition process. The model is able to make many more predictions than can be verified experimentally, on a behavioural level, with brain imaging techniques or with cell recordings. This model is therefore a good example of the type of model that we hope will emerge for many more brain processes.

10.2 An interconnecting workspace hypothesis

It is increasingly clear that complex cognitive tasks can only be solved with the flexible cooperation of many specialized modules. In contrast to most existing models of brain functions, it is highly fascinating to observe how flexible we humans are in coping with the complex world around us. In contrast to most models, which produce very stereotyped behaviour, humans display the ability to master highly skilled functions while still maintaining the ability to produce novel solutions that have not been trained extensively. For example, we can drive a car with such ease that little attention and effort seems necessary to do so. This is despite the fact that we have to react to the unknown environment, for example, in following a road that we have not driven on before. However, when very novel and critical circumstances occur we are able to engage ourselves with many resources applied to this problem, which typically go along with very attentive and often very arduous mental activity.

10.2.1 The global workspace

From a system-level perspective we expect that specialized processors can have a high degree of autonomy while still being able to engage, if necessary, in more

global system activities. It is, however, still largely unknown how such flexibility can be achieved in a robust way. In the remainder of this chapter we will review one speculative, yet far-reaching idea as to how this could be achieved in the brain, which was proposed by *Stanislas Dehaene, Michel Kerszberg, and Jean-Pierre Changeux*. In Fig. 10.6 we have reproduced their illustration of the principal idea behind their proposal. In this the authors divided brain functions into five basic subsystems, ranging from a perceptual system, which is our window to the world, to the motor system responsible for executing desired goals. The three remaining subsystems are a memory system, an evaluative system, and an attentional system, all of which are basic ingredients for complex cognitive tasks as viewed by many cognitive scientists. The main point made by Dehaene and colleagues is that these subsystems must be able to work alone or in combinations in a flexible manner, and they speculate that this may be achieved through some form of network between the subsystems. Such an interconnecting network would form a *global workspace*, which would form the basis of our ability to solve complex tasks in a flexible manner.

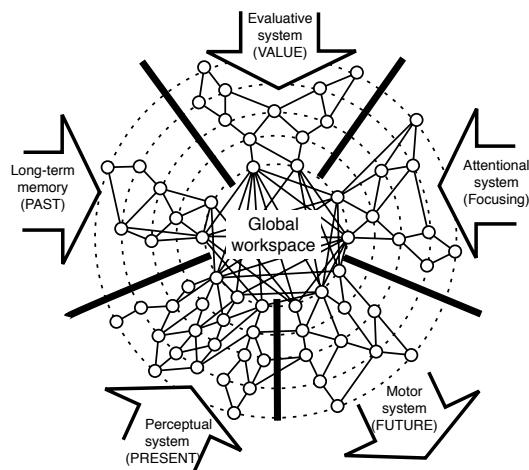


Fig. 10.6 Illustration of the workspace hypothesis. Two computational spaces can be distinguished, the subnetworks with localized and specific computational specialization, and an interconnecting network that is the platform of the global workspace [adapted from Dehaene, Kerszberg, and Changeux, *Proceedings of the National Academy of Science* 95: 14529–34 (1998)].

In contrast to the more localized basic processing networks that make up the five basic subsystems as illustrated in Fig. 10.6, the workspace has to be a more global computational space in the brain. Projections between cortical areas are indeed abundant, and associated fibres of cortico-cortical connections originate predominantly from pyramidal neurons in layers II and III, as mentioned in Chapter 5. The authors suggest that a large portion of the global workspace could hence be localized within these layers. This suggestion has interesting consequences for the interpretation of anatomical findings concerning these layers. We have mentioned that the extent of layers within the cortex varies considerably. Layers II and III are, for example, elevated in the dorsolateral prefrontal cortex, an area that has been implicated in a variety of types of complex mental processing. For example, it was shown to become active in early stages of learning a sequence of numbers, and patients with lesions in this area are thought to have difficulties in switching between different rules in more challenging categorization tasks. It is difficult to record brain activity

from specific layers, and direct verifications of the model discussed below are therefore challenging. In such situations it becomes necessary to work out other predictions that can be verified experimentally.

10.2.2 Demonstration of the global workspace in the Stroop task

Dehaene and colleagues demonstrated the principal idea in much more detail with a model that sheds light on the processing of the *Stroop task* when following the global workspace hypothesis. The Stroop task is illustrated in Fig. 10.7. In a common form it consists of a set of words that name colours, where each word can be printed in a different colour. In the Stroop task a subject is asked to either read the word or name the colour in which the word is written (all relatively rapidly). We are highly trained in reading, so that the display of a word does not render any problems in pronouncing the word. However, the naming of the colour can initially cause some problems as the image of the word prompts us to read the word rather than to ignore the content and to report the colour of the letters. It initially takes some effort to suppress the reading of the word until we have automated our response after a few trials. The explanation offered by Dehaene and colleagues is that the global workspace has to become active to 're-wire' the commonly active word-naming configuration of the brain.

task image	word naming	colour naming
grey	grey	black
black	black	grey

Fig. 10.7 In a Stroop task a word for a colour, written in a colour that can be different from the meaning of the word, is shown to a subject who is asked to perform either a word-naming or colour-naming task.

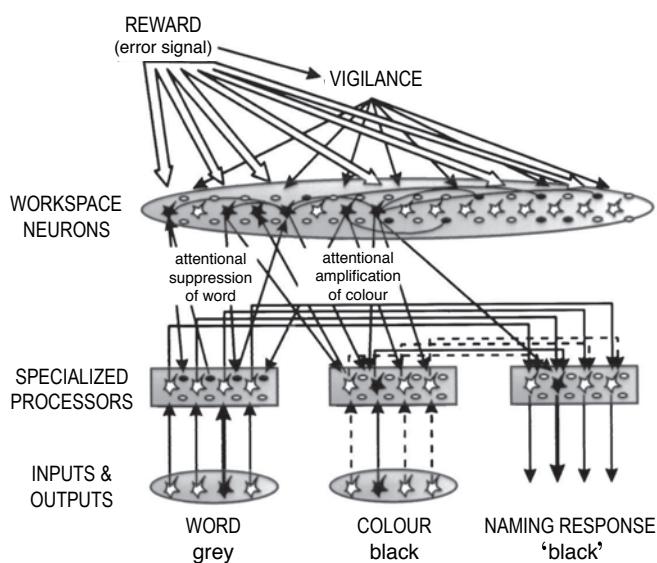


Fig. 10.8 Global workspace model that is able to reproduce several experimental findings in the Stroop task [adapted from Dehaene, Kerszberg, and Changeux, *Proceedings of the National Academy of Science* 95:14529–34 (1998)].

To demonstrate this idea further the authors developed a model that can be tested on a Stroop task. This is illustrated in Fig. 10.8. The model includes three specialized processors, one that indicates the meaning of the word that is displayed, one that indicates the colour of the word that is displayed, and one that indicates the response of the system. In the more standard word-

reading task we can assume that there is a tight coupling between the first input processor and the response processor so that the system could initially report easily the content of the word. The second task, that of naming the colour in which the word is written, then requires a suppression of these connections and the enhancement of the connections between the ‘colour’ input processor and the response processor. This is achieved by the top-down influence of workspace nodes on the nodes in the specialized processors.

What prompts the workspace neurons to become active or to change their behaviour in the first place? Basically, the system has to be told what to do. This is achieved in the form of a reward signal that only indicates a mismatch between the desired response and actual response of the system. The large error signal then triggers a vigilance parameter to increase, which in turn allows the workspace neurons to become active that effectively reconfigure the system to allow a correct response within the required task. The vigilance parameter decreases once the system responds correctly. The authors argue that an increased vigilance (and therefore an increased activity of workspace nodes) could parallel the increased mental effort that is felt by subjects during the initial period after switching the task.

The model makes several predictions, such as increased activity in the workspace network in novel tasks, that can be tested experimentally. The model demonstrates an interesting flexibility to solve different tasks, and we can imagine how this flexibility of coupling specialized subsystems in the brain could be responsible for our ability to cope with the changing environment in which we live.

10.3 The anticipating brain

The examples in the previous sections showed how important layered architectures with bottom-up and top-down information flow are for cognitive processes. We now attempt to generalize brain-style information processing principles into a more general hypothesis of how the brain implements cognitive functions. Central to this emerging brain theory is the view of the brain as an anticipating memory system, as mentioned in the introductory chapter. In particular, we are trying to incorporate several factors which we deem essential in realizing cognitive functions.

- (1) The brain can develop a model of the world, which can be used to anticipate or predict the environment.
- (2) The inverse of the model can be used to recognize causes by evoking internal concepts.
- (3) Hierarchical representations are essential to capture the richness of the world.
- (4) Internal concepts are learned through matching the brain’s hypotheses with input from the world.
- (5) An agent can learn actively by testing hypothesis through actions.
- (6) The temporal domain is an important degree of freedom.

The remainder of this chapter is a discussion of the anticipating brain hypothesis and related model implementations. We start with a general overview of the anticipating brain which should show more clearly how the points above are related. We then outline some specific models and discuss their relation to statistical methods of generative density estimation. Related ideas have been expressed for some years, in particular by *Ulric Neisser*, who coined the *analysis by synthesis*, and by *Douglas Hofstadter* in the *copycat project*. However, the presentation in this section is particularly based on work by *Geoffrey Hinton*, *Peter Dayan*, *Terrance Sejnowski*, *Karl Friston*, and *Jeff Hawkins*, together with their colleagues.

10.3.1 The brain as anticipatory system in a probabilistic framework

To formalize the thesis we need to introduce some notations which help to discuss the systems in a compact way. We start by denoting a sensory state with a vector \mathbf{s} , where the components can be any relevant quantity, such as spikes or firing rates of primary sensory neurons. Also, these sensory states can describe any kinds of sensations, including olfaction, touch, vestibular senses, proprioceptive feedback, and vision. Sensory states are caused by physical processes in the environment, and we can formalize this by writing the sensory state as a function of a *causal state*, \mathbf{c} ,

$$\mathbf{s} = g(\mathbf{c}). \quad (10.1)$$

The function g describes the physical process of generating the sensory response. We can refine this notation further by recognizing that the brain can respond with changing patterns to stimulations from the world. The reasons for different responses include noise somewhere in the system and/or the dependency of the system on hidden variables. In any case, it is thus appropriate to consider the probability that a certain sensory state is evoked by a given causal state,

$$p(\mathbf{s}|\mathbf{c}). \quad (10.2)$$

This describes the conditional probability (or conditional probability density) of having a sensory state given a specific set of causes. Thus, a sensory state, as considered here, can be different for identical presentations of a sensory scene. Only the probability of finding a specific sensory state depends on the environmental condition.

We introduced the term ‘causes’ above, and this term should be defined further. While it is likely that everyone understands, to some extent, what is meant by the word ‘cause’, it is interesting to realize that there can be different opinions on the causes in many examples. For example, imagine sitting in a concert hall and listening to a symphony. What are the causes of our experience in this situation? Or, more precisely, on which level should we define causes? Should the orchestra as a whole be considered a cause, or an individual player, or her instrument, or even parts of the instrument. While it might seem a bit niggling to contemplate this issue, resolving it is central to our thesis and highlights two important functions of brain processing. First, we propose that

one of the major goals of the brain is to learn what causes are by forming internal *concepts*. Second, the brain must learn concepts at different levels of abstraction, from some concrete representations of sensory states, such as edges of objects or sound fragments, to more abstract representations such as going to a concert. Learning concepts and predicting causes in our environment is thus central to the thesis developed in this chapter.

To discuss and develop concrete examples of predictive layered memory systems in the following, we need to elaborate on some notations. When considering learning of concepts which can explain causes, it will be important to consider systems that can explore the environment actively. The notation of 10.2 would only be appropriate for a passive observer. Instead, we should explicitly consider that the carrier of the brain, called an *agent* in the following, can interact with the environment. For example, the agent could direct its gaze to specific locations in space, or the agent could touch an object or pick it up to test some hypothesis about the object or the environment. We denote an action of the agent with a vector \mathbf{a} to allow for distributed representations of motor output. We thus consider the probability of sensory states, given causes in the environment, and specific actions of the agent by writing:

$$p(\mathbf{s}|\mathbf{c}, \mathbf{a}). \quad (10.3)$$

The central conjecture followed now is that the brain is trying to match sensory input with internally generated states. The internal representations of sensory states in primary sensory cortex are denoted by \mathbf{s}' . While these states depend on input from the environment, they also depend on expectations from higher cortical areas. More specifically, the distribution of the cortical sensory states, \mathbf{s}' , depends on the primary input from the environment, s , and some states, c' , of higher cortical areas:

$$p(\mathbf{s}'|s, c'). \quad (10.4)$$

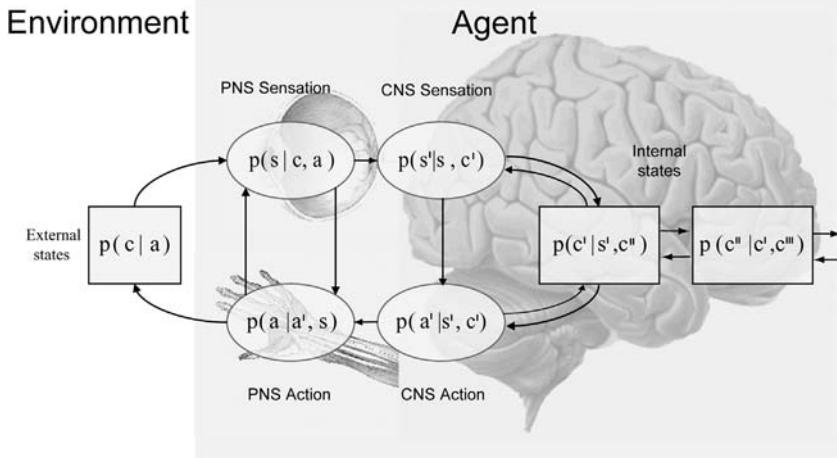
We call these higher-order cortical representations *concepts*. Concepts on a certain level of cortical representations depend in turn on concepts of higher-order cortical representations, $c'', c''',$ etc., which in turn are influenced by input from lower cortical areas.

On an abstract level, we see the brain as a *generative model* of the world, G , which can generate probability distributions of concepts on different levels, ultimately generating expectations of sensory states:

$$p(\mathbf{s}'; G). \quad (10.5)$$

The inverse of the model can be seen as a *recognition model*, Q , which evokes internal concepts from causes in the environment. The world model, which is embedded in the central nervous system, is dissected according to the above discussion in Fig. 10.9 with a layered structure that includes the necessary bidirectional connections. Such related models have been termed *deep belief networks*; ‘believe networks’ for their ability to generate expectations, and ‘deep’ for their layered architecture. At this stage it is not easy to clearly separate the generative model from the recognition model, since the whole system is highly interactive, but the distinction of a generative and recognition model can sometimes be made explicit, as discussed below. The model illustrated

in this figure can be viewed as a *Bayesian network* or *causal graphical model*, since it specifies the causal relations between conditional probability functions that produce the joined density function of the environment-agent system. The nodes represent random variables in Bayesian networks, often representing high-level concepts such as whether it is raining or not, although the model could be a large network of probabilistic neurons.¹



¹This model contains loops which are not commonly covered in the most basic discussions of Bayesian networks.

Fig. 10.9 Illustration of an agent-environment system with probabilistic notations describing various components of the system. The illustration includes the environment which causes sensory experiences of the agent. The interface between the environment is illustrated as the peripheral nervous system (PNS) with exemplary sensory and action components. The central nervous system (CNS) is a predictive memory system with layered representations. [Extended from Friston, *Philosophical Transactions of the Royal Society B* 360, 815–36 (2005)].

The concepts at different levels of cortical representation have to be learned in an *self-supervised* way through the interaction with the environment and engrained into a memory system. For example, the early visual system can learn to recognize different sequences of retinal patterns. Sequences of these concepts can then be learned by higher-order cortical areas. In turn, higher order concepts that are evoked by specific sensory input can influence the expectations of concepts in lower cortical representations, ultimately anticipating specific patterns of sensory input. We discuss below a more concrete system that shows some of these abilities.

Testing how good the world model is can only be achieved through interactions with the environment. This can be seen as hypothesis testing, or *inference*, of the world model with environmental data. We presented the state of the environment, $p(\mathbf{c}|\mathbf{a})$ as a probabilistic quantity, which does not only mean that causes are noisy, but incorporates that causes are changing beyond our control. The conditional probability on the actions generated by the agent describes that the world states can be influenced by actions generated by the agent. Thus, hypothesis testing by the agent is somewhat different to common inference techniques in statistics in that the agent seems to be able to actively interact with the environment. Activations of specific concepts can guide specific actions, which, in turn, can manipulate the environment in a specific way, which in turn can guide further learning. For example, when whistling to a child, he might first recognize the funny face and unusual shape of the mouth. The child might then form a hypothesis that the mouth has something to do with the whistling sound and might touch the mouth to verify this hypothesis. Such *active learning* might be necessary to reduce the demands on learning in

large systems.

Many components of the theory we just outlined are untested or need more considerations of implementation issues. However, some recent models that implement and elaborate on the principal ideas outlined in this section are discussed in the following sections.

10.3.2 The Boltzmann machine

There are several methods of constructing models which are able to learn expectations of sensory states. We encountered an important model in Chapter 8, the attractor neural network (ANN), which is able to learn patterns (sensory states). After learning, the system can recall (generate) sensory data from partial input, and thereby recognize unseen or noisy versions of patterns. In this sense, ANN models can be seen as predictive memory systems. These simple recurrent networks can be trained with Hebbian autocorrelation rules so that the corresponding dynamic system has point attractors. Thus, given partial input of a sensory state, basic ANNs always produce the same answer and cannot provide us with a series of answers reflecting the probability that the partial input was generated by different causes in the environment with similar sensory states. To get different answers, we can consider such networks with probabilistic update rules, as discussed in Section 8.2.1. However, we saw that such systems are still dominated by point attractor states, and we need to generalize the system further to allow it to learn more generally dynamic solutions.

A general dynamic system can be constructed by building a recurrent system with hidden nodes. We have already seen that hidden nodes are necessary in feedforward mapping networks (perceptrons) to provide the system with enough internal representations to form sufficient approximations. Similarly, the hidden states in recurrent networks, together with their adjustable connections, provide the system with enough degrees of freedom to approximate any dynamical system. While this has been recognized for a long time, finding practical training rules for such systems have been a major challenge.

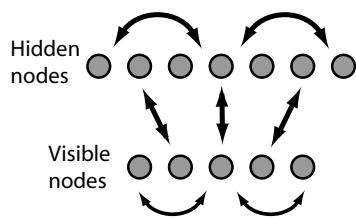


Fig. 10.10 A Boltzmann machine with one visible and one hidden layer.

It was possible to use supervised Hebbian training for ANN networks in Chapter 8 since all the nodes in such networks are visible and can be supervised with sensory states. In more general models, we now need to distinguish visible nodes and hidden nodes. Such a general recurrent network is shown in Fig. 10.10. Visible nodes, labelled with a superscript v in the following, are either input or output nodes, whereas hidden nodes, labelled with a superscript h , are not connected directly to the external world. Such systems can still be described by an energy function, as discussed in Chapter 8 (Section 8.4.2). The energy between two nodes that are symmetrically connected with strength w_{ij} is

$$H^{nm} = -\frac{1}{2} \sum_{ij} w_{ij} s_i^n s_j^m. \quad (10.6)$$

The state variables, s , have superscripts n or m which can have values (v) or (h) to indicate visible and hidden nodes. We consider again the probabilistic

update rule,

$$p(s_i^n = +1) = \frac{1}{1 + \exp(-\beta \sum_j w_{ij} s_j^n)}, \quad (10.7)$$

with inverse temperature, β , which is called the Glauber dynamics in physics and describes the competitive interaction between minimizing the energy and the randomizing thermal force. The probability distribution for such a stochastic system is called the Boltzmann–Gibbs distribution. Following this distribution, the distribution of visible states, in thermal equilibrium, is given by

$$p(\mathbf{s}^v; \mathbf{w}) = \frac{1}{Z} \sum_{m \in h} \exp(-\beta H^{vm}), \quad (10.8)$$

where we summed over all hidden states. In other words, this function describes the distribution of visible states of a Boltzmann machine with specific parameters, \mathbf{w} , representing the weights of the recurrent network. The normalization term, $Z = \sum_{n,m} \exp(-\beta H^{nm})$, is called the *partition function*, which provides the correct normalization so that the sum of the probabilities of all states sums to one. These stochastic networks with symmetrical connections have been termed *Boltzmann machines* by Ackley, Hinton and Sejnowski. Networks with asymmetrical connections, called *Helmholtz machines*, are considered in Section 10.3.4.

Let us consider the case where we have chosen enough hidden nodes so that the system can, given the right weight values, implement a generative model of a given world. Thus, by choosing the right weight values, we want this dynamical system to approximate the probability function, $p(\mathbf{s}^v)$, of the sensory states (states of visible nodes) caused by the environment. To derive a learning rule, we need to define an objective function. In this case, we want to minimize the difference between two density functions. A common measure for the difference between two probabilistic distributions is the Kulbach–Leibler divergence (see Appendix C.6),

$$\text{KL}(p(\mathbf{s}^v), p(\mathbf{s}^v; \mathbf{w})) = \sum_{\mathbf{s}}^v p(\mathbf{s}^v) \log \frac{p(\mathbf{s}^v)}{p(\mathbf{s}^v; \mathbf{w})} \quad (10.9)$$

$$= \sum_{\mathbf{s}}^v p(\mathbf{s}^v) \log p(\mathbf{s}^v) - \sum_{\mathbf{s}}^v p(\mathbf{s}^v) \log p(\mathbf{s}^v; \mathbf{w}). \quad (10.10)$$

To minimize this divergence with a gradient method, we need to calculate the derivative of this ‘distance measure’ with respect to the weights. The first term in the difference in eqn 10.10 is the entropy (see Appendix D) of sensory states, which does not depend on the weights of the Boltzmann machine. Minimizing the Kulbach–Leibler divergence is therefore equivalent to maximizing the average log-likelihood function,

$$l(\mathbf{w}) = \sum_{\mathbf{s}}^v p(\mathbf{s}^v) \log p(\mathbf{s}^v; \mathbf{w}) = \langle \log p(\mathbf{s}^v; \mathbf{w}) \rangle. \quad (10.11)$$

In other words, we treat the probability distribution produced by the Boltzmann machine as a function of the parameters, w_{ij} , and choose the parameters

which maximize the likelihood of the training data (the actual world states). Therefore, the averages of the model are evaluated over actual visible states generated by the environment. The log-likelihood of the model increases the better the model approximates the world. A standard method of maximizing this function is gradient ascent, for which we need to calculate the derivative of $l(\mathbf{w})$ with respect to the weights. We omit the detailed derivation here, but we note that the resulting learning rule can be written in the form

$$\Delta w_{ij} = \eta \frac{\partial l}{\partial w_{ij}} = \eta \frac{\beta}{2} (\langle s_i s_j \rangle_{\text{clamped}} - \langle s_i s_j \rangle_{\text{free}}). \quad (10.12)$$

The meaning of the terms on the right-hand side is as follows. The term labelled ‘clamped’ is the thermal average of the correlation between two nodes when the states of the visible nodes are fixed. The term labelled ‘free’ is the thermal average when the recurrent system is running freely. The Boltzmann machine can thus be trained, in principle, to represent any arbitrary density functions, given that the network has a sufficient number of hidden nodes.

This result is encouraging as it gives as an exact algorithm to train general recurrent networks to approximate arbitrary density functions. The learning rule looks interesting since the clamped phase could be associated with a sensory driven agent during an awake state, whereas the freely running state could be associated with a sleep phase. Unfortunately, it turns out that this learning rule is too demanding in practice. The reason for this is that the averages, indicated by the angular brackets in eqn 10.12, have to be evaluated at thermal equilibrium. Thus, after applying each sensory state, the system has to run for a long time to minimize the initial transient response of the system. The same has to be done for the freely running phase. Even when the system reaches equilibrium, it has to be sampled for a long time to allow sufficient accuracy of the averages so that the difference of the two terms is meaningful. Further, the applicability of the gradient method can be questioned since such methods are even problematic in recurrent systems without hidden states since small changes of system parameters (weights) can trigger large changes in the dynamics of the dynamical systems. These problems prevented, until recently, more practical progress in this area. Recently, Hinton and colleagues developed more practical, and biologically more plausible, systems which are described next.

10.3.3 The restricted Boltzmann machine and contrastive Hebbian learning

Training of the Boltzmann machine with the above rule is challenging because the states of the nodes are always changing. Even with the visible states clamped, the states of the hidden nodes are continuously changing for two reasons. First, the update rule is probabilistic, which means that even with constant activity of the visible nodes, hidden nodes receive variable input. Second, the recurrent connections between hidden nodes can change the states of the hidden nodes rapidly and generate rich dynamics in the system. We certainly want to keep the probabilistic update rule since we need to generate different responses of the system in response to sensory data. However,

we can simplify the system by eliminating recurrent connections within each layer, although connections between the layers are still bidirectional. While the simplification of omitting collateral connections is potentially severe, much of the abilities of general recurrent networks with hidden nodes can be recovered through the use of many layers which bring back indirect recurrencies. A *restricted Boltzmann machine* (RBM) is shown in Fig. 10.11.

When applying the learning rule of eqn 10.12 to one layer of an RBM, we can expect faster convergence of the rule due to the restricted dynamics in the hidden layer. We can also write the learning rule in a slightly different form by using the following procedure. A sensory input state is applied to the input layer, which triggers some probabilistic recognition in the hidden layer. The states of the visible and hidden nodes can then be used to update the expectation value of the correlation between these nodes, $\langle s_i^v s_j^h \rangle^0$, at the initial time step. The pattern in the hidden layer can then be used to approximately reconstruct the pattern of visible nodes. This *alternating Gibbs sampling* is illustrated in Fig. 10.12 for a connection between one visible node and one hidden node, although this learning can be done in parallel for all connections. The learning rule can then be written in form,

$$\Delta w_{ij} \propto \langle s_i^v s_j^h \rangle^0 - \langle s_i^v s_j^h \rangle^\infty. \quad (10.13)$$

Alternating Gibbs sampling becomes equivalent to the Boltzmann machine

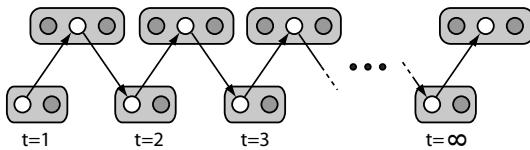


Fig. 10.11 Restricted Boltzmann machine in which recurrences within each later are removed.

learning rule (eqn 10.12) when repeating this procedure for an infinite number of time steps, at which point it produces pure fantasies. However, this procedure still requires averaging over long sequences of simulated network activities, and sufficient evaluations of thermal averages can still take a long time. Also, the learning rule of eqn 10.13 does not seem to correspond to biological learning. While developmental learning also takes some time, it does not seem reasonable that the brain produces and evaluates long sequences of responses to individual sensory stimulations. Instead, it seems more reasonable to allow some finite number of alternations between hidden responses and the reconstruction of sensory states. While this does not formally correspond to the mathematically derived gradient leaning rule, it is an important step in solving the learning problem for practical problems, which is a form of *contrastive divergence* introduced by Geoffrey Hinton. It is heuristically clear that such a restricted training procedure can work. In each step we create only a rough approximation of ideal average fantasies, but the system learns the environment from many examples, so that it continuously improves its expectations. While it might be reasonable to use initially longer sequences, as infants might do, Hinton and colleagues showed that learning with only a few reconstructions is able to self-organize the system. The self-organization, which is based on input from the environment, is able to form internal representations that can be

Fig. 10.12 Alternating Gibbs sampling.

used to generate reasonable sensory expectations and which can also be used to recognize learned and novel sensory patterns.

The basic Boltzmann machine with a visible and hidden layer can easily be combined into hierarchical networks by using the activities of hidden nodes in one layer as inputs to the next layer. Hinton and colleagues have demonstrated the power of restricted Boltzmann machines for a number of examples. For example, they applied layered RBMs as auto-encoders where restricted alternating Gibbs sampling was used as pre-training to find appropriate initial internal representations that could be fine-tuned with backpropagation techniques to yield results surpassing support vector machines. However, for our discussions of brain functions it is not even necessary to yield perfect solutions in a machine learning sense, and machines can indeed outperform humans in some classification tasks solved by machine learning methods. For us, it is more important to understand how the brain works.

10.3.4 The Helmholtz machine

In the section above, we concentrated on the Boltzmann machine, which is a model of stochastic units with symmetrical weights. Before demonstrating such machines further, we briefly mention here the cousins of such symmetrical models, which are models of stochastic units where the bottom-up and top-down weights between the layers can be different. An illustration of such a network structure with two layers is shown in Fig. 10.13.

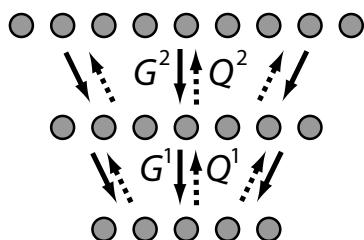


Fig. 10.13 A deep belief network called the Helmholtz machine with two hidden layers. The generative model, \mathbf{G} , is shown with solid arrows, and the recognition model, \mathbf{Q} , is shown with dashed arrows.

$$F(Q, G) = -L(G) + \sum_{\mathbf{c}} KL(p(\mathbf{c}; \mathbf{s}, Q), p(\mathbf{c}|\mathbf{s}; G)). \quad (10.14)$$

Minimizing the free energy corresponds to maximizing the log-likelihood of the generative model and minimizing the Kulback–Leibler divergence between the densities of causes produced by the recognition model and the densities of causes produced by the generative model, for a given set of visible states.

Helmholz machines can be trained with the *wake-sleep* algorithm. During the wake phase of this algorithm, data are applied to the input layer of the model and the generative (top-down) weights are trained. In a subsequent sleep phase, random sequences are produced by the topmost layer and propagated down with the generative model to the input layer. In this phase, the recognition (bottom-up) weights are trained. This algorithm has some resemblance to the expectation-maximization algorithm discussed below in Section 10.3.6. In the case of stochastic sigmoidal neurons, the training algorithms for both, the recognition and generative weights, take the form of Hebbian-type delta rules.

Simulation

To illustrate the function of an anticipating brain model, we briefly outline a demonstration by the Hinton group. The online demonstration can be run in a browser from <http://www.cs.toronto.edu/~hinton/adi>, and a stand alone version of this demonstration is available at this book's resource page. MATLAB source code for restricted Boltzmann machines are available at Hinton's home page. An image of the demonstration program is shown in Fig. 10.14. The model consists of a combination of restricted Boltzmann machines and a Helmholtz machine. The input layer is called the *model retina* in the figure, and the system also contains a *recognition-readout-and-stimulation* layer. The model retina is used to apply images of handwritten characters to the system. The recognition-readout-and-stimulation layer is a brain imaging and stimulation device from and to the uppermost RBM layer. This device is trained by providing labels as inputs to the RBM for the purpose of 'reading the mind' of the system and to give it high-level instructions. This device learns to recognize patterns in the uppermost layer and map them to their meaning, as supplied during supervised learning of this device. This is somewhat analogous to *brain-computer interfaces* developed with different brain-imaging devices such as EEG, fMRI, or implanted electrodes. The advantage of the simulated device is that it can read the activity of every neuron in the upper RBM layer. The device can also be used with the learned connections in the opposite direction to stimulate the upper RBM layer with typical patterns for certain image categories.

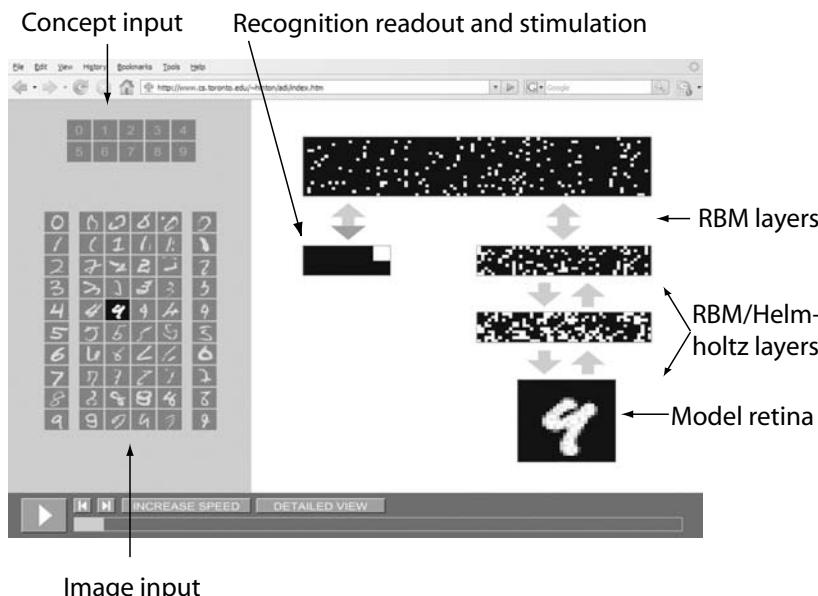


Fig. 10.14 Simulation of restricted Boltzmann machine by Geoffrey Hinton and colleagues, available at www.cs.toronto.edu/~hinton/adi.

The model for this demonstration was trained on images of handwritten numbers from a large database. Some example images can be seen on the left-hand side. All layers of this model were first treated as RBMs with symmetrical

weights. Specifically, these were trained by applying images of handwritten characters to the model retina and using three steps of alternating Gibbs sampling for training the different layers. The evolving representations in each layer are thus purely unsupervised. After this basic training, the model was allowed, for fine-tuning purposes, to develop different weight values for the recognition and generative model as in Helmholtz machines with a wake–sleep training algorithm as mentioned above.

The simulations provided by Hinton demonstrate the ability of the system after training. The system can be tested in two ways, either by supplying a handwritten image and asking for recognition, or by asking the system to produce images of a certain letter. These two modes can be initiated by selecting either an image or by selecting a letter category on the left-hand side. In the example shown in Fig. 10.14, we selected an example of an image of the number 4. When running the simulation, this image triggers response patterns in the layers. These patterns change in every time step, due to the probabilistic nature of the updating rule. The recognition read-out of the uppermost layer does, therefore, also fluctuate. In the shown example, the response of the system is 4, but the letter 9 is also frequently reported. This makes sense, as this image does also look somewhat like the letter 9. A histogram of responses can be constructed when counting the responses over time, which, when properly normalized, corresponds to an estimate of the probability over high-level concepts generated by this sensory state. Thus, this mode tests the recognition ability of the model.

The stimulation device connected to the upper RBM layer allows us to instruct the system to ‘visualize’ specific letters, which corresponds to testing the generative ability of the model. For example, if we ask the system to visualize a letter 4 by evoking corresponding patterns in the upper layer, the system responds with varying images on the model retina. There is not a single right answer, and the answers of the system change with time. In this way, the system produces examples of possible images of letter 4, proportional to some likelihood that these images are encountered in the sensory world on which the system was trained. The probabilistic nature of the system much better resembles human abilities to produce a variety of responses, in contrast to the mapping networks discussed in Chapter 6 which were only able to produce single answers for each input.

10.3.5 Probabilistic reasoning: causal models and Bayesian networks

The models discussed in this section are aimed at describing and implementing general learning machines which are able to self-organize from experience. The Boltzmann machine is a good example of an anticipating brain system that can learn density functions. It is useful to relate the concepts of an anticipatory brain system to other formulations of such ideas. We argued that learning of concepts is the basis of forming a general understanding of the environment and to enable sophisticated anticipation of causes. In recent years, statistical models have been invented, together with graphical representations and efficient inference algorithms, to formalize statistical reasoning in such causal models.

The principal idea behind these more abstract models (in terms of brain modelling) are outlined in the rest of this section before returning to a more specific brain model in the last section of this chapter. The methods discussed here are useful in their own right, but they are also increasingly discussed in the computational neuroscience literature.

To describe these ideas, we consider the example of forecasting whether it will rain tomorrow. More specifically, we would like to estimate the probability of rain given certain observations. For example, we could measure the trend of atmospheric pressure with a barometer, or see if the weather report on the evening news calls for rain. Since we can not predict the precise value of these variables at any given time, we treat these concepts as random variables. There are, of course, many factors that influence the weather conditions; the atmospheric pressure being one of them. In contrast, although we think (or at least hope) that there is a high correlation between the weather prediction on the evening news (which takes several factors into account) and the probability of rain the next day, this prediction is certainly not the cause of rain. Rather, it is somewhat indirect in that some causes of rain influence the prediction of rain on the evening news.

The causal relations of the different factors in this example are illustrated with a *graphical model* in Fig. 10.15. In such *Bayesian networks*, each circle represents a random variable (R , rain; A , atmospheric pressure; W , weather report (forecast); X , other causal factors of rain). The arrows between them represent conditional probabilities. Thus, the density function of the whole model can be factorized due to the conditional independence of nodes without arrows between them as

$$P(R, A, X, W) = P(R|A, X)P(W|A, X)P(A)P(X). \quad (10.15)$$

This expression is much shorter than the general *diagnostic* statistical model that we would need to evaluate without the knowledge of the causal relationships. Of course, the different conditional distributions need to be estimated from data, but once these quantities are known, one can answer specific questions, such as how likely it is that it rains given that the weather forecast calls for rain.

A useful variant of causal models is the *dynamic Bayesian network* (DBN), which takes temporal aspects into account. For example, in the model shown in Fig. 10.16, we consider whether it is raining at our favourite vacation destination. Since we are not there, we can only watch the weather channel which reports on that day whether there was rain or not. Thus, only the random variable $W(t)$ at time t is observable, and the random variable $R(t)$ is hidden. The information on the weather network is not entirely conclusive, since the report covers a large area and might not be accurate for our specific vacation destination. The model takes into account that it is more likely to rain when it rained the previous day. Networks with the simple connectivity pattern as shown in 10.16, in which nodes are conditionally independent from other nodes give the parent nodes, are called *Markov chains*. Also, we usually consider models in which the laws (conditional probabilities) do not depend on time (are *stationary*). Models with these conditions, and with the structure of Fig. 10.16 that includes one hidden variable and one observable variable, are

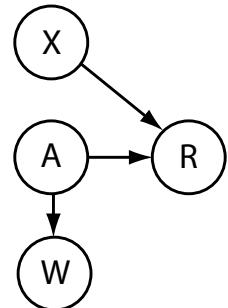


Fig. 10.15 A Bayesian network with four random variables (R , rain; A , atmospheric pressure; W , weather report (forecast); X , other causal factors of rain).

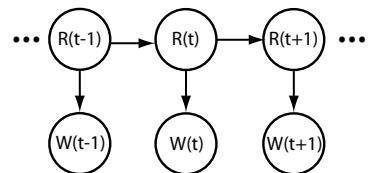


Fig. 10.16 A dynamic Bayesian model with one hidden variable (R) and one observable (W). Each of the variables obeys a first-order Markov condition. Such models are also called hidden Markov models.

called *hidden Markov models*. Efficient algorithms have been found to perform probabilistic reasoning (statistical inference), and such models are increasingly used to describe mental processes.

10.3.6 Expectation maximization

As stressed in Section 10.3.1, we can view the anticipating brain as a *generative model*, which is able to produce data that can be compared with the data from the environment, and whose inverse, , can be used to recognize causes in the environment. It is, of course, a major question in neuroscience how such causal models are formed in the brain, and we have discussed already some examples in Section 10.3.1. Here we view the problem in a slightly different way, in that we assume that a general form of a model is given, and in which the problem is to estimate the parameters of the corresponding generative/recognition models in an unsupervised (or self-supervised) way.

A widely applicable technique of parameter estimation in such situations is (EM). To introduce the idea behind EM, we follow an example of density estimation in a very simple world. In this simple world, data are generated with equal likelihood from two Gaussian distributions, one with mean $\mu_1 = -1$ and standard deviation $\sigma_1 = 2$, the other with mean $\mu_2 = 4$ and standard deviation $\sigma_2 = 0.5$. These two distributions are illustrated in Fig. 10.17A with dashed lines. Let us assume that we know that the world consists only of data from two Gaussian distributions with equal likelihood, but that we do not know the specific realizations (parameters) of these distributions. The pre-knowledge of two Gaussian distributions encodes a specific which makes up this . In this simple example, we have chosen the heuristics to match the actual data-generating system (world), that is, we have explicitly used some knowledge of the world. As argued above, we can think of the brain as a flexible dynamic system, the parameters of which can be adjusted to match world distributions, and it is also possible that, through evolution, this system has evolved to parameterize common concepts in our world.

Learning the parameters of the two Gaussians would be easy if we had access to the information about which data point was produced by which Gaussian, that is, which cause produced the specific examples. Unfortunately, we can only observe the data without a teacher label that could supervise the learning. We choose therefore a self-supervised strategy, which repeats the following two steps until convergence:

E-step: We make assumptions of training labels (or the probability that the data were produced by a specific cause) from the current model (expectation step); and

M-step: use this hypothesis to update the parameters of the model to maximize the observations (maximization step).

Since we do not know appropriate parameters yet, we just chose some arbitrary values as the starting point. In the example shown in Fig. 10.17A we used $\mu_1 = 2$, $\mu_2 = -2$, $\sigma_1 = \sigma_2 = 1$. These distributions are shown with solid

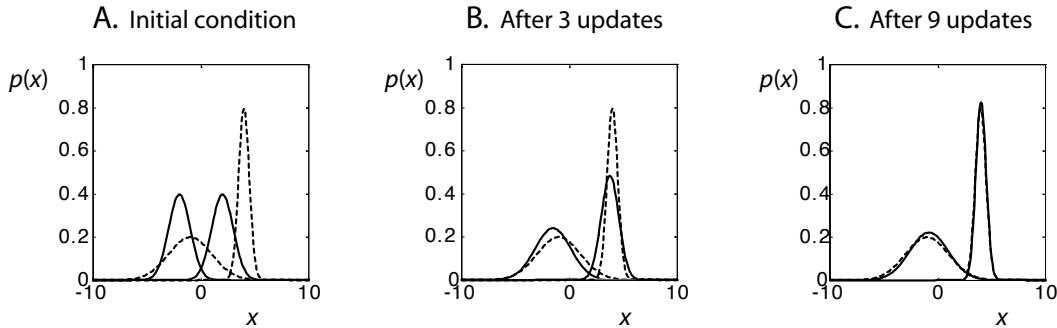


Fig. 10.17 Example of the expectation maximization (EM) algorithm for a world model with two Gaussian distributions. The Gaussian distributions of the world data (input data) are shown with dashed lines. (A) The generative model, shown with solid lines, is initialized with arbitrary parameters. In the EM algorithm, the unlabelled input data are labelled with a recognition model, which is, in this example, the inverse of the generative model. These labelled data are then used for parameter estimation of the generative model. The results of learning are shown in (B) after three iterations, and in (C) after nine iterations.

lines. Comparing the generated data with the environmental data corresponds to hypothesis testing.

The results are not yet very satisfactory, but we can use the generative model to express our *expectation* of the data. Specifically, we can assign each data point to the class which produces the larger probability within the current world model. Thus, we are using our specific hypothesis here as a *recognition model*. In the example we can use Bayes' rule to invert the generative model into a recognition model as detailed in the simulation section below. If this inversion is not possible, then we can introduce a separate recognition model, Q , to approximate the inverse of the generative model. Such a recognition model can be learned with similar methods and interleaved with the generative model.

Of course, the recognition with the recognition model early in learning is not expected to be exact, but estimation of new parameters from the recognized data in the M-step to maximize the expectation can be expected to be better than the model with the initial arbitrary values. The new model can then be compared to the data again and, when necessary, be used to generate new expectations from which the model is refined. This procedure is known as the *expectation maximization* (EM) algorithm. The distributions after three and nine such iterations, where we have chosen new data points in each iteration, are shown in Figs 10.17B and C.

Simulation

The program used to produce Fig. 10.17 is shown in Table 10.1. The vector x_0 , defined in Line 2, is used to plot the distributions later in the program. The arbitrary random initial conditions of the distribution parameters are set in Line 3. Line 4 defines an *inline function* of a properly normalized Gaussian since this function is used several times in the program. An inline function is an alternative to writing a separate function file. It defines the name of

Table 10.1 Program ExpectationMaximization.m

```

1  %% 1d example EM algorithm
2  clear; hold on; x0=-10:0.1:10;
3  var1=1; var2=1; mu1=-2; mu2=2;
4  normal= @(x,mu,var) exp(-(x-mu).^2/(2*var))/sqrt(2*pi*var);
5  while 1
6  %%plot distribution
7  clf; hold on;
8  plot(x0, normal(x0,-1,4), 'k:');
9  plot(x0, normal(x0,4,.25), 'k:');
10 plot(x0, normal(x0,mu1,var1), 'r');
11 plot(x0, normal(x0,mu2,var2), 'b');
12 waitforbuttonpress;
13 %% data
14 x=[2*randn(50,1)-1;0.5*randn(50,1)+4];
15 %% recognition
16 c=normal(x,mu1,var1)>normal(x,mu2,var2);
17 %% maximization
18 mu1=sum(x(c>0.5))/sum(c);
19 var1=sum((x(c>0.5)-mu1).^2)/sum(c);
20 mu2=sum(x(c<0.5))/(100-sum(c));
21 var2=sum((x(c<0.5)-mu2).^2)/(100-sum(c));
22 end

```

the functions, followed by a list of parameters and an expression, as shown in Line 4. The rest of the program consist of an infinite loop produced with the statement `while 1`, which is always true. The program has thus to be interrupted by closing the figure window or with the interruption command `Ctrl C`. In Lines 7–12, we produce plots of the real-world models (dotted lines) and the model distributions (plotted with a red and a blue curve when running the program). The command `waitforbuttonpress` is used in Line 12 so that we can see the results after each iteration.

In Line 14 we produce new random data in each iteration. Recognition of this data is done in Line 16 by inverting the generative model using Bayes' formula,

$$P(c|\mathbf{x}; G) = \frac{P(\mathbf{x}|c; G)P(c; G)}{P(\mathbf{x}; G)}. \quad (10.16)$$

In this specific example, we know that the data are equally distributed from each Gaussian so that the *prior distribution over causes*, $P(c; G)$ is 1/2 for each cause. Also, the *marginal distribution of data* is equally distributed, so that we can ignore this normalizing factor. The recognition model in Line 16 uses the Bayesian decision criterion, in which the data point is assigned to the cause with a larger *recognition distribution*, $P(c|\mathbf{x}; G)$. Using the labels of the data generated by the recognition model, we can then use the data to obtain new estimates of the parameters for each Gaussian in Lines 17–21.

Note that when testing the system for a long time, it can happen that one of the distributions is dominating the recognition model so that only data from one distribution are generated. The model of one Gaussian would then be *explaining away* data from the other cause. More practical solutions must take such factors into account.

10.4 Adaptive resonance theory

This book has introduced many important concepts underlying cognitive processes, including learning, different forms of memory, self-organization, attention, and anticipation. This last section outlines an important theory that combines many of these concepts and explains how they are related. The basic ideas were introduced in 1976 by *Stephen Grossberg*, and *adaptive resonance theory* (ART) was introduced more formally in 1987 by *Carpenter* and *Grossberg*. We concentrate here on the basic ideas of this theory and discuss a simplified version of the ART1 model for binary patterns. The theory has since been advanced in several directions, including models for real-valued patterns (ART2, FuzzyART), versions for supervised learning (ARTMAP, fuzzy ARTMAP) and several other variants to model specific aspects of cognitive functions.

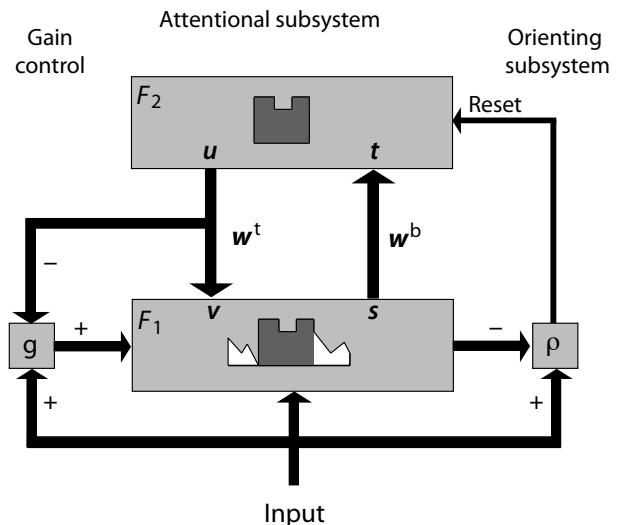
10.4.1 The basic model

We argued above that the brain is a hierarchical memory system that learns from data to anticipate causes in the environment. We have seen, with the help of the Boltzmann machine, how brain-generated sensory (top-down) states can be compared to physical evidence (bottom-up) to guide the self-organisation of useful representations in hierarchical structures. Furthermore, we mentioned more abstract Bayesian approaches to causal models based on learned concepts, and how such models can be used for probabilistic reasoning. The theory discussed in this section specifies more directly how bottom-up and top-down processes interact to guide learning.

One major challenge for advanced learning machines, and one that the brain seems to address particularly well, is the plasticity–stability dilemma, as mentioned in Section 7.2.3. The dilemma is that we want a learning system to learn new concepts or refine learned concepts quickly when appropriate, yet the system should be stable enough to not overthrow the gained experience and world model it acquired over its development. More specifically, when a pattern is observed in the environment, the question is how should this experience change our world model, should it change our acquired concepts, or should it learn the new input as a new concept? Also, by how much should a new input change an existing concept, and when is an input sufficiently different to everything the system has experienced before to grant the creation of a new concept?

These questions are addressed by *adaptive resonance theory* (ART). The basic idea of ART is outlined in Fig. 10.18. This example model has two layers labelled F_1 and F_2 . The arrows represent weight matrices as discussed in Chapter 6, and the letters relate the figure to the implementation discussed below. The model can be divided into three subsystems as noted on the top of

Fig. 10.18 Outline of adaptive resonance theory (ART). The model has two layers, F_1 and F_2 . An input is first represented with basic features in layer F_1 , which in turn can activate concepts (more abstract features, or categories) in layer F_2 . The arrows represent weight matrices that can transform representations as discussed in Chapter 6. The labels relate to the variables as used in the implementation discussed in the text. The output of layer F_1 is s , which is weighted with the bottom-up weight w^b to form the effective input t for layer F_2 . Similarly, u is the output of layer F_2 , which is weighted with the top-down weight w^t to form the effective top-down input v for layer F_1 . The parameter ρ is called the vigilance parameter, and g describes the gain mechanisms.



the figure, with an *attentional subsystem*, an *orienting subsystem*, and a *gain control subsystem*. We outline in the following briefly the basic functionality of the system, which should explain some of the nomenclature.

In the first layer, the input is represented by features, and the decomposition is achieved by the mapping indicated by the arrows. Without top-down expectations at this stage, layer F_1 also receives some unspecific gain input, g , as further specified below and indicated on the left in Fig. 10.18. The strong activation in F_1 is then mapped to a higher cortical area, F_2 . Let us call the features in F_2 ‘categories’ to distinguish them more easily from the features in F_1 , and also to allude to the fact that these categories are more abstract than the features in F_1 . There is competition among categories in F_2 which causes a specific category to dominate the activity in F_2 , similar to the mechanisms studied in Chapter 7 in the form of the centre-surround neural field theory. Thus, the selection of a winning category is indicated as selecting specific features in the figure, but the relations can be more complicated through the mapping with the bottom-up weight matrix w^b between F_1 and F_2 . Selection of a category in F_2 cancels the gain input.

We are here specifically interested in the continuous refinement of the bottom-up weights w^b and the top-down weights w^t . That is, if a new instance of a category is experienced, then this example should update the representation of that category. For example, when cars with little tail wings appeared in the 1960s, there was no need to think that these were rockets. But the representation of possible features of cars had certainly to be changed. The confirmation and refinement of the car category is achieved in the model outlined in Fig. 10.18 through a resonant state. This occurs when the activation of a specific category in F_2 is mapped back to F_1 , a process that corresponds to the attentional system discussed earlier, such as when the resulting states are not deemed sufficiently different from the initial input to trigger a search for

another category. Thus, the top-down signalling pathway can confirm some patterns in F_1 and can suppress others. This confirmation process can iterate for some time, creating an attentional short-term memory trace. This process requires some form of matching between the input and some category, but since an equivalence is not required, such a process is better termed as *resonance*. The corresponding resonant state can then refine the prototype vector \mathbf{w}_i^b for the selected category i , as well as the inverse mapping, \mathbf{w}_i^t , through Hebbian learning, hence the name of the theory. This learning process reinforces old features while also taking new features into account.

Now consider the case when the input is not a car and should be treated as a new category. Such a process is enabled through the mechanisms indicated on the right in Fig. 10.18. This pathway determines whether the state in F_1 , which is modified by top-down expectation, is sufficiently different from the original input. The similarity is compared to a threshold, ρ , called *vigilance*. A difference exceeding this threshold indicates a non-match with the selected category and triggers a new search by generating a reset signal. The reset signal cancels the top-down expectation and inhibits the currently selected category. The cancellation of the top-down signal triggers again unspecified gain, which, in turn, initiates the search for a new category. Such a process resembles an *orienting* mechanism in a visual search process, hence the label of this subsystem. Furthermore, the top-down enhancement of features in F_1 resembles attentional processes, and the necessary inactivation of the categories in F_2 in the case of a new search resembles some *inhibition of return* (IOR) in the attentional search literature.

The search process can select a new category, and it is possible that the new category supports a resonant state. However, it is also important that a new category can be created when none of the existing categories are appropriate. The proper switching between a resonant state and search is crucial in the network implementation. This process depends on many factors, including the way bottom-up and top-down signals are combined in F_1 , the specific normalization of weights and the activity in layer F_1 , as well as the choice of the vigilance parameter. The vigilance parameter might itself be altered, for example, through supervision, which could increase the vigilance parameter so that the system pays more attention to details in the features making up a category.

This brief outline of ART was intended to focus on the basic idea behind this theory, to describe how it solves the plasticity dilemma, to introduce the important aspects of resonant states, and to describe the important roles of attentional top-down processes and aroused states for learning novel categories. Such a framework has been implemented in many models to explain a variety of behavioural findings, and further studies in this area are highly recommended. While this theory explains many behavioural findings, ultimately, the specific implementation in the brain has to be clarified. It is important to examine how bottom-up (feedforward) and top-down (feedback) signals are combined in the brain. Or, more specifically, the question is how such models are realized within laminar cortical architectures. A lot of progress has recently been made in this area, such as the model shown in Fig. 5.7.

10.4.2 ART1 equations

As outlined above, adaptive resonance theory covers important principles of information processing in the brain, including iterations between bottom-up and top-down processing, resonant states, gain control and reset. In this section we describe a specific example implementation for binary input patterns called ART1, which we can use to demonstrate ART processing with the letter patterns used in Chapters 6 and 8. Analogue input patterns are covered with similar models known as ART2. The nodes in the ART architecture are described, as usual, as leaky integrators that receive excitatory and inhibitory inputs.

The dynamic of the internal states of nodes in layer F_1 are described by the equation

$$\tau \frac{dx_i}{dt} = -x_i + (1 - a_1 x_i) I_i^+ - (b_1 + c_1 x_i) I_i^- . \quad (10.17)$$

The inputs are modulated by the current activities in these equations to keep the activity bounded between $-b_1 c_1$ and a_1 . This form of modulation is called *shunting* in the case of inhibition, but does not have an explicit name for excitation. The constants a_1 , b_1 , and c_1 are all positive.

The F_1 layer receives three excitatory inputs,

$$I_i^+ = I_i + d_1 v_i + b_1 g, \quad (10.18)$$

where \mathbf{I} is the input pattern applied to the network, \mathbf{v} is the input generated from activities in layer F_2 as specified further below, and g is the input from the gain system. The output of the gain control system depends on the activation in layer F_2 and the input. It is 1 if input is provided but layer F_2 is not yet active, and is 0 otherwise. This non-specific input is necessary to facilitate the activation in layer F_1 to be able to activate layer F_2 in the case of a new search. Stability of the solutions requires that all constants are positive and that $\max(1, d_1) < b_1 < d_1 + 1$. Finally, the output of the F_1 units is given by the threshold activation function

$$s_i = \begin{cases} 1 & \text{for } x_i > 0 \\ 0 & \text{otherwise} \end{cases} . \quad (10.19)$$

The activation of the nodes in layer F_1 are mapped with the bottom-up weight matrix, \mathbf{w}^b to determine the effective input of layer F_2 .

$$\mathbf{t} = \mathbf{w}^b \mathbf{s}, \quad (10.20)$$

In the full ART1 model, the activations of F_2 nodes are also governed by a leaky integrator dynamic similar to eqn 10.17. However, since this layer is competitive, we need to use coupled differential equations as discussed in Chapter 7 for the dynamic neural field model. Rather than implementing this full model here, we simplify this part of the model in the following with a winner-takes-all maximum calculation, similar to the discussions in Section 7.2.2. Thus, the output of layer F_2 in the simplified model is given by

$$u_i = \begin{cases} 1 & \text{if } i = \text{argmax}(\mathbf{t}) \\ 0 & \text{otherwise} \end{cases} , \quad (10.21)$$

where the function `argmaxt` returns the index of the maximal element in \mathbf{t} . This output is weighted with the matrix \mathbf{w}^t to determine the top-down input to layer F_1 ,

$$\mathbf{v} = \mathbf{w}^t \mathbf{u}. \quad (10.22)$$

Learning of the weight matrices is produced by perceptron-like Hebbian learning rules. That is, the top-down weight matrix changes according to

$$\frac{dw_{ij}^t}{dt} = u_j(s_i - w_{ij}^t). \quad (10.23)$$

That is, only connections between the winning node in F_2 and active nodes in F_1 are changed. The weight vector for the winning node in F_2 represents the current prototype for this category which is made more similar to the current resonating example, \mathbf{s} . Learning of the bottom-up weights is similar, although complicated by the interacting nature of the competitive layer in F_2 . This learning requires therefore some form of normalization and is given by

$$\frac{dw_{ij}^b}{dt} = ku_i((1 - w_{ij}^b)Ls_j - w_{ij}^b \sum_{k \neq j} s_k), \quad (10.24)$$

with constants k and L . Note that the plasticity dynamics should be on a much smaller time-scale than the response dynamics of the resonating state. Thus, τ in eqn 10.17 should be $\tau \ll 1$.

10.4.3 Simplified dynamics for unsupervised letter clustering

We have already made some simplifications of the competitive network dynamic above by using a winner-take-all step instead of the full dynamic implementation. While this prevents us from modelling the specific time course of such processes, we are, in the following, mainly interested in demonstrating the self-organized clustering properties of such models. For this purpose, we can even simplify the model dynamic further.

Before input to the model is given ($\mathbf{I} = 0$), both the gain and the top-down input are zero ($g = 0$, $\mathbf{v} = 0$). Thus, the equilibrium activity of F_1 nodes ($dx/dt = 0$) is,

$$x_i = \frac{-b_1}{1 + c_1}. \quad (10.25)$$

Similarly, during the early processing stages before top-down signals become effective, \mathbf{v} is effectively zero, but the gain is now $g = 1$. The corresponding equilibrium values are

$$x_i = \frac{I_i}{1 + a_1(I_i + b_1) + c_1}. \quad (10.26)$$

When the top-down input becomes effective, the gain will be $g = 0$, so that at this stage the equilibrium value is

$$x_i = \frac{I_i + d_1 v_i + b_1}{I_1 + a_1(I_i + d_1 v_i) + c_1}. \quad (10.27)$$

In the following simulations, we use a model with three corresponding stages. We use the equilibrium states before applying inputs as initial values for the weight matrices. For the bottom-up weights, a simple scaled version of the inverse of the top-down weights is used.

As an example of ART processing, the simplified model is applied to learning visual representations of the letters of the alphabet, similar to examples in previous chapters. The model should be able to classify letter vectors as used in Chapters 6 and 8, but in this case we do not provide labels, or give specific examples, for each letter. Instead, we require the system to learn letter categories and prototype vectors from examples. The simulations shown here were performed with the program outlined below.

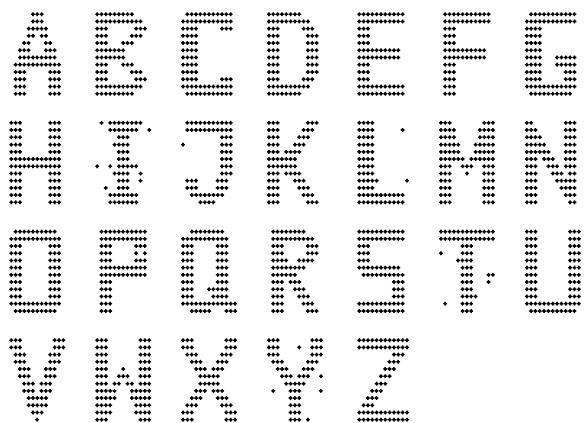


Fig. 10.19 Example of training the simplified ART1 model on noisy versions of binary letter patterns. This figure shows the prototype letters, given by the top-down weight vectors w_i^t for each category node i in layer F_2 , after training this network in an unsupervised way on 100 noisy examples of the letters used in Chapters 6 and 8.

In this example we used the same pattern for the letters as used in previous chapters. Each of these letter patterns is modified with 10% noise and then provided as training vector. The results of training with the simplified model of the basic ART architecture on such patterns is shown in Fig. 10.19 after presenting 100 examples of each letter. The figure shows the prototype vector for each of the 26 categories corresponding to each node in layer F_2 . All letters are learned as different categories. While some prototype vectors are still somewhat noisy, most of the categories have been able to extract the underlying pattern perfectly from the noisy examples through ongoing in-class learning.

Simulation

The simulation of the letter clustering was produced with the program shown in Table 10.2. The specific parameters used in this example are specified in Lines 4 and 5. Following the discussions in the last section, the bottom-up weight matrix, wb , and the top-down weight matrix, wt , are initialized in Line 6, and the activities of nodes in layers F_1 and F_2 , $x1$ and $x2$, are initialized in Line 7. We use the letter patterns from Chapter 6, provided in file **pattern1**, as inputs, where each letter is represented with an array of $12 \times 13 = 156$ binary components. The model has 26 output nodes to represent each of the letters,

although a different number could be provided. The letter matrices are loaded into the workspace in Line 9, and are reshaped into pattern vectors analogously to previous examples.

The main model simulation begins at Line 12 of the program with a loop over 100 iterations, where in each iteration one input pattern is presented to the network (Line 13). The specific input is generated in Lines 14 and 15 and consists of a letter from the letter file `pattern1` in which 16 random bits are flipped. A flag called `IOR` is set in Line 16, which inhibits return to a rejected category when its value is zero,

In the while loop starting at Line 18, the eligible categories are iterated over until no all categories are tried. The search for the most promising category is achieved in parallel; only the rejection of the selected category is serial. The network is updated in Lines 19–26 as outlined in the last section. At first, the equilibrium value of F_1 nodes without F_2 feedback is calculated, and this activity is then transformed into input to F_2 . The equilibrium value of F_1 nodes with top-down feedback and turned off gain is finally calculated. This value is used in Line 28 to decide if the pattern matches sufficiently the chosen category. Such a resonant state is then used to update the weight matrices in Line 29, and the search (while) loop is stopped with the `break` command. Otherwise, the category is rejected by setting the `IOR` flag to zero, and the search is continued among the remaining categories.

At the end, the prototypes for each category, encoded in the weight matrices, are displayed (Lines 37–40).

10.5 Where to go from here

We hope that we have provided enough background in computational neuroscience for you to follow-up more specific issues of your interest in the research literature. Also, we hope we have convinced you that analytical and computational studies can advance our understanding of how the brain works, by quantifying hypotheses that can lead to more precise predictions, which, in turn, can be verified experimentally. Even if you concentrate on experimental techniques, it is vital to connect them to more quantitative endeavours. For those concentrating on theoretical techniques, we hope that we have made it clear that theoretical studies must be rooted in experimental knowledge. A closer cooperation between theoreticians and experimentalist is currently emerging, and we strongly believe that such collaborations can considerably advance our understanding of brain functions.

Many aspects of brain function are still unknown, and the brain is one of the most challenging systems on our planet. There are still many very fundamental questions which are unanswered, including many details of the information processing mechanisms of the brain. However, there have also been quite specific proposals, such as the brain as an anticipatory memory system, which should be taken seriously and which should guide further experimental investigations.

Some brain areas have been targeted by modellers, and major progress in understanding these areas is expected. The cerebellum has been a promising area due to its elaborate architecture, but strong hypotheses have also been

Table 10.2 Program ART.m

```

1  %% ART1 network for letter recognition
2  clear;
3  % Parameters & initial values
4  n1=12*13; n2=26; % number of input and output nodes
5  a1=1; b1=1.5; c1=1; d1=1; L=2; rho=0.9;
6  wt=2*(b1-1)/d1*ones(n1,n2); wb=0.5*L/(L-1+n1)*ones(n2,n1);
7  x1=-b1/(1+c1)*ones(n2,1); x2=zeros(n2,1);
8  %% training vectors
9  load pattern1; rPat=reshape(pattern1', n1, n2); % matrix of inputs
10
11 %% ART update
12 for run=1:100;
13     for pat=1:n2
14         I=rPat(:,pat);
15         x=randperm(n1); I(x(1:16))=1-I(x(1:16)); %adding noise
16         IOR=ones(n2,1);
17
18         while sum(IOR)>0;
19             x1=I./(1+a1*(I+b1)+c1); % activ. of F1 with ext input
20             s1=(x1>0)*1; % output of F1
21             t=wb*s1; % input to F2
22             [maxv,maxi]=max(IOR.*t); % winner
23             u=zeros(n2,1); u(maxi)=1;% takes all
24             v=wt*u; % top-down input to F1
25             x1=(I+d1*v-b1)./(1+a1*(I+d1*v)+c1);% new activ. of F1
26             s1=(x1>0)*1; % new output of F1
27
28             if sum(s1)/sum(I) > rho; %resonant state
29                 wt(:,maxi)=wt(:,maxi)+0.1*(s1-wt(:,maxi));
30                 wb=wt'*L/(L-1+sum(s1));
31                 break
32             else
33                 IOR(maxi)=0; % search for other node
34             end
35         end
36     end
37 end
38 %% Display category prototypes
39 for i=1:n2
40     thisWeightArray=reshape(wt(:,i),13, 12)';
41     format +; disp(thisWeightArray-0.5); format;
42 end

```

advanced for the hippocampus, the basal ganglia, and others. A current challenge in computational neuroscience is the multitude of models with diverse aims. This multitude of models is expected, and desirable, since each model is designed to answer specific questions, but it is also necessary to extract principal knowledge from such studies, which can be compared to other types of investigations. The progression of our understanding of brain functions will also certainly lead to more convergence in modelling approaches.

A better understanding of brain processes will ultimately lead to the development of new health-care applications such as advanced rehabilitation treatment after brain damage. In fact, the study of the consequences of specific brain damage alone, is an important application area. Studying rehabilitation techniques takes a lot of effort with patients, and modelling studies can provide flexible and cost-efficient investigations of specific questions. Such modelling efforts could at least narrow down more specific questions to be answered in complementary studies. Computational studies must certainly become a more integral part of neuroscience research in the future.

Further reading

Most of this chapter follows some example papers which are cited in the figure captions. The example of a model of attentive vision by Deco and Zihl is described in Rolls and Deco (2001). This book also elaborates on such models and discusses further aspects of vision.

The discussions of anticipatory systems have been compiled from a variety of sources. The paper by Karl Friston (2005) is a great read and a reminder of important features of cortical architecture, and the book by Jeff Hawkins (2003) is a motivational must-read. Anticipatory systems have been studied by Robert Rosen (1985). A good first reading of restricted Boltzmann machines is Hinton (2007).

The principles of adaptive resonance theory have been proposed by Grossberg (1976), and the first concrete implementation was due to Carpenter and Grossberg (1987). I also recommend the book by Daniel Levine (2000), which not only discusses ART, but also neural network architectures in conjunction with cognitive models. The implementation of ART1 discussed in this chapter was mainly derived from Freeman (1994).

Edmund T. Rolls and Gustavo Deco (2001), *Computational neuroscience of vision*, Oxford University Press.

Karl Friston (2005), *A theory of cortical responses*,

in *Philosophical Transactions of the Royal Society B* 360, 815–36.

Jeff Hawkins with Sandra Blakeslee (2004), *On intelligence*, Henry Holt and Company.

Robert Rosen (1985), *Anticipatory systems: Philosophical, mathematical and methodological foundations*, Pergamon Press.

Geoffrey E. Hinton (2007), *Learning Multiple Layers of Representation*, in *Trends in Cognitive Sciences* 11: 428–434.

Stephen Grossberg (1976), *Adaptive pattern classification and universal recoding: Feedback, expectation, olfaction, and illusions*, in *Biological Cybernetics* 23: 187–202.

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

Daniel S. Levine (2000), *Introduction to neural and cognitive modeling*, Lawrence Erlbaum, 2nd edition.

James A. Freeman (1994), *Simulating neural networks with Mathematica*, Addison-Wesley.

This page intentionally left blank

Some useful mathematics



A.1 Vector and matrix notations

We frequently use vector and matrix notation in this book as it is extremely convenient for specifying neural network models. It is a shorthand notation for otherwise lengthy looking formulas, and formulas written in this notation can easily be entered into MATLAB. We consider three basic data types:

(1) Scalar:

$$a \text{ for example } 41 \quad (\text{A.1})$$

(2) Vector:

$$\mathbf{a} \text{ or component-wise } \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \text{ for example } \begin{pmatrix} 41 \\ 7 \\ 13 \end{pmatrix} \quad (\text{A.2})$$

(3) Matrix:

$$\mathbf{a} \text{ or component-wise } \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \text{ for example } \begin{pmatrix} 41 & 12 \\ 7 & 45 \\ 13 & 9 \end{pmatrix} \quad (\text{A.3})$$

We used bold face to indicate both a vector and a matrix because the difference is usually apparent from the circumstances. A matrix is just a collection of scalars or vectors. We talk about an $n \times m$ matrix where n is the number of rows and m is the number of columns. A scalar is thus a 1×1 matrix, and a vector of length n can be considered an $n \times 1$ matrix. A similar collection of data is called *array* in computer science. However, a matrix is difference because we also define operations on these data collections. The rules of calculating with matrices can be applied to scalars and vectors.

We define how to add and multiply two matrices so that we can use them in algebraic equations. The *sum of two matrices* is defined as the sum of the individual components

$$(\mathbf{a} + \mathbf{b})_{ij} = \mathbf{a}_{ij} + \mathbf{b}_{ij}. \quad (\text{A.4})$$

For example, \mathbf{a} and \mathbf{b} are 3×2 matrices, then

$$\mathbf{a} + \mathbf{b} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \\ a_{31} + b_{31} & a_{32} + b_{32} \end{pmatrix} \quad (\text{A.5})$$

Matrix multiplication is defined as

$$(\mathbf{a} * \mathbf{b})_{ij} = \sum_k \mathbf{a}_{ik} \mathbf{b}_{kj}. \quad (\text{A.6})$$

A.1 Vector and matrix notations	323
A.2 Distance measures	325
A.3 The δ -function	326

The matrix multiplication is hence only defined as multiplication matrices **a** and **b** where the number of columns of the matrix **a** is equal to the number of rows of matrix **b**. For example, for two square matrices with two rows and two columns, their product is given by

$$\mathbf{a} * \mathbf{b} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix} \quad (\text{A.7})$$

A handy rule for matrix multiplications is illustrated in Fig. A.1. Each component in the resulting matrix is calculated from the sum of two multiplicative terms. The rule for multiplying two matrices is tedious but straightforward and can easily be implemented in a computer. It is the default when multiplying variables of the matrix type in MATLAB. If we want to multiply each component of a matrix by the corresponding component in a second matrix, we just have to include the operator ‘.*’ between the matrices in MATLAB where the dot in front of the multiplication sign indicates ‘component-wise’.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

Fig. A.1 Illustration of a matrix multiplication. Each element in the resulting matrix consists of terms that are taken from the corresponding row of the first matrix and column of the second matrix. Thus in the example we calculate the highlighted element from the components of the first row of the first matrix and the second column of the second matrix. From these rows and columns we add all the terms that consist of the element-wise multiplication of the terms.

Another useful definition is the *transpose* of a matrix. This operation, indicated usually by a superscript *t* or a prime ('). The later is used in MATLAB. Taking the transpose of a matrix means that the matrix is rotated 90 degrees; the first row becomes the first column, the second row becomes the second column, etc. For example, the transpose of the example in A.3 is

$$\mathbf{a}' = \begin{pmatrix} 41 & 7 & 13 \\ 12 & 45 & 9 \end{pmatrix} \quad (\text{A.8})$$

The transpose of a vector transforms a column vector into a row vector and vice versa.

As already mentioned, matrices were invented to simplify the notations for systems of coupled algebraic equations. Consider, for example, the system of three equations

$$41x_1 + 12x_2 = 17 \quad (\text{A.9})$$

$$7x_1 + 45x_2 = -83 \quad (\text{A.10})$$

$$13x_1 + 9x_2 = -5. \quad (\text{A.11})$$

This can be written as

$$\mathbf{ax} = \mathbf{b} \quad (\text{A.12})$$

with the matrix \mathbf{a} as in the example of A.3, the vector $\mathbf{x} = (x_1 \ x_2)'$, and the vector $\mathbf{b} = (17 \ -83 \ -5)'$. Solutions of such equation systems can be formulated by using matrix notation, and many corresponding routines are implemented in MATLAB.

A.2 Distance measures

We are often confronted with comparing two vectors. It is straightforward comparing two vectors to decide if they are the same by inspecting the equivalence of each component. However, if they are different we would like to put a value, d , on the difference that gives us some measure of how different they are. There are many possible different definitions of such a difference measure, although we should impose some obvious restrictions on such measures:

- The value should be $d = 0$ if the vectors are the same. We do not allow negative values because we wish to interpret this value as a distance.
- The value should be positive, $d > 0$, if they are not the same.
- The distance from vector \mathbf{a} to \mathbf{b} should be the same as from \mathbf{b} to \mathbf{a} .

We sometimes have to evaluate the distance between two binary vectors \mathbf{a} and \mathbf{b} , in which the components have only two possible values for example $a_i, b_i \in \{0, 1\}$. A common value for the distance of such vectors is obtained by counting how many components (bits) are different. This measure is called the *Hamming distance*, d^h . If we normalize this value to the number of components of the vector, N , then we get a value between 0 and 1. This *normalized Hamming distance* can be calculated using the formula

$$d^h(\mathbf{a}, \mathbf{b}) = \frac{1}{N} \sum_i a_i(1 - b_i) + (1 - a_i)b_i. \quad (\text{A.13})$$

However, it is not obvious how to generalize this measure to vectors with real numbered components as we would have to judge the amount of the difference among the components, using their possible values, which we might not know *a priori*. One possible definition is to use the *normalized dot product* between vectors. The *dot product*, or *inner product*, of two column vectors \mathbf{a} and \mathbf{b} is given by

$$\mathbf{a}'\mathbf{b} = \sum_i a_i b_i. \quad (\text{A.14})$$

We used the transpose of the second vector so that this is only a special case of a matrix multiplication with a row vector \mathbf{a}' and a column vector \mathbf{b} . In a geometrical interpretation of vectors (see Fig. A.2) this number is proportional to the cosine of the angle between the two vectors,

$$\mathbf{a}\mathbf{b}' = \|\mathbf{a}\| \|\mathbf{b}'\| \cos(\alpha), \quad (\text{A.15})$$

where $\|\mathbf{a}\|$ is the *length* or *norm* of vector \mathbf{a} . The *Euclidean norm* of the vector is defined as

$$\|\mathbf{a}\| = \sqrt{\sum_i a_i^2}, \quad (\text{A.16})$$

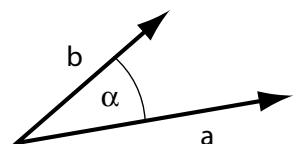


Fig. A.2 Graphical representation of two vectors in a two-dimensional space with an angle α between them.

which can also be written with the help of a dot product as

$$\|\mathbf{a}\| = \sqrt{\mathbf{a}'\mathbf{a}}. \quad (\text{A.17})$$

The cosine of the angle α between two vectors is a number between -1 and 1 that is only zero when the vectors are pointing in the same direction. The absolute value of this number is therefore a possible definition of the similarity of two vectors,

$$d^{\text{dot}} = \left| \frac{\mathbf{a}\mathbf{b}'}{\|\mathbf{a}\| \|\mathbf{b}'\|} \right|. \quad (\text{A.18})$$

Other definitions, such as taking the square of the normalized dot product, are also valid. Another definition that is sometimes used as measure of the distance between two vectors with positive real numbered components $a_i, b_i \in \mathbf{R}^+$ has the form of the *Pearson correlation coefficient* from statistics, namely

$$d^{\text{Pearson}} = \frac{\|\mathbf{a}\mathbf{b}'\| - \|\mathbf{a}\| \|\mathbf{b}'\|}{\sqrt{(\|\mathbf{a}^2\| - \|\mathbf{a}\|^2)(\|\mathbf{b}^2\| - \|\mathbf{b}\|^2)}}. \quad (\text{A.19})$$

We call this distance measure the *Pearson distance*.

A.3 The δ -function

The δ -function is a very convenient notation, which is formally a functional since it is only defined as an operation over a function. One can think of it as a density function that is zero except for its arguments for which it is infinite, and the integral over the δ -function is one; that is,

$$\int_{-\infty}^{\infty} \delta(x = x_1) dx = 1. \quad (\text{A.20})$$

It is usually used as an integration kernel with other functions as in

$$\int_{-\infty}^{\infty} \delta(x_1) f(x) dx = f(x_1). \quad (\text{A.21})$$

The delta function is useful for writing discrete events in a continuous form.

Numerical calculus

B

B.1 Differences and sums

We are often interested how a variable, such as the membrane potential or population rate of cell assemblies, change with time. Let us call this quantity $x(t)$ for now, where we indicated that it changes with time. The change of this variable from time t to time $t' = t + \Delta t$ is then

$$\Delta x = x(t + \Delta t) - x(t). \quad (\text{B.1})$$

The quantity Δt is the finite difference in time. For a continuously changing quantity we could also think about the instantaneous change value, dx , by considering an infinitesimally small time step. Formally,

$$dx = \lim_{\Delta t \rightarrow 0} \Delta x = \lim_{\Delta t \rightarrow 0} (x(t + \Delta t) - x(t)). \quad (\text{B.2})$$

The infinitesimally small time step is often written as dt . Calculating with such infinitesimal quantities is covered in the mathematical discipline of calculus, but on the computer we have always finite differences and we need to consider very small time steps to approximate continuous formulation. With discrete time steps, differential become differences and integrals become sums

$$dx \rightarrow \Delta x \quad (\text{B.3})$$

$$\int dx \rightarrow \Delta x \sum \quad (\text{B.4})$$

Note the factor of Δx in front of the summation in the last equation. It is easy to forget this factor when replacing integrals with sums. The following demonstrates this discretization for a differential equation.

B.2 Numerical integration of an initial value problem

The time dynamics of several models in this book are defined with differential equations that specify the change of a quantity dx for an infinitesimal time step dt . This change is specified by a function $f(x, t)$, which may depend on the quantity x and sometimes also explicitly by the time. Mathematically this is expressed by

$$\tau \frac{dx}{dt} = f(x, t), \quad (\text{B.5})$$

B.1 Differences and sums	327
B.2 Numerical integration of an initial value problem	327
B.3 Euler method	328
B.4 Higher-order methods	330
B.5 Adaptive Runge–Kutta	331
Further reading	334

which is a first-order differential equation with one independent variable and time constant τ . We outline here the principles of some numerical integration techniques for this simple case, but the methods can be generalized easily to more complex coupled differential equations and differential equations that contain higher-order differentials. We are concerned with calculating the values of the quantity $x(t)$ for specific times $t > t_0$ when the value of the quantity at time $t = t_0$ is known,

$$x(t_0) = x_0. \quad (\text{B.6})$$

This is called an *initial value problem*. We often refer to the initial value problem in the text as numerical integration. However, numerical integration can also refer to other numerical tasks such as numerically approximating integrals like $\int_{x_1}^{x_2} f(x)dx$, or to finding solutions of differential equations with other constraints.

B.3 Euler method

Digital computers can only represent discrete numbers and not infinitesimally small changes. We must therefore express the continuous dynamic given by eqn B.5 in discrete time steps. We call this procedure *discretization*. The simplest form is to use small but finite time steps between two consecutive times t_1 and t_2 . We write this time step as

$$\Delta t = t_2 - t_1. \quad (\text{B.7})$$

The continuous process is recovered in the limit $\Delta t \rightarrow 0$, which we call the *continuum limit*. The change of the quantity between the two time steps can be expressed as

$$\Delta x(t + \Delta t) = x(t + \Delta t) - x(t). \quad (\text{B.8})$$

The differential eqn B.5 can thus be discretized by writing it as a difference equation

$$\frac{\Delta x(t + \Delta t)}{\Delta t} = f(x(t), t). \quad (\text{B.9})$$

We have assigned the value of the change specified on the right-hand side at time t to the change at time $t + \Delta t$, though we could have also chosen $f(x(t + \Delta t), t)$ as the right hand side. The numerical values would then be a little bit different and the discrete systems are indeed a little bit different. However, we have to make all the following choices in the light of the continuum limit in which we should recover the same answers for the different methods. Substituting eqn B.8 into eqn B.9 gives

$$x(t + \Delta t) = x(t) + \Delta t f(x(t), t). \quad (\text{B.10})$$

This equation simply states the meaning of a differential in a discrete situation. The value of variable x at the next time step is given by the value of the variable at the previous time step times plus Δt the change specified by the function f . This iterative method tracks the development of quantity x from an initial value to later times by summing up the changes. This simplest method of numerical integration, which extrapolates changes linearly, is known as the *Euler method*.

Table B.1 Values of different numerical solutions of the differential equation $dx/dt = t - x + 1$ with initial condition $x(0) = 1$

time t	Exact solution	Euler method	Runge–Kutta (adaptive)
0	1	1	1
0.2	1.01873075	1.0400	1.01873077
0.4	1.07032005	1.1120	1.07032008
0.6	1.14881164	1.2096	1.14881168
0.8	1.24932896	1.3277	1.24932901
1.0	1.36787944	1.4621	1.36787949
1.2	1.50119421	1.6097	1.50119426
1.4	1.64659696	1.7678	1.64659701
1.6	1.80189652	1.9342	1.80189656
1.8	1.96529889	2.1074	1.96529893
2.0	2.13533528	2.2859	2.13533532

To illustrate this procedure we follow a specific example, given by the function

$$f(x, t) = 1 - x + t. \quad (\text{B.11})$$

The corresponding differential equation (inserting the function into eqn B.5) can be solved analytically. With the initial condition $x(0) = 1$, this is given by

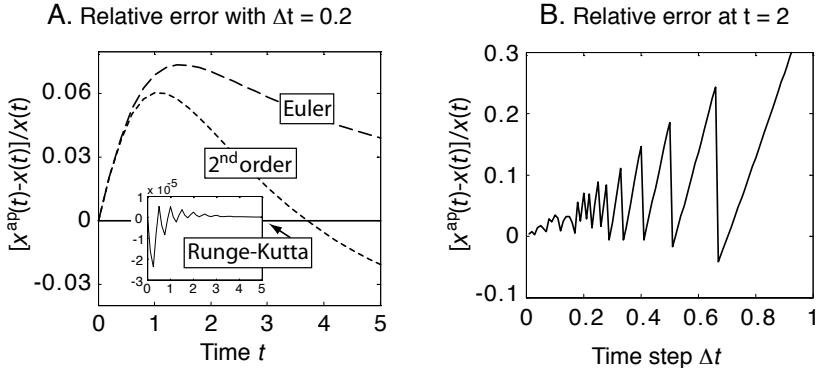
$$x(t) = t + e^{-t}, \quad (\text{B.12})$$

which can be verified by inserting this function into eqn B.5. Values for this function at different time steps are shown in Table B.1 in the column labelled ‘Exact solution’. The Euler method for this example is

$$x(t + \Delta t) = x(t) + (1 - x(t))\Delta t. \quad (\text{B.13})$$

The approximation values calculated with the Euler method using a time step of $\Delta t = 0.2$ are included in the column labelled ‘Euler method’ in Table B.1. The relative difference between these two columns, $[x^{\text{ap}}(t) - x(t)]/x(t)$, where x^{ap} is the approximation from different numerical integration methods discussed in this appendix, is plotted in Fig. B.1A. The Euler method is overestimating the true value in this particular example since the slope of the solution is increasing only slowly and we overestimated the slope by taking the finite integration time step. The relative error is decreasing with time since the function becomes more linear on larger scales. The error at $t = 2$ for different integration time steps Δt is shown in Fig. B.1B. This is a non-monotone function so that tuning Δt is not always simple.

Fig. B.1 (A) The relative difference between numerical solutions and the solution for the differential equation $dx/dt = t - x + 1$ with the initial condition $x(0) = 1$. For the Euler method and the second order method we set the integration time step to $\Delta t = 0.2$. The higher order Runge-Kutta method uses an adaptive integration time step. The absolute error is much smaller than in the other cases, and differences less than $O(10^{-5})$ are depicted in the insert. (B) Relative error of the Euler method for different values of the integration time step Δt at time $t = 2$.



B.4 Higher-order methods

In the Euler method just outlined we have only taken the first derivative of the function f (the slope of the function) into account and have used this to extrapolate linearly to the value of x at the next time step $t + \Delta t$. We can improve the accuracy of the approximation if we also take into account higher-order derivatives (curvature terms). We can formalize this by considering the *Taylor expansion* of the function around the time for which we want to estimate the value. The Taylor expansion is given by

$$x(t + \Delta t) = x(t) + \Delta t \frac{dx}{dt} + \frac{1}{2}(\Delta t)^2 \frac{d^2x}{dt^2} + O((\Delta t)^3), \quad (\text{B.14})$$

where the last term stands for all possible higher-order terms. This equation with the first two terms is the Euler method (eqn B.10) since the derivative is given by $f(x, t)$ as stated by eqn B.5. If we consider the next order term in the Taylor expansion for our example, we get the approximation

$$x(t + \Delta t) = x(t) + (1 - x + t)\Delta t + \frac{1}{2}(1 - x)(\Delta t)^2, \quad (\text{B.15})$$

which adds a term proportional to $(\Delta t)^2$ to the previous expression. The second order relative difference of the second order solution to the exact solution is also shown in Fig. B.1A. While this method is a little bit better for this example for intermediate extrapolation times, the error increases again for large times.

If we use higher-order terms we can further improve the approximations of the exact solution. This method requires that we know higher derivatives of the function f explicitly. In case they are difficult to calculate analytically, we can also estimate them numerically, although this introduces further numerical errors. This method is therefore not used directly but rather indirectly in the following way. If we use only the first derivative (slope) of the function at time t we assume that this is the same for subsequent times. If this is changing (for example, if the function has a curvature) then we use the slope at a time between t and $t + \Delta t$ as a better guess for the average slope in this interval. Thus, instead of using $f(x, t)$ in eqn B.10 we should use

$$f(x, t) \rightarrow f(\tilde{x}, t + \alpha\Delta t), \quad (\text{B.16})$$

where α is a parameter. The value \tilde{x} should be the value of x at $t + \alpha\Delta t$, that is, $\tilde{x} = x(t + \alpha\Delta t)$, which we don't know *a priori*. We could, however, guess this using the first-order Euler method, that is,

$$\tilde{x} = x(t) + \alpha\Delta t f(x(t), t). \quad (\text{B.17})$$

A common choice for the parameter α is $\alpha = 1/2$, in which case this method is called the *midpoint method*. This method cancels error terms of first order and corresponds therefore to a second-order method. The midpoint method requires two function calls, but no higher derivatives are needed. Note that we made several assumptions that might often be appropriate, but not always. Therefore, higher-order methods can usually improve the approximation of the solution $x(t)$, but this does not have to be the case in all circumstances.

B.5 Adaptive Runge–Kutta

It is not necessary to stop at the second-order and midpoint approximation and it is possible (though tedious) to work out solutions for higher-order approximations. The method most commonly used for numerical integrations is thereby a fourth-order method known as the *Runge–Kutta method*. This method is given by the following set of equations

$$k_1 = \Delta t f(x(t), t) \quad (\text{B.18})$$

$$k_2 = \Delta t f(x(t) + \frac{1}{2}k_1, t + \frac{1}{2}\Delta t) \quad (\text{B.19})$$

$$k_3 = \Delta t f(x(t) + \frac{1}{2}k_2, t + \frac{1}{2}\Delta t) \quad (\text{B.20})$$

$$k_4 = \Delta t f(x(t) + k_3, t + \Delta t) \quad (\text{B.21})$$

$$x(t + \Delta t) = x(t) + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4, \quad (\text{B.22})$$

which is a direct generalization of the midpoint method. The conclusions we drew for the midpoint method still hold. This fourth-order method is often better than lower-order methods, although this cannot always be guaranteed.

Applying any of these numerical methods requires that we check that the results we get do not critically depend on the choice of the parameters such as the time step Δt . At the very least we should check that the numerical results do not change more than within a certain range that we demand of the solutions when modifying the time step parameter considerably (for example, double it and halve it). However, many numerical solvers have already implemented such strategies with algorithms that change the time step as long as the variations in the results are lower than the error bound we give to the system. Such algorithms are called *adaptive time step methods*. The numerical solution of our example solved with the MATLAB routine `ode45()`, which is basically a Runge–Kutta of at least fourth order with adaptive time steps, is included in Table B.1. The maximum of the absolute difference is less than $5 * 10^{-8}$ with the standard parameters used by MATLAB. This is six orders of magnitude better than the Euler approximation.

There are several other methods for numerical integration, each having different strengths and weaknesses in certain application domains. For example,

Table B.2 Some MATLAB functions for numerical integrations of first-order differential equations [adapted from Hanselman and Littlefield, *Mastering MATLAB® 5*, Prentice Hall (1998)]

MATLAB function	Method	Comments
<code>ode23</code>	Runge–Kutta (order 2–3)	For problems that are non-stiff or that are discontinuous, or where lower accuracy is acceptable
<code>ode45</code>	Runge–Kutta (order 4–5)	Non-stiff problems, often tried first for new problem
<code>ode15s</code>	Multistep (order 1–5)	Stiff problems, try if <code>ode45</code> fails or is too inefficient
<code>ode113</code>	Adams–Bashford–Moulton (order 1–13)	Non-stiff-problems, multistep method, where $f(x, t)$ is expensive to compute, not suitable for discontinuities
<code>ode23s</code>	Modified Rosenbrock (order 2)	Stiff problems with discontinuities, lower accuracy is acceptable

the Runge–Kutta method breaks down if there are very sudden changes in the exact solution for which the linear interpolations used in the midpoint approach break down. MATLAB provides implementations of several other solvers that should be considered for various types of problems. A summary of such solvers is provided in Table B.2. The different solvers are provided within MATLAB with the same generic function calls, so that a switch between solvers can easily be achieved by changing only the name of the function.

Simulation

The results shown in Fig. B.1 are produced with the program in Table B.3. In Line 1 we make sure to clear the workspace (`clear`) and possible figure (`clf`), and hold the plot so that several lines can be drawn in the same figure. The numerical integrations with the Euler method (Lines 3–7) and the second-order method (Lines 8–12) are direct translations of the formulas above. In Line 15 we use the MATLAB function `ode45()`, which is an efficient implementation of an adaptive higher-order Runge–Kutta method. The right-hand side of the differential equation is thereby provided in a function we have to write. In this example we use in Line 14 a MATLAB construct called *anonymous function* to define this function in the same file. Lines 16–20 are then used to generate a plot of results used in Fig. B.1A.

The results in Fig. B.1B are produced by the rest of the code. It starts by calling `figure`, which produces first a new figure window so that the previous plot is not overwritten. We then run the Euler method many times with different time steps Δt .

Table B.3 Program app_numinte.m

```
1 clear; clf; hold on;
2 dt=0.2; tend=5;
3 %% Euler method
4 x=1; xEuler=x;
5 for t=dt:dt:tend
6     x=x+dt*(1-x+t); xEuler=[xEuler,x];
7 end
8 %% Second order
9 x=1; xSecond=x;
10 for t=dt:dt:tend
11     x=x+dt*(1-x+t)+dt^2*(1-x)/2; xSecond=[xSecond,x];
12 end
13 %% Runge-Kutta
14 ode_function= @(t,x,flag) 1-x+t;
15 [t,xRK]=ode45(ode_function,[0 tend],1,[]);
16 %% Plotting results
17 x=t+exp(-t); plot(t,(xRK-x)./x,'r');
18 t=0:dt:tend; x=t+exp(-t);
19 plot(t,(xEuler-x)./x);
20 plot(t,(xSecond-x)./x,'g');
21 %% Plot dt error
22 figure; xDIFF=[];
23 for dt=0.01:0.01:1
24     x=1; for t=dt:dt:2; x=x+dt*(1-x+t); end
25     xDIFF=[xDIFF,(x-2-exp(-2))/(2+exp(-2))];
26 end
27 dt=0.01:0.01:1; plot(dt,xDIFF);
```

Further reading

The book by Press *et al.* (1992) is a standard reference you will find on the desks of most people applying computer simulations. The book by Flower (1995) is a book that is more teaching-oriented with clear descriptions and explicit program examples.

William H. Press, Saul A. Teukolsky, William

T. Vetterling, and Brian P. Flannery (1992), *Numerical recipes in C: The art of scientific computing*, Cambridge University Press, 2nd edition.

B. H. Flowers (1995), *An introduction to numerical methods in C++*, Oxford University Press.

Basic probability theory

C

This appendix reviews briefly the basic properties of random numbers and the probabilistic framework. This includes a list of some common *probability density functions* used in this book. This section is not about statistics, which is primarily concerned with hypothesis testing. Rather, we review here some concepts that provide a useful description framework of brain processes.

C.1 Random numbers and their probability (density) function

The most important concept introduced in this appendix is that of *random numbers*, which are denoted by capital letters in this appendix to distinguish them from regular numbers written in lower case. A random variable, X , is a quantity that can have different values each time the variable is inspected, such as in measurements in experiments. This is fundamentally different to a regular variable, x , which does not change its value once it is assigned. A random number is thus a new mathematical concept, not included in the regular mathematics of numbers. A specific value of a random number is still meaningful as it might influence specific processes in a deterministic way. However, since a random number can change every time it is inspected, it is also useful to describe more general properties when drawing examples many times. The frequency with which numbers can occur is then the most useful quantity to take into account. This frequency is captured by the mathematical construct of a *probability*.

We can formalize this with some compact notations. We speak of a *discrete random variable* in the case of discrete numbers for the possible values of a random number. A *continuous random variable* is a random variable that has possible values in a continuous set of numbers. There is, in principle, not much difference between these two kinds of random variables, except that the mathematical formulation has to be slightly different to be mathematically correct. For example, the *probability function*,

$$P_X(x) = P(X = x) \quad (\text{C.1})$$

describes the frequency with which each possible value x of a discrete variable X occurs. Note that x is a regular variable, not a random variable. The value of $P_X(x)$ gives the fraction of the times we get a value x for the random variable X if we draw many examples of the random variable.¹ From this definition it follows that the frequency of having any of the possible values is equal to one,

C.1	Random numbers and their probability (density) function	335
C.2	Moments: mean, variance, etc.	336
C.3	Examples of probability (density) functions	338
C.4	Cumulative probability (density) function and the Gaussian error function	340
C.5	Functions of random variables and the central limit theorem	341
C.6	Measuring the difference between distributions	342
	Further reading	344

¹Probabilities are sometimes written as a percentage, but we will stick to the fractional notation.

which is the normalization condition

$$\sum_x P_X(x) = 1. \quad (\text{C.2})$$

In the case of continuous random numbers we have an infinite number of possible values x so that the fraction for each number becomes infinitesimally small. It is then appropriate to write the probability distribution function as $P_X(x) = p_X(x)dx$, where $p_X(x)$ is the *probability density function* (pdf). The sum in eqn C.2 then becomes an integral, and normalization condition for a continuous random variable is

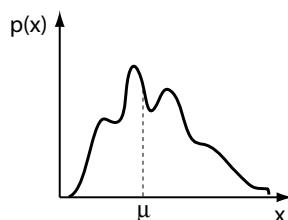
$$\int_x p_X(x)dx = 1. \quad (\text{C.3})$$

We will formulate the rest of this section in terms of continuous random variables. The corresponding formulas for discrete random variables can easily be deduced by replacing the integrals over the pdf with sums over the probability function. It is also possible to use the δ -function, outlined in Appendix A, to write discrete random processes in a continuous form.

C.2 Moments: mean, variance, etc.

In the following we only consider independent random values that are drawn from identical pdfs, often labeled as iid (independent and identically distributed) data. That is, we do not consider cases where there is a different probabilities of getting certain numbers when having a specific number in a previous trial. The static probability density function describes, then, all we can know about the corresponding random variable.

Let us consider the arbitrary pdf, $p_X(x)$, with the following graph:



Such a distribution is called *multimodal* because it has several peaks. Since this is a pdf, the area under this curve must be equal to one, as stated in eqn C.3. It would be useful to have this function parameterized in an analytical format. Most pdfs have to be approximated from experiments, and a common method is then to fit a function to the data. However, sometimes it is sufficient to know at least some basic characteristics of the functions. For example, we might ask what the most frequent value is when drawing many examples. This number is given by the largest peak value of the distribution. A more common quantity

to know is the expected arithmetic average of those numbers, which is called the *mean*, *expected value*, or *expectation value* of the distribution, defined by

$$\mu = \int_{-\infty}^{\infty} xf(x)dx. \quad (\text{C.4})$$

In the discrete case, this formula corresponds to the formula of calculating an arithmetic average, where we add up all the different numbers together with their frequency. Formally, we need to distinguish between a quantity calculated from random numbers and quantities calculated from the pdfs. If we treat the pdf as fundamental, then the arithmetic average is like an estimation of the mean. This is usually how it is viewed. However, we could also be pragmatic and say that we only have a collection of measurements so that the numbers are the ‘real’ thing, and that pdfs are only a mathematical construct. While this is mainly a philosophical debate, we try to be consistent in calling the quantities derived from data ‘estimates’ of the quantities defined through pdfs.

The mean of a distribution is not the only interesting quantity that characterizes a distribution. For example, we might want to ask what the *median* value is for which it is equally likely to find a value lower or larger than this value. Furthermore, the spread of the pdf around the mean is also very revealing as it gives us a sense of how spread the values are. This spread is often characterized by the standard deviation (std), or its square, which is called *variance*, σ^2 , and is defined as

$$\sigma^2 = \int_{-\infty}^{\infty} (x - \mu)^2 f(x)dx. \quad (\text{C.5})$$

This quantity is generally not enough to characterize the probability function uniquely; this is only possible if we know all moments of a distribution, where the *n*th *moment about the mean* is defined as

$$m^n = \int_{-\infty}^{\infty} (x - \mu)^n f(x)dx. \quad (\text{C.6})$$

The *variance* is the second moment about the mean,

$$\sigma^2 = \int_{-\infty}^{\infty} (x - \mu)^2 f(x)dx. \quad (\text{C.7})$$

Higher moments specify further characteristics such as the kurtosis and skewness of the distribution. Moments higher than this have not been given explicit names. Knowing all moments of a distribution is equivalent in knowing the distribution precisely, and knowing a pdf is equivalent in knowing everything we could know about a random variable.

In case the distribution function is not given, moments have to be estimated from data. For example, the mean can be estimated from a sample of measurements by the *sample mean*,

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i, \quad (\text{C.8})$$

and the variance either from the *biased sample variance*,

$$s_1^2 = \frac{1}{n} \sum_{i=1}^n (\bar{x} - x_i)^2, \quad (\text{C.9})$$

or the *unbiased sample variance*

$$s_2^2 = \frac{1}{n-1} \sum_{i=1}^n (\bar{x} - x_i)^2. \quad (\text{C.10})$$

A statistic is said to be biased if the mean of the sampling distribution is not equal to the parameter that is intended to be estimated. Knowing all moments uniquely specifies a pdf.

C.3 Examples of probability (density) functions

There is an infinite number of possible pdfs. However, some specific forms have been very useful for describing some specific processes and have thus been given names. The following is a list of some frequently used examples.

C.3.1 Bernoulli distribution

A Bernoulli random variable is a variable from an experiment that has two possible outcomes: success with probability p ; or failure, with probability $(1-p)$.

Probability function:

$$P(\text{success}) = p; P(\text{failure}) = 1 - p$$

mean: p

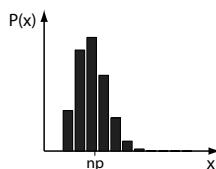
variance: $p(1-p)$

C.3.2 Binomial distribution

The number of successes in n Bernoulli trials with probability of success p is binomially distributed. Note that the binomial coefficient is defined as

$$\binom{n}{x} = \frac{n!}{x!(n-x)!} \quad (\text{C.11})$$

and is given by the MATLAB function `nchoosek`.



Probability function:

$$P(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

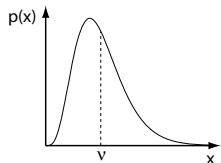
mean: np

variance: $np(1-p)$

C.3.3 Chi-square distribution

The sum of the squares of normally distributed random numbers is chi-square distributed and depends on a parameter ν that is equal to the mean. Γ is the

gamma function included in MATLAB as `gamma`.



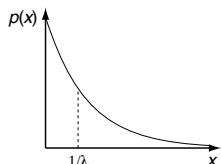
Probability density function:

$$p(x) = \frac{x^{(\nu-2)/2} e^{-x/2}}{2^{\nu/2} \Gamma(\nu/2)}$$

mean: ν
variance: 2ν

C.3.4 Exponential distribution

Distributions of time between events that are Poisson distributed. Depends on one parameter λ , which is called the hazard function.



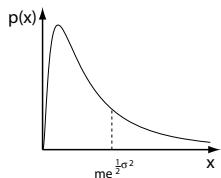
Probability density function:

$$p(x) = \lambda e^{-\lambda x}$$

mean: $1/\lambda$
variance: $1/\lambda^2$

C.3.5 Lognormal distribution

Distribution that is constrained to be zero at $x = 0$ and has a few large numbers. It depends on two parameters, the scale parameter m (median) and the shape parameter σ .



Probability density function:

$$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{[\log(x/m)]^2}{2\sigma^2}}$$

mean: $me^{\frac{1}{2}\sigma^2}$
variance: $m^2 e^{\sigma^2} (1 - e^{\sigma^2})$

C.3.6 Multinomial distribution

Generalization of the binomial distribution where n trials can have k outcomes, each with different probabilities p_i .

Probability function:

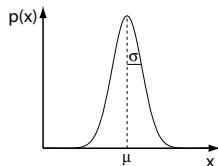
$$P(x_i) = n! \prod_{i=1}^k (p_i^{x_i} / x_i!)$$

mean: np_i
variance: $np_i(1 - p_i)$

C.3.7 Normal (Gaussian) distribution

Limit of the binomial distribution for a large number of trials. Depends on two parameters, the mean μ and the standard deviation σ . The importance of the

normal distribution stems from the central limit theorem outlined below.



Probability density function:

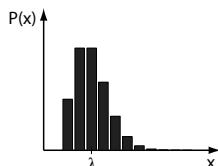
$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

mean: μ

variance: σ^2

C.3.8 Poisson distribution

This discrete distribution is frequently used to describe the number of rare but open-ended events and to model spike trains of cortical neurons. It is closely related to the exponential distribution given above. The parameter λ is equal to the mean and the variance.



Probability function:

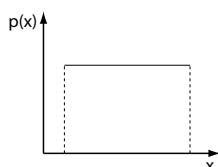
$$P(x) = \frac{\lambda^x}{x!} e^{-\lambda}$$

mean: λ

variance: λ

C.3.9 Uniform distribution

Equally distributed random numbers in the interval $a \leq x \leq b$. Pseudo-random variables with this distribution are often generated by routines in many programming languages.



Probability density function:

$$p(x) = \frac{1}{b-a}$$

mean: $(a+b)/2$

variance: $(b-a)^2/12$

C.4 Cumulative probability (density) function and the Gaussian error function

The probability of having a value x for the random variable X in the range of $x_1 \leq x \leq x_2$ is given by

$$P(x_1 \leq X \leq x_2) = \int_{x_1}^{x_2} p(x) dx. \quad (\text{C.12})$$

Note that we have shortened the notation by replacing the notation $P_X(x_1 \leq X \leq x_2)$ by $P(x_1 \leq X \leq x_2)$ to simplify the following expressions. In the main text we often need to calculate the probability that a normally (or Gaussian) distributed variable has values between $x_1 = 0$ and $x_2 = y$. The probability of eqn C.12 then becomes a function of y . This defines the *Gaussian error*

function

$$\frac{1}{\sqrt{2\pi}\sigma} \int_0^y e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx = \frac{1}{2} \operatorname{erf}\left(\frac{y-\mu}{\sqrt{2}\sigma}\right). \quad (\text{C.13})$$

This Gaussian error function (erf) for normally distributed variables (Gaussian distribution with mean $\mu = 0$ and variance $\sigma = 1$) is commonly tabulated in books on statistics. Programming libraries also frequently include routines that return the values for specific arguments. In MATLAB this is implemented by the routine `erf`, and values for the inverse of the error function are returned by the routine `erfinv`.

Another special case of eqn C.12 is when x_1 in the equation is equal to the lowest possible value of the random variable (usually $-\infty$). The integral in eqn C.12 then corresponds to the probability that a random variable has a value smaller than a certain value, say y . This function of y is called the *cumulative density function* (cdf),²

$$P^{\text{cum}}(x < y) = \int_{-\infty}^y p(x)dx, \quad (\text{C.14})$$

which we will utilize further below.

²Note that this is a probability function, not a density function.

C.5 Functions of random variables and the central limit theorem

A function of a random variable X ,

$$Y = f(X), \quad (\text{C.15})$$

is also a random variable, Y , and we often need to know what the pdf of this new random variable is. Calculating with functions of random variables is a bit different to regular functions and some care has been given in such situations. Let us illustrate how to do this with an example. Say we have an equally distributed random variable X as commonly approximated with pseudo-random number generators on a computer. The probability density function of this variable is given by

$$p_X(x) = \begin{cases} 1 & \text{for } 0 \leq x \leq 1, \\ 0 & \text{otherwise.} \end{cases} \quad (\text{C.16})$$

We are seeking the probability density function $p_Y(y)$ of the random variable

$$Y = e^{-X^2}. \quad (\text{C.17})$$

The random number Y is **not** Gaussian distributed as we might think naively. To calculate the probability density function we can employ the cumulative density function eqn C.14 by noting that

$$P(Y \leq y) = P(e^{-X^2} \leq y) = P(X \geq \sqrt{-\ln y}). \quad (\text{C.18})$$

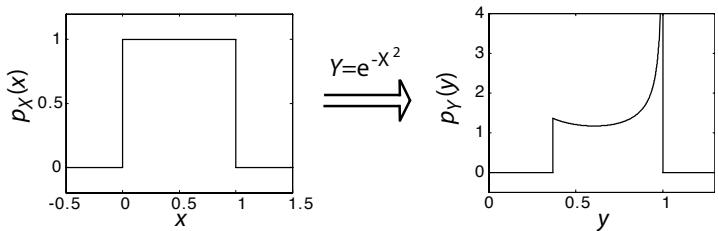
Thus, the cumulative probability function of Y can be calculated from the cumulative probability function of X ,

$$P(X \geq \sqrt{-\ln y}) = \begin{cases} \int_{\sqrt{-\ln y}}^1 p_X(x)dx = 1 - \sqrt{-\ln y} & \text{for } e^{-1} \leq y \leq 1, \\ 0 & \text{otherwise.} \end{cases} \quad (\text{C.19})$$

The probability density function of Y is the derivative of this function,

$$p_Y(y) = \begin{cases} 1 - \sqrt{-\ln y} & \text{for } e^{-1} \leq y \leq 1, \\ 0 & \text{otherwise.} \end{cases} \quad (\text{C.20})$$

The probability density functions of X and Y are shown below.



A special function of random variables, which is of particular interest it can approximate many processes in nature, is the sum of many random variables. For example, such a sum occurs if we calculate averages from measured quantities, that is,

$$\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i, \quad (\text{C.21})$$

and we are interested in the probability density function of such random variables. This function depends, of course, on the specific density function of the random variables X_i . However, there is an important observation summarized in the *central limit theorem* that we can often utilize in this book. This theorem states that the average (normalized sum) of n random variables that are drawn from any distribution with mean μ and variance σ is approximately normally distributed with mean μ and variance σ/n for a sufficiently large sample size n . The approximation is, in practice, often very good also for small sample sizes. For example, the normalized sum of only seven uniformly distributed pseudo-random numbers is often used as a pseudo-random number for a normal distribution.

C.6 Measuring the difference between distributions

An important practical consideration is how to measure the similarity of difference between two density functions, say the density function p and the density function q . Note that such a measure is a matter of definition, similar to distance measures of real numbers or functions discussed in Appendix A.2. However, a proper distance measure, d , should be zero if the items to be compared, a and b , are the same, its value should be positive otherwise, and a distance measure should be symmetrical, meaning that $d(a, b) = d(b, a)$. The following popular measure of similarity between two density functions is not symmetric and is hence not called a distance. It is called *Kulbach–Leibler divergence* and

is given by

$$d^{\text{KL}}(p, q) = \int p(x) \log\left(\frac{p(x)}{q(x)}\right) dx \quad (\text{C.22})$$

$$= \int p(x) \log(p(x)) dx - \int p(x) \log(q(x)) dx \quad (\text{C.23})$$

This measure is zero if $p = q$ since $\log(1) = 0$. This measure is related to the information gain or relative entropy in information theory as discussed in Appendix D.

C.6.1 Marginal, joined, and conditional distributions

When we have probability density functions of several variables, we can distinguish several cases. This is illustrated here in the case of two variables, while generalizations to more dimensions are straight forward. The most general information we can have about a process that depends on two random variables is the *joined distribution*,

$$p(x, y), \quad (\text{C.24})$$

which is a two dimensional function. The slice of this function, given the value of one variable, say y , is the *conditional distribution*,

$$p(x|y). \quad (\text{C.25})$$

If we add over all realizations of y we get the *marginal distribution*,

$$p(x) = \int p(x, y) dy. \quad (\text{C.26})$$

Two random variables x and y are called independent if

$$p(x, y) = p(x)p(y), \quad (\text{C.27})$$

which is the same as saying

$$p(x|y) = p(x). \quad (\text{C.28})$$

Note that we can decompose the joined distribution functions into the product of a conditional and a marginal distribution,

$$p(x, y) = p(x|y)p(y) = p(y|x)p(x), \quad (\text{C.29})$$

which is sometimes called the product rule. If we divide this equation by $p(x)$, we get the identity

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}, \quad (\text{C.30})$$

which is called *Bayes theorem*. This theorem is important because it tells us how to combine a *prior* knowledge, such as the expected distribution of stimuli, $P(s)$, with some evidence as measured by a *likelihood*, $P(\mathbf{r}|s)$, to get the *posterior* distribution $P(s|\mathbf{r})$ from which the optimal estimate (in a statistical sense) of the stimulus can be calculated with eqn 7.31. $P(r)$ is the proper normalization so that the left-hand side is again a probability.

Further reading

The book by Evans et al. (2000) is a useful collection of probability functions, but not an introduction to statistics. Wikipedia also has a good collection of probability functions.

Merran Evans, Nicholas Hastings, and Brian Peacock (2000), *Statistical distributions*, John Wiley & Sons, 3rd edition.

Basic information theory

D

Information theory, which is an important application area of probability theory, was originally invented to describe communication channels. It was brought to wide attention through an article by *Claude Shannon* in 1948. Information theory has been widely applied to brain theory. For example, some researchers have attempted to quantify the information in spike trains with formal methods of information theory, and the study of the ‘neural code’ was popular in the late 1990s. Concepts of information theory are used in this book at several places, including the discussion of coding in neural maps and the formulation of the anticipating brain. The following is a basic introduction to some of the constructs invented in this theory.

D.1	Communication channel and information gain	345
D.2	Entropy, the average information gain	348
D.3	Mutual information and channel capacity	353
D.4	Information and sparseness in the inferior-temporal cortex	354
D.5	Surprise	357
	Further reading	359

D.1 Communication channel and information gain

One can talk very casually about information and how information is transmitted by signals over telegraph wires or in neural systems. However, it is useful to define information more formally so that we can quantify the amount of information that is or can be transmitted in different systems. This was done in 1948 by *Claude Shannon* who studied the transmission of information over a general *communication channel* as shown in Fig. D.1. A message x_i , one of several possible messages of a set X from an information source, is converted into a signal $s_i = f(x_i)$ that can be transmitted over the communication channel to a receiver. The receiver converts the received signal r_i back to a message $y_i = g(r_i)$, one of a set Y of possible output messages, that can be interpreted by the destination receiver. Shannon included a noise input η in the channel that is intended to capture faulty transmissions, for example due to physical noise in the transmission channel or faulty message conversion. In the noiseless case the receiver receives the sent signal $r_i = s_i$, which can be converted into the original message when the receiver does the inverse function of the encoding done by the transmitter, that is, $g = f^{-1}$. Note that we can consider several noise models such as additive noise, for example, $r = s + \eta$, or multiplicative noise, for example, $r_i = s_i * \eta$, where η is a random variable.

What is the amount of information we gain when receiving a message y_i ? This depends highly on our expectations. For example, if I tell you that the first name of ‘Shannon’ is ‘Claude’ then you gain no information if you have read carefully the text above where it was already mentioned. If you thought it was either ‘Albert’ or ‘Claude’ you gain some information. You gain more information from my message if you thought it was one of four possible choices

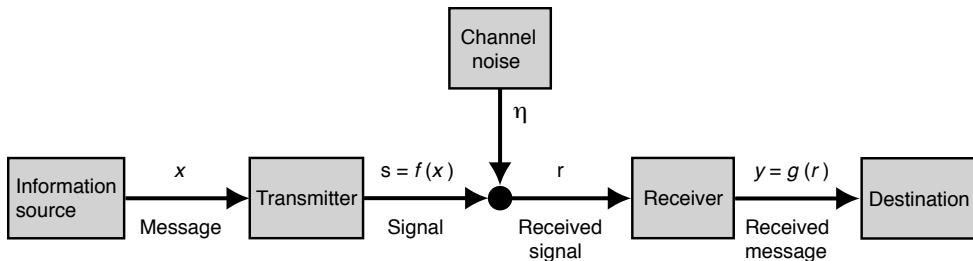


Fig. D.1 The communication channel as studied by Shannon. A message x is converted into a signal $s = f(x)$ by a transmitter that sends this signal subsequently to the receiver. The receiver generally receives a distorted signal r consisting of the sent signal s convoluted by noise η . The received signal is then converted into a message $y = g(r)$. [Adapted from Shannon, *The Bell System Technical Journal* 27: 379–423 (1948).]

with equal likelihood. Information depends on the set of possible messages and their frequencies, and a measure of information gain should therefore be a function of the probability $p_i = P(y_i)$ of a message. Furthermore, independent information should be additive, that is, if I tell you that the middle name of Claude Shannon is ‘Elwood’, then you gain a bit of information independent of and additional to the information I gave you previously. The probability of independent messages is the product of the probabilities of the individual messages. We thus need a function (of the probabilities) that has the property

$$f(xy) = f(x) + f(y). \quad (\text{D.1})$$

The logarithm has these characteristics, and we therefore define the information gain when receiving a message y_i as

$$I(y_i) = -\log_2(p_i). \quad (\text{D.2})$$

The minus sign makes the information positive as the probabilities are less than 1. Note that we use the logarithm with basis 2 and no additional proportionality constant, which is just a convention. This convention defines the units of one *bit*. If we have two possible states with equal likelihood, then the transmission of one state allows us to gain 1 bit of information.

Let us illustrate the concept further with information transmitted by hand signals as illustrated in Fig. D.2. In Fig. D.2A we illustrate a case where we use only the thumb in two possible locations, either horizontal or up, to transmit a message, say ‘0’ and ‘1’. If both cases are equally likely, that is, $P(0) = P(1) = 1/2$, then we receive an information gain of

$$I = -\log_2(0.5) = 1 \quad (\text{D.3})$$

bit of information. If we use two fingers with two possible locations and the code of the total number of fingers to transmit information, as illustrated in Fig. D.2B, then we can transmit three different messages. The gain of information from each signal is therefore $I = -\log_2(1/3) = 1.585$ bits with equally frequent signals. In the code illustrated in Fig. D.2B it does not matter whether the pointing finger is extended or if the thumb is up. In contrast, in Fig. D.2C

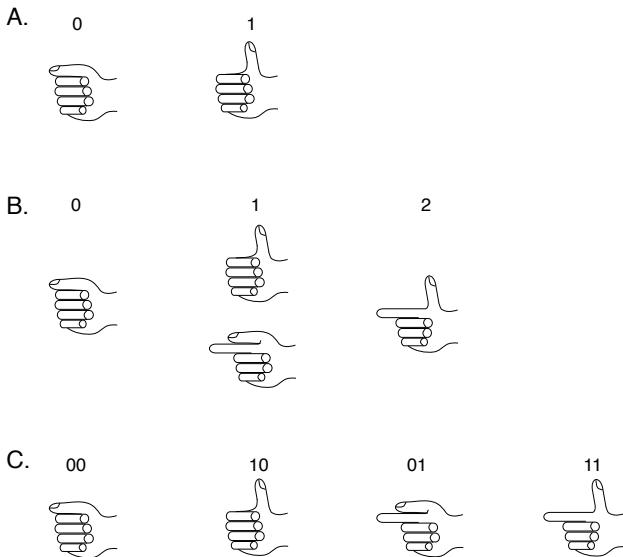


Fig. D.2 Example of hand signals used to transmit messages. (A) Only two positions of the thumb are used to encode two possible messages ‘0’ or ‘1’. (B) Two fingers are used to encode three messages by the total number of extended fingers. (C) Two fingers are used with a different code, in which the position of each of the two fingers used for the signalling is essential.

we have changed the code to transmit a message with two fingers where now the position of the finger matters. We can then transmit four different messages, and each equally likely signal would convey 2 bits of information.

We demonstrated the information gain with discrete events that had a certain probability distribution. Discrete events are also most appropriate when thinking about message transmission with a finite alphabet. However, it is mathematically often more convenient to write the following formulations in continuous form. The information gain is thus often written in the form

$$I(x) = -\log_2 P(x). \quad (\text{D.4})$$

Note that this formulation can still be used for discrete events. The probability function $P(x)$ then has non-vanishing values only for a set of discrete values x . If we consider a continuous set X of events x then we have to take the corresponding probability density function $p(x)$ into account (see also Appendix C). The probability of an event x is then an infinitesimally small quantity $P(x) = p(x)dx$, and the information gain for one of those events goes to infinity. However, averages of continuous sets are still defined as we will see in the following, and it is easy to replace integrals with sums (and vice versa) in the formulas below.

We have so far only considered the cases where a message conveyed a definitive statement. In contrast, if I tell you that the first name of Shannon is ‘Claude’ with 90% probability out of four possibilities, then there is still some uncertainty left which we have to take into account. We have then to distinguish the probability of an event before the message was sent, called the *a priori* or *prior* probability $P^{\text{prior}}(x)$, and the probability of an event after taking the information into account, which is called the *posterior* probability

$P^{\text{posterior}}(x)$. The information gain is then defined by

$$I(x) = - \sum_x P^{\text{posterior}}(x) \log_2 \frac{P^{\text{prior}}(x)}{P^{\text{posterior}}(x)}. \quad (\text{D.5})$$

In the previous examples we knew the precise answer after the message was received, so that $P^{\text{posterior}}(x) = 1$. We also wrote the prior probability simply as $P^{\text{prior}}(x) = P(x)$. Equation D.4 is therefore only a special case of the more general formula, eqn D.5.

D.2 Entropy, the average information gain

We have used equally likely signals in the previous numerical examples. It is, however, often the case that not all messages are equally likely, and the amount of information gain is therefore different for different messages. For example, let's go back to the example illustrated in Fig. D.2A and consider the case where the thumb is up in 3/4 of all messages sent out and only horizontal in 1/4 of the cases. When we receive a signal of thumb horizontal, then we gain $I(0) = -\log_2(1/4) = 2$ bits of information, whereas the signal of 'thumb up' conveys only $I(1) = -\log_2(3/4) = 0.415$ bits of information. The average amount of information in the message set of the information source is

$$S(X) = - \sum_i p_i \log_2(p_i), \quad (\text{D.6})$$

which is called the *entropy* of the information source. The entropy is a quantity of the message set and is not defined for an individual message. The entropy of the set of possible received messages $S(Y)$ is defined in a similar way. The average amount of information gained by the transmission of a signal in the previous example is $S = 1/4*2 + 3/4*0.415 = 0.8113$. The entropy is a measure of the average information gain we can expect from a signal in a communication channel, though it is good to keep in mind that individual messages can have larger and smaller information gain values. Some rare events might indeed convey a very large amount of information.

The entropy of a message set with N possible messages that are all equally likely is

$$S(X) = - \sum_{i=1}^N \frac{1}{N} \log_2 \left(\frac{1}{N} \right) = \log_2(N). \quad (\text{D.7})$$

The entropy is hence equivalent to the logarithm of the number of possible states for equally likely states. This is a useful fact to keep in mind. More generally, the entropy measures the effective number of states by weighting the logarithm of the number of states with the frequency of their occurrences.

We can formulate the entropy for continuous distributions in an analogous way. We again write the set with capital letters, for example, X , and examples from this set with lower case letters, for example, x . The entropy of a set X with probability density function $p(x)$ is then defined as

$$S(X) = - \int_X p(x) \log_2 p(x) dx. \quad (\text{D.8})$$

For example, the entropy of a Gaussian-distributed set with mean μ and variance σ^2 ,

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (\text{D.9})$$

is given by

$$S = \frac{1}{2} \log_2(2\pi e \sigma^2). \quad (\text{D.10})$$

This entropy does depends on the variance, not the mean. This is intuitively clear as the average information gain is only a relative quantity and information is transmitted by variations of a signal around its mean value.

D.2.1 Difficulties in measuring entropy

Measuring entropies is a data hungry task. This comes partly from the fact that we have to estimate probability distributions. To do this we have to observe many instances, from which we can, for example, construct a histogram. The histogram gives us the numbers of the frequencies with which events occur within a certain resolution determined by the bin size of the histogram. We can often get some reasonable estimations of the probability function if we have at least some prior knowledge about the form of the probability function. We can then fit the histograms with the expected function to fix the unknown parameters to values consistent with the experimental observations. Often, however, we do not know *a priori* the form of the distribution functions, and suitable guesses from the data are common. We have then to check that the results do not depend critically on the choice of distribution function.

A further difficulty is that information depends on the logarithm of probabilities. This means that events with small probabilities have a large factor in the entropy, and reliable measurements of rare events do demand a large amount of representative data. To get probability distributions from frequency histograms we have to normalize the histograms so that the sum over all data becomes one (see Appendix C),

$$\int dx P(x) = 1. \quad (\text{D.11})$$

Due to this normalization procedure we tend to overestimate the information content of the events measured if we miss out some rare events with high information content. Estimations of entropy therefore tend to overestimate the true entropy. This is a potential danger for suitable interpretations of experimental results. For example, a large entropy per spike, much larger than, for example, 2 bits, would rule out rate codes and would therefore indicate that temporal structures in the spike trains are used to transmit information. An overestimation of entropies from experimental data can obscure such conclusions. Some methods have been developed to compensate for such systematic shifts of small data sets, but such methods cannot, of course, completely compensate for real data.

D.2.2 Entropy of a spike train with temporal coding

It is very instructive, and important for the comparisons with experimental data outlined below, to calculate the maximum entropy of a spike train with temporal coding up to a certain precision. To do this we introduce small time bins Δt , small enough that one time bin can contain at most one spike. We can then view the spike train as a binary string with 0 in a time bin corresponding to no spike in this time interval and 1 in a time bin in which a spike has occurred. An example is illustrated in Fig. D.3. With small time bins we can assume a binary string with many more 0s than 1s, indeed many more than illustrated in the figure. We will also assume that the events in the bins are independent. Dependence would lower the entropy as mentioned before, and we are interested here in estimating the maximum information carried in a spike train.

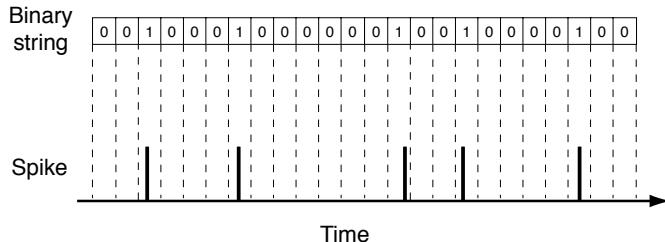


Fig. D.3 Representation of a spike train as a binary string with time resolution Δt .

The firing rate r of the spike train fixes the number of 1s and 0s in the spike train of length T . We can then calculate the number of possible spike patterns with the fixed number of spikes in the sequence of length T . The logarithm to base 2 of this number is the entropy of the spike train. This can thus be calculated to be

$$S = -\frac{T}{\Delta t \ln 2} [r\Delta t \ln(r\Delta t) + (1-r\Delta t) \ln(1-r\Delta t)]. \quad (\text{D.12})$$

For time bins much smaller than the inverse firing rate, that is, $\Delta t \ll 1/r$, this is approximately equal to

$$S \approx Tr \log_2 \left(\frac{e}{r\Delta t} \right). \quad (\text{D.13})$$

The term $N = Tr$ is the number of spikes in the spike train, so that the average entropy per spike is given by

$$\frac{S}{N} \approx \log_2 \left(\frac{e}{r\Delta t} \right). \quad (\text{D.14})$$

This is the maximum entropy per spike of a spike train for a given timing precision Δt when we can assign to each individual spike pattern a unique message or sensory event. It is maximum in the sense that we considered only the noiseless case, and noise would diminish the information gain of each message.

The maximum entropy per spike is plotted for three different firing rates in Fig. D.4A. What is remarkable is that the entropy per spike is for the parameters shown always larger than one bit. The reason for this is that the interspike

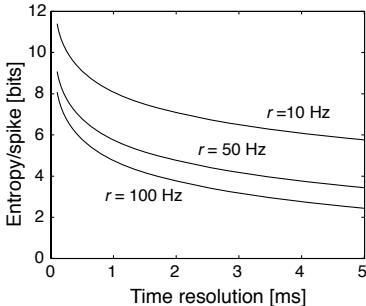
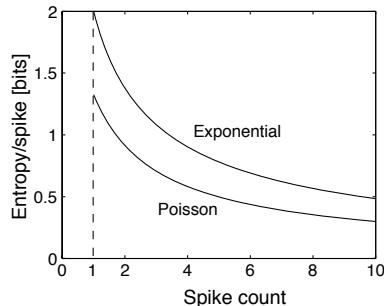
A. Maximum entropy with spike code**B. Maximum entropy with rate code**

Fig. D.4 (A) Maximum entropy per spike for spike trains with different firing rates as a function of the size of the time bins, which sets the resolution of the firing times. (B) Maximum entropy per spike for spike trains from which the information is transmitted with a rate code. The two different curves correspond to two different spike statistics of the spike train, a Poisson and an exponential probability of spike counts. Spike trains with exponential spike distributions can convey the maximum information with a rate code for fixed variance.

intervals are long so that the occurrence of a spike becomes highly relevant. Also, we have normalized the entropy to the number of spikes, which can be a bit misleading; the absence of a spike can also bear information, and the measure of the entropy takes this into account. We have only normalized the spike train entropy to the total number of spikes to make it easier to compare the entropies of different spike trains and coding schemes.

Figure D.4A illustrates that the maximal entropy decreases with increasing precision interval Δt (decreasing time precision). This is because the number of patterns that can be represented by the spike train decreases with fewer time intervals in a spike train of length T . The entropy per spike also decreases with increasing firing rate because the individual spike is then not so surprising and bears less information. We should also keep in mind that the transmission of the information contained in a spike train is lowered by noise in the transmission process and by spike correlations. However, the maximum entropy as calculated above gives us the scale to which we can compare measurements of information transmission in the brain.

D.2.3 Entropy of a rate code

We can compare the entropy of a spike train above, in which we considered all possible spike patterns (which is therefore a time code), with other coding schemes that do not employ all possible spike patterns, such as a rate code. In the rate code only the number of spikes n in a certain interval T is relevant. The entropy of observing N spikes in a time interval T can be calculated by applying the definition of the entropy,

$$\begin{aligned} S(N; T) &= - \sum_n P(n) \log_2 P(n) \\ &= -\frac{1}{\ln 2} \sum_n P(n) \ln P(n) \end{aligned} \quad (\text{D.15})$$

where $P(n)$ is the probability of observing n spikes in the time interval T . For example, let us calculate the entropy for a *Poisson spike train* of length T . The probability of observing n spikes, when we expect $N = rT$ spikes, is given by

the Poisson distribution,

$$P(n) = \frac{N^n e^{-N}}{n!}. \quad (\text{D.16})$$

Inserting this probability into the log part of formula D.15 we get

$$S(N; T) = -\frac{1}{\ln 2} \sum_n P(n)(n \ln N - N - \ln(n!)). \quad (\text{D.17})$$

We can approximate the last term with Stirling's formula

$$\ln(n!) \approx (n + \frac{1}{2}) \ln n - n + \frac{1}{2} \ln(2\pi). \quad (\text{D.18})$$

At this point we should remember the definition of the expectation value $E(x)$ of a quantity x ,

$$E(x) = - \sum_x P(x)x, \quad (\text{D.19})$$

and the expectation value of n is just $E(n) = N$. The entropy of a Poisson spike train with rate code is therefore

$$S(N; T) \approx \frac{1}{2} \log_2 N - \frac{1}{2} \log_2 2\pi. \quad (\text{D.20})$$

We divide this value again by the number of spikes so that we get values for the entropy per spike. This is plotted in Fig. D.4B. The entropy of the rate code is, of course, much smaller than the entropy of a spike train in which we allowed the precise spike timing to be relevant. The length of time over which we have to integrate to achieve a particular spike count depends on the firing rate. To get a spike count of 2 we have to integrate on average for 40 ms if a neuron fires with mean firing rate of 50 Hz.

We calculated the entropy of a spike train with rate coding for the example of a Poisson spike train. Spike trains with other probability distributions of spike counts can have different entropies within the rate code. We can ask what would be the distribution of the spike count with fixed mean and variance that would result in the maximum entropy. The answer is that the entropy under these conditions is maximized by an exponential distribution. An exponentially distributed spike train

$$P(n) \propto e^{-\lambda n} \quad (\text{D.21})$$

with average firing rate $r = 1/[T(\exp(\lambda) - 1)]$ has an entropy of

$$S(N; T) = \log_2(1 + N) + N \log_2(1 + \frac{1}{N}). \quad (\text{D.22})$$

The entropy per spike for this distribution is also shown in Fig. D.4B. We can see that the entropy of such spike trains is moderately larger than the entropy for the Poisson spike train and reaches a value of precisely 2 bits/spike in the limit where a single spike carries all the information. The average information in a rate code can therefore be reasonably high so that the information that flows through the brain even with this code can be enormous.

D.3 Mutual information and channel capacity

So far we have only considered noise-free signal transmission and receivers that invert precisely the encoding of the transmitter ($g = f^{-1}$). This corresponds to perfect transmission and unambiguous decoding. However, in all practical applications of information transmission, including information transmission in neural systems, we must take noise into account. We have then to distinguish the received signal y from the sending signal x . The question we are most interested in is what a received signal y can tell us about an event x . Indeed, we will later attempt to reconstruct an event, such as a sensory input to the neural system, from the firing pattern of neurons. For now we want to ask, what the average information gain is from what a message y tells us about the set of events that could happen. We therefore have to consider conditional probabilities such as $P(x|y)$ (or corresponding density functions). The average information gain with such conditional probabilities over both sets is

$$\int_y P(y)S(X|y)dy = \int_X \int_Y p(y)p(x|y) \log_2 \frac{p(x|y)}{p(x)} dx dy. \quad (\text{D.23})$$

We can express the conditional probabilities with joint distributions,

$$P(x|y) = \frac{P(x,y)}{P(y)}, \quad (\text{D.24})$$

so that we can also write eqn D.23 as

$$I^{\text{mutual}} = \int_X \int_Y p(x,y) \log_2 \frac{p(x,y)}{p(x)p(y)} dx dy. \quad (\text{D.25})$$

This quantity is called the *cross-entropy* or *mutual information*. It is symmetrical in the arguments (which motivated the term mutual) and describes the average amount of information that can be gained by receiving a message y when a signal x is sent (or vice versa). We can also rewrite the mutual information as difference between the sum of the individual entropies $S(X)$ and $S(Y)$ and the entropy of the joint distributions $S(X,Y)$,

$$I^{\text{mutual}}(X,Y) = S(X) + S(Y) - S(X,Y). \quad (\text{D.26})$$

The joint entropy in the case of independent events in X and Y is $S(X,Y) = S(X) + S(Y)$. The mutual information in this case is equal to zero,

$$I^{\text{mutual}}(\text{all } y \in Y \text{ independent of all } x \in X) = 0, \quad (\text{D.27})$$

which reflects the fact that received messages are independent of sent messages. However, for information transmission we want messages such that the received message and the sent message are dependent. Mutual information not equal to zero then reflects some correlation of the events in X and Y . The mutual information in a communication channel describes the average amount of information we can expect to flow between input and output events.

An amount of information that can be transmitted by a change in the signal is proportional to the logarithm of the number of states the signal can have.

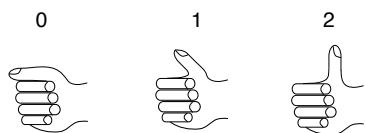


Fig. D.5 Example of hand signals using the thumb in three different states to transmit messages.

For example, in Fig. D.2A we have considered only two allowed positions of the thumb and could therefore send one bit of information in each time interval in which the thumb can change. We can increase the number of states the thumb can take as illustrated in Fig. D.5. There we allowed for three positions ($0^\circ, 45^\circ, 90^\circ$) and we could therefore transmit $I = \log_2 1/3 \approx 1.6$ bit of information at every time step with equally likely messages. We could increase the number of thumb states further until it will be difficult for the observer to distinguish the differences. There is therefore a limiting factor given by the resolution of our visual system and the noise introduced by the motor system of the sender. It is intuitively clear that the amount of information that can be transmitted over a noisy communication channel is limited by the ratio of the signal to noise or, to be more precise, by the ratios of the variance of these quantities, as the information is conveyed by changes in signals (see eqn D.10).

We can calculate the mutual information of a channel with noise for particular noise models and probability distributions for the input signal and the noise. It is common to consider additive Gaussian noise as well as Gaussian-distributed input signals. It turns out that the amount of information transmitted in such a Gaussian communication channel is the best we can achieve. The amount of information that can be transmitted through a Gaussian channel is called the *channel capacity*. The information we can transmit through a noisy channel is therefore always less than the channel capacity, which is given by

$$I \leq \frac{1}{2} \log_2 \left(1 + \frac{\langle x^2 \rangle}{\langle \eta_{\text{eff}}^2 \rangle} \right), \quad (\text{D.28})$$

where $\langle x^2 \rangle$ is the variance of the input signal and $\langle \eta_{\text{eff}}^2 \rangle$ is the effective variance of the channel noise, considering the noise as transmitted itself as signal through the channel. The ratio $\langle x^2 \rangle / \langle \eta_{\text{eff}}^2 \rangle$ is called the *signal to noise ratio* (SNR). It is interesting to note that the information transmission with given noise in the transmission channel can be increased by increasing the variability of the input signals. It seems that the high variability of spike trains, as discussed in Chapter 3, is therefore particularly suited for transmission in noisy neural systems. There are indeed several possible mechanisms that can increase the variability of neuronal response, some of which are discussed in Chapter 8.

D.4 Information and sparseness in the inferior-temporal cortex

D.4.1 Population information

To estimate the information content in a set of neurons, one has to record from several cells in an area simultaneously if the responses of the population neurons are not independent of each other. If, on the other hand, we assume that each neuron responds with a certain independent probability distribution to each stimulus, then we can use neurons that have been recorded independently. *Edmund Rolls* and colleagues have used such independent recordings from neurons in the inferior-temporal (IT) cortex to estimate the information in a population of neurons. The authors used neurons that responded to a set

of 20 faces and used a statistical estimate for the decoding by fitting a Gaussian to the fluctuations of the neuronal response of each neuron. The results of the mutual information between the 20 face stimuli and the response of C recorded neurons is shown in Fig. D.6A with star symbols. The information quickly rises with the number of neurons taken into account in the analysis until the information gain levels off.

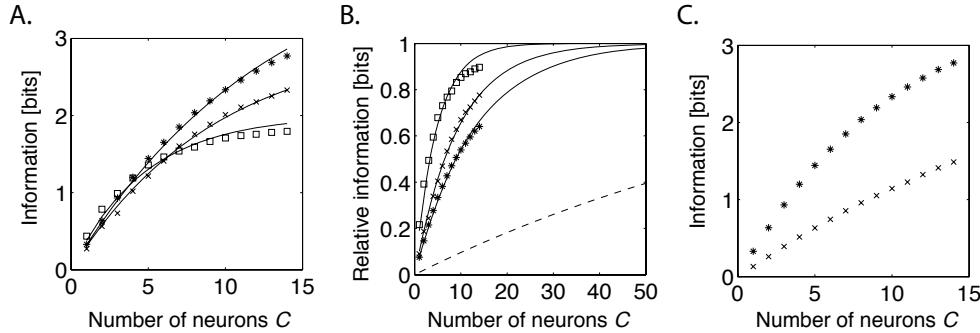


Fig. D.6 (A) Estimate of mutual information between face stimuli and firing rate responses of C cells in the inferior-temporal cortex. The set of stimuli consisted of 20 faces (stars), 8 faces (crosses), and 4 faces (squares). (B) The information in the population of cells relative to the number of stimuli in the stimulus set. The solid lines are fits of the cell data to the ceiling model (eqn D.29). The dashed line illustrates the values of the ceiling model for a stimulus set of 10,000 items and $y = 0.01$. (C) Estimate of mutual information with 20 faces when the neuronal response is derived from the spike count in 500 ms (stars) and 50 ms (crosses). [Data from Rolls, Treves, and Tovee, *Experimental Brain Research* 114: 149–62 (1997).]

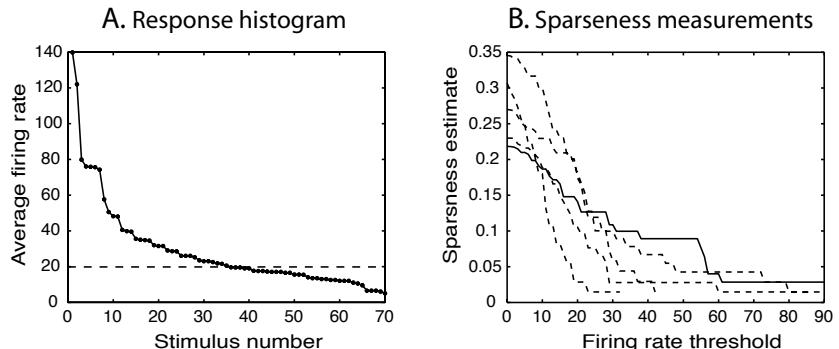
The information for a stimulus set of 20 faces can be compared with the responses of the neurons to a reduced set of faces. The information, when analysed with a set of 8 faces (crosses) and a set of 4 faces (squares), is also shown in Fig. D.6A. We see that the levelling off of the curves depends on the size of the stimulus set, and the linear regime of the scaling extends further with increasing size of the stimulus set. The deviation from the linear envelope for larger numbers of neurons used for decoding can be expected because the entropy of a set of N equally likely objects is given by $S_{\max} = \log_2 N$. The entropy for a fixed size of the stimulus set cannot therefore continue to grow. The deviation of the linear scaling is therefore a *ceiling effect*. The data points are indeed well fitted with a simple empirical model describing the ceiling effect by

$$I(C) = S_{\max}(1 - (1 - y)^C), \quad (\text{D.29})$$

where the parameter y is the fraction of information carried by each neuron. The fits reveal a decreasing fraction y of information carried by each neuron with increasing size of the stimulus set. This can be expected as each new stimulus that has to be represented by a neuron diminishes the ability to discriminate among of the other stimuli represented by the neuron.

On the basis of the simple ceiling effect model we can also plot the data of Fig. D.6A relative to the maximum entropy of the stimulus set. This is shown in Fig. D.6B. From this it seems that only a small number of neurons are necessary to accurately represent a stimulus set. This conclusion is, however,

Fig. D.7 (A) Average firing rates of a neuron in the IT cortex in response to each individual stimulus in a set of 70 stimuli (faces and objects). The responses are sorted in descending order. The horizontal dashed line indicates the spontaneous firing rate of this neuron. (B) The sparseness derived from five IT neurons calculated from eqn 7.27 in which responses lower than the firing rate threshold above the spontaneous firing rate are ignored. The solid line corresponds to the data shown in (A). [Data from Rolls and Tovee, *Journal of Physiology* 73: 713–26 (1995).]



obscured by the use of small stimuli sets in the experiments. If we extrapolate the data using the ceiling model we find that much larger numbers of neurons are necessary to gain sufficient information about a presented stimulus. An example of a set size of 10,000 items with $y = 0.01$ is shown as a dashed line in Fig. D.6B. Larger set sizes with smaller y value would result in an even smaller increase of the information gain with the number of cells in the population. Also, the information gains in Fig. D.6A and B are overestimates if we think that the amount of information in shorter time intervals is relevant. In the experiments a 500 ms time interval following the stimulus was used to estimate the information gain. The estimates for a smaller time interval (50 ms) are shown in Fig. D.6C as crosses and compared to the information gain estimated with a 500 ms time interval (stars) from the previous analysis. The information gain therefore increases only slowly with the number of neurons in the population, which suggests a distributed representation of information.

D.4.2 Sparseness of object representations

We still have not answered the question of how sparse the representation in the IT cortex is. To do this we use the measured responses of individual neurons directly as input to eqn 7.27. The average firing rate of one particular IT neuron, averaged over many trials, in response to 70 individual stimuli is shown in Fig. D.7A. The responses are thereby ordered so that the stimuli with the largest responses are given a small stimulus number. We indicate the spontaneous firing rate of this neuron, against which we have to judge the response of the neuron, as a horizontal dashed line. The individual average firing rate response of a neuron to some stimuli are markedly different from the spontaneous firing of the neuron, whereas most such differences are only minor. If all such differences are relevant, then we see that the neuron responded to practically all stimuli so that we have to conclude that a very distributed code is used. However, if only large responses are relevant, then we would conclude a more sparse representation because only a small number of stimuli drive the cell with large responses.

To take the relevance in the firing rate into account we include in the following a firing rate threshold and count only cell responses above this threshold as

relevant. The relevant firing rate of the neuron for a given stimulus is therefore the mean firing rate for responses above the threshold minus the spontaneous firing rate, and is zero for firing rates below the threshold. These relevant firing rates are used in eqn 7.27 to estimate the sparseness. The sparseness estimate from data from five different neurons as a function of the firing rate threshold are shown in Fig. D.7B. The solid line corresponds to the cell shown in Fig. D.7A. All cells indicate fairly small values of sparseness. A value of around 0.3 is only reached if we also take small deviations of the spontaneous firing rate into account as the relevant signal. The estimated sparseness, however, becomes much smaller if we take only high firing rates into account. It is also likely that this estimation of the sparseness in IT responses is an overestimation of the sparseness that we would get for much larger stimuli sets. The data set used in the experiments had several images that often drive responses in IT cells. The reported results indicate that sparsely distributed representations, roughly on the order of $a = 0.1$, may be employed in the brain.

D.5 Surprise

The above section reviewed classical concepts of information theory, in particular the average information (entropy) of different codes. When thinking about brain processing in such terms one must recognize that we are talking about a large number of possible states of the environment and the brain itself. Many of the concepts from classical information theory were defined for finite and even small communication codes. Also, it is useful to define information-theoretic measures about the average information gain of specific signals.

To define such a measure, we consider a system with a set of responses, r . Our knowledge about the system can be characterized by the *prior*, $P(r)$, for example by building a model that produces the characteristic distribution of responses. We then probe the system with a specific signal, s , and measure the responses to this specific signal. We can then update our knowledge of the system, which is the *posterior*, $P(r|s)$. If the response of the system is as expected, than we are not *surprised* about the response to this input and there is no need to update our model of the system. The posterior is then equal to the prior. However, the difference of the posterior is large for responses that surprise us. We can turn this observation around and define the *surprise* as the difference between the posterior and prior distribution. A common measure to quantify this difference is the Kulbach–Leibler divergence (see Appendix C.6), and the surprise is hence defined as

$$I^S(s) = \int_R P(r|s) \log_2 \frac{P(r|s)}{P(r)} dr. \quad (\text{D.30})$$

Another way of thinking about this measure is to write it as a difference

$$I^S(s) = - \int_R P(r|s) (\log_2 P(r) - \log_2 P(r|s)) dr. \quad (\text{D.31})$$

The first term, $I(s) = \int_R P(r|s) \log_2 P(r) dr$, is the total amount of information associated with each stimulus. However, not all the information is about the

stimulus since some of the responses can also be expected generally from the system. We thus need to subtract the information not related to the stimulus, which can be measured from the variations when the stimulus is held fixed, $I(R|s) = \int_R P(r|s) \log_2 P(r|s) dr$. This quantity defined as surprise looks like an information measure. However, this measure is based on conditional probabilities so that this quantity does not necessarily have the additivity feature that we demand from information measures.

Examples for the stimulus-dependent surprise per spike are shown in Fig. D.8 for two visual neurons in the inferior-temporal cortex that respond to faces. The average firing rate of the responding neurons shown in Fig. D.8A was 54 Hz. Responses with rates around this firing rate bear little surprise value. Some responses with high firing rates for this neuron carried high surprise values of around 2 bits per spike. However, the average surprise value in the responses of this neuron to all 20 faces in the stimulus set was only 0.55 bits. This value is not surprisingly high and does not suggest the need for a code within the firing pattern of these neurons. Another example is shown in Fig. D.8B. The surprise values are again lowest around the mean firing rates of the neuron, which is higher than that of the neuron shown on the left. This neuron carried the largest surprise values for small firing rates and carried an average surprise value similar to that in the previous example.

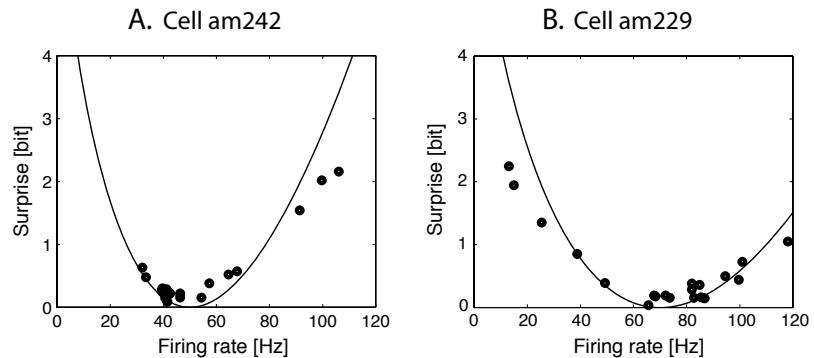


Fig. D.8 Stimulus-dependent surprise for two cells in the inferior-temporal cortex that respond to faces. The curve is 5 times the universal expectation for small time windows (eqn D.32). [Data from Rolls, Treves, Tovee, and Panzeri, *Journal of Computational Neuroscience* 4: 309–33 (1997).]

The values for the surprise rate were calculated for 500 ms spike trains during the presentation of the faces. Several studies have also shown that much of the information about a stimulus is transmitted in much smaller time intervals. For very small time intervals, so that at most one spike can occur during this time, there is not much freedom for a coding scheme, so that we can expect a universal curve, which is given by

$$I^s(\Delta t \ll 1) = r^s \log_2 \left(\frac{r^s}{r} \right) + \frac{r - r^s}{\ln 2}, \quad (\text{D.32})$$

where r^s is the specific response rate for stimulus s , and r is the average firing rate over all stimuli. This universal surprise curve, divided by the average firing rate r and multiplied by a factor of 5, is plotted in Fig. D.8 as a solid line. We scaled the curve up because the experimental data were obtained in large time windows for which the values can be higher. What is striking, however, is that

the experimental data follow approximately the universal curve for small time windows. This is another hint that elaborate codes of spiking pattern might not be employed by neurons in the central nervous system.

A related formulation of surprise in a Bayesian view was proposed by *Pierre Baldi*, who also applied it to neuroscientific research with his colleague *Laurrent Itti*. In a Bayesian view, we consider a distribution of possible models that can explain the data. A new example, \mathbf{x} , can be used to update our world view by refining the distribution of models that can explain the data. Thus, we consider distributions of possible models, $P(\theta)$, where different models are specified with different parameters θ . The Bayesian surprise is defined as the change between the prior and posterior distribution of models,

$$S(\mathbf{x}) = \int_{\theta} P(\theta|\mathbf{x}) \log_2 \frac{P(\theta|\mathbf{x})}{P(\theta)} d\theta. \quad (\text{D.33})$$

New evidence is surprising when not anticipated by the system, and surprising data should thus trigger learning as discussed further in Chapter 10.

Further reading

The (two) classical articles by Claude Shannon (1948) are worth reading. A well known and wonderfully written book by Rieke *et al.* (1996) explores neural codes in nervous systems in much more detail and explains information theory on the way. The article by Allesandro Treves (2001) gives a good overview of research about the ‘neural code’ and is even recommended as an addition to the above book. The book chapter by Pierre Balidi also introduces surprise and is worth exploring, and his web page contains further papers where his concept of surprise is studied in neuroscience.

Claude Shannon (1948), *A mathematical theory*

of communication in *Bell System Technical Journal*, 27.

Fred Rieke, David Warland, Rob de Ruyter van Steveninck, and William Bialek (1996), *Spikes: Exploring the neural code*, MIT Press.

Allesandro Treves (2001), *Information coding in higher sensory and memory areas*, in *Handbook of biological physics IV*, Stan Gielen (ed.), Elsevier.

Pierre Baldi (2002), *A computational theory of surprise*, in *Information, coding, and mathematics*, M. Blaum (ed.), 1–25, Kluwer.

This page intentionally left blank

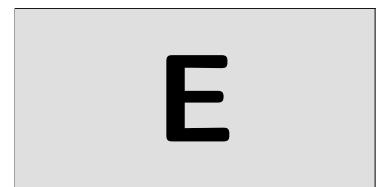
A brief introduction to MATLAB

This brief MATLAB tutorial assumes little programming experience, although experienced might want to look through it when switching from another programming language. MATLAB is an interactive programming environment for scientific computing. It is very convenient for several reasons, including its interpreted execution mode, which allows fast and interactive program development, advanced graphic routines, which allow easy and versatile visualization of results, a large collection of predefined routines and algorithms, which makes it unnecessary to implement known algorithms from scratch, and the use of matrix notations, which allows a compact and efficient implementation of mathematical equations. MATLAB stands for matrix laboratory, which emphasizes the fact that most operations are array or matrix oriented. Similar programming environments are provided by the open source systems called *Scilab* and *Octave*. The Octave system seems to emphasize syntactic compatibility with MATLAB, while Scilab is a fully fledged alternative to MATLAB with similar interactive tools. While the syntax and names of some routines in Scilab are sometimes slightly different, the distribution includes a converter for MATLAB programs. We concentrate here on the introduction to MATLAB and outline briefly the conversion of the programs in this book to the freely available cousin systems at the end of this appendix.

E.1 The MATLAB programming environment

MATLAB¹ is a programming environment and collection of tools to write programs, execute them, and visualize results. MATLAB has to be installed on your computer to run the programs in this book. It is commercially available for many computer systems, including Windows, Mac, and UNIX systems. The MATLAB web page includes a set of brief tutorial videos, also accessible from the *demos* link from the MATLAB desktop, which are highly recommended for learning MATLAB.

As already mentioned, there are several reasons why MATLAB is easy to use and appropriate for our programming need. MATLAB is an interpreted language, which means that commands can be executed directly by an interpreter program. This makes the time-consuming compilation steps of other programming languages redundant and allows a more interactive working mode. A disadvantage of this operational mode is that the programs could be less efficient compared to compiled programs. However, there are two possible solution



E.1 The MATLAB programming environment	361
E.2 A first project: modeling the world	371
E.3 Octave	373
E.4 Scilab	375
Further reading	377

¹ MATLAB and Simulink are registered trademarks, and MATLAB Compiler is a trademark of The MathWorks, Inc.

to this problem in case efficiency become a concern. The first is that the implementations of many MATLAB functions is very efficient and are themselves pre-compiled. MATLAB functions, specifically when used on whole matrices, can therefore outperform less well-designed compiled code. It is thus recommended to use matrix notations instead of explicit component-wise operations whenever possible. A second possible solutions to increase the performance is to use the MATLAB compiler to either produce compiled MATLAB code in `.mex` files or to translate MATLAB programs into compilable language such as C.

A further advantage of MATLAB is that the programming syntax supports matrix notations. This makes the code very compact and comparable to the mathematical notations used in the book. MATLAB code is even useful as compact notation to describe algorithms, and it is hence useful to go through the MATLAB code in this book even when not running the programs in the MATLAB environment. Furthermore, MATLAB has very powerful visualization routines, and the new versions of MATLAB include tools for documentation and publishing of codes and results. Finally, MATLAB includes implementations of many mathematical and scientific methods on which we can base our programs. For example, MATLAB includes many functions and algorithms for linear algebra and to solve systems of differential equations. Specialized collections of functions and algorithms, called a ‘toolbox’ in MATLAB, can be purchased in addition to the basic MATLAB package or imported from third parties, including many freely available programs and tools published by researchers. For example, the MATLAB Neural Network Toolbox incorporates functions for building and analysing standard neural networks. This toolbox covers many algorithms particularly suitable for connectionist modelling and neural network applications, including advanced back-propagation algorithms mentioned in Chapter 6. A similar toolbox, called NETLAB, is freely available and contains many advanced machine learning methods. We do not use these toolboxes in this book, and only use very few examples with functions from other toolboxes. Instead, we concentrate in this book on illustrating simulations with minimal code based on basic MATLAB commands and functions.

E.1.1 Starting a MATLAB session

Starting MATLAB opens the MATLAB desktop as shown in Fig. E.1 for MATLAB version 7. All of the basic features used in this book can be found in earlier versions of MATLAB, while later versions are enhanced in terms of the user interfaces and some added functionalities. The MATLAB desktop is comprised of several windows which can be customized or undocked (moving them into an own window). A list of these tools are available under the *desktop menu*, and includes tools such as the *command window*, *editor*, *workspace*, etc. We will use some of these tools later, but for now we only need the *MATLAB command window*. We can thus close the other windows if they are open (such as the *launch pad* or the *current directory window*); we can always get them back from the *desktop menu*. Alternatively, we can undock the command window by clicking the arrow button on the command window bar. An undocked command window is illustrated on the left in Fig. E.2. Older versions of MATLAB

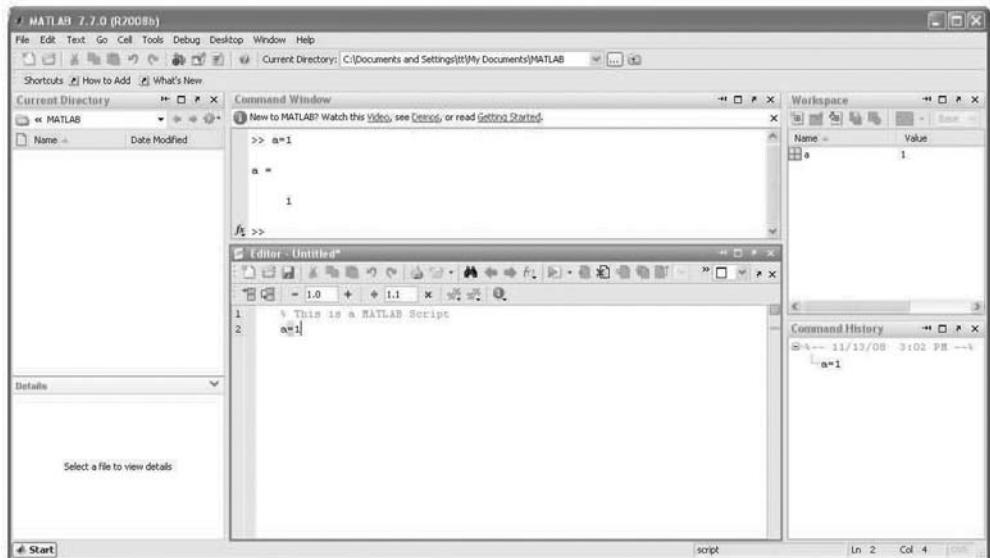


Fig. E.1 The MATLAB *desktop window* of MATLAB Version 7.

start directly with a command window or simply with a MATLAB command prompt `>>` in a standard system window. The command window is our control centre for accessing the essential MATLAB functionalities.

E.1.2 Basic variables in MATLAB

The MATLAB programming environment is interactive in that all commands can be typed into the command window after the command prompt (see Fig. E.2). The commands are interpreted directly, and the result is returned to (and displayed in) the command window. For example, a variable is created and assigned a value with the `=` operator, such as

```
>> a=3
```

```
a =
```

```
3
```

Ending a command with semicolon (`;`) suppresses the printing of the result on screen. It is therefore generally included in our programs unless we want to view some intermediate results. All text after a percentage sign (`%`) is not interpreted and thus treated as comment,

```
>> b='Hello World!'; % delete the semicolon to echo Hello World!
```

This example also demonstrates that the type of a variable, such as being an integer, a real number, or a string, is determined by the values assigned to the

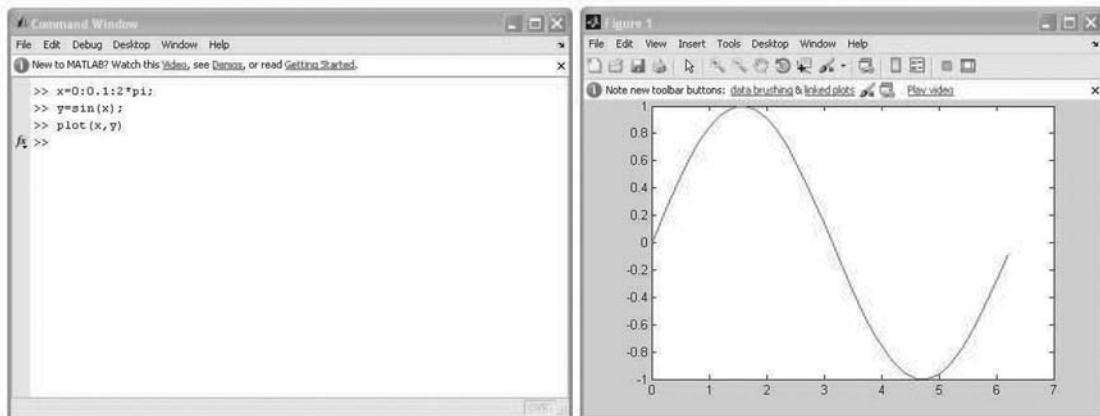


Fig. E.2 A MATLAB *command window* (left) and a MATLAB *figure window* (right) displaying the results of the function `plot_sin` developed in the text.

²While dynamic typing is convenient, a disadvantage is that a mistyped variable name can not be detected by the compiler. Inspecting the list of created variables is thus a useful step for debugging.

elements. This is called *dynamic typing*. Thus, variables do not have to be declared as in some other programming languages.²

All the variables that are created by a program are kept in a buffer called *workspace*. These variable can be viewed with the command `whos` or displayed in the *workspace* window of the MATLAB desktop. For example, after declaring the variables above, the `whos` command results in the responds

```
>> whos
  Name      Size            Bytes  Class       Attributes
  a            1x1              8  double
  b           1x12             24  char
```

It displays the name, the size, and the type (class) of the variable. The size is often useful to check the orientation of matrices as we will see below. The variables in the workspace can be used as long as MATLAB is running and as long as it is not cleared with the command `clear`. The workspace can be saved with the command `save filename`, which creates a file `filename.mat` with internal MATLAB format. The saved workspace can be reloaded into MATLAB with the command `load filename`. The load command can also be used to import data in ASCII format. The workspace is very convenient as we can run a program within a MATLAB session and can then work interactively with the results, for example, to plot some of the generated data.

Variables in MATLAB are generally matrices (or data arrays), which is very convenient for most of our purposes. Matrices include scalars (1×1 matrix) and vectors ($1 \times N$ matrix) as special cases. Values can be assigned to matrix elements in several ways. The most basic one is using square brackets and separating rows by a semicolon within the square brackets, for example (see Fig. E.2),

```
>> a=[1 2 3; 4 5 6; 7 8 9]
```

```
a =
1     2     3
4     5     6
7     8     9
```

A vector of elements with consecutive values can be assigned by column operators like

```
>> v=0:2:4
```

```
v =
0     2     4
```

Furthermore, the MATLAB desktop includes an *array editor*, and data in ASCII files can be assigned to matrices when loaded into MATLAB. Also, MATLAB functions often return matrices which can be used to assign values to matrix elements. For example, a uniformly distributed random 3×3 matrix can be generated with the command

```
>> b=rand(3)
```

```
b =
0.9501    0.4860    0.4565
0.2311    0.8913    0.0185
0.6068    0.7621    0.8214
```

The multiplication of two matrices (following the matrix multiplication rules illustrated in Appendix A) can be done in MATLAB by typing

```
>> c=a*b
c =
3.2329    4.5549    2.9577
8.5973   10.9730    6.8468
13.9616   17.3911   10.7360
```

This is equivalent to

```
c=zeros(3);
for i=1:3
    for j=1:3
        for k=1:3
            c(i,j)=c(i,j)+a(i,k)*b(k,j);
        end
    end
end
```

which is the common way of writing matrix multiplications in other programming languages. Formulating operations on whole matrices, rather than on the individual components separately, is not only more convenient and clear, but can enhance the programs performance considerably. Whenever possible, operations on whole matrices should be used. This is likely to be the major change in your programming style when converting from another programming language to MATLAB. The performance disadvantage of an interpreted language is often negligible when using operations on whole matrices.

The transpose operation of a matrix changes columns to rows. Thus, a row vector such as v can be changed to a column vector with the MATLAB transpose operator ('),

```
>> v'
```

```
ans =
```

```
0  
2  
4
```

which can then be used in a matrix-vector multiplication like

```
>> a*v'
```

```
ans =
```

```
16  
34  
52
```

The inconsistent operation $a*v$ does produce an error,

```
>> a*v  
??? Error using ==> mtimes  
Inner matrix dimensions must agree.
```

Component-wise operations in matrix multiplications (*), divisions (/) and potentiation ^ are indicated with a dot modifier such as

```
>> v.^2
```

```
ans =
```

```
0     4     16
```

The most common operators and basic programming constructs in MATLAB are similar to those in other programming languages and are listed in Table E.1.

E.1.3 Control flow and conditional operations

Besides the assignments of values to variables, and the availability of basic data structures such as arrays, a programming language needs a few basic operations

Table E.1 Basic programming contracts in MATLAB.

Programming construct	Command	Syntax
Assignment	=	<code>a=b</code>
Arithmetic operations	add multiplication division power	<code>a+b</code> <code>a*b</code> (matrix), <code>a.*b</code> (element-wise) <code>a/b</code> (matrix), <code>a./b</code> (element-wise) <code>a^b</code> (matrix), <code>a.^b</code> (element-wise)
Relational operators	equal not equal less than	<code>a==b</code> <code>a~=b</code> <code>a<b</code>
Logical operators	AND OR	<code>a & b</code> <code>a b</code>
Loop	for while	<code>for index=start:increment:end</code> statement <code>end</code> <code>while expression</code> statement <code>end</code>
Conditional command	if statement	<code>if logical expressions</code> statement <code>elseif logical expressions</code> statement <code>else</code> statement <code>end</code>
Function		<code>function [x,y,...]=name(a,b,...)</code>

for building loops and for controlling the flow of a program with conditional statements (see Table E.1). For example, the *for loop* can be used to create the elements of the vector *v* above, such as

```
>> for i=1:3; v(i)=2*(i-1); end  
>> v
```

v =

0 2 4

Table E.1 lists, in addition, the syntax of a while loop. An example of a conditional statement within a loop is

```
>> for i=1:10; if i>4 & i<=7; v2(i)=1; end; end  
>> v2
```

v2 =

0 0 0 0 1 1 1

In this loop, the statement *v2(i)=1* is only executed when the loop index is larger than 4 and less or equal to 7. Thus, when *i*=5, the array *v2* with 5 elements is created, and since only the elements *v2(5)* is set to 1, the previous elements are set to 0 by default. The loop adds then the two element *v2(6)* and *v2(7)*. Such a vector can also be created by assigning the values 1 to a specified range of indices,

```
>> v3(4:7)=1
```

v3 =

0 0 0 1 1 1 1

A 1×7 array is thereby created with elements set to 0, and only the specified elements are overwritten with the new value 1. Another method to write compact loops in MATLAB is to use vectors as index specifiers. For example, another way to create a vector with values such as *v2* or *v3* is

```
>> i=1:10
```

i =

1 2 3 4 5 6 7 8 9 10

```
>> v4(i>4 & i<=7)=1
```

v4 =

0 0 0 0 1 1 1

E.1.4 Creating MATLAB programs

If we want to repeat a series of commands, it is convenient to write this list of commands into an ASCII file with extension ‘.m’. Any ASCII editor (for example; WordPad, Emacs, etc.) can be used. The MATLAB package contains an editor that has the advantage of colouring the content of MATLAB programs for better readability and also provides direct links to other MATLAB tools. The list of commands in the ASCII file (e.g. *prog1.m*) is called a *script* in MATLAB and makes up a MATLAB program. This program can be executed with a run button in the MATLAB editor or by calling the name of the file within the command window (for example, by typing *prog1*). We assumed here that the program file is in the current directory of the MATLAB session or in one of the search paths that can be specified in MATLAB. The MATLAB desktop includes a ‘current directory’ window (see desktop menu). Some older MATLAB versions have instead a ‘path browser’. Alternatively, one can specify absolute path when calling a program, or change the current directories with UNIX-style commands such as *cd* in the command window (see Fig. E.3).

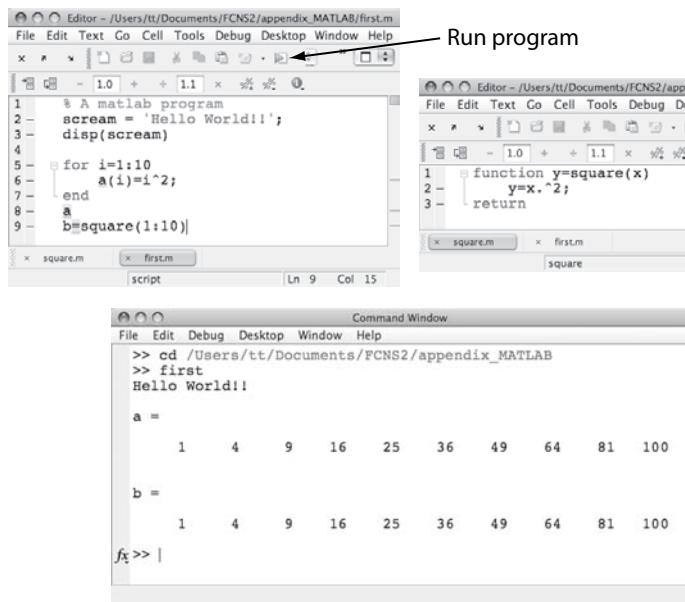


Fig. E.3 Two editor windows and a command window.

Functions are another way to encapsulate code. They have the additional advantage that they can be pre-compiled with the MATLAB Compiler™ available from MathWorks, Inc. Functions are kept in files with extension ‘.m’ which start with the command line like

```
function y=f(a,b)
```

Table E.2 MATLAB functions used in the programs of this book. This table provides only a brief explanation, while the MATLAB command `help cmd`, where `cmd` is any of the functions listed here, provides more detailed explanations.

Name	Brief description	Name	Brief description
abs	absolute functions	mod	modulus function
axis	sets axis limits	num2str	converts number to string
bar	produces bar plot	ode45	ordinary differential equation solver
ceil	round to larger integer	ones	produces matrix with unit elements
colormap	colour matrix for surface plots	plot	plot lines graphs
cos	cosine function	plot3	plot 3-dimensional graphs
diag	diagonal elements of a matrix	prod	product of elements
disp	display in command window	rand	uniformly distributed random variable
errorbar	plot with error bars	randn	normally distributed random variable
exp	exponential function	randperm	random permutations
fft	fast Fourier transform	reshape	reshaping a matrix
find	index of non-zero elements	set	sets values of parameters in structure
floor	round to smaller integer	sign	sign function
hist	produces histogram	sin	sine function
int2str	converts integer to string	sqrt	square root function
isempty	true if array is empty	std	calculates standard deviation
length	length of a vector	subplot	figure with multiple subfigures
log	logarithmic function	sum	sum of elements
lsqcurvefit	least mean square curve fitting (statistics toolbox)	surf	surface plot
max	maximum value and index	title	writes title on plot
min	minimum value and index	view	set viewing angle of 3D plot
mean	calculates mean	xlabel	label on x-axis of a plot
meshgrid	creates matrix to plot grid	ylabel	label on y-axis of a plot
		zeros	creates matrix of zero elements

where the variables `a` and `b` are passed to the function and `y` contains the values returned by the function. The return values can be assigned to a variable in the calling MATLAB script and added to the workspace. All other internal variables of the functions are local to the function and will not appear in the workspace of the calling script.

MATLAB has a rich collection of predefined functions implementing many algorithms, mathematical constructs, and advanced graphic handling, as well as general information and help functions. You can always search for some keywords using the useful command `lookfor` followed by the keyword (search term). This command lists all the names of the functions that include the keywords in a short description in the function file within the first *comment lines* after the function declaration in the function file. The command `help`, followed by the function name, displays the first block of comment lines in a function file. This description of functions is usually sufficient to know how to use a function. A list of all functions used in the examples of this book is listed in Table E.2.

E.1.5 Graphics

MATLAB is a great tool for producing scientific graphics. We want to illustrate this by writing our first program in MATLAB: calculating and plotting the sine function. The program is

```
x=0:0.1:2*pi;
y=sin(x);
plot(x,y)
```

The first line assigns elements to a vector **x** starting with $x(1) = 0$ and incrementing the value of each further component by 0.1 until the value 2π is reached (the variable **pi** has the appropriate value in MATLAB). The last element is $x(63) = 6.2$. The second line calls the MATLAB function **sin** with the vector **x** and assigns the results to a vector **y**. The third line calls a MATLAB plotting routine. You can type these lines into an ASCII file that you can name **plot_sin.m**. The code can be executed by typing **plot_sin** as illustrated in the *command window* in Fig. E.2.³ The execution of this program starts a *figure window* with the plot of the sine function as illustrated on the right in Fig. E.2.

The appearance of a plot can easily be changed by changing the attributes of the plot. There are several functions that help in performing this task, for example, the function **axis** that can be used to set the limits of the axis. New versions of MATLAB provide window-based interfaces to the attributes of the plot. However, there are also two basic commands, **get** and **set**, that we find useful. The command **get(gca)** returns a list with the axis properties currently in effect. This command is useful for finding out what properties exist. The variable **gca** (get current axis) is the *axis handle*, which is a variable that points to a memory location where all the attribute variables are kept. The attributes of the axis can be changed with the **set** command. For example, if we want to change the size of the labels we can type **set(gca,'fontsize',18)**. There is also a handle for the current figure **gcf** that can be used to get and set other attributes of the figure. MATLAB provides many routines to produce various special forms of plots including plots with error-bars, histograms, three-dimensional graphics, and multi-plot figures.

E.2 A first project: modelling the world

Suppose there is a simple world with a creature that can be in three distinct states, sleep (state value 1), eat (state value 2), and study (state value 3). An agent, which is a device that can sense environmental states and can generate actions, is observing this creature with poor sensors, which add white (Gaussian) noise to the true state. Our aim is to build a model of the behaviour of the creature which can be used by the agent to observe the states of the creature with some accuracy despite the limited sensors. For this exercise, the function **creature_state()** is available on the books resource page on the web. This function returns the current state of the creature. Try to create an agent program that predicts the current state of the creature. In the following we discuss some simple approaches.

A simulation program that implements a specific agent a with simple world model (a model of the creature), which also evaluates the accuracy of the model, is given in Table E.3. This program, also available on the web, is provided in file **main.m**. This program can be downloaded into the working directory of MATLAB and executed by typing **main** into the command window, or by opening the file in the MATLAB editor and starting it from there by pressing

³Provided that the MATLAB session points to the folder in which you placed the code.

Table E.3 Program main.m

```

1 % Project 1: simulation of agent which models simple creature
2 clear; correct=0;
3
4 for trial=1:1000
5     x=creature_state();
6     s=x+randn();
7
8     %% perception model
9     if (s<1.5) x_predict=1;
10    elseif (s<2.5) x_predict=2;
11    else x_predict=3;
12    end
13
14    %% calculate accuracy
15    if (x==x_predict) correct=correct+1; end
16    end
17
18 disp(['percentage correct: ',num2str(correct/1000)]);

```

the icon with the green triangle. The program reports the percentage of correct perceptions of the creature's state.

Line 1 of the program uses a comment indicator (%) to outline the purpose of the program. Line 2 clears the workspace to erase all eventual existing variables, and sets a counter for the number of correct perceptions to zero. Line 4 starts a loop over 1000 trials. In each trial, a creature state is pulled by calling the function `creature_state()` and recording this state value in variable `x`. The sensory state `s` is then calculated by adding a random number to this value. The value of the random number is generated from a normal distribution, a Gaussian distribution with mean zero and unit variance, with the MATLAB function `randn()`.

We are now ready to build a model for the agent to interpret the sensory state. In the example shown, this model is given in Lines 8–12. This model assumes that a sensory value below 1.5 corresponds to the state of a sleeping creature (Line 9), a sensory value between 1.5 and 2.5 corresponds to the creature eating (Line 10), and a higher value corresponds to the creature studying (Line 11). Note that we made several assumptions by defining this model, which might be unreasonable in real-world applications. For example, we used our knowledge that there are three states with ideal values of 1, 2, and 3 to build the model for the agent. Furthermore, we used the knowledge that the sensors are adding independent noise to these states in order to come up with the decision boundaries. The major challenge for real agents is to build models without this explicit knowledge. When running the program we find that a little bit over 50% of the cases are correctly perceived by the agent. While this

is a good start, one could do better. Try some of your own ideas . . .

. . . Did you succeed in getting better results? It is certainly not easy to guess some better model, and it is time to inspect the data more carefully. For example, we can plot the number of times each state occurs. For this we can write a loop to record the states in a vector,

```
>> for i=1:1000; a(i)=creature_state(); end
```

and then plot a histogram with the MATLAB function `hist()`,

```
>> hist(a)
```

The result is shown in Fig. E.4. This histogram shows that not all states are equally likely as we implicitly assumed in the above agent model. The third state is indeed much less likely. We could use this knowledge in a modified model in which we predict that the agent is sleeping for sensory states less than 1.5 and is eating otherwise. This modified model, which completely ignores study states, predicts around 65% of the states correctly. Many machine learning methods suffer from such ‘explaining away’ solutions for imbalanced data, as further discussed in Chapter 10.

It is important to recognize that 100% accuracy is not achievable with the inherent limitations of the sensors. However, higher recognition rates could be achieved with better world (creature + sensor) models. The main question is how to find such a model. We certainly should use observed data in a better way. For example, we could use several observations to estimate how many states are produced by function `creature_state()` and their relative frequency. Such parameter estimation is a basic form of learning from data. Many models in science take such an approach by proposing a parametric model and estimating parameters from the data by model fitting. The main challenge with this approach is how complex we should make the model. It is much easier to fit a more complex model with many parameters to example data, but the increased flexibility decreases the prediction ability of such models. Much progress has been made in machine learning by considering such questions (as discussed in Chapter 6), but those approaches only work well in limited worlds, certainly much more restricted than the world we live in. More powerful methods can be expected by learning how the brain solves such problems, as discussed in Chapter 10.

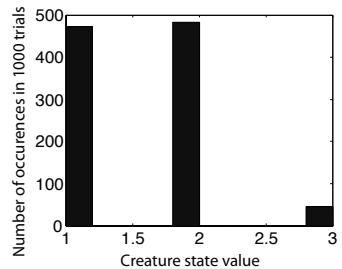


Fig. E.4 The MATLAB desktop window histogram of states produced by function `creature_state()` from 1000 trials.

E.3 Octave

The following two sections describe alternative programming environments to MATLAB, as well as the conversion of the example programs in this book to these environments. Both of these programming systems are open source environments and have general public licenses for non-commercial use. The converted programs are available on the resource page for this book.

We start with the programming environment called *Octave*, which is freely available under the GNU general public license. Most of the programs in this book run directly in this programming environment, and we concentrate in this section on outlining some changes that had to be made to some of the programs.

A collection of the modified programs is available on the book resource web page. The programs were tested with version 3.0.1 under Windows and Mac OS. Octave is available through links at <http://www.gnu.org/software/octave/>. The installation requires the additional installation of a graphics package, such as gnuplot or Java graphics. Some distributions contain the SciTE editor which can be used in this environment. An example of the environment is shown in Fig. E.5

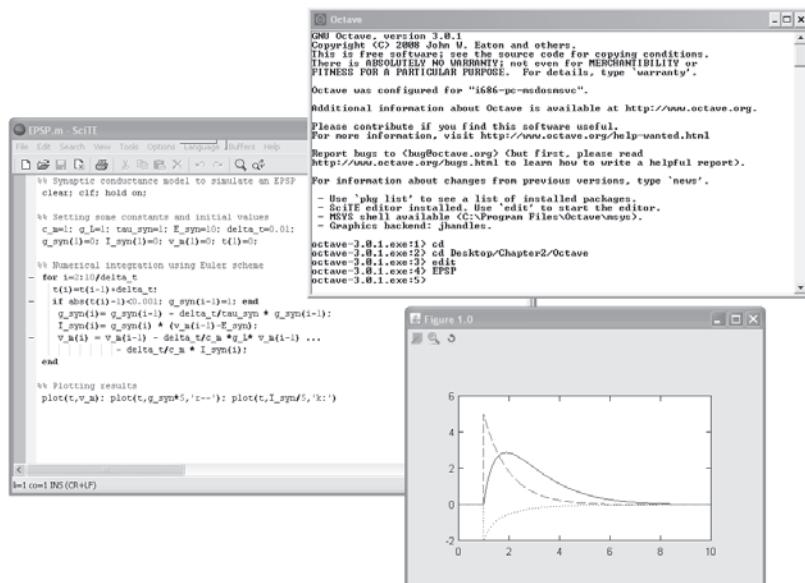


Fig. E.5 The Octave programming environment with the main console, and editor called *SciTE*, and a graphics window.

In Chapter 2, only the program *wilson.m*, had to be modified since it uses a library function to solve the differential equation. An ODE solver is provided in Octave with function *lsode('ode_file',x,t)*. Since additional parameter passing to the function in *ode_file* seems not to be provided, we have to declare the external input, *I_ext*, as a global variable. Similar changes had to be made in the other programs that use an ODE solver, in Chapters 7 and 8.

In Chapter 3, the only change needed is the plotting of results in program *if_sim*. While Octave supports the common subplot function, it does not support the specification of positions as in MATLAB. The program is thus simplified to only plot the time evolution of the voltage since the time of spikes is indicated when these values cross the threshold, indicated by the dashed line.

In Chapter 4, program *weightDistribution* requires a change for the least square fitting function, which is provided in Octave by the function *leasqr()*. The order of the *x*-values and parameters is also different in the Octave equivalent, so that the function *normal()* had also to be changed. Program *oja.m* does not require any changes, nor do any of the programs in Chapter 5.

In Chapter 6, the *errorbar()* function is used in program *perceptronTest* to plot results. While the *errorbar()* function is generally supported by Octave, the version tested with the Java graphics package produced an error. We

therefore plotted small lines with additional plot commands. The MATLAB interface for LIBSVM also works with Octave. In the program `som.m` of Chapter 7, the function `waitforbuttonpress` is not supported in the tested versions. Thus, we changed the Octave version to an automatically running version with a fixed number of iterations and some fixed delay using the function `pause()`. Similar to the changes in Chapter 2, changes were made to program `dnf.m` and the corresponding function `rnn_ode.m` to use the ODE solver provided by Octave.

In Chapter 8, analogue to the changes described above, changes were made to programs `ann_cont.m` and `lorenz.m`, together with their corresponding ODE functions, so enable the ODE solver in Octave. Also, the time step had to be made small in `lorenzOctave.m` to get reasonable results, since the ODE solver does not include and adaptive time step. We changed the plotting of the error bars in program `ann_sparse` as explained above. Finally, in Chapter 10, we changed the program `ExpectationMaximization.m` into a self-running movie to avoid the MATLAB function `waitforbuttonpress` as already encountered.

E.4 Scilab

Scilab is another scientific programming environment similar to MATLAB. This software package is freely available under the CeCILL software license, a license compatible to the GNU general public license. It is developed by the Scilab consortium, initiated by the French research centre INRIA. The Scilab package includes a MATLAB import facility that can be used to translate MATLAB programs to Scilab. In this section, we comment briefly on changes made to the example programs of this book to run under Scilab. References to Scilab documentations are provided on the Scilab home page at www.scilab.org. A screen shot of the Scilab environment is shown in Fig. E.6. A Scilab script can be run from the execute menu in the editor, or by calling `exec("filename.sce")`.

Scilab requires some changes of all the original MATLAB programs used in this book, although some are minor. Specifically, all comment lines had to be changed from `%` to `%%`. Another factor is the default orientation of vectors. The statement `a(3)=1` produces a row vector in MATLAB and a column vector in Scilab. Also, the plotting environment holds the previous plots by default so that the `hold on;` command of MATLAB needs to be removed. There are some additional differences between MATLAB and Scilab, specifically when it comes to suing predefined functions. These differences are mainly a concern when translating existing MATLAB programs. The graphical interfaces and the editor, called SciPad, can make Scilab easier to use when programming in Scilab compared to Octave. The modified example programs for Scilab have the extension `.sce` and are included in the folder called `Scilab` for each chapter. These programs were tested with version 5.1.1 for Windows. In the following we comment on additional changes specific to some programs.

In Chapter 2, program `hh.sce`, the dimension in the definition of vector `x` in Line 9 had to be changed. Also, in Line 25, the scalar 1 had to be replaced with the one element vector `[1]`. In program `wilson_euler.sce`, we also changed the dimensions in the definition of vector `x` on Line 8, and changed the MAT-

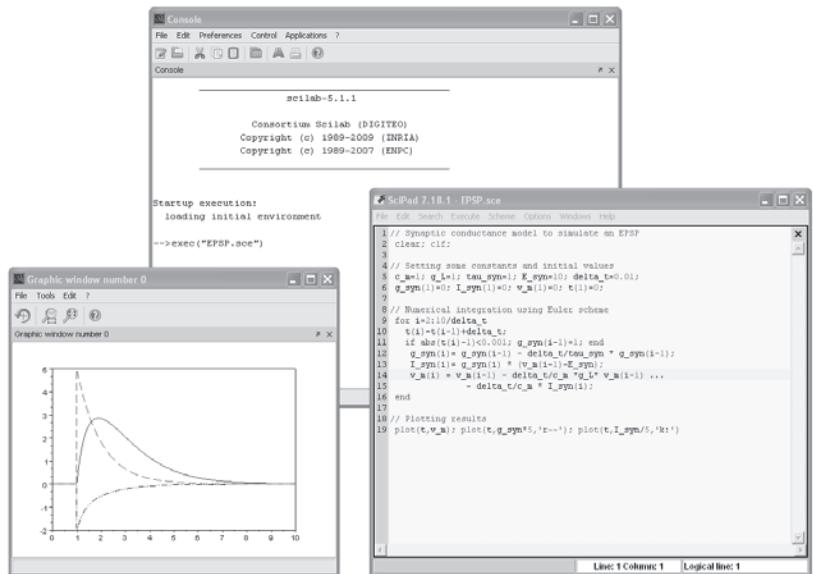


Fig. E.6 The Scilab programming environment with *console*, and editor called *SciPad*, and a graphics window.

LAB keyword **switch** to the Scilab equivalent **select**. The major change in this chapter is how the ODE solver is used in Scilab compared to MATLAB. The ODE-solver is provided in function **ode()**. The function with the ODE equations is included in the file **wilson.sce** of the Scilab version. Variables of the surrounding procedure are thereby accessible within this function, so that there is no need to pass the external input, **Lext**, or to declare a global variable as in the Octave version. The function is ended with **endfunction** instead of the MATLAB **return**. The function call to the ODE solver is now slightly different, and some orientations of vectors had to be adjusted.

In Chapter 3, the main change in the **if.sim** program is the plotting of the results. While Scilab supports the common subplot function, it does not currently support the specification of positions as in MATLAB. The program is thus simplified to only plot the time evolution of the voltage since the time of spikes is anyhow indicated when these values cross the threshold, indicated by the dashed line. In the program **poisson_spiketrain** we made the call to a uniform random variable a proper function call, **rand** → **rand()**, and replaced some MATLAB functions with the corresponding Scilab names, **hist()** → **histplot()**, and **std()** → **st_deviation()**.

In Chapter 4, program **weightDistribution** requires some changes of functions within this program. A function **hist()** is provided to count the number of values in each bin. The Scilab function **datafit()** is used for the function fitting, and we moved the function **normal()** into the file as required by criterion function **G()**. Finally, names of predefined constants start with %. The value for π is therefore returned by **%pi**. In program **oja.sce**, we removed the **hold** on command and changed the function call to generate a normal distributed random number to **rand(...,'normal')**.

In Chapter 5, only program `IzhikevichRanNet.m` requires minor changes on Lines 16 and 17. The normal distributed random variable is generated with Scilab function `randn(...,'normal')`, and the vector for the firing times has to be transposed in Line 17.

In Chapter 6, the first two programs, `displayLetter()` and `perceptronTrain` read in letters from an ASCII file and use a function to reshape a matrix. In these programs, the MATLAB `load` command is replaced with the `read()` function, and the MATLAB `reshape()` function with the Scilab `matrix()` function. Also, the variable `error` had to be replaced since this is a function in Scilab. In program `perceptronTest`, the function `randomFlipMatrix()` is added into the main script, and the MATLAB function for random permutations is replaced with the Scilab function `grand()` with option '`prm`'. To get the row vector with the maximum along the columns of a matrix, the letter '`r`' must be added to the `max` function in Scilab. Plotting error bars can be done with the `errbar` function in Scilab. A major difference in Scilab is the notation of component-wise division. For example, the inverse of each component of a matrix A with n rows and m columns is $1./A$, whereas in Scilab we need a unity matrix of same size, `ones(n,m)./A`. Changes are made accordingly in program `mlp.sce`. A scilab interface to LIBSVM is available from Scilab at http://www.scilab.org/contrib/index_contrib.php?page=displayCon...tribution&fileID=185.

In Chapter 7, we replaced the interactive version with MATLAB's `waitForbuttonpress`, with a self-running loop over a fixed number of interactions. However, the Scilab command `pause` can be used to wait for input. The program `dnf.sce` includes the ODE function, as already used in Chapter 2. Analogue changes were made in the programs `ann_cont.m` and `lorenz.m` of Chapter 8. In program `ann_sparse.sce`, the functions `mean()` and `st_deviation()`, have to include the row specifier to be compatible with the MATLAB version. In program `TDlearning2.sce` of Chapter 9, we renamed `gamma` to `gamma0` to not interfere with a function of the same name in Scilab. The remaining programs require no new changes.

Further reading

The MATLAB web (2008) and demos link in the MATLAB desktop provide links to very useful brief tutorial videos on basic MATLAB operations and advanced topics. There are also many other written tutorials on the link, and MathWorks Inc. maintains a list on their web site (MATLAB tutorial list, 2008). The book by Wallisch *et al.* (2009) provides a project-based introductory tutorial in the first section, specifically addressing a biologically-minded audience. In addition, later sections of this book include an introduction of using MATLAB for data analysis in neuro-

science and a brief introduction to neuroscience modelling.

There are also a variety of tutorials available for the use of Scilab and Octave. Scilab maintains a list of tutorials and online books (Scilab tutorial list, 2008). Octave has a comprehensive online manual (Octave manual, 2008), and there are a variety of online tutorials, such the Octave wiki maintained by AIMS (2008).

- MATLAB ‘getting started’ video (2008). Retrieved November 14, 2008. <http://www.mathworks.com/demos/matlab/getting-started-with-matlab-video-tutorial.html>
- MATLAB tutorial list (2008), retrieved November 14, 2008,
http://www.mathworks.com/matlabcentral/1/link_exchange/MATLAB/Tutorials.
- P. Wallisch, M. Lusignan, M. Benayoun, T.I. Baker, A.S. Dickey, N.G. Hatsopoulos (2009), *MATLAB for Neuroscientists*, Academic Press.
- Scilab tutorial list (2008), retrieved November 14, 2008, http://www.scilab.org/publications/index_publications.php?page=freebooks
- Octave Manual list (2008), retrieved November 14, 2008, <http://www.gnu.org/software/octave/doc/interpreter>.
- AIMS, Octave wiki (2008). Retrieved November 14, 2008. <http://www.aims.ac.za/resources/tutorials/octave>

Index

- α -function 28
- δ -function 326
- δ -rule 150–2, 153
 - generalized 160–3, 164
- Abbott, L. 17, 106, 108, 116, 286
- Abeles, M. 130
- ABS model 102–3
- absolute function 253
- abstraction 7
- acetylcholine (Ach) 26, 83–4, 219
- action–perception loop 16
- action potentials 23
 - generation of 33–46
- activation functions 37
 - in feedforward networks 159, 166
 - IF neuron 57, 59, 69–70
 - input and 69–70
 - interpretation of 78
 - non-monotonic 246
 - population 76, 78, 79–81
 - threshold 198–9
- activation vector 157
- activity packet/bubble 182, 193, 196–201
 - in complementary memory systems 269–70
 - in rotation network 204–5
- actor–critic scheme 282–6
- adaptation 11, 87–8
- adaptive resonance theory (ART) 313
 - ART1 equations 316–17
 - basic model 313–15
 - simulation 318–19, 320
 - unsupervised letter clustering 317–19, 320
- adaptive time step methods 46, 331
- adiabatic limit 77–8
- Adrian, E. 70–1
- after-depolarizing potential (ADP) 44
- algorithms 11, 12
 - batch 168–9
 - design 169
 - genetic 169–70
 - node creation 169
 - online learning algorithms 168–9
 - pruning 169
 - weight decay 110, 113–16, 169
- alpha-function 28
- alternating Gibbs sampling 305–6, 308
- Amari, S. 198, 213
- Amit, D. 219
- amnesia 218
 - amnesic phase 230
- AMPA channels and receptors 27, 83, 97–9
- amygdala 285
- analysis, levels of 11–13, 16
- ANN *see* attractor neural network (ANN)
- anticipatory brain system 14–16, 298–9
- Boltzmann machine 302–4
- expectation maximization 310–13
- Helmholtz machine 306
- in probabilistic framework 299–302
- probabilistic reasoning 308–10
- restricted Boltzmann machine (RBM) 304–6
 - simulation of 307–8
- arbor 24
- Artola, A. 97–8, 102, 116
- associative memory 87–94
- associators 87–116, 217
- asymmetrical networks 243–6
- asymptotic states 76, 192–3, 217

- asynchronous gain 132
- attentive vision 291–5
- attractor neural network (ANN)
 - 193, 217
 - exercises on 246
 - point-attractor neural networks 219–33
 - sparse attractor networks 233–8
- attractors 239–40
- attractor states 198–201, 220–33
- auto-associative networks/memory 215–46, 262–4
 - hippocampus and 215–19
- axial delay 138, 141, 278
- axons 22–4, 88–91
 - in schematic connectivity patterns 127, 128
 - staining techniques 124
- background firing 131–3
- back-projection 123
- Baddeley, A. 264
- Barto, A. 253, 255, 279, 282, 286
- Bar-Yam, Y. 258, 286
- basal ganglia 282–5
- basin of attraction 228, 231–2, 246
- basket cells 23, 274
- batch algorithm 168–9
- Bayesian network 301, 308–10
 - dynamic (DBN) 309–10
- Bayes's theorem 208–10, 343
- BCM theory 102–3
- Bear, M. 50
- Bernoulli distribution 338
- Bi, G. 96–7, 99–100, 116
- Bienenstock, E. 102, 113
- binary functions *see* Boolean functions
- binding problem 267
- binomial distribution 338, 339
- Bishop, C. 84, 179
- Bliss, T. 94, 95, 218
- Blue Brain Project 7
- Boltzmann machine 302–4
 - restricted (RBM) 304–6, 307–8
- Boolean functions 148–50
 - AND 82, 146
 - OR 149–50
 - XOR 149, 150, 177
- bouton 23, 24, 98, 128
- brain development 87–8
- brain-imaging techniques 8, 121
- brain organization 119–20
 - anatomy, large-scale 120–1
 - columnar 125–7
 - cortical parameters 128–9
 - hierarchical cortex 121–3
 - layered structure of neocortex 124–5
- neocortical layer connectivity 127–8
- rapid data transmission 123–4
- brainstem 121
- brain theory
 - anticipatory 298–313
 - computational 11–12, 13–16
 - existence of 8–13
- Braver, T. 264–5
- Broadbent, D. 266
- Bröcher, S. 116
- Brodmann, K. 120–1
- Brunel, N. 84
- Burges, C. 179
- bursting neurons 43–4, 97
- cable equation/theory 47–8
- Caianiello, E. 101
- calcium hypothesis 97–9
- CaM kinase II 97
- CANN 193; *see also* dynamic neural field (DNF) theory/model
- Carpenter, G. 313, 321
- cation influx channels 42
- causal models 301, 308–10
- causal state 299
- central limit theorem 342
- cerebellum, motor control and 273–4
- Changeux, J. 296
- channel capacity 353–4
- chaotic networks 238–46
 - asymmetrical networks 243–5
 - attractors 239–40
- Cohen–Grossberg theorem 242–3
- Lyapunov functions 240–2
- non-monotonic networks 246
 - simulation 240, 241, 245–6
- chemical pathway modelling 98–9
- chi-square distribution 338–9
- Churchland, P. 17

- classifiers
 large-margin 173–5
 soft-margin 175–6
- climbing fibres 274
- clustering 200–1
- Cohen, J. 264–5
- Cohen, M. 242
- Cohen–Grossberg theorem 242–3, 246
- coincidence detectors 71–2
- committee machine 251
- communication channel 345–8
- compartmental models 46–50
- competition 192
- complete transmission chain 131
- composite pattern 258, 260
- computational theory 11–12, 13–16
- concepts 200
- conditional distribution 343
- conditioning 216, 278
- Hebbian learning and 91–2
 - reinforcement learning problem and 275–6
- connectivity maps 121–3
- Connors, B. 50
- context units 170–1
- continuum limit 328
- continuous attraction neural network (CANN) 193; *see also* dynamic neural field (DNF) theory/model
- contrastive divergence 305
- convergence rate 131
- Cooper, L. 102, 113
- cooperation, local 192
- cornus ammonis (CA) 218–19, 263
- cortical feature maps 126–7, 181–2, 215
- self-organizing (SOMs) 4, 183–90
- cortical magnification 126
- corticothalamic loops 123
- cost function 151, 160, 175
- coupled attractor networks 257
- imprinted and composite patterns 258
 - signal-to-noise analysis 259–62
- Covey, A. 252
- Cowan, J. 84, 213
- Cowan, N. 266, 270
- Cramér–Rao bound 209
- cross-correlation function 102, 107–8
- cross-entropy 173, 353
- cross-talk 225–7, 254
- cumulative probability density function 341
- Dale’s principle 243–5
- Dayan, P. 17, 284, 286, 299
- decaying activity 192
- decision boundary 174
- Deco, G. 291, 294, 321
- decoding 207
- Bayesian 208–9
 - mechanism implementations 210–12
 - with tuning curves 209–10
- deep belief networks 300–1
- definition (of computational neuroscience) 1
- degradation, graceful 93–4
- Dehaene, S. 296, 297
- delta rule 150–2, 153
- generalized 160–3, 164
- dendrites 22–3
- dendritic processing 26–33
- dentate gyrus (DG) 218–19, 263
- design algorithms 169
- deterministic chaos 240
- differential equations xii, 28–9, 327
- in Hodgkin–Huxley model 36–7
- discretization 328
- distance measures 325–6
- distributed representation 93, 205–11
- distribution
- posterior 208–9, 343, 357
 - prior 312, 357
- divergence rate 131
- diverging–converging chains 130–1
- divide-and-conquer strategy 253
- divisive inhibition 82–3
- Doidge, N. 17
- dopamine (DA) 26, 98–9
- dopaminergic neurons 283–4, 285
- dorsal visual pathway 253, 294
- dorsal visual stream 121
- dot product 325–6
- Doya, K. 284

- dynamic neural field (DNF)
 - theory/model 190
 - asymptotic states and 192–3
 - attractor state analysis 198–201
 - centre-surround interaction
 - kernel 191–2
 - competitive representations 195–8
 - interacting-reverberating-memory hypothesis and 268–70
 - noisy population decoding and 211–12
 - simulation 194–5, 212
- dynamic systems theory
 - asymmetrical networks 243–6
 - attractors 239–40
 - Cohen–Grossberg theorem 242–3
 - Lyapunov functions 240–2
 - non-monotonic networks 246
 - terminology of 238–9
- edge enhancement 270
- educational resources vii, xv–xvii
- eligibility trace 278
- Elman, J. 171
- Elman-net 171
- emergence 10–11
- encoding 207–8
- energy function 241
- Enoki, R. 95
- entorhinal cortex (EC) 218
- entropy 348–52
 - cross-entropy 353
 - entropic error function 167
- environmental interactions 5, 15–16
- epileptic seizures 240
- episodic memory 216, 218–19, 265
- equations of motion 238, 239
- error-back-propagation 161–3, 164, 166
- Euler method 44, 328–30, 333
- evoked field potentials (EFPs) 94–5
- excitatory postsynaptic potentials (EPSPs) 27, 50, 95–6
- excitatory synapse 27
- expansion recoding 165, 234–5, 274
- expectation maximization 310–13
- experimental data, use of 2–3, 6–7, 16
- exponential distribution 339
- fatigue 44, 63
- fault tolerance 93–4
- feature vector 144–6, 165–6, 211
- feedback systems 14–15
 - feedback controllers 271–3
- feedforward mapping networks
 - multilayer perceptron 155–73
 - simple perceptron 143–55
- ferromagnetic phase 228
- Fine, A. 95
- firing rate 44
 - instantaneous 74
 - neural code and 70–3
 - population averages 74–5
- firing threshold 55, 56, 59
- first passage time 57–8, 70
- fixpoint 222–3
- fMRI 8, 121
- Fokker–Planck equations 106
- forward model control 273
- fractals 240
- Freud, S. 88
- Friston, K. 299, 321
- frontal lobe 120
- frustrated systems 228–9
- functional plasticity 87–8
- fusiform neurons 125
- future directions 319, 321
- GABA receptors 27
- Gabor functions 291–2
- gain control 109
- gain function 37, 69, 78;
 - see also* activation function
- gamma-aminobutyric acid (GABA) 26, 27
- Gardner, E. 235
- Gaussian distribution 340
- Gaussian error function 340–1
- generalization 93, 159–60
- generalized delta rule 160–3, 164
- genetic algorithms 169–70
- Gerstner, W. 76, 77–8, 84
- glial cells 21, 23–4
- Glimcher, P. 17
- glutamate (Glu) 26, 27, 98
- glutamate (Glu) receptors 27
 - AMPAR 27, 83, 97–9
 - NMDAR 27, 83, 98, 200–1

- Goldstein, E. 179
 Golgi staining 124, 125
 graceful degradation 93–4
 gradient descent 151–2, 161,
 166–8
 granule cells 273
 Grossberg, S. 127–8, 188, 219, 243,
 313, 321
 growing activity 192
 Gurney, K. 284
 Gutfreund, H. 220
- Hamming distance 325
 Hasselmo, M. 219
 Hawkins, J. 17, 299, 321
 Haykin, S. 179
 head direction 202–5
 Hebb, D. 64, 88–9
 Hebbian learning/plasticity 88–91
 associative mechanisms and
 recurrent networks 219
 asymmetrical and symmetrical
 96–7
 basic rule 101, 113
 in conditioning framework 91–2
 contrastive 304–6
 covariance rule 101–2, 109–10
 delta rule and 152
 features of 93–4
 mathematical formulation of
 99–105
 in rate models 100–4
 simulation 103–4, 110, 111
 in spike models 99–100
 trace rule 204
- Helmholtz machine 306
 Hemmen, L. 76
 Hertz, J. 179
 hetero-association 262–4
 heterosynaptic LTD 102
 hidden nodes 156–64, 169, 171,
 254–5, 302–6
 Hilgetag, C. 122
 Hinton, G. 256–7, 286, 299,
 303–8, 321
 hippocampus 197–8, 265
 auto-associative network and
 215–19
 sequence learning 263–4
 Hodgkin, A. 33
- Hodgkin–Huxley model 33–4,
 35–7
 simplification of 41–2
 simulation 37–8, 39–40, 50
 Hofstadter, D. 299
 homosynaptic LTD 102
 Hopfield, J. 219, 263
 Houk, J. 286
 Hubel, D. 125, 126–7, 181
 Huxley, A. 33
 hybrid methods 168
 hypercolumn 126–7, 181–2
 hyperpolarization 34–5
- Idiart, M. 266
 idiothetic cues 203
 independent component analysis
 (ICA) 115
 inferior-temporal (IT) cortex
 neurons 195–6
 population information in
 354–6
 receptive fields in 290–4
 sparseness in 356–7
 information theory 345
 channel capacity 354
 communication channel 345–8
 entropy (average information
 gain) 348–53
 inferior-temporal cortex
 sparseness 354–7
 information gain 345–8
 mutual information 353–4
 surprise 357–9
 inhibitory postsynaptic potentials
 (IPSPs) 27
 inhibitory synapse 27, 82–3, 274
 initial value problem 327–8
 inner product 165, 176, 325
 integrate-and-fire (IF) neuron
 leaky 54–5, 56–7, 58, 84
 noise models 67–70
 response of 55–6
 simulation 56–7, 58
 spike-response model 59–61, 62
 interacting-reverberating-memory
 hypothesis 268–70
 intermediate-term memory 219,
 264–5
 inverse model control 272–3

- ion channels 25
 - leakage channels 25, 34
 - neurotransmitter-gated 25, 26–7
 - voltage-gated 25, 34–5
- ion pumps 25, 35
- Izhikevich, E. 17, 61, 84, 137–42
- Izhikevich neuron 61, 63
 - exercises with 84, 142
 - networks of 137–42
- Jacobs, R. 253, 255, 286
- Jessell, T. 50
- joined distribution 343
- Jordan, M. 253, 255, 286
- Kandel, E. 50
- kernel function/machines 165–6, 176, 191–2
- Kerzberg, M. 296
- Kirchhoff's law 28, 36
- Koch, C. 50
- Kohonen, T. 184, 213
- Kohonen model 184–7
- Krogh, A. 179
- Kronecker symbol 233
- Kulbach–Leibler divergence 303, 342–3
- laminar models 127–8
- lateral geniculate nucleus (LGN) 6
- leakage channels 25, 34
- leaky integrator xii, 54, 71–2
- learning
 - advanced 166–8
 - delta rule 150–5
 - error-back-propagation 161–3, 164, 166
 - motor 270–4
 - perceptual 216
 - procedural 216
 - Q-learning 280–1, 284
 - reinforcement 274–86
 - retrieval phase 219
 - sequence 262–4
 - temporal difference 278–82
 - see also* Hebbian learning/plasticity
- learning algorithms 151–2
- learning phase 107, 204, 219
- learning rate 101
 - adaptive 167
- letter clustering 317–19, 320
- Levenberg–Marquardt method 167–8
- Levine, D. 321
- limit cycle 239
- limited capacity 265–70
- linear function 80, 81
- Lisman, J. 97, 116, 266
- load
 - capacity 228, 260
 - parameter 226–9, 260–1
- local representation 205–6
- logistic function *see* sigmoid function
- lognormal distribution 339
- Lømo, T. 94, 95, 218
- long-term depression (LTD) 94, 96–9, 102, 108
- long-term memory 218–19, 264–5
- long-term potentiation (LTP) 94–9, 102, 108
- look-up table 146, 147–8, 149
- Lorenz system 239–40
- Luck, S. 266–8
- Lyapunov functions 240–2, 243
- Maass, W. 84
- mapping functions 145–7
- mapping networks
 - feed-forward networks 143–79
 - modular 251–7
- maps, hierarchical 289–95; *see also* cortical feature maps
- marginal distribution 343
- Markov chains and models 309–10
- Markram, H. 96
- Marr, D. 11–13, 14, 218, 219
- Martinotti cells 23, 125, 127
- mathematical formulas, use of xi–xii
- MATLAB programming/simulations xii, 361
 - adaptive resonance theory (ART) 318–19, 320
- anticipating brain model 307–8
- basic variables in 363–6, 367
- chaotic networks 240, 241, 245–6
- conditional operations 366, 368

- continuous time ANN model 222, 223–4
 control flow 366, 368
 converting to Octave 373–5
 converting to Scilab 375–7
 creating programs 369–70
 dynamic neural field (DNF)
 model 194–5, 212
 error-back-propagation algorithm 162–3
 expectation maximization 311–13
 a first project 371–3
 fixpoint ANN model 222–3
 graphics 370–1
 Hebbian covariance rule 110, 111
 Hebbian plasticity rule 103–4
 Hodgkin–Huxley model 38, 39–40
 Izhikevich networks 138–40
 leaky IF neuron 56–7, 58
 Lorenz system 240, 241
 netlet 136–7
 numerical integration 331–3
 ODE solver 44–6
 perceptron 152–5, 156, 157
 Poisson spike train generation 66
 principal component analysis 115–16
 programming environment 361–2
 self-organizing map (SOM) 186–7, 212
 sparse attractor network 237–8
 starting a session 362–3
 support vector machine 176–8
 synaptic model 30–2
 temporal difference learning 281–2, 285, 286
 Wilson model 44
 matrix notations 323–5
 maximal storage capacity 235
 maximum likelihood estimate 208–9
 McCulloch, W. 63–4, 84
 McCulloch–Pitts neuron 63–4, 80
 McLeod, P. 179
 mean square error (MSE) 151, 152, 167
 medial temporal lobe 218
 membrane potential 24, 50
 memory
 anticipatory 15–16, 298–313
 associative 87–94
 auto-associative 215–46, 262–4
 complementary systems 70
 declarative/non-declarative 215–16
 episodic 216, 218–19, 265
 intermediate-term 219, 264–5
 long-term 218–19, 264–5
 memory activity 192
 semantic 216, 265
 short-term 171, 204, 215, 264
 types of 215–17
 working 193, 196–7, 215, 264–70
 Merzenich, M. 189
 Mexican-hat function 191
 midpoint method 331
 Miller, G. 265
 Miller, K. 106
 Minsky, M. 251, 286
 Miyake, A. 286
 mixture of experts 252–3
 models 2–3, 5–7
 exercises on 16
 see also specific models
 modular mapping networks 251–2
 mixture of experts 252–3
 product of experts 256–7
 ‘what-and-where’ task 253–5
 momentum 166–7
 Morita, M. 246
 mossy fibres 91, 219, 274
 motor learning and control 270–1
 cerebellum and 273–4
 feedback controllers 271–3
 MRI, functional (fMRI) 8, 121
 Müller, B. 179
 multilayer perceptron (MLP) 155–7
 advanced concepts 165–79
 biological plausibility of 163–5
 exercises on 178–9
 generalization in 159–63
 update rule for 157–9
 multinomial distribution 339
 Munro, P. 102, 113
 mutual information 353–4, 355
 myelin sheath 41
 natural gradient algorithm 167–8
 nature–nurture debate 255
 negative feedback control 271–2

- Neisser, U. 299
 Nelson, S. 116
 neocortex
 anatomy, large-scale 120–1
 columnar organization and cortical modules 125–7
 connectivity in 121–3, 127–8
 hierarchical organization 121–3
 layered structure of 124–5
 parameters in 128–9
 rapid data transmission in 123–4
 see also cortical feature maps
 Nernst equation 24
 net input 63–4
 netlets 134–7
 networks 4–5, 9–11
 background activity immunity of 131–2
 exercises on 142
 fully connected 131
 large 133
 netlets 134–7
 self-organizing architecture 169–70
 stochastic 172
 see also specific types of network
 neural code, firing rate hypothesis and 70–1
 neurites 22, 124, 125
 neuroinformatics 122
 neuromodulators 219
 neurons 3–4
 biological background 21–2
 information-processing mechanisms 23–4
 neuronal chains 130–1
 neuronal death 130
 neuron simulators 49–50
 structural properties 22–3
 see also specific types of neurons
 neuroscience, specializations in 1–2
 neurotransmitters 26–7, 83–4, 285
 Ng, A. 179
 Nilsson, N. 251, 286
 Nissl staining 124, 125
 NMDA receptors and channels 27, 83, 200–1
 Ca²⁺ and 98
 node creation algorithm 169
 nodes 9–10; *see also specific types of node*
 noise 16
 advantage of 230–2
 in coupled attractor networks 259–62
 decoding mechanisms and 211–12
 dynamic neural field model and 200, 201
 information theory and 353–4
 integrate-and-fire (IF) neuron noise models 67–70
 network immunity to 131–3
 in point-attractor neural networks 221–2, 224–33
 population average model and 79
 response characteristics and 37, 40
 simulated annealing 168
 non-monotonic networks 246
 noradrenaline 219
 norepinephrine 219
 normal distribution 339–40
 numerical calculus
 differences and sums 327
 Euler method 44, 328–30, 333
 higher-order methods 330–1
 numerical integration of initial value problem 327–8
 Oberley, H. 265
 objective function 151, 172–3, 255, 303
 object recognition 253–5, 289–95
 occipital lobe 120, 121
 Octave 373–5
 ocular dominance columns 126
 Ohm's law 29, 35
 oligodendrocytes 23–4
 Oja, E. 113
 Oja's rule 113–16
 O'Neill, M. 122
 online learning algorithms 168–9
 optical character recognition (OCR) 143–5
 O'Reilly, R. 264–5
 organization (central nervous system), levels of 3–5
 orientation columns 126–7
 Oscam's razor 7
 overfitting 160, 176

- paired-pulse facilitation 95
 Palmer, R. 179
 pandemonium 251
 Paradiso, M. 50
 parallel distributed processing 10
 parallel fibres 274
 parallel search 294–5
 paramagnetic phase 228
 parietal lobe 120
 path integration 202–5
 pattern completion 90, 93, 217
 Pearson distance 326
 perception 15–16
 perceptron *see* multilayer perceptron; simple perceptron
 perceptron learning rule 152
 perfect integrator 71–2
 phase transition 228
 phosphorylation 97–8, 99
 Pitts, W. 63–4, 84
 plasticity–stability dilemma 188, 313
 Poggio, T. 11
 point attractors 239, 243, 246, 292–3
 point-attractor neural network (ANN) 200, 215, 217, 220–1
 continuous time model 221–2, 223–4
 fixpoint model 221–3, 224–7
 network dynamics and training 220–4
 noisy weights and diluted attractor networks 232–3
 phase diagram 227–30
 signal-to-noise analysis 224–7
 simulation 222–4
 spurious states and advantage of noise 230–2
 Poisson distribution 65, 340
 Poisson spike trains 65–6, 68, 106–8, 351–2
 polychrony 141–2
 Poo, M. 96–7, 99–100
 population coding 205–12
 population/rate models 64, 74
 decoding 210–12
 excitatory and inhibitory separation 105
 firing rates and population averages 74–5
 Hebbian learning in 100–3
 information in 354–6
 mixed 105
 motivations for population dynamics 76–8
 non-classical synapses 81–4
 perceptrons and 147–8
 probabilistic population coding 207–12
 rapid response 78–9
 slow varying input dynamics 75–6
 weight distributions in 109–10
 population vector decoding 210–12
 posterior distribution 208–9, 343, 357
 postsynaptic potential (PSP) 26, 27–8
 non-linear superposition of 32–3
 Pouget, A. 213
 power spectrum analysis 138, 140
 principal component analysis (PCA) 114–16
 principle of parsimony 7
 prior distribution 312, 357
 probabilistic mapping networks 172–3
 probabilistic population coding 207–12
 probabilistic reasoning 308–10
 probability theory 16
 central limit theorem 342
 cumulative probability (density) function 340–1
 distribution differences’ measurement 342–3
 Gaussian error function 340–1
 moments 336–8
 probability (density) functions, examples of 338–40
 random numbers 335–6
 random variables’ function 341–2
 product of experts 256–7
 proprioceptive feedback 203, 272
 prototype 93, 146–7
 pruning algorithms 169
 pseudo-inverse method 233
 Purkinje cells 96–7, 274
 pyramidal neurons 22–3, 125, 127
 Q-learning 280–1, 284–5

- radial-basis function 80, 81, 166, 185
- random network 130–5, 137–40
- random variables xii, 16
- Ranvier nodes 41
- rate code 71
 - entropy of 351–2
- rate models *see* population/rate models
- receptive field size 290
- recognition model 300–1, 306, 310–11
- recurrence 219
- recurrent networks 131, 204, 215–46, 302–5
- Izhikevich 137–42
- modular 257–62
- see also* dynamic neural field (DNF) theory/model
- Redgrave, P. 284
- refractory period 38, 40
- regularization 160
- reinforcement learning 274–5
 - actor–critic scheme 282–6
 - problem of 275–6
 - temporal delta rule 277–8
 - temporal difference learning 278–82
- Reinhardt, J. 179
- replica method 229
- representation 11–12
 - classes of 205–6
 - competitive 195–8
 - distributed 205–13
 - sparseness of 206–7, 356–7
- representational plasticity 189–90
- Rescorla–Wagner theory 278, 279
- response properties 6
- resting potential 24, 34–5, 50
- retina, model 254, 307
- retrieval phase 219
- retrograde messengers 98
- reversal potential 24
- Rolls, E. 235, 286, 289, 321, 354
- Rosen, R. 321
- Rosenblatt, F. 149
- Rossum, M. 116
- rotation networks 203–5
- Rumelhart, D. 179
- Runge–Kutta method 329, 331–3
- sample
 - mean 337
 - variance 337–8
- Samsonovich, A. 270
- schizophrenic states 231
- Schölkopf, B. 179
- Schultz, W. 283
- Schwartz, J. 50
- Scilab 375–7
- Segev, I. 50
- Sejnowski, T. 17, 101, 299, 303
- self-organizing maps (SOMs) 4, 183–90
 - basic cortical map model 183–4
 - Kohonen model 184–6
 - ongoing refinements 188–90
 - simulation 186–7, 212
- Selfridge, O. 251, 286
- sensory feature vector 144–6
- sensory feedback 272–3
- sensory states 299–306
- sequence learning 262–4
- serial search 294–5
- serotonin 285
- Shah, P. 286
- Shannon, C. 345
- Shepherd, G. 50
- short-term memory 171, 204, 215, 264
- shunting inhibition 27, 82–3
- sigma–pi node 81–4, 252
- sigmoid function 80, 81, 159
- signal-to-noise analysis
 - coupled attractor network 259–62
 - point-attractor neural network 224–7
- signal-to-noise ratio (SNR) 354
- simple perceptron
 - Boolean functions 148–50
 - delta rule 150–2, 153
 - exercise on 178–9
 - mapping functions 145–7
 - optical character recognition (OCR) 143–5
 - population node as 147–8
 - simulation 152–5, 156, 157
- simulated annealing 168
- Singer, W. 97–8, 102, 116
- Smola, A. 179
- softmax function 172

- Sompolinsky, H. 220
 Song, S. 106, 108
 sparseness 206–7
 in inferior-temporal (IT) cortex 356–7
 sparse attractor networks 233–8
 spatial representation 202
 spike-response model 59–61, 62
 spike-time variability
 activation function and input 69–70
 biological irregularities 64–6
 noise models 67
 simulating variability 67–9
 spike timing accuracy 73
 spike timing dependent plasticity 96–7, 99–100, 140–2
 examples with spiking neurons 106–9
 exercises on 116
 spike train entropy 350–1
 spiking neurons 53
 activation function 57, 59
 integrate-and-fire (IF) neuron
 response 55–7, 58
 Izhikevich neuron 61, 63
 leaky integrate-and-fire neuron 54–5
 McCulloch–Pitts neuron 63–4
 population models 74–84
 spike-response model 59–61, 62
 spike timing dependent plasticity (STDP) examples 106–9
 spin glasses 228–9
 spin models 228–9
 spiny neurons 283–4
 spurious states 230–2
 spurious synchronization hypothesis 266–8
 Squire, L. 215–16
 staining techniques 124–5
 state space 239
 state value function 276
 state vector 239
 stationary states 76, 217;
 see also asymptotic states
 statistical learning 143, 217
 steady state 108–9, 135–6
 Steinbuch, K. 89
 stellate neurons 23, 125, 127
 step function 80, 81
 stimulus-dependent surprise 357–9
 stochastic escape 276
 stochastic networks 172
 stochastic threshold 67
 storage capacity 233–5
 Strickland, M. 179
 Stringer, S. 213, 286, 289
 Stroop task 297
 structural plasticity 87–8
 associations and 89–91
 structural risk minimization 176
 substantia nigra pars compacta (SNc) 283
 substantia nigra pars reticulata (SNr) 284
 supervised learning 168, 275
 support vector machines (SVMs) 173
 kernel function and 176
 large-margin classifiers 173–5
 simulation 176–8
 soft-margin classifiers 175–6
 surprise, stimulus-dependent 357–9
 Sutton, R. 279, 282, 286
 synapses 23–4
 basic mechanisms 26–33
 chemical 26
 excitatory 26–7, 82–3
 inhibitory 27, 82–3
 long-term depression (LTD) 94, 96–9, 102, 108
 long-term potentiation (LTP) 94–9, 102, 108
 model of 27–32
 networks with non-classical 81–4
 synaptic model 27–32
 synaptic plasticity 87–94
 calcium hypothesis and 97–9
 chemical pathway modelling 98–9
 typical experiments 94–6
 see also spike timing dependent plasticity
 synaptic scaling 105–16
 competitive 110, 112–13
 synaptic strength value 55
 synaptic weight 55, 90–1
 diverging–converging chains and 131
 synfire chains 130

- Takeuchi, A. 213
- task decomposition 255
- Taylor expansion 330
- teacher 161
- temporal credit assignment problem 275, 278
- temporal delta rule 277–8
- temporal difference learning 278–82, 284–6
- temporal lobe 120
 - medial 218
- thermal noise 228
- Thorpe, S. 14, 123–4
- threshold-linear function 80, 81
- threshold node 148–50
- time step methods, adaptive 46, 331
- trace rule 204
- trace term 204
- training 152–4
- training phase 219
- trajectory 239–40
- transfer function 80
- Treves, A. 234, 286
- Tuckwell, C. 50
- tuning curves 6, 185–6, 195–8
 - decoding 209–10, 211
 - exercise on 212
 - orientation 181–2, 190
- Turrigiano, g. 116
- underfitting
- uniform distribution 340
- universal function approximator 158, 164
- Vapnik, V. 173, 176
- variability simulation 67–9
- variance 337
- vector notations 323–5
- vision
 - attentive 291–5
 - VisNet 289–91
 - visual pathways 253–5
 - visual scene analysis 14–15, 123–4
 - visual search 291–5
 - VisNet 289–91
- Vogel, E. 266–8
- von der Malsburg, C. 183–4, 213
- Wang, X. 84
- Watkins, C. 280
- weight, negative/positive 104–5, 243
- weight decay 110, 113–16, 169
- weight distributions 105–16
 - simulation 110, 111
- weight matrix 104, 200–1, 243–6
- ‘what-and-where’ task 253–5
- white matter 23–4
- Wickens, J. 98
- Widrow, B. 89
- Wiesel, T. 125, 126–7, 181
- Willshaw, D. 113, 183–4, 213
- Wilson, H. 42, 50, 84, 213, 270
- Wilson model 41–6
- winner-take-all architecture 172, 317
- Wolpert, D. 13
- working memory 193, 196–7, 215, 264
 - limited capacity 265–70
- workspace hypothesis 295–8
- world model 300–2
- Young, M. 122
- Zhang, K. 213
- Zhou, X. 189