

Reinforcement Learning

An Introduction
second edition

Richard S. Sutton and Andrew G. Barto



Adaptive Computation and Machine Learning

Francis Bach, series editor

A complete list of books published in the Adaptive Computation and Machine Learning series appears at the back of this book.

Reinforcement Learning:

An Introduction

second edition

Richard S. Sutton and Andrew G. Barto

The MIT Press

Cambridge, Massachusetts

London, England

© 2018, 2020 Richard S. Sutton and Andrew G. Barto

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the copyright holder. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 2.0 Generic License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/2.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

This book was set in 10/12, CMR by Westchester Publishing Services. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Names: Sutton, Richard S., author. | Barto, Andrew G., author.

Title: Reinforcement learning : an introduction / Richard S. Sutton and Andrew G. Barto.

Description: Second edition. | Cambridge, MA : The MIT Press, [2018] | Series: Adaptive computation and machine learning series | Includes bibliographical references and index.

Identifiers: LCCN 2018023826 | ISBN 9780262039246 (hardcover : alk. paper)

Subjects: LCSH: Reinforcement learning.

Classification: LCC Q325.6 .R45 2018 | DDC 006.3/1--dc23 LC record available

at <https://lccn.loc.gov/2018023826>

10 9 8 7 6 5 4 3 2 1

In memory of A. Harry Klopf

Contents

Preface to the Second Edition	xiii
Preface to the First Edition	xvii
Summary of Notation	xix
1 Introduction	1
1.1 Reinforcement Learning	1
1.2 Examples	4
1.3 Elements of Reinforcement Learning	6
1.4 Limitations and Scope	7
1.5 An Extended Example: Tic-Tac-Toe	8
1.6 Summary	13
1.7 Early History of Reinforcement Learning	13
I Tabular Solution Methods	23
2 Multi-armed Bandits	25
2.1 A k -armed Bandit Problem	25
2.2 Action-value Methods	27
2.3 The 10-armed Testbed	28
2.4 Incremental Implementation	30
2.5 Tracking a Nonstationary Problem	32
2.6 Optimistic Initial Values	34
2.7 Upper-Confidence-Bound Action Selection	35
2.8 Gradient Bandit Algorithms	37
2.9 Associative Search (Contextual Bandits)	41
2.10 Summary	42

3 Finite Markov Decision Processes	47
3.1 The Agent–Environment Interface	47
3.2 Goals and Rewards	53
3.3 Returns and Episodes	54
3.4 Unified Notation for Episodic and Continuing Tasks	57
3.5 Policies and Value Functions	58
3.6 Optimal Policies and Optimal Value Functions	62
3.7 Optimality and Approximation	67
3.8 Summary	68
4 Dynamic Programming	73
4.1 Policy Evaluation (Prediction)	74
4.2 Policy Improvement	76
4.3 Policy Iteration	80
4.4 Value Iteration	82
4.5 Asynchronous Dynamic Programming	85
4.6 Generalized Policy Iteration	86
4.7 Efficiency of Dynamic Programming	87
4.8 Summary	88
5 Monte Carlo Methods	91
5.1 Monte Carlo Prediction	92
5.2 Monte Carlo Estimation of Action Values	96
5.3 Monte Carlo Control	97
5.4 Monte Carlo Control without Exploring Starts	100
5.5 Off-policy Prediction via Importance Sampling	103
5.6 Incremental Implementation	109
5.7 Off-policy Monte Carlo Control	110
5.8 *Discounting-aware Importance Sampling	112
5.9 *Per-decision Importance Sampling	114
5.10 Summary	115
6 Temporal-Difference Learning	119
6.1 TD Prediction	119
6.2 Advantages of TD Prediction Methods	124
6.3 Optimality of TD(0)	126
6.4 Sarsa: On-policy TD Control	129
6.5 Q-learning: Off-policy TD Control	131
6.6 Expected Sarsa	133
6.7 Maximization Bias and Double Learning	134
6.8 Games, Afterstates, and Other Special Cases	136
6.9 Summary	138

7 n-step Bootstrapping	141
7.1 n-step TD Prediction	142
7.2 n-step Sarsa	145
7.3 n-step Off-policy Learning	148
7.4 *Per-decision Methods with Control Variates	150
7.5 Off-policy Learning Without Importance Sampling: The n -step Tree Backup Algorithm	152
7.6 *A Unifying Algorithm: n -step $Q(\sigma)$	154
7.7 Summary	157
8 Planning and Learning with Tabular Methods	159
8.1 Models and Planning	159
8.2 Dyna: Integrated Planning, Acting, and Learning	161
8.3 When the Model Is Wrong	166
8.4 Prioritized Sweeping	168
8.5 Expected vs. Sample Updates	172
8.6 Trajectory Sampling	174
8.7 Real-time Dynamic Programming	177
8.8 Planning at Decision Time	180
8.9 Heuristic Search	181
8.10 Rollout Algorithms	183
8.11 Monte Carlo Tree Search	185
8.12 Summary of the Chapter	188
8.13 Summary of Part I: Dimensions	189
II Approximate Solution Methods	195
9 On-policy Prediction with Approximation	197
9.1 Value-function Approximation	198
9.2 The Prediction Objective (\overline{VE})	199
9.3 Stochastic-gradient and Semi-gradient Methods	200
9.4 Linear Methods	204
9.5 Feature Construction for Linear Methods	210
9.5.1 Polynomials	210
9.5.2 Fourier Basis	211
9.5.3 Coarse Coding	215
9.5.4 Tile Coding	217
9.5.5 Radial Basis Functions	221
9.6 Selecting Step-Size Parameters Manually	222
9.7 Nonlinear Function Approximation: Artificial Neural Networks	223
9.8 Least-Squares TD	228

9.9	Memory-based Function Approximation	230
9.10	Kernel-based Function Approximation	232
9.11	Looking Deeper at On-policy Learning: Interest and Emphasis	234
9.12	Summary	236
10	On-policy Control with Approximation	243
10.1	Episodic Semi-gradient Control	243
10.2	Semi-gradient n -step Sarsa	247
10.3	Average Reward: A New Problem Setting for Continuing Tasks	249
10.4	Deprecating the Discounted Setting	253
10.5	Differential Semi-gradient n -step Sarsa	255
10.6	Summary	256
11	*Off-policy Methods with Approximation	257
11.1	Semi-gradient Methods	258
11.2	Examples of Off-policy Divergence	260
11.3	The Deadly Triad	264
11.4	Linear Value-function Geometry	266
11.5	Gradient Descent in the Bellman Error	269
11.6	The Bellman Error is Not Learnable	274
11.7	Gradient-TD Methods	278
11.8	Emphatic-TD Methods	281
11.9	Reducing Variance	283
11.10	Summary	284
12	Eligibility Traces	287
12.1	The λ -return	288
12.2	TD(λ)	292
12.3	n -step Truncated λ -return Methods	295
12.4	Redoing Updates: Online λ -return Algorithm	297
12.5	True Online TD(λ)	299
12.6	*Dutch Traces in Monte Carlo Learning	301
12.7	Sarsa(λ)	303
12.8	Variable λ and γ	307
12.9	Off-policy Traces with Control Variates	309
12.10	Watkins's Q(λ) to Tree-Backup(λ)	312
12.11	Stable Off-policy Methods with Traces	314
12.12	Implementation Issues	316
12.13	Conclusions	317

13 Policy Gradient Methods	321
13.1 Policy Approximation and its Advantages	322
13.2 The Policy Gradient Theorem	324
13.3 REINFORCE: Monte Carlo Policy Gradient	326
13.4 REINFORCE with Baseline	329
13.5 Actor–Critic Methods	331
13.6 Policy Gradient for Continuing Problems	333
13.7 Policy Parameterization for Continuous Actions	335
13.8 Summary	337
III Looking Deeper	339
14 Psychology	341
14.1 Prediction and Control	342
14.2 Classical Conditioning	343
14.2.1 Blocking and Higher-order Conditioning	345
14.2.2 The Rescorla–Wagner Model	346
14.2.3 The TD Model	349
14.2.4 TD Model Simulations	350
14.3 Instrumental Conditioning	357
14.4 Delayed Reinforcement	361
14.5 Cognitive Maps	363
14.6 Habitual and Goal-directed Behavior	364
14.7 Summary	368
15 Neuroscience	377
15.1 Neuroscience Basics	378
15.2 Reward Signals, Reinforcement Signals, Values, and Prediction Errors	380
15.3 The Reward Prediction Error Hypothesis	381
15.4 Dopamine	383
15.5 Experimental Support for the Reward Prediction Error Hypothesis	387
15.6 TD Error/Dopamine Correspondence	390
15.7 Neural Actor–Critic	395
15.8 Actor and Critic Learning Rules	398
15.9 Hedonistic Neurons	402
15.10 Collective Reinforcement Learning	404
15.11 Model-based Methods in the Brain	407
15.12 Addiction	409
15.13 Summary	410

16 Applications and Case Studies	421
16.1 TD-Gammon	421
16.2 Samuel’s Checkers Player	426
16.3 Watson’s Daily-Double Wagering	429
16.4 Optimizing Memory Control	432
16.5 Human-level Video Game Play	436
16.6 Mastering the Game of Go	441
16.6.1 AlphaGo	444
16.6.2 AlphaGo Zero	447
16.7 Personalized Web Services	450
16.8 Thermal Soaring	453
17 Frontiers	459
17.1 General Value Functions and Auxiliary Tasks	459
17.2 Temporal Abstraction via Options	461
17.3 Observations and State	464
17.4 Designing Reward Signals	469
17.5 Remaining Issues	472
17.6 Reinforcement Learning and the Future of Artificial Intelligence	475
References	481
Index	519

Preface to the Second Edition

The twenty years since the publication of the first edition of this book have seen tremendous progress in artificial intelligence, propelled in large part by advances in machine learning, including advances in reinforcement learning. Although the impressive computational power that became available is responsible for some of these advances, new developments in theory and algorithms have been driving forces as well. In the face of this progress, a second edition of our 1998 book was long overdue, and we finally began the project in 2012. Our goal for the second edition was the same as our goal for the first: to provide a clear and simple account of the key ideas and algorithms of reinforcement learning that is accessible to readers in all the related disciplines. The edition remains an introduction, and we retain a focus on core, online learning algorithms. This edition includes some new topics that rose to importance over the intervening years, and we expanded coverage of topics that we now understand better. But we made no attempt to provide comprehensive coverage of the field, which has exploded in many different directions. We apologize for having to leave out all but a handful of these contributions.

As in the first edition, we chose not to produce a rigorous formal treatment of reinforcement learning, or to formulate it in the most general terms. However, our deeper understanding of some topics since the first edition required a bit more mathematics to explain; we have set off the more mathematical parts in shaded boxes that the non-mathematically-inclined may choose to skip. We also use a slightly different notation than was used in the first edition. In teaching, we have found that the new notation helps to address some common points of confusion. It emphasizes the difference between random variables, denoted with capital letters, and their instantiations, denoted in lower case. For example, the state, action, and reward at time step t are denoted S_t , A_t , and R_t , while their possible values might be denoted s , a , and r . Along with this, it is natural to use lower case for value functions (e.g., v_π) and restrict capitals to their tabular estimates (e.g., $Q_t(s, a)$). Approximate value functions are deterministic functions of random parameters and are thus also in lower case (e.g., $\hat{v}(s, \mathbf{w}_t) \approx v_\pi(s)$). Vectors, such as the weight vector \mathbf{w}_t (formerly $\boldsymbol{\theta}_t$) and the feature vector \mathbf{x}_t (formerly ϕ_t), are bold and written in lowercase even if they are random variables. Uppercase bold is reserved for matrices. In the first edition we used special notations, $\mathcal{P}_{ss'}^a$ and $\mathcal{R}_{ss'}^a$, for the transition probabilities and expected rewards. One weakness of that notation is that it still did not fully characterize the dynamics of the rewards, giving only their expectations, which is sufficient for dynamic programming but not for reinforcement learning. Another weakness

is the excess of subscripts and superscripts. In this edition we use the explicit notation of $p(s', r|s, a)$ for the joint probability for the next state and reward given the current state and action. All the changes in notation are summarized in a table on page xix.

The second edition is significantly expanded, and its top-level organization has been changed. After the introductory first chapter, the second edition is divided into three new parts. The first part (Chapters 2–8) treats as much of reinforcement learning as possible without going beyond the tabular case for which exact solutions can be found. We cover both learning and planning methods for the tabular case, as well as their unification in n -step methods and in Dyna. Many algorithms presented in this part are new to the second edition, including UCB, Expected Sarsa, Double learning, tree-backup, $Q(\sigma)$, RTDP, and MCTS. Doing the tabular case first, and thoroughly, enables core ideas to be developed in the simplest possible setting. The second part of the book (Chapters 9–13) is then devoted to extending the ideas to function approximation. It has new sections on artificial neural networks, the fourier basis, LSTD, kernel-based methods, Gradient-TD and Emphatic-TD methods, average-reward methods, true online TD(λ), and policy-gradient methods. The second edition significantly expands the treatment of off-policy learning, first for the tabular case in Chapters 5–7, then with function approximation in Chapters 11 and 12. Another change is that the second edition separates the forward-view idea of n -step bootstrapping (now treated more fully in Chapter 7) from the backward-view idea of eligibility traces (now treated independently in Chapter 12). The third part of the book has large new chapters on reinforcement learning’s relationships to psychology (Chapter 14) and neuroscience (Chapter 15), as well as an updated case-studies chapter including Atari game playing, Watson’s wagering strategy, and the Go playing programs AlphaGo and AlphaGo Zero (Chapter 16). Still, out of necessity we have included only a small subset of all that has been done in the field. Our choices reflect our long-standing interests in inexpensive model-free methods that should scale well to large applications. The final chapter now includes a discussion of the future societal impacts of reinforcement learning. For better or worse, the second edition is about twice as large as the first.

This book is designed to be used as the primary text for a one- or two-semester course on reinforcement learning. For a one-semester course, the first ten chapters should be covered in order and form a good core, to which can be added material from the other chapters, from other books such as Bertsekas and Tsitsiklis (1996), Wiering and van Otterlo (2012), and Szepesvari (2010), or from the literature, according to taste. Depending of the students’ background, some additional material on online supervised learning may be helpful. The ideas of options and option models are a natural addition (Sutton, Precup and Singh, 1999). A two-semester course can cover all the chapters as well as supplementary material. The book can also be used as part of broader courses on machine learning, artificial intelligence, or neural networks. In this case, it may be desirable to cover only a subset of the material. We recommend covering Chapter 1 for a brief overview, Chapter 2 through Section 2.4, Chapter 3, and then selecting sections from the remaining chapters according to time and interests. Chapter 6 is the most important for the subject and for the rest of the book. A course focusing on machine learning or neural networks should cover Chapters 9 and 10, and a course focusing on artificial intelligence or planning should cover Chapter 8. Throughout the book, sections and chapters that are more difficult and not essential to the rest of the book are marked

with a *. These can be omitted on first reading without creating problems later on. Some exercises are also marked with a * to indicate that they are more advanced and not essential to understanding the basic material of the chapter.

Most chapters end with a section entitled “Bibliographical and Historical Remarks,” wherein we credit the sources of the ideas presented in that chapter, provide pointers to further reading and ongoing research, and describe relevant historical background. Despite our attempts to make these sections authoritative and complete, we have undoubtedly left out some important prior work. For that we again apologize, and we welcome corrections and extensions for incorporation into the electronic version of the book.

Like the first edition, this edition of the book is dedicated to the memory of A. Harry Klopff. It was Harry who introduced us to each other, and it was his ideas about the brain and artificial intelligence that launched our long excursion into reinforcement learning. Trained in neurophysiology and long interested in machine intelligence, Harry was a senior scientist affiliated with the Avionics Directorate of the Air Force Office of Scientific Research (AFOSR) at Wright-Patterson Air Force Base, Ohio. He was dissatisfied with the great importance attributed to equilibrium-seeking processes, including homeostasis and error-correcting pattern classification methods, in explaining natural intelligence and in providing a basis for machine intelligence. He noted that systems that try to maximize something (whatever that might be) are qualitatively different from equilibrium-seeking systems, and he argued that maximizing systems hold the key to understanding important aspects of natural intelligence and for building artificial intelligences. Harry was instrumental in obtaining funding from AFOSR for a project to assess the scientific merit of these and related ideas. This project was conducted in the late 1970s at the University of Massachusetts Amherst (UMass Amherst), initially under the direction of Michael Arbib, William Kilmer, and Nico Spinelli, professors in the Department of Computer and Information Science at UMass Amherst, and founding members of the Cybernetics Center for Systems Neuroscience at the University, a farsighted group focusing on the intersection of neuroscience and artificial intelligence. Barto, a recent PhD from the University of Michigan, was hired as post doctoral researcher on the project. Meanwhile, Sutton, an undergraduate studying computer science and psychology at Stanford, had been corresponding with Harry regarding their mutual interest in the role of stimulus timing in classical conditioning. Harry suggested to the UMass group that Sutton would be a great addition to the project. Thus, Sutton became a UMass graduate student, whose PhD was directed by Barto, who had become an Associate Professor. The study of reinforcement learning as presented in this book is rightfully an outcome of that project instigated by Harry and inspired by his ideas. Further, Harry was responsible for bringing us, the authors, together in what has been a long and enjoyable interaction. By dedicating this book to Harry we honor his essential contributions, not only to the field of reinforcement learning, but also to our collaboration. We also thank Professors Arbib, Kilmer, and Spinelli for the opportunity they provided to us to begin exploring these ideas. Finally, we thank AFOSR for generous support over the early years of our research, and the NSF for its generous support over many of the following years.

We have very many people to thank for their inspiration and help with this second edition. Everyone we acknowledged for their inspiration and help with the first edition

deserve our deepest gratitude for this edition as well, which would not exist were it not for their contributions to edition number one. To that long list we must add many others who contributed specifically to the second edition. Our students over the many years that we have taught this material contributed in countless ways: exposing errors, offering fixes, and—not the least—being confused in places where we could have explained things better. We especially thank Martha Steenstrup for reading and providing detailed comments throughout. The chapters on psychology and neuroscience could not have been written without the help of many experts in those fields. We thank John Moore for his patient tutoring over many many years on animal learning experiments, theory, and neuroscience, and for his careful reading of multiple drafts of Chapters 14 and 15. We also thank Matt Botvinick, Nathaniel Daw, Peter Dayan, and Yael Niv for their penetrating comments on drafts of these chapter, their essential guidance through the massive literature, and their interception of many of our errors in early drafts. Of course, the remaining errors in these chapters—and there must still be some—are totally our own. We thank Phil Thomas for helping us make these chapters accessible to non-psychologists and non-neuroscientists, and we thank Peter Sterling for helping us improve the exposition. We are grateful to Jim Houk for introducing us to the subject of information processing in the basal ganglia and for alerting us to other relevant aspects of neuroscience. José Martínez, Terry Sejnowski, David Silver, Gerry Tesauro, Georgios Theocharous, and Phil Thomas generously helped us understand details of their reinforcement learning applications for inclusion in the case-studies chapter, and they provided helpful comments on drafts of these sections. Special thanks are owed to David Silver for helping us better understand Monte Carlo Tree Search and the DeepMind Go-playing programs. We thank George Konidaris for his help with the section on the Fourier basis. Emilio Cartoni, Thomas Cederborg, Stefan Dernbach, Clemens Rosenbaum, Patrick Taylor, Thomas Colin, and Pierre-Luc Bacon helped us in a number important ways for which we are most grateful.

Sutton would also like to thank the members of the Reinforcement Learning and Artificial Intelligence laboratory at the University of Alberta for contributions to the second edition. He owes a particular debt to Rupam Mahmood for essential contributions to the treatment of off-policy Monte Carlo methods in Chapter 5, to Hamid Maei for helping develop the perspective on off-policy learning presented in Chapter 11, to Eric Graves for conducting the experiments in Chapter 13, to Shangtong Zhang for replicating and thus verifying almost all the experimental results, to Kris De Asis for improving the new technical content of Chapters 7 and 12, and to Harm van Seijen for insights that led to the separation of n -step methods from eligibility traces and (along with Hado van Hasselt) for the ideas involving exact equivalence of forward and backward views of eligibility traces presented in Chapter 12. Sutton also gratefully acknowledges the support and freedom he was granted by the Government of Alberta and the National Science and Engineering Research Council of Canada throughout the period during which the second edition was conceived and written. In particular, he would like to thank Randy Goebel for creating a supportive and far-sighted environment for research in Alberta. He would also like to thank DeepMind their support in the last six months of writing the book.

Finally, we owe thanks to the many careful readers of drafts of the second edition that we posted on the internet. They found many errors that we had missed and alerted us to potential points of confusion.

Preface to the First Edition

We first came to focus on what is now known as reinforcement learning in late 1979. We were both at the University of Massachusetts, working on one of the earliest projects to revive the idea that networks of neuronlike adaptive elements might prove to be a promising approach to artificial adaptive intelligence. The project explored the “heterostatic theory of adaptive systems” developed by A. Harry Klopf. Harry’s work was a rich source of ideas, and we were permitted to explore them critically and compare them with the long history of prior work in adaptive systems. Our task became one of teasing the ideas apart and understanding their relationships and relative importance. This continues today, but in 1979 we came to realize that perhaps the simplest of the ideas, which had long been taken for granted, had received surprisingly little attention from a computational perspective. This was simply the idea of a learning system that *wants* something, that adapts its behavior in order to maximize a special signal from its environment. This was the idea of a “hedonistic” learning system, or, as we would say now, the idea of reinforcement learning.

Like others, we had a sense that reinforcement learning had been thoroughly explored in the early days of cybernetics and artificial intelligence. On closer inspection, though, we found that it had been explored only slightly. While reinforcement learning had clearly motivated some of the earliest computational studies of learning, most of these researchers had gone on to other things, such as pattern classification, supervised learning, and adaptive control, or they had abandoned the study of learning altogether. As a result, the special issues involved in learning how to get something from the environment received relatively little attention. In retrospect, focusing on this idea was the critical step that set this branch of research in motion. Little progress could be made in the computational study of reinforcement learning until it was recognized that such a fundamental idea had not yet been thoroughly explored.

The field has come a long way since then, evolving and maturing in several directions. Reinforcement learning has gradually become one of the most active research areas in machine learning, artificial intelligence, and neural network research. The field has developed strong mathematical foundations and impressive applications. The computational study of reinforcement learning is now a large field, with hundreds of active researchers around the world in diverse disciplines such as psychology, control theory, artificial intelligence, and neuroscience. Particularly important have been the contributions establishing and developing the relationships to the theory of optimal control and dynamic programming.

The overall problem of learning from interaction to achieve goals is still far from being solved, but our understanding of it has improved significantly. We can now place component ideas, such as temporal-difference learning, dynamic programming, and function approximation, within a coherent perspective with respect to the overall problem.

Our goal in writing this book was to provide a clear and simple account of the key ideas and algorithms of reinforcement learning. We wanted our treatment to be accessible to readers in all of the related disciplines, but we could not cover all of these perspectives in detail. For the most part, our treatment takes the point of view of artificial intelligence and engineering. Coverage of connections to other fields we leave to others or to another time. We also chose not to produce a rigorous formal treatment of reinforcement learning. We did not reach for the highest possible level of mathematical abstraction and did not rely on a theorem–proof format. We tried to choose a level of mathematical detail that points the mathematically inclined in the right directions without distracting from the simplicity and potential generality of the underlying ideas.

In some sense we have been working toward this book for thirty years, and we have lots of people to thank. First, we thank those who have personally helped us develop the overall view presented in this book: Harry Klopf, for helping us recognize that reinforcement learning needed to be revived; Chris Watkins, Dimitri Bertsekas, John Tsitsiklis, and Paul Werbos, for helping us see the value of the relationships to dynamic programming; John Moore and Jim Kehoe, for insights and inspirations from animal learning theory; Oliver Selfridge, for emphasizing the breadth and importance of adaptation; and, more generally, our colleagues and students who have contributed in countless ways: Ron Williams, Charles Anderson, Satinder Singh, Sridhar Mahadevan, Steve Bradtke, Bob Crites, Peter Dayan, and Leemon Baird. Our view of reinforcement learning has been significantly enriched by discussions with Paul Cohen, Paul Utgoff, Martha Steenstrup, Gerry Tesauro, Mike Jordan, Leslie Kaelbling, Andrew Moore, Chris Atkeson, Tom Mitchell, Nils Nilsson, Stuart Russell, Tom Dietterich, Tom Dean, and Bob Narendra. We thank Michael Littman, Gerry Tesauro, Bob Crites, Satinder Singh, and Wei Zhang for providing specifics of Sections 4.7, 15.1, 15.4, 15.5, and 15.6 respectively. We thank the Air Force Office of Scientific Research, the National Science Foundation, and GTE Laboratories for their long and farsighted support.

We also wish to thank the many people who have read drafts of this book and provided valuable comments, including Tom Kalt, John Tsitsiklis, Paweł Cichosz, Olle Gällmo, Chuck Anderson, Stuart Russell, Ben Van Roy, Paul Steenstrup, Paul Cohen, Sridhar Mahadevan, Jette Randlov, Brian Sheppard, Thomas O’Connell, Richard Coggins, Cristina Versino, John H. Hiett, Andreas Badelt, Jay Ponte, Joe Beck, Justus Piater, Martha Steenstrup, Satinder Singh, Tommi Jaakkola, Dimitri Bertsekas, Torbjörn Ekman, Christina Björkman, Jakob Carlström, and Olle Palmgren. Finally, we thank Gwyn Mitchell for helping in many ways, and Harry Stanton and Bob Prior for being our champions at MIT Press.

Summary of Notation

Capital letters are used for random variables, whereas lower case letters are used for the values of random variables and for scalar functions. Quantities that are required to be real-valued vectors are written in bold and in lower case (even if random variables). Matrices are bold capitals.

\doteq	equality relationship that is true by definition
\approx	approximately equal
\propto	proportional to
$\Pr\{X=x\}$	probability that a random variable X takes on the value x
$X \sim p$	random variable X selected from distribution $p(x) \doteq \Pr\{X=x\}$
$\mathbb{E}[X]$	expectation of a random variable X , i.e., $\mathbb{E}[X] \doteq \sum_x p(x)x$
$\arg \max_a f(a)$	a value of a at which $f(a)$ takes its maximal value
$\ln x$	natural logarithm of x
$e^x, \exp(x)$	the base of the natural logarithm, $e \approx 2.71828$, carried to power x ; $e^{\ln x} = x$
\mathbb{R}	set of real numbers
$f : \mathcal{X} \rightarrow \mathcal{Y}$	function f from elements of set \mathcal{X} to elements of set \mathcal{Y}
\leftarrow	assignment
$(a, b]$	the real interval between a and b including b but not including a
ε	probability of taking a random action in an ε -greedy policy
α, β	step-size parameters
γ	discount-rate parameter
λ	decay-rate parameter for eligibility traces
$\mathbb{1}_{\textit{predicate}}$	indicator function ($\mathbb{1}_{\textit{predicate}} \doteq 1$ if the <i>predicate</i> is true, else 0)

In a multi-arm bandit problem:

k	number of actions (arms)
t	discrete time step or play number
$q_*(a)$	true value (expected reward) of action a
$Q_t(a)$	estimate at time t of $q_*(a)$
$N_t(a)$	number of times action a has been selected up prior to time t
$H_t(a)$	learned preference for selecting action a at time t
$\pi_t(a)$	probability of selecting action a at time t
\bar{R}_t	estimate at time t of the expected reward given π_t

In a Markov Decision Process:

s, s'	states
a	an action
r	a reward
\mathcal{S}	set of all nonterminal states
\mathcal{S}^+	set of all states, including the terminal state
$\mathcal{A}(s)$	set of all actions available in state s
\mathcal{R}	set of all possible rewards, a finite subset of \mathbb{R}
\subset	subset of (e.g., $\mathcal{R} \subset \mathbb{R}$)
\in	is an element of; e.g. ($s \in \mathcal{S}, r \in \mathcal{R}$)
$ \mathcal{S} $	number of elements in set \mathcal{S}
t	discrete time step
$T, T(t)$	final time step of an episode, or of the episode including time step t
A_t	action at time t
S_t	state at time t , typically due, stochastically, to S_{t-1} and A_{t-1}
R_t	reward at time t , typically due, stochastically, to S_{t-1} and A_{t-1}
π	policy (decision-making rule)
$\pi(s)$	action taken in state s under <i>deterministic</i> policy π
$\pi(a s)$	probability of taking action a in state s under <i>stochastic</i> policy π
G_t	return following time t
h	horizon, the time step one looks up to in a forward view
$G_{t:t+n}, G_{t:h}$	n -step return from $t + 1$ to $t + n$, or to h (discounted and corrected)
$\tilde{G}_{t:h}$	flat return (undiscounted and uncorrected) from $t + 1$ to h (Section 5.8)
G_t^λ	λ -return (Section 12.1)
$G_{t:h}^\lambda$	truncated, corrected λ -return (Section 12.3)
$G_t^{\lambda_s}, G_t^{\lambda_a}$	λ -return, corrected by estimated state, or action, values (Section 12.8)
$p(s', r s, a)$	probability of transition to state s' with reward r , from state s and action a
$p(s' s, a)$	probability of transition to state s' , from state s taking action a
$r(s, a)$	expected immediate reward from state s after action a
$r(s, a, s')$	expected immediate reward on transition from s to s' under action a
$v_\pi(s)$	value of state s under policy π (expected return)
$v_*(s)$	value of state s under the optimal policy
$q_\pi(s, a)$	value of taking action a in state s under policy π
$q_*(s, a)$	value of taking action a in state s under the optimal policy
V, V_t	array estimates of state-value function v_π or v_*
Q, Q_t	array estimates of action-value function q_π or q_*
$\bar{V}_t(s)$	expected approximate action value; for example, $\bar{V}_t(s) \doteq \sum_a \pi(a s)Q_t(s, a)$
U_t	target for estimate at time t

δ_t	temporal-difference (TD) error at t (a random variable) (Section 6.1)
δ_t^s, δ_t^a	state- and action-specific forms of the TD error (Section 12.9)
n	in n -step methods, n is the number of steps of bootstrapping
d	dimensionality—the number of components of \mathbf{w}
d'	alternate dimensionality—the number of components of $\boldsymbol{\theta}$
\mathbf{w}, \mathbf{w}_t	d -vector of weights underlying an approximate value function
$w_i, w_{t,i}$	i th component of learnable weight vector
$\hat{v}(s, \mathbf{w})$	approximate value of state s given weight vector \mathbf{w}
$v_{\mathbf{w}}(s)$	alternate notation for $\hat{v}(s, \mathbf{w})$
$\hat{q}(s, a, \mathbf{w})$	approximate value of state-action pair s, a given weight vector \mathbf{w}
$\nabla \hat{v}(s, \mathbf{w})$	column vector of partial derivatives of $\hat{v}(s, \mathbf{w})$ with respect to \mathbf{w}
$\nabla \hat{q}(s, a, \mathbf{w})$	column vector of partial derivatives of $\hat{q}(s, a, \mathbf{w})$ with respect to \mathbf{w}
$\mathbf{x}(s)$	vector of features visible when in state s
$\mathbf{x}(s, a)$	vector of features visible when in state s taking action a
$x_i(s), x_i(s, a)$	i th component of vector $\mathbf{x}(s)$ or $\mathbf{x}(s, a)$
\mathbf{x}_t	shorthand for $\mathbf{x}(S_t)$ or $\mathbf{x}(S_t, A_t)$
$\mathbf{w}^\top \mathbf{x}$	inner product of vectors, $\mathbf{w}^\top \mathbf{x} \doteq \sum_i w_i x_i$; for example, $\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s)$
\mathbf{v}, \mathbf{v}_t	secondary d -vector of weights, used to learn \mathbf{w} (Chapter 11)
\mathbf{z}_t	d -vector of eligibility traces at time t (Chapter 12)
$\boldsymbol{\theta}, \boldsymbol{\theta}_t$	parameter vector of target policy (Chapter 13)
$\pi(a s, \boldsymbol{\theta})$	probability of taking action a in state s given parameter vector $\boldsymbol{\theta}$
$\pi_{\boldsymbol{\theta}}$	policy corresponding to parameter $\boldsymbol{\theta}$
$\nabla \pi(a s, \boldsymbol{\theta})$	column vector of partial derivatives of $\pi(a s, \boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$
$J(\boldsymbol{\theta})$	performance measure for the policy $\pi_{\boldsymbol{\theta}}$
$\nabla J(\boldsymbol{\theta})$	column vector of partial derivatives of $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$
$h(s, a, \boldsymbol{\theta})$	preference for selecting action a in state s based on $\boldsymbol{\theta}$
$b(a s)$	behavior policy used to select actions while learning about target policy π
$b(s)$	a baseline function $b : \mathcal{S} \mapsto \mathbb{R}$ for policy-gradient methods
b	branching factor for an MDP or search tree
$\rho_{t:h}$	importance sampling ratio for time t through time h (Section 5.5)
ρ_t	importance sampling ratio for time t alone, $\rho_t \doteq \rho_{t:t}$
$r(\pi)$	average reward (reward rate) for policy π (Section 10.3)
\bar{R}_t	estimate of $r(\pi)$ at time t
$\mu(s)$	on-policy distribution over states (Section 9.2)
$\boldsymbol{\mu}$	$ \mathcal{S} $ -vector of the $\mu(s)$ for all $s \in \mathcal{S}$
$\ v\ _{\mu}^2$	μ -weighted squared norm of value function v , i.e., $\ v\ _{\mu}^2 \doteq \sum_{s \in \mathcal{S}} \mu(s) v(s)^2$
$\eta(s)$	expected number of visits to state s per episode (page 199)
Π	projection operator for value functions (page 268)
B_{π}	Bellman operator for value functions (Section 11.4)

A	$d \times d$ matrix $\mathbf{A} \doteq \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top]$
b	d -dimensional vector $\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t]$
w_{TD}	TD fixed point $\mathbf{w}_{\text{TD}} \doteq \mathbf{A}^{-1}\mathbf{b}$ (a d -vector, Section 9.4)
I	identity matrix
P	$ \mathcal{S} \times \mathcal{S} $ matrix of state-transition probabilities under π
D	$ \mathcal{S} \times \mathcal{S} $ diagonal matrix with μ on its diagonal
X	$ \mathcal{S} \times d$ matrix with the $\mathbf{x}(s)$ as its rows
$\bar{\delta}_{\mathbf{w}}(s)$	Bellman error (expected TD error) for $v_{\mathbf{w}}$ at state s (Section 11.4)
$\bar{\delta}_{\mathbf{w}}$, BE	Bellman error vector, with components $\bar{\delta}_{\mathbf{w}}(s)$
$\overline{\text{VE}}(\mathbf{w})$	mean square value error $\overline{\text{VE}}(\mathbf{w}) \doteq \ v_{\mathbf{w}} - v_{\pi}\ _{\mu}^2$ (Section 9.2)
$\overline{\text{BE}}(\mathbf{w})$	mean square Bellman error $\overline{\text{BE}}(\mathbf{w}) \doteq \ \bar{\delta}_{\mathbf{w}}\ _{\mu}^2$
$\overline{\text{PBE}}(\mathbf{w})$	mean square projected Bellman error $\overline{\text{PBE}}(\mathbf{w}) \doteq \ \Pi \bar{\delta}_{\mathbf{w}}\ _{\mu}^2$
$\overline{\text{TDE}}(\mathbf{w})$	mean square temporal-difference error $\overline{\text{TDE}}(\mathbf{w}) \doteq \mathbb{E}_b[\rho_t \delta_t^2]$ (Section 11.5)
$\overline{\text{RE}}(\mathbf{w})$	mean square return error (Section 11.6)

Chapter 1

Introduction

The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning. When an infant plays, waves its arms, or looks about, it has no explicit teacher, but it does have a direct sensorimotor connection to its environment. Exercising this connection produces a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals. Throughout our lives, such interactions are undoubtedly a major source of knowledge about our environment and ourselves. Whether we are learning to drive a car or to hold a conversation, we are acutely aware of how our environment responds to what we do, and we seek to influence what happens through our behavior. Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence.

In this book we explore a *computational* approach to learning from interaction. Rather than directly theorizing about how people or animals learn, we primarily explore idealized learning situations and evaluate the effectiveness of various learning methods.of an artificial intelligence researcher or engineer. We explore designs for machines that are effective in solving learning problems of scientific or economic interest, evaluating the designs through mathematical analysis or computational experiments. The approach we explore, called *reinforcement learning*, is much more focused on goal-directed learning from interaction than are other approaches to machine learning.

1.1 Reinforcement Learning

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning.

Reinforcement learning, like many topics whose names end with “ing,” such as machine learning and mountaineering, is simultaneously a problem, a class of solution methods that work well on the problem, and the field that studies this problem and its solution methods. It is convenient to use a single name for all three things, but at the same time essential to keep the three conceptually separate. In particular, the distinction between problems and solution methods is very important in reinforcement learning; failing to make this distinction is the source of many confusions.

We formalize the problem of reinforcement learning using ideas from dynamical systems theory, specifically, as the optimal control of incompletely-known Markov decision processes. The details of this formalization must wait until Chapter 3, but the basic idea is simply to capture the most important aspects of the real problem facing a learning agent interacting over time with its environment to achieve a goal. A learning agent must be able to sense the state of its environment to some extent and must be able to take actions that affect the state. The agent also must have a goal or goals relating to the state of the environment. Markov decision processes are intended to include just these three aspects—sensation, action, and goal—in their simplest possible forms without trivializing any of them. Any method that is well suited to solving such problems we consider to be a reinforcement learning method.

Reinforcement learning is different from *supervised learning*, the kind of learning studied in most current research in the field of machine learning. Supervised learning is learning from a training set of labeled examples provided by a knowledgeable external supervisor. Each example is a description of a situation together with a specification—the label—of the correct action the system should take in that situation, which is often to identify a category to which the situation belongs. The object of this kind of learning is for the system to extrapolate, or generalize, its responses so that it acts correctly in situations not present in the training set. This is an important kind of learning, but alone it is not adequate for learning from interaction. In interactive problems it is often impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act. In uncharted territory—where one would expect learning to be most beneficial—an agent must be able to learn from its own experience.

Reinforcement learning is also different from what machine learning researchers call *unsupervised learning*, which is typically about finding structure hidden in collections of unlabeled data. The terms supervised learning and unsupervised learning would seem to exhaustively classify machine learning paradigms, but they do not. Although one might be tempted to think of reinforcement learning as a kind of unsupervised learning because it does not rely on examples of correct behavior, reinforcement learning is trying to maximize a reward signal instead of trying to find hidden structure. Uncovering structure in an agent’s experience can certainly be useful in reinforcement learning, but by itself does not address the reinforcement learning problem of maximizing a reward signal. We therefore consider reinforcement learning to be a third machine learning paradigm, alongside supervised learning and unsupervised learning and perhaps other paradigms.

One of the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-off between exploration and exploitation. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to *exploit* what it has already experienced in order to obtain reward, but it also has to *explore* in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions *and* progressively favor those that appear to be best. On a stochastic task, each action must be tried many times to gain a reliable estimate of its expected reward. The exploration–exploitation dilemma has been intensively studied by mathematicians for many decades, yet remains unresolved. For now, we simply note that the entire issue of balancing exploration and exploitation does not even arise in supervised and unsupervised learning, at least in the purest forms of these paradigms.

Another key feature of reinforcement learning is that it explicitly considers the *whole* problem of a goal-directed agent interacting with an uncertain environment. This is in contrast to many approaches that consider subproblems without addressing how they might fit into a larger picture. For example, we have mentioned that many machine learning researchers have studied supervised learning without specifying how such an ability would ultimately be useful. Other researchers have developed theories of planning with general goals, but without considering planning’s role in real-time decision making, or the question of where the predictive models necessary for planning would come from. Although these approaches have yielded many useful results, their focus on isolated subproblems is a significant limitation.

Reinforcement learning takes the opposite tack, starting with a complete, interactive, goal-seeking agent. All reinforcement learning agents have explicit goals, can sense aspects of their environments, and can choose actions to influence their environments. Moreover, it is usually assumed from the beginning that the agent has to operate despite significant uncertainty about the environment it faces. When reinforcement learning involves planning, it has to address the interplay between planning and real-time action selection, as well as the question of how environment models are acquired and improved. When reinforcement learning involves supervised learning, it does so for specific reasons that determine which capabilities are critical and which are not. For learning research to make progress, important subproblems have to be isolated and studied, but they should be subproblems that play clear roles in complete, interactive, goal-seeking agents, even if all the details of the complete agent cannot yet be filled in.

By a complete, interactive, goal-seeking agent we do not always mean something like a complete organism or robot. These are clearly examples, but a complete, interactive, goal-seeking agent can also be a component of a larger behaving system. In this case, the agent directly interacts with the rest of the larger system and indirectly interacts with the larger system’s environment. A simple example is an agent that monitors the charge level of robot’s battery and sends commands to the robot’s control architecture. This agent’s environment is the rest of the robot together with the robot’s environment. It is

important to look beyond the most obvious examples of agents and their environments to appreciate the generality of the reinforcement learning framework.

One of the most exciting aspects of modern reinforcement learning is its substantive and fruitful interactions with other engineering and scientific disciplines. Reinforcement learning is part of a decades-long trend within artificial intelligence and machine learning toward greater integration with statistics, optimization, and other mathematical subjects. For example, the ability of some reinforcement learning methods to learn with parameterized approximators addresses the classical “curse of dimensionality” in operations research and control theory. More distinctively, reinforcement learning has also interacted strongly with psychology and neuroscience, with substantial benefits going both ways. Of all the forms of machine learning, reinforcement learning is the closest to the kind of learning that humans and other animals do, and many of the core algorithms of reinforcement learning were originally inspired by biological learning systems. Reinforcement learning has also given back, both through a psychological model of animal learning that better matches some of the empirical data, and through an influential model of parts of the brain’s reward system. The body of this book develops the ideas of reinforcement learning that pertain to engineering and artificial intelligence, with connections to psychology and neuroscience summarized in Chapters 14 and 15.

Finally, reinforcement learning is also part of a larger trend in artificial intelligence back toward simple general principles. Since the late 1960s, many artificial intelligence researchers presumed that there are no general principles to be discovered, that intelligence is instead due to the possession of a vast number of special purpose tricks, procedures, and heuristics. It was sometimes said that if we could just get enough relevant facts into a machine, say one million, or one billion, then it would become intelligent. Methods based on general principles, such as search or learning, were characterized as “weak methods,” whereas those based on specific knowledge were called “strong methods.” This view is uncommon today. From our point of view, it was premature: too little effort had been put into the search for general principles to conclude that there were none. Modern artificial intelligence now includes much research looking for general principles of learning, search, and decision making. It is not clear how far back the pendulum will swing, but reinforcement learning research is certainly part of the swing back toward simpler and fewer general principles of artificial intelligence.

1.2 Examples

A good way to understand reinforcement learning is to consider some of the examples and possible applications that have guided its development.

- A master chess player makes a move. The choice is informed both by planning—anticipating possible replies and counterreplies—and by immediate, intuitive judgments of the desirability of particular positions and moves.
- An adaptive controller adjusts parameters of a petroleum refinery’s operation in real time. The controller optimizes the yield/cost/quality trade-off on the basis

of specified marginal costs without sticking strictly to the set points originally suggested by engineers.

- A gazelle calf struggles to its feet minutes after being born. Half an hour later it is running at 20 miles per hour.
- A mobile robot decides whether it should enter a new room in search of more trash to collect or start trying to find its way back to its battery recharging station. It makes its decision based on the current charge level of its battery and how quickly and easily it has been able to find the recharger in the past.
- Phil prepares his breakfast. Closely examined, even this apparently mundane activity reveals a complex web of conditional behavior and interlocking goal–subgoal relationships: walking to the cupboard, opening it, selecting a cereal box, then reaching for, grasping, and retrieving the box. Other complex, tuned, interactive sequences of behavior are required to obtain a bowl, spoon, and milk carton. Each step involves a series of eye movements to obtain information and to guide reaching and locomotion. Rapid judgments are continually made about how to carry the objects or whether it is better to ferry some of them to the dining table before obtaining others. Each step is guided by goals, such as grasping a spoon or getting to the refrigerator, and is in service of other goals, such as having the spoon to eat with once the cereal is prepared and ultimately obtaining nourishment. Whether he is aware of it or not, Phil is accessing information about the state of his body that determines his nutritional needs, level of hunger, and food preferences.

These examples share features that are so basic that they are easy to overlook. All involve *interaction* between an active decision-making agent and its environment, within which the agent seeks to achieve a *goal* despite *uncertainty* about its environment. The agent’s actions are permitted to affect the future state of the environment (e.g., the next chess position, the level of reservoirs of the refinery, the robot’s next location and the future charge level of its battery), thereby affecting the actions and opportunities available to the agent at later times. Correct choice requires taking into account indirect, delayed consequences of actions, and thus may require foresight or planning.

At the same time, in all of these examples the effects of actions cannot be fully predicted; thus the agent must monitor its environment frequently and react appropriately. For example, Phil must watch the milk he pours into his cereal bowl to keep it from overflowing. All these examples involve goals that are explicit in the sense that the agent can judge progress toward its goal based on what it can sense directly. The chess player knows whether or not he wins, the refinery controller knows how much petroleum is being produced, the gazelle calf knows when it falls, the mobile robot knows when its batteries run down, and Phil knows whether or not he is enjoying his breakfast.

In all of these examples the agent can use its experience to improve its performance over time. The chess player refines the intuition he uses to evaluate positions, thereby improving his play; the gazelle calf improves the efficiency with which it can run; Phil learns to streamline making his breakfast. The knowledge the agent brings to the task at the start—either from previous experience with related tasks or built into it by design or

evolution— influences what is useful or easy to learn, but interaction with the environment is essential for adjusting behavior to exploit specific features of the task.

1.3 Elements of Reinforcement Learning

Beyond the agent and the environment, one can identify four main subelements of a reinforcement learning system: a *policy*, a *reward signal*, a *value function*, and, optionally, a *model* of the environment.

A *policy* defines the learning agent’s way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. It corresponds to what in psychology would be called a set of stimulus–response rules or associations. In some cases the policy may be a simple function or lookup table, whereas in others it may involve extensive computation such as a search process. The policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior. In general, policies may be stochastic, specifying probabilities for each action.

A *reward signal* defines the goal of a reinforcement learning problem. On each time step, the environment sends to the reinforcement learning agent a single number called the *reward*. The agent’s sole objective is to maximize the total reward it receives over the long run. The reward signal thus defines what are the good and bad events for the agent. In a biological system, we might think of rewards as analogous to the experiences of pleasure or pain. They are the immediate and defining features of the problem faced by the agent. The reward signal is the primary basis for altering the policy; if an action selected by the policy is followed by low reward, then the policy may be changed to select some other action in that situation in the future. In general, reward signals may be stochastic functions of the state of the environment and the actions taken.

Whereas the reward signal indicates what is good in an immediate sense, a *value function* specifies what is good in the long run. Roughly speaking, the *value* of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state. Whereas rewards determine the immediate, intrinsic desirability of environmental states, values indicate the *long-term* desirability of states after taking into account the states that are likely to follow and the rewards available in those states. For example, a state might always yield a low immediate reward but still have a high value because it is regularly followed by other states that yield high rewards. Or the reverse could be true. To make a human analogy, rewards are somewhat like pleasure (if high) and pain (if low), whereas values correspond to a more refined and farsighted judgment of how pleased or displeased we are that our environment is in a particular state.

Rewards are in a sense primary, whereas values, as predictions of rewards, are secondary. Without rewards there could be no values, and the only purpose of estimating values is to achieve more reward. Nevertheless, it is values with which we are most concerned when making and evaluating decisions. Action choices are made based on value judgments. We seek actions that bring about states of highest value, not highest reward, because these actions obtain the greatest amount of reward for us over the long run. Unfortunately, it is much harder to determine values than it is to determine rewards. Rewards are basically given directly by the environment, but values must be estimated and re-estimated from

the sequences of observations an agent makes over its entire lifetime. In fact, the most important component of almost all reinforcement learning algorithms we consider is a method for efficiently estimating values. The central role of value estimation is arguably the most important thing that has been learned about reinforcement learning over the last six decades.

The fourth and final element of some reinforcement learning systems is a *model* of the environment. This is something that mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave. For example, given a state and action, the model might predict the resultant next state and next reward. Models are used for *planning*, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced. Methods for solving reinforcement learning problems that use models and planning are called *model-based* methods, as opposed to simpler *model-free* methods that are explicitly trial-and-error learners—viewed as almost the *opposite* of planning. In Chapter 8 we explore reinforcement learning systems that simultaneously learn by trial and error, learn a model of the environment, and use the model for planning. Modern reinforcement learning spans the spectrum from low-level, trial-and-error learning to high-level, deliberative planning.

1.4 Limitations and Scope

Reinforcement learning relies heavily on the concept of state—as input to the policy and value function, and as both input to and output from the model. Informally, we can think of the state as a signal conveying to the agent some sense of “how the environment is” at a particular time. The formal definition of state as we use it here is given by the framework of Markov decision processes presented in Chapter 3. More generally, however, we encourage the reader to follow the informal meaning and think of the state as whatever information is available to the agent about its environment. In effect, we assume that the state signal is produced by some preprocessing system that is nominally part of the agent’s environment. We do not address the issues of constructing, changing, or learning the state signal in this book (other than briefly in Section 17.3). We take this approach not because we consider state representation to be unimportant, but in order to focus fully on the decision-making issues. In other words, our concern in this book is not with designing the state signal, but with deciding what action to take as a function of whatever state signal is available.

Most of the reinforcement learning methods we consider in this book are structured around estimating value functions, but it is not strictly necessary to do this to solve reinforcement learning problems. For example, solution methods such as genetic algorithms, genetic programming, simulated annealing, and other optimization methods never estimate value functions. These methods apply multiple static policies each interacting over an extended period of time with a separate instance of the environment. The policies that obtain the most reward, and random variations of them, are carried over to the next generation of policies, and the process repeats. We call these *evolutionary* methods because their operation is analogous to the way biological evolution produces organisms

with skilled behavior even if they do not learn during their individual lifetimes. If the space of policies is sufficiently small, or can be structured so that good policies are common or easy to find—or if a lot of time is available for the search—then evolutionary methods can be effective. In addition, evolutionary methods have advantages on problems in which the learning agent cannot sense the complete state of its environment.

Our focus is on reinforcement learning methods that learn while interacting with the environment, which evolutionary methods do not do. Methods able to take advantage of the details of individual behavioral interactions can be much more efficient than evolutionary methods in many cases. Evolutionary methods ignore much of the useful structure of the reinforcement learning problem: they do not use the fact that the policy they are searching for is a function from states to actions; they do not notice which states an individual passes through during its lifetime, or which actions it selects. In some cases such information can be misleading (e.g., when states are misperceived), but more often it should enable more efficient search. Although evolution and learning share many features and naturally work together, we do not consider evolutionary methods by themselves to be especially well suited to reinforcement learning problems and, accordingly, we do not cover them in this book.

1.5 An Extended Example: Tic-Tac-Toe

To illustrate the general idea of reinforcement learning and contrast it with other approaches, we next consider a single example in more detail.

Consider the familiar child’s game of tic-tac-toe. Two players take turns playing on a three-by-three board. One player plays Xs and the other Os until one player wins by placing three marks in a row, horizontally, vertically, or diagonally, as the X player has in the game shown to the right. If the board fills up with neither player getting three in a row, then the game is a draw. Because a skilled player can play so as never to lose, let us assume that we are playing against an imperfect player, one whose play is sometimes incorrect and allows us to win. For the moment, in fact, let us consider draws and losses to be equally bad for us. How might we construct a player that will find the imperfections in its opponent’s play and learn to maximize its chances of winning?

X	O	O
O	X	X
		X

Although this is a simple problem, it cannot readily be solved in a satisfactory way through classical techniques. For example, the classical “minimax” solution from game theory is not correct here because it assumes a particular way of playing by the opponent. For example, a minimax player would never reach a game state from which it could lose, even if in fact it always won from that state because of incorrect play by the opponent. Classical optimization methods for sequential decision problems, such as dynamic programming, can *compute* an optimal solution for any opponent, but require as input a complete specification of that opponent, including the probabilities with which the opponent makes each move in each board state. Let us assume that this information is not available *a priori* for this problem, as it is not for the vast majority of problems of

practical interest. On the other hand, such information can be estimated from experience, in this case by playing many games against the opponent. About the best one can do on this problem is first to learn a model of the opponent’s behavior, up to some level of confidence, and then apply dynamic programming to compute an optimal solution given the approximate opponent model. In the end, this is not that different from some of the reinforcement learning methods we examine later in this book.

An evolutionary method applied to this problem would directly search the space of possible policies for one with a high probability of winning against the opponent. Here, a policy is a rule that tells the player what move to make for every state of the game—every possible configuration of Xs and Os on the three-by-three board. For each policy considered, an estimate of its winning probability would be obtained by playing some number of games against the opponent. This evaluation would then direct which policy or policies were considered next. A typical evolutionary method would hill-climb in policy space, successively generating and evaluating policies in an attempt to obtain incremental improvements. Or, perhaps, a genetic-style algorithm could be used that would maintain and evaluate a population of policies. Literally hundreds of different optimization methods could be applied.

Here is how the tic-tac-toe problem would be approached with a method making use of a value function. First we would set up a table of numbers, one for each possible state of the game. Each number will be the latest estimate of the probability of our winning from that state. We treat this estimate as the state’s *value*, and the whole table is the learned value function. State A has higher value than state B, or is considered “better” than state B, if the current estimate of the probability of our winning from A is higher than it is from B. Assuming we always play Xs, then for all states with three Xs in a row the probability of winning is 1, because we have already won. Similarly, for all states with three Os in a row, or that are filled up, the correct probability is 0, as we cannot win from them. We set the initial values of all the other states to 0.5, representing a guess that we have a 50% chance of winning.

We then play many games against the opponent. To select our moves we examine the states that would result from each of our possible moves (one for each blank space on the board) and look up their current values in the table. Most of the time we move *greedily*, selecting the move that leads to the state with greatest value, that is, with the highest estimated probability of winning. Occasionally, however, we select randomly from among the other moves instead. These are called *exploratory* moves because they cause us to experience states that we might otherwise never see. A sequence of moves made and considered during a game can be diagrammed as in Figure 1.1.

While we are playing, we change the values of the states in which we find ourselves during the game. We attempt to make them more accurate estimates of the probabilities of winning. To do this, we “back up” the value of the state after each greedy move to the state before the move, as suggested by the arrows in Figure 1.1. More precisely, the current value of the earlier state is updated to be closer to the value of the later state. This can be done by moving the earlier state’s value a fraction of the way toward the value of the later state. If we let S_t denote the state before the greedy move, and S_{t+1} the state after that move, then the update to the estimated value of S_t , denoted $V(S_t)$,

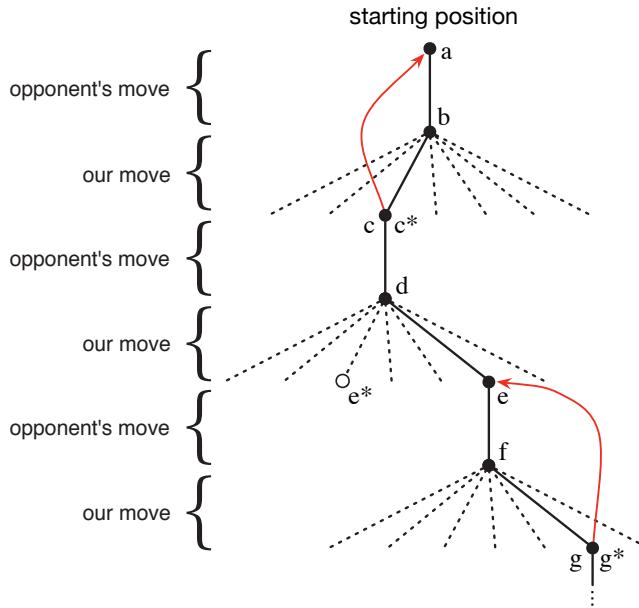


Figure 1.1: A sequence of tic-tac-toe moves. The solid black lines represent the moves taken during a game; the dashed lines represent moves that we (our reinforcement learning player) considered but did not make. The * indicates the move currently estimated to be the best. Our second move was an exploratory move, meaning that it was taken even though another sibling move, the one leading to e^* , was ranked higher. Exploratory moves do not result in any learning, but each of our other moves does, causing updates as suggested by the red arrows in which estimated values are moved up the tree from later nodes to earlier nodes as detailed in the text.

can be written as

$$V(S_t) \leftarrow V(S_t) + \alpha [V(S_{t+1}) - V(S_t)],$$

where α is a small positive fraction called the *step-size parameter*, which influences the rate of learning. This update rule is an example of a *temporal-difference* learning method, so called because its changes are based on a difference, $V(S_{t+1}) - V(S_t)$, between estimates at two successive times.

The method described above performs quite well on this task. For example, if the step-size parameter is reduced properly over time, then this method converges, for any fixed opponent, to the true probabilities of winning from each state given optimal play by our player. Furthermore, the moves then taken (except on exploratory moves) are in fact the optimal moves against this (imperfect) opponent. In other words, the method converges to an optimal policy for playing the game against this opponent. If the step-size parameter is not reduced all the way to zero over time, then this player also plays well against opponents that slowly change their way of playing.

This example illustrates the differences between evolutionary methods and methods that learn value functions. To evaluate a policy, an evolutionary method holds the policy fixed and plays many games against the opponent or simulates many games using a model of the opponent. The frequency of wins gives an unbiased estimate of the probability of winning with that policy, and can be used to direct the next policy selection. But each policy change is made only after many games, and only the final outcome of each game is used: what happens *during* the games is ignored. For example, if the player wins, then *all* of its behavior in the game is given credit, independently of how specific moves might have been critical to the win. Credit is even given to moves that never occurred! Value function methods, in contrast, allow individual states to be evaluated. In the end, evolutionary and value function methods both search the space of policies, but learning a value function takes advantage of information available during the course of play.

This simple example illustrates some of the key features of reinforcement learning methods. First, there is the emphasis on learning while interacting with an environment, in this case with an opponent player. Second, there is a clear goal, and correct behavior requires planning or foresight that takes into account delayed effects of one's choices. For example, the simple reinforcement learning player would learn to set up multi-move traps for a shortsighted opponent. It is a striking feature of the reinforcement learning solution that it can achieve the effects of planning and lookahead without using a model of the opponent and without conducting an explicit search over possible sequences of future states and actions.

While this example illustrates some of the key features of reinforcement learning, it is so simple that it might give the impression that reinforcement learning is more limited than it really is. Although tic-tac-toe is a two-person game, reinforcement learning also applies in the case in which there is no external adversary, that is, in the case of a “game against nature.” Reinforcement learning also is not restricted to problems in which behavior breaks down into separate episodes, like the separate games of tic-tac-toe, with reward only at the end of each episode. It is just as applicable when behavior continues indefinitely and when rewards of various magnitudes can be received at any time. Reinforcement learning is also applicable to problems that do not even break down into discrete time steps like the plays of tic-tac-toe. The general principles apply to continuous-time problems as well, although the theory gets more complicated and we omit it from this introductory treatment.

Tic-tac-toe has a relatively small, finite state set, whereas reinforcement learning can be used when the state set is very large, or even infinite. For example, Gerry Tesauro (1992, 1995) combined the algorithm described above with an artificial neural network to learn to play backgammon, which has approximately 10^{20} states. With this many states it is impossible ever to experience more than a small fraction of them. Tesauro’s program learned to play far better than any previous program and eventually better than the world’s best human players (see Section 16.1). The artificial neural network provides the program with the ability to generalize from its experience, so that in new states it selects moves based on information saved from similar states faced in the past, as determined by the network. How well a reinforcement learning system can work in problems with such large state sets is intimately tied to how appropriately it can generalize from past

experience. It is in this role that we have the greatest need for supervised learning methods within reinforcement learning. Artificial neural networks and deep learning (Section 9.7) are not the only, or necessarily the best, way to do this.

In this tic-tac-toe example, learning started with no prior knowledge beyond the rules of the game, but reinforcement learning by no means entails a tabula rasa view of learning and intelligence. On the contrary, prior information can be incorporated into reinforcement learning in a variety of ways that can be critical for efficient learning (e.g., see Sections 9.5, 17.4, and 13.1). We also have access to the true state in the tic-tac-toe example, whereas reinforcement learning can also be applied when part of the state is hidden, or when different states appear to the learner to be the same.

Finally, the tic-tac-toe player was able to look ahead and know the states that would result from each of its possible moves. To do this, it had to have a model of the game that allowed it to foresee how its environment would change in response to moves that it might never make. Many problems are like this, but in others even a short-term model of the effects of actions is lacking. Reinforcement learning can be applied in either case. A model is not required, but models can easily be used if they are available or can be learned (Chapter 8).

On the other hand, there are reinforcement learning methods that do not need any kind of environment model at all. Model-free systems cannot even think about how their environments will change in response to a single action. The tic-tac-toe player is model-free in this sense with respect to its opponent: it has no model of its opponent of any kind. Because models have to be reasonably accurate to be useful, model-free methods can have advantages over more complex methods when the real bottleneck in solving a problem is the difficulty of constructing a sufficiently accurate environment model. Model-free methods are also important building blocks for model-based methods. In this book we devote several chapters to model-free methods before we discuss how they can be used as components of more complex model-based methods.

Reinforcement learning can be used at both high and low levels in a system. Although the tic-tac-toe player learned only about the basic moves of the game, nothing prevents reinforcement learning from working at higher levels where each of the “actions” may itself be the application of a possibly elaborate problem-solving method. In hierarchical learning systems, reinforcement learning can work simultaneously on several levels.

Exercise 1.1: Self-Play Suppose, instead of playing against a random opponent, the reinforcement learning algorithm described above played against itself, with both sides learning. What do you think would happen in this case? Would it learn a different policy for selecting moves? □

Exercise 1.2: Symmetries Many tic-tac-toe positions appear different but are really the same because of symmetries. How might we amend the learning process described above to take advantage of this? In what ways would this change improve the learning process? Now think again. Suppose the opponent did not take advantage of symmetries. In that case, should we? Is it true, then, that symmetrically equivalent positions should necessarily have the same value? □

Exercise 1.3: Greedy Play Suppose the reinforcement learning player was *greedy*, that is, it always played the move that brought it to the position that it rated the best. Might it

learn to play better, or worse, than a nongreedy player? What problems might occur? \square

Exercise 1.4: Learning from Exploration Suppose learning updates occurred after *all* moves, including exploratory moves. If the step-size parameter is appropriately reduced over time (but not the tendency to explore), then the state values would converge to a different set of probabilities. What (conceptually) are the two sets of probabilities computed when we do, and when we do not, learn from exploratory moves? Assuming that we do continue to make exploratory moves, which set of probabilities might be better to learn? Which would result in more wins? \square

Exercise 1.5: Other Improvements Can you think of other ways to improve the reinforcement learning player? Can you think of any better way to solve the tic-tac-toe problem as posed? \square

1.6 Summary

Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision making. It is distinguished from other computational approaches by its emphasis on learning by an agent from direct interaction with its environment, without requiring exemplary supervision or complete models of the environment. In our opinion, reinforcement learning is the first field to seriously address the computational issues that arise when learning from interaction with an environment in order to achieve long-term goals.

Reinforcement learning uses the formal framework of Markov decision processes to define the interaction between a learning agent and its environment in terms of states, actions, and rewards. This framework is intended to be a simple way of representing essential features of the artificial intelligence problem. These features include a sense of cause and effect, a sense of uncertainty and nondeterminism, and the existence of explicit goals.

The concepts of value and value function are key to most of the reinforcement learning methods that we consider in this book. We take the position that value functions are important for efficient search in the space of policies. The use of value functions distinguishes reinforcement learning methods from evolutionary methods that search directly in policy space guided by evaluations of entire policies.

1.7 Early History of Reinforcement Learning

The early history of reinforcement learning has two main threads, both long and rich, that were pursued independently before intertwining in modern reinforcement learning. One thread concerns learning by trial and error, and originated in the psychology of animal learning. This thread runs through some of the earliest work in artificial intelligence and led to the revival of reinforcement learning in the early 1980s. The second thread concerns the problem of optimal control and its solution using value functions and dynamic programming. For the most part, this thread did not involve learning. The two threads were mostly independent, but became interrelated to some extent around a

third, less distinct thread concerning temporal-difference methods such as that used in the tic-tac-toe example in this chapter. All three threads came together in the late 1980s to produce the modern field of reinforcement learning as we present it in this book.

The thread focusing on trial-and-error learning is the one with which we are most familiar and about which we have the most to say in this brief history. Before doing that, however, we briefly discuss the optimal control thread.

The term “optimal control” came into use in the late 1950s to describe the problem of designing a controller to minimize or maximize a measure of a dynamical system’s behavior over time. One of the approaches to this problem was developed in the mid-1950s by Richard Bellman and others through extending a nineteenth century theory of Hamilton and Jacobi. This approach uses the concepts of a dynamical system’s state and of a value function, or “optimal return function,” to define a functional equation, now often called the Bellman equation. The class of methods for solving optimal control problems by solving this equation came to be known as dynamic programming (Bellman, 1957a). Bellman (1957b) also introduced the discrete stochastic version of the optimal control problem known as Markov decision processes (MDPs). Ronald Howard (1960) devised the policy iteration method for MDPs. All of these are essential elements underlying the theory and algorithms of modern reinforcement learning.

Dynamic programming is widely considered the only feasible way of solving general stochastic optimal control problems. It suffers from what Bellman called “the curse of dimensionality,” meaning that its computational requirements grow exponentially with the number of state variables, but it is still far more efficient and more widely applicable than any other general method. Dynamic programming has been extensively developed since the late 1950s, including extensions to partially observable MDPs (surveyed by Lovejoy, 1991), many applications (surveyed by White, 1985, 1988, 1993), approximation methods (surveyed by Rust, 1996), and asynchronous methods (Bertsekas, 1982, 1983). Many excellent modern treatments of dynamic programming are available (e.g., Bertsekas, 2005, 2012; Puterman, 1994; Ross, 1983; Whittle, 1982, 1983). Bryson (1996) provides an authoritative history of optimal control.

Connections between optimal control and dynamic programming, on the one hand, and learning, on the other, were slow to be recognized. We cannot be sure about what accounted for this separation, but its main cause was likely the separation between the disciplines involved and their different goals. Also contributing may have been the prevalent view of dynamic programming as an off-line computation depending essentially on accurate system models and analytic solutions to the Bellman equation. Further, the simplest form of dynamic programming is a computation that proceeds backwards in time, making it difficult to see how it could be involved in a learning process that must proceed in a forward direction. Some of the earliest work in dynamic programming, such as that by Bellman and Dreyfus (1959), might now be classified as following a learning approach. Witten’s (1977) work (discussed below) certainly qualifies as a combination of learning and dynamic-programming ideas. Werbos (1987) argued explicitly for greater interrelation of dynamic programming and learning methods and for dynamic programming’s relevance to understanding neural and cognitive mechanisms. For us the full integration of dynamic programming methods with online learning did not occur

until the work of Chris Watkins in 1989, whose treatment of reinforcement learning using the MDP formalism has been widely adopted. Since then these relationships have been extensively developed by many researchers, most particularly by Dimitri Bertsekas and John Tsitsiklis (1996), who coined the term “neurodynamic programming” to refer to the combination of dynamic programming and artificial neural networks. Another term currently in use is “approximate dynamic programming.” These various approaches emphasize different aspects of the subject, but they all share with reinforcement learning an interest in circumventing the classical shortcomings of dynamic programming.

We consider all of the work in optimal control also to be, in a sense, work in reinforcement learning. We define a reinforcement learning method as any effective way of solving reinforcement learning problems, and it is now clear that these problems are closely related to optimal control problems, particularly stochastic optimal control problems such as those formulated as MDPs. Accordingly, we must consider the solution methods of optimal control, such as dynamic programming, also to be reinforcement learning methods. Because almost all of the conventional methods require complete knowledge of the system to be controlled, it feels a little unnatural to say that they are part of reinforcement *learning*. On the other hand, many dynamic programming algorithms are incremental and iterative. Like learning methods, they gradually reach the correct answer through successive approximations. As we show in the rest of this book, these similarities are far more than superficial. The theories and solution methods for the cases of complete and incomplete knowledge are so closely related that we feel they must be considered together as part of the same subject matter.

Let us return now to the other major thread leading to the modern field of reinforcement learning, the thread centered on the idea of trial-and-error learning. We only touch on the major points of contact here, taking up this topic in more detail in Section 14.3. According to American psychologist R. S. Woodworth (1938) the idea of trial-and-error learning goes as far back as the 1850s to Alexander Bain’s discussion of learning by “groping and experiment” and more explicitly to the British ethologist and psychologist Conway Lloyd Morgan’s 1894 use of the term to describe his observations of animal behavior. Perhaps the first to succinctly express the essence of trial-and-error learning as a principle of learning was Edward Thorndike:

Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond. (Thorndike, 1911, p. 244)

Thorndike called this the “Law of Effect” because it describes the effect of reinforcing events on the tendency to select actions. Thorndike later modified the law to better account for subsequent data on animal learning (such as differences between the effects of reward and punishment), and the law in its various forms has generated considerable controversy among learning theorists (e.g., see Gallistel, 2005; Herrnstein, 1970; Kimble,

1961, 1967; Mazur, 1994). Despite this, the Law of Effect—in one form or another—is widely regarded as a basic principle underlying much behavior (e.g., Hilgard and Bower, 1975; Dennett, 1978; Campbell, 1960; Cziko, 1995). It is the basis of the influential learning theories of Clark Hull (1943, 1952) and the influential experimental methods of B. F. Skinner (1938).

The term “reinforcement” in the context of animal learning came into use well after Thorndike’s expression of the Law of Effect, first appearing in this context (to the best of our knowledge) in the 1927 English translation of Pavlov’s monograph on conditioned reflexes. Pavlov described reinforcement as the strengthening of a pattern of behavior due to an animal receiving a stimulus—a reinforcer—in an appropriate temporal relationship with another stimulus or with a response. Some psychologists extended the idea of reinforcement to include weakening as well as strengthening of behavior, and extended the idea of a reinforcer to include possibly the omission or termination of stimulus. To be considered a reinforcer, the strengthening or weakening must persist after the reinforcer is withdrawn; a stimulus that merely attracts an animal’s attention or that energizes its behavior without producing lasting changes would not be considered a reinforcer.

The idea of implementing trial-and-error learning in a computer appeared among the earliest thoughts about the possibility of artificial intelligence. In a 1948 report, Alan Turing described a design for a “pleasure-pain system” that worked along the lines of the Law of Effect:

When a configuration is reached for which the action is undetermined, a random choice for the missing data is made and the appropriate entry is made in the description, tentatively, and is applied. When a pain stimulus occurs all tentative entries are cancelled, and when a pleasure stimulus occurs they are all made permanent. (Turing, 1948)

Many ingenious electro-mechanical machines were constructed that demonstrated trial-and-error learning. The earliest may have been a machine built by Thomas Ross (1933) that was able to find its way through a simple maze and remember the path through the settings of switches. In 1951 W. Grey Walter built a version of his “mechanical tortoise” (Walter, 1950) capable of a simple form of learning. In 1952 Claude Shannon demonstrated a maze-running mouse named Theseus that used trial and error to find its way through a maze, with the maze itself remembering the successful directions via magnets and relays under its floor (see also Shannon, 1951). J. A. Deutsch (1954) described a maze-solving machine based on his behavior theory (Deutsch, 1953) that has some properties in common with model-based reinforcement learning (Chapter 8). In his PhD dissertation, Marvin Minsky (1954) discussed computational models of reinforcement learning and described his construction of an analog machine composed of components he called SNARCs (Stochastic Neural-Analog Reinforcement Calculators) meant to resemble modifiable synaptic connections in the brain (Chapter 15). The web site cyberneticzoo.com contains a wealth of information on these and many other electro-mechanical learning machines.

Building electro-mechanical learning machines gave way to programming digital computers to perform various types of learning, some of which implemented trial-and-error learning. Farley and Clark (1954) described a digital simulation of a neural-network

learning machine that learned by trial and error. But their interests soon shifted from trial-and-error learning to generalization and pattern recognition, that is, from reinforcement learning to supervised learning (Clark and Farley, 1955). This began a pattern of confusion about the relationship between these types of learning. Many researchers seemed to believe that they were studying reinforcement learning when they were actually studying supervised learning. For example, artificial neural network pioneers such as Rosenblatt (1962) and Widrow and Hoff (1960) were clearly motivated by reinforcement learning—they used the language of rewards and punishments—but the systems they studied were supervised learning systems suitable for pattern recognition and perceptual learning. Even today, some researchers and textbooks minimize or blur the distinction between these types of learning. For example, some textbooks have used the term “trial-and-error” to describe artificial neural networks that learn from training examples. This is an understandable confusion because these networks use error information to update connection weights, but this misses the essential character of trial-and-error learning as selecting actions on the basis of evaluative feedback that does not rely on knowledge of what the correct action should be.

Partly as a result of these confusions, research into genuine trial-and-error learning became rare in the 1960s and 1970s, although there were notable exceptions. In the 1960s the terms “reinforcement” and “reinforcement learning” were used in the engineering literature for the first time to describe engineering uses of trial-and-error learning (e.g., Waltz and Fu, 1965; Mendel, 1966; Fu, 1970; Mendel and McLaren, 1970). Particularly influential was Minsky’s paper “Steps Toward Artificial Intelligence” (Minsky, 1961), which discussed several issues relevant to trial-and-error learning, including prediction, expectation, and what he called the *basic credit-assignment problem for complex reinforcement learning systems*: How do you distribute credit for success among the many decisions that may have been involved in producing it? All of the methods we discuss in this book are, in a sense, directed toward solving this problem. Minsky’s paper is well worth reading today.

In the next few paragraphs we discuss some of the other exceptions and partial exceptions to the relative neglect of computational and theoretical study of genuine trial-and-error learning in the 1960s and 1970s.

One exception was the work of the New Zealand researcher John Andreae, who developed a system called STeLLA that learned by trial and error in interaction with its environment. This system included an internal model of the world and, later, an “internal monologue” to deal with problems of hidden state (Andreae, 1963, 1969; Andreae and Cashin, 1969). Andreae’s later work (1977) placed more emphasis on learning from a teacher, but still included learning by trial and error, with the generation of novel events being one of the system’s goals. A feature of this work was a “leakback process,” elaborated more fully in Andreae (1998), that implemented a credit-assignment mechanism similar to the backing-up update operations that we describe. Unfortunately, his pioneering research was not well known and did not greatly impact subsequent reinforcement learning research. Recent summaries are available (Andreae, 2017a,b).

More influential was the work of Donald Michie. In 1961 and 1963 he described a simple trial-and-error learning system for learning how to play tic-tac-toe (or naughts

and crosses) called MENACE (for Matchbox Educable Naughts and Crosses Engine). It consisted of a matchbox for each possible game position, each matchbox containing a number of colored beads, a different color for each possible move from that position. By drawing a bead at random from the matchbox corresponding to the current game position, one could determine MENACE's move. When a game was over, beads were added to or removed from the boxes used during play to reward or punish MENACE's decisions. Michie and Chambers (1968) described another tic-tac-toe reinforcement learner called GLEE (Game Learning Expectimaxing Engine) and a reinforcement learning controller called BOXES. They applied BOXES to the task of learning to balance a pole hinged to a movable cart on the basis of a failure signal occurring only when the pole fell or the cart reached the end of a track. This task was adapted from the earlier work of Widrow and Smith (1964), who used supervised learning methods, assuming instruction from a teacher already able to balance the pole. Michie and Chambers's version of pole-balancing is one of the best early examples of a reinforcement learning task under conditions of incomplete knowledge. It influenced much later work in reinforcement learning, beginning with some of our own studies (Barto, Sutton, and Anderson, 1983; Sutton, 1984). Michie (1974) consistently emphasized trial and error and learning as essential aspects of artificial intelligence.

Widrow, Gupta, and Maitra (1973) modified the Least-Mean-Square (LMS) algorithm of Widrow and Hoff (1960) to produce a reinforcement learning rule that could learn from success and failure signals instead of from training examples. They called this form of learning "selective bootstrap adaptation" and described it as "learning with a critic" instead of "learning with a teacher." They analyzed this rule and showed how it could learn to play blackjack. This was an isolated foray into reinforcement learning by Widrow, whose contributions to supervised learning were much more influential. Our use of the term "critic" is derived from Widrow, Gupta, and Maitra's paper. Buchanan, Mitchell, Smith, and Johnson (1978) independently used the term critic in the context of machine learning (see also Dietterich and Buchanan, 1984), but for them a critic was an expert system able to do more than evaluate performance.

Research on *learning automata* had a more direct influence on the trial-and-error thread leading to modern reinforcement learning research. These are methods for solving a nonassociative, purely selectional learning problem known as the *k*-armed bandit by analogy to a slot machine, or "one-armed bandit," except with *k* levers (see Chapter 2). Learning automata are simple, low-memory machines for improving the probability of reward in these problems. Learning automata originated with work in the 1960s of the Russian mathematician and physicist M. L. Tsetlin and colleagues (published posthumously in Tsetlin, 1973) and has been extensively developed since then within engineering (see Narendra and Thathachar, 1974, 1989). These developments included the study of *stochastic learning automata*, which are methods for updating action probabilities on the basis of reward signals. Although not developed in the tradition of stochastic learning automata, Harth and Tzanakou's (1974) Alopex algorithm (for *Algorithm of pattern extraction*) is a stochastic method for detecting correlations between actions and reinforcement that influenced some of our early research (Barto, Sutton, and Brouwer, 1981). Stochastic learning automata were foreshadowed by earlier work in psychology, beginning with William Estes' (1950) effort toward a statistical theory of learning and further developed by others (e.g., Bush and Mosteller, 1955; Sternberg, 1963).

The statistical learning theories developed in psychology were adopted by researchers in economics, leading to a thread of research in that field devoted to reinforcement learning. This work began in 1973 with the application of Bush and Mosteller’s learning theory to a collection of classical economic models (Cross, 1973). One goal of this research was to study artificial agents that act more like real people than do traditional idealized economic agents (Arthur, 1991). This approach expanded to the study of reinforcement learning in the context of game theory. Reinforcement learning in economics developed largely independently of the early work in reinforcement learning in artificial intelligence, though game theory remains a topic of interest in both fields (beyond the scope of this book). Camerer (2011) discusses the reinforcement learning tradition in economics, and Nowé, Vrancx, and De Hauwere (2012) provide an overview of the subject from the point of view of multi-agent extensions to the approach that we introduce in this book. Reinforcement learning in the context of game theory is a much different subject than reinforcement learning used in programs to play tic-tac-toe, checkers, and other recreational games. See, for example, Szita (2012) for an overview of this aspect of reinforcement learning and games.

John Holland (1975) outlined a general theory of adaptive systems based on selectional principles. His early work concerned trial and error primarily in its nonassociative form, as in evolutionary methods and the *k*-armed bandit. In 1976 and more fully in 1986, he introduced *classifier systems*, true reinforcement learning systems including association and value functions. A key component of Holland’s classifier systems was the “bucket-brigade algorithm” for credit assignment, which is closely related to the temporal difference algorithm used in our tic-tac-toe example and discussed in Chapter 6. Another key component was a *genetic algorithm*, an evolutionary method whose role was to evolve useful representations. Classifier systems have been extensively developed by many researchers to form a major branch of reinforcement learning research (reviewed by Urbanowicz and Moore, 2009), but genetic algorithms—which we do not consider to be reinforcement learning systems by themselves—have received much more attention, as have other approaches to evolutionary computation (e.g., Fogel, Owens and Walsh, 1966; Koza, 1992).

The individual most responsible for reviving the trial-and-error thread of reinforcement learning within artificial intelligence was Harry Klop (1972, 1975, 1982). Klop recognized that essential aspects of adaptive behavior were being lost as learning researchers came to focus almost exclusively on supervised learning. What was missing, according to Klop, were the hedonic aspects of behavior: the drive to achieve some result from the environment, to control the environment toward desired ends and away from undesired ends (see Section 15.9). This is the essential idea of trial-and-error learning. Klop’s ideas were especially influential on the authors because our assessment of them (Barto and Sutton, 1981a) led to our appreciation of the distinction between supervised and reinforcement learning, and to our eventual focus on reinforcement learning. Much of the early work that we and colleagues accomplished was directed toward showing that reinforcement learning and supervised learning were indeed different (Barto, Sutton, and Brouwer, 1981; Barto and Sutton, 1981b; Barto and Anandan, 1985). Other studies showed how reinforcement learning could address important problems in artificial neural

network learning, in particular, how it could produce learning algorithms for multilayer networks (Barto, Anderson, and Sutton, 1982; Barto and Anderson, 1985; Barto, 1985, 1986; Barto and Jordan, 1987; see Section 15.10).

We turn now to the third thread to the history of reinforcement learning, that concerning temporal-difference learning. Temporal-difference learning methods are distinctive in being driven by the difference between temporally successive estimates of the same quantity—for example, of the probability of winning in the tic-tac-toe example. This thread is smaller and less distinct than the other two, but it has played a particularly important role in the field, in part because temporal-difference methods seem to be new and unique to reinforcement learning.

The origins of temporal-difference learning are in part in animal learning psychology, in particular, in the notion of *secondary reinforcers*. A secondary reinforcer is a stimulus that has been paired with a primary reinforcer such as food or pain and, as a result, has come to take on similar reinforcing properties. Minsky (1954) may have been the first to realize that this psychological principle could be important for artificial learning systems. Arthur Samuel (1959) was the first to propose and implement a learning method that included temporal-difference ideas, as part of his celebrated checkers-playing program (Section 16.2).

Samuel made no reference to Minsky's work or to possible connections to animal learning. His inspiration apparently came from Claude Shannon's (1950) suggestion that a computer could be programmed to use an evaluation function to play chess, and that it might be able to improve its play by modifying this function online. (It is possible that these ideas of Shannon's also influenced Bellman, but we know of no evidence for this.) Minsky (1961) extensively discussed Samuel's work in his "Steps" paper, suggesting the connection to secondary reinforcement theories, both natural and artificial.

As we have discussed, in the decade following the work of Minsky and Samuel, little computational work was done on trial-and-error learning, and apparently no computational work at all was done on temporal-difference learning. In 1972, Klopf brought trial-and-error learning together with an important component of temporal-difference learning. Klopf was interested in principles that would scale to learning in large systems, and thus was intrigued by notions of local reinforcement, whereby subcomponents of an overall learning system could reinforce one another. He developed the idea of "generalized reinforcement," whereby every component (nominally, every neuron) views all of its inputs in reinforcement terms: excitatory inputs as rewards and inhibitory inputs as punishments. This is not the same idea as what we now know as temporal-difference learning, and in retrospect it is farther from it than was Samuel's work. On the other hand, Klopf linked the idea with trial-and-error learning and related it to the massive empirical database of animal learning psychology.

Sutton (1978a,b,c) developed Klopf's ideas further, particularly the links to animal learning theories, describing learning rules driven by changes in temporally successive predictions. He and Barto refined these ideas and developed a psychological model of classical conditioning based on temporal-difference learning (Sutton and Barto, 1981a; Barto and Sutton, 1982). There followed several other influential psychological models of classical conditioning based on temporal-difference learning (e.g., Klopf, 1988; Moore et al.,

1986; Sutton and Barto, 1987, 1990). Some neuroscience models developed at this time are well interpreted in terms of temporal-difference learning (Hawkins and Kandel, 1984; Byrne, Gingrich, and Baxter, 1990; Gelperin, Hopfield, and Tank, 1985; Tesauro, 1986; Friston et al., 1994), although in most cases there was no historical connection.

Our early work on temporal-difference learning was strongly influenced by animal learning theories and by Klopf's work. Relationships to Minsky's "Steps" paper and to Samuel's checkers players were recognized only afterward. By 1981, however, we were fully aware of all the prior work mentioned above as part of the temporal-difference and trial-and-error threads. At this time we developed a method for using temporal-difference learning combined with trial-and-error learning, known as the *actor–critic architecture*, and applied this method to Michie and Chambers's pole-balancing problem (Barto, Sutton, and Anderson, 1983). This method was extensively studied in Sutton's (1984) PhD dissertation and extended to use backpropagation neural networks in Anderson's (1986) PhD dissertation. Around this time, Holland (1986) incorporated temporal-difference ideas explicitly into his classifier systems in the form of his bucket-brigade algorithm. A key step was taken by Sutton (1988) by separating temporal-difference learning from control, treating it as a general prediction method. That paper also introduced the $\text{TD}(\lambda)$ algorithm and proved some of its convergence properties.

As we were finalizing our work on the actor–critic architecture in 1981, we discovered a paper by Ian Witten (1977, 1976a) which appears to be the earliest publication of a temporal-difference learning rule. He proposed the method that we now call tabular $\text{TD}(0)$ for use as part of an adaptive controller for solving MDPs. This work was first submitted for journal publication in 1974 and also appeared in Witten's 1976 PhD dissertation. Witten's work was a descendant of Andreae's early experiments with STeLLA and other trial-and-error learning systems. Thus, Witten's 1977 paper spanned both major threads of reinforcement learning research—trial-and-error learning and optimal control—while making a distinct early contribution to temporal-difference learning.

The temporal-difference and optimal control threads were fully brought together in 1989 with Chris Watkins's development of Q-learning. This work extended and integrated prior work in all three threads of reinforcement learning research. Paul Werbos (1987) contributed to this integration by arguing for the convergence of trial-and-error learning and dynamic programming since 1977. By the time of Watkins's work there had been tremendous growth in reinforcement learning research, primarily in the machine learning subfield of artificial intelligence, but also in artificial neural networks and artificial intelligence more broadly. In 1992, the remarkable success of Gerry Tesauro's backgammon playing program, TD-Gammon, brought additional attention to the field.

In the time since publication of the first edition of this book, a flourishing subfield of neuroscience developed that focuses on the relationship between reinforcement learning algorithms and reinforcement learning in the nervous system. Most responsible for this is an uncanny similarity between the behavior of temporal-difference algorithms and the activity of dopamine producing neurons in the brain, as pointed out by a number of researchers (Friston et al., 1994; Barto, 1995a; Houk, Adams, and Barto, 1995; Montague, Dayan, and Sejnowski, 1996; and Schultz, Dayan, and Montague, 1997). Chapter 15 provides an introduction to this exciting aspect of reinforcement learning. Other important

contributions made in the recent history of reinforcement learning are too numerous to mention in this brief account; we cite many of these at the end of the individual chapters in which they arise.

Bibliographical Remarks

For additional general coverage of reinforcement learning, we refer the reader to the books by Szepesvari (2010), Bertsekas and Tsitsiklis (1996), Kaelbling (1993a), and Sugiyama, Hachiya, and Morimura (2013). Books that take a control or operations research perspective include those of Si, Barto, Powell, and Wunsch (2004), Powell (2011), Lewis and Liu (2012), and Bertsekas (2012). Cao’s (2009) review places reinforcement learning in the context of other approaches to learning and optimization of stochastic dynamic systems. Three special issues of the journal *Machine Learning* focus on reinforcement learning: Sutton (1992a), Kaelbling (1996), and Singh (2002). Useful surveys are provided by Barto (1995b); Kaelbling, Littman, and Moore (1996); and Keerthi and Ravindran (1997). The volume edited by Weiring and van Otterlo (2012) provides an excellent overview of recent developments.

- 1.2** The example of Phil’s breakfast in this chapter was inspired by Agre (1988).
- 1.5** The temporal-difference method used in the tic-tac-toe example is developed in Chapter 6.

Part I: Tabular Solution Methods

In this part of the book we describe almost all the core ideas of reinforcement learning algorithms in their simplest forms: that in which the state and action spaces are small enough for the approximate value functions to be represented as arrays, or *tables*. In this case, the methods can often find exact solutions, that is, they can often find exactly the optimal value function and the optimal policy. This contrasts with the approximate methods described in the next part of the book, which only find approximate solutions, but which in return can be applied effectively to much larger problems.

The first chapter of this part of the book describes solution methods for the special case of the reinforcement learning problem in which there is only a single state, called bandit problems. The second chapter describes the general problem formulation that we treat throughout the rest of the book—finite Markov decision processes—and its main ideas including Bellman equations and value functions.

The next three chapters describe three fundamental classes of methods for solving finite Markov decision problems: dynamic programming, Monte Carlo methods, and temporal-difference learning. Each class of methods has its strengths and weaknesses. Dynamic programming methods are well developed mathematically, but require a complete and accurate model of the environment. Monte Carlo methods don't require a model and are conceptually simple, but are not well suited for step-by-step incremental computation. Finally, temporal-difference methods require no model and are fully incremental, but are more complex to analyze. The methods also differ in several ways with respect to their efficiency and speed of convergence.

The remaining two chapters describe how these three classes of methods can be combined to obtain the best features of each of them. In one chapter we describe how the strengths of Monte Carlo methods can be combined with the strengths of temporal-difference methods via multi-step bootstrapping methods. In the final chapter of this part of the book we show how temporal-difference learning methods can be combined with model learning and planning methods (such as dynamic programming) for a complete and unified solution to the tabular reinforcement learning problem.

Chapter 2

Multi-armed Bandits

The most important feature distinguishing reinforcement learning from other types of learning is that it uses training information that *evaluates* the actions taken rather than *instructs* by giving correct actions. This is what creates the need for active exploration, for an explicit search for good behavior. Purely evaluative feedback indicates how good the action taken was, but not whether it was the best or the worst action possible. Purely instructive feedback, on the other hand, indicates the correct action to take, independently of the action actually taken. This kind of feedback is the basis of supervised learning, which includes large parts of pattern classification, artificial neural networks, and system identification. In their pure forms, these two kinds of feedback are quite distinct: evaluative feedback depends entirely on the action taken, whereas instructive feedback is independent of the action taken.

In this chapter we study the evaluative aspect of reinforcement learning in a simplified setting, one that does not involve learning to act in more than one situation. This *nonassociative* setting is the one in which most prior work involving evaluative feedback has been done, and it avoids much of the complexity of the full reinforcement learning problem. Studying this case enables us to see most clearly how evaluative feedback differs from, and yet can be combined with, instructive feedback.

The particular nonassociative, evaluative feedback problem that we explore is a simple version of the k -armed bandit problem. We use this problem to introduce a number of basic learning methods which we extend in later chapters to apply to the full reinforcement learning problem. At the end of this chapter, we take a step closer to the full reinforcement learning problem by discussing what happens when the bandit problem becomes associative, that is, when the best action depends on the situation.

2.1 A k -armed Bandit Problem

Consider the following learning problem. You are faced repeatedly with a choice among k different options, or actions. After each choice you receive a numerical reward chosen from a stationary probability distribution that depends on the action you selected. Your

objective is to maximize the expected total reward over some time period, for example, over 1000 action selections, or *time steps*.

This is the original form of the *k-armed bandit problem*, so named by analogy to a slot machine, or “one-armed bandit,” except that it has k levers instead of one. Each action selection is like a play of one of the slot machine’s levers, and the rewards are the payoffs for hitting the jackpot. Through repeated action selections you are to maximize your winnings by concentrating your actions on the best levers. Another analogy is that of a doctor choosing between experimental treatments for a series of seriously ill patients. Each action is the selection of a treatment, and each reward is the survival or well-being of the patient. Today the term “bandit problem” is sometimes used for a generalization of the problem described above, but in this book we use it to refer just to this simple case.

In our k -armed bandit problem, each of the k actions has an expected or mean reward given that that action is selected; let us call this the *value* of that action. We denote the action selected on time step t as A_t , and the corresponding reward as R_t . The value then of an arbitrary action a , denoted $q_*(a)$, is the expected reward given that a is selected:

$$q_*(a) \doteq \mathbb{E}[R_t | A_t = a].$$

If you knew the value of each action, then it would be trivial to solve the k -armed bandit problem: you would always select the action with highest value. We assume that you do not know the action values with certainty, although you may have estimates. We denote the estimated value of action a at time step t as $Q_t(a)$. We would like $Q_t(a)$ to be close to $q_*(a)$.

If you maintain estimates of the action values, then at any time step there is at least one action whose estimated value is greatest. We call these the *greedy* actions. When you select one of these actions, we say that you are *exploiting* your current knowledge of the values of the actions. If instead you select one of the nongreedy actions, then we say you are *exploring*, because this enables you to improve your estimate of the nongreedy action’s value. Exploitation is the right thing to do to maximize the expected reward on the one step, but exploration may produce the greater total reward in the long run. For example, suppose a greedy action’s value is known with certainty, while several other actions are estimated to be nearly as good but with substantial uncertainty. The uncertainty is such that at least one of these other actions probably is actually better than the greedy action, but you don’t know which one. If you have many time steps ahead on which to make action selections, then it may be better to explore the nongreedy actions and discover which of them are better than the greedy action. Reward is lower in the short run, during exploration, but higher in the long run because after you have discovered the better actions, you can exploit them many times. Because it is not possible both to explore and to exploit with any single action selection, one often refers to the “conflict” between exploration and exploitation.

In any specific case, whether it is better to explore or exploit depends in a complex way on the precise values of the estimates, uncertainties, and the number of remaining steps. There are many sophisticated methods for balancing exploration and exploitation for particular mathematical formulations of the k -armed bandit and related problems.

However, most of these methods make strong assumptions about stationarity and prior knowledge that are either violated or impossible to verify in most applications and in the full reinforcement learning problem that we consider in subsequent chapters. The guarantees of optimality or bounded loss for these methods are of little comfort when the assumptions of their theory do not apply.

In this book we do not worry about balancing exploration and exploitation in a sophisticated way; we worry only about balancing them at all. In this chapter we present several simple balancing methods for the k -armed bandit problem and show that they work much better than methods that always exploit. The need to balance exploration and exploitation is a distinctive challenge that arises in reinforcement learning; the simplicity of our version of the k -armed bandit problem enables us to show this in a particularly clear form.

2.2 Action-value Methods

We begin by looking more closely at methods for estimating the values of actions and for using the estimates to make action selection decisions, which we collectively call *action-value methods*. Recall that the true value of an action is the mean reward when that action is selected. One natural way to estimate this is by averaging the rewards actually received:

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}, \quad (2.1)$$

where $\mathbb{1}_{\text{predicate}}$ denotes the random variable that is 1 if *predicate* is true and 0 if it is not. If the denominator is zero, then we instead define $Q_t(a)$ as some default value, such as 0. As the denominator goes to infinity, by the law of large numbers, $Q_t(a)$ converges to $q_*(a)$. We call this the *sample-average* method for estimating action values because each estimate is an average of the sample of relevant rewards. Of course this is just one way to estimate action values, and not necessarily the best one. Nevertheless, for now let us stay with this simple estimation method and turn to the question of how the estimates might be used to select actions.

The simplest action selection rule is to select one of the actions with the highest estimated value, that is, one of the greedy actions as defined in the previous section. If there is more than one greedy action, then a selection is made among them in some arbitrary way, perhaps randomly. We write this *greedy* action selection method as

$$A_t \doteq \operatorname{argmax}_a Q_t(a), \quad (2.2)$$

where argmax_a denotes the action a for which the expression that follows is maximized (with ties broken arbitrarily). Greedy action selection always exploits current knowledge to maximize immediate reward; it spends no time at all sampling apparently inferior actions to see if they might really be better. A simple alternative is to behave greedily most of the time, but every once in a while, say with small probability ε , instead select randomly

from among all the actions with equal probability, independently of the action-value estimates. We call methods using this near-greedy action selection rule ε -greedy methods. An advantage of these methods is that, in the limit as the number of steps increases, every action will be sampled an infinite number of times, thus ensuring that all the $Q_t(a)$ converge to their respective $q_*(a)$. This of course implies that the probability of selecting the optimal action converges to greater than $1 - \varepsilon$, that is, to near certainty. These are just asymptotic guarantees, however, and say little about the practical effectiveness of the methods.

Exercise 2.1 In ε -greedy action selection, for the case of two actions and $\varepsilon = 0.5$, what is the probability that the greedy action is selected? \square

2.3 The 10-armed Testbed

To roughly assess the relative effectiveness of the greedy and ε -greedy action-value methods, we compared them numerically on a suite of test problems. This was a set of 2000 randomly generated k -armed bandit problems with $k = 10$. For each bandit problem, such as the one shown in Figure 2.1, the action values, $q_*(a)$, $a = 1, \dots, 10$,

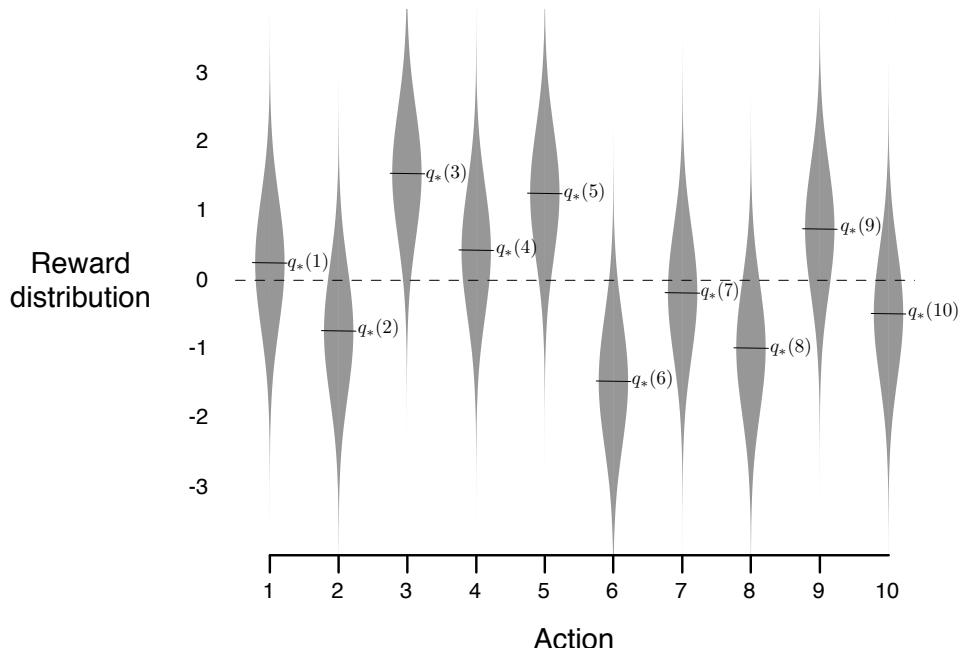


Figure 2.1: An example bandit problem from the 10-armed testbed. The true value $q_*(a)$ of each of the ten actions was selected according to a normal distribution with mean zero and unit variance, and then the actual rewards were selected according to a mean $q_*(a)$, unit-variance normal distribution, as suggested by these gray distributions.

were selected according to a normal (Gaussian) distribution with mean 0 and variance 1. Then, when a learning method applied to that problem selected action A_t at time step t , the actual reward, R_t , was selected from a normal distribution with mean $q_*(A_t)$ and variance 1. These distributions are shown in gray in Figure 2.1. We call this suite of test tasks the *10-armed testbed*. For any learning method, we can measure its performance and behavior as it improves with experience over 1000 time steps when applied to one of the bandit problems. This makes up one *run*. Repeating this for 2000 independent runs, each with a different bandit problem, we obtained measures of the learning algorithm's average behavior.

Figure 2.2 compares a greedy method with two ε -greedy methods ($\varepsilon = 0.01$ and $\varepsilon = 0.1$), as described above, on the 10-armed testbed. All the methods formed their action-value estimates using the sample-average technique (with an initial estimate of 0). The upper graph shows the increase in expected reward with experience. The greedy method improved slightly faster than the other methods at the very beginning, but then leveled off at a lower level. It achieved a reward-per-step of only about 1, compared with the best possible of about 1.55 on this testbed. The greedy method performed significantly worse in the long run because it often got stuck performing suboptimal actions. The lower graph

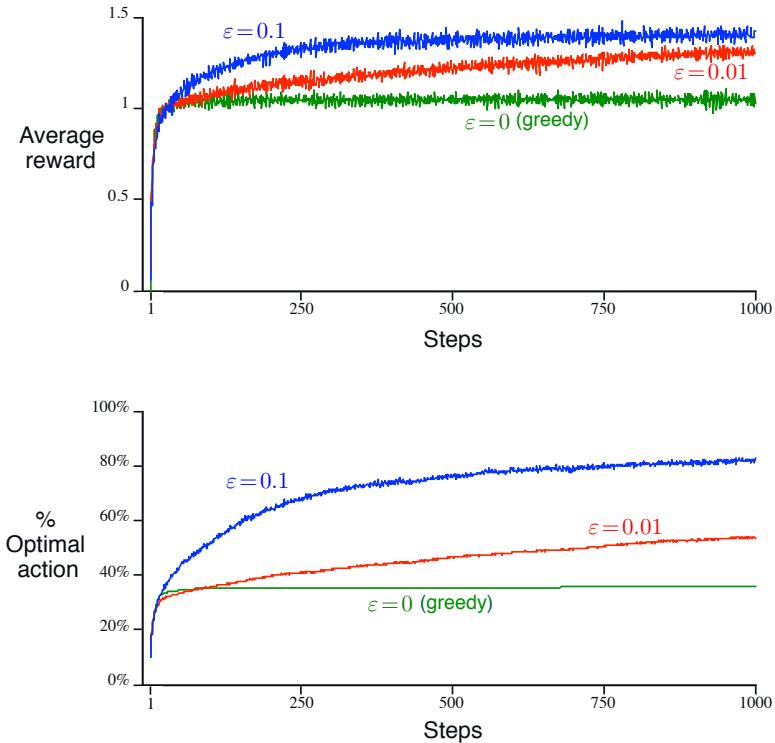


Figure 2.2: Average performance of ε -greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems. All methods used sample averages as their action-value estimates.

shows that the greedy method found the optimal action in only approximately one-third of the tasks. In the other two-thirds, its initial samples of the optimal action were disappointing, and it never returned to it. The ε -greedy methods eventually performed better because they continued to explore and to improve their chances of recognizing the optimal action. The $\varepsilon = 0.1$ method explored more, and usually found the optimal action earlier, but it never selected that action more than 91% of the time. The $\varepsilon = 0.01$ method improved more slowly, but eventually would perform better than the $\varepsilon = 0.1$ method on both performance measures shown in the figure. It is also possible to reduce ε over time to try to get the best of both high and low values.

The advantage of ε -greedy over greedy methods depends on the task. For example, suppose the reward variance had been larger, say 10 instead of 1. With noisier rewards it takes more exploration to find the optimal action, and ε -greedy methods should fare even better relative to the greedy method. On the other hand, if the reward variances were zero, then the greedy method would know the true value of each action after trying it once. In this case the greedy method might actually perform best because it would soon find the optimal action and then never explore. But even in the deterministic case there is a large advantage to exploring if we weaken some of the other assumptions. For example, suppose the bandit task were nonstationary, that is, the true values of the actions changed over time. In this case exploration is needed even in the deterministic case to make sure one of the nongreedy actions has not changed to become better than the greedy one. As we shall see in the next few chapters, nonstationarity is the case most commonly encountered in reinforcement learning. Even if the underlying task is stationary and deterministic, the learner faces a set of banditlike decision tasks each of which changes over time as learning proceeds and the agent's decision-making policy changes. Reinforcement learning requires a balance between exploration and exploitation.

Exercise 2.2: Bandit example Consider a k -armed bandit problem with $k = 4$ actions, denoted 1, 2, 3, and 4. Consider applying to this problem a bandit algorithm using ε -greedy action selection, sample-average action-value estimates, and initial estimates of $Q_1(a) = 0$, for all a . Suppose the initial sequence of actions and rewards is $A_1 = 1$, $R_1 = -1$, $A_2 = 2$, $R_2 = 1$, $A_3 = 2$, $R_3 = -2$, $A_4 = 2$, $R_4 = 2$, $A_5 = 3$, $R_5 = 0$. On some of these time steps the ε case may have occurred, causing an action to be selected at random. On which time steps did this definitely occur? On which time steps could this possibly have occurred? \square

Exercise 2.3 In the comparison shown in Figure 2.2, which method will perform best in the long run in terms of cumulative reward and probability of selecting the best action? How much better will it be? Express your answer quantitatively. \square

2.4 Incremental Implementation

The action-value methods we have discussed so far all estimate action values as sample averages of observed rewards. We now turn to the question of how these averages can be computed in a computationally efficient manner, in particular, with constant memory and constant per-time-step computation.

To simplify notation we concentrate on a single action. Let R_i now denote the reward received after the i th selection of *this action*, and let Q_n denote the estimate of its action value after it has been selected $n - 1$ times, which we can now write simply as

$$Q_n = \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1}.$$

The obvious implementation would be to maintain a record of all the rewards and then perform this computation whenever the estimated value was needed. However, if this is done, then the memory and computational requirements would grow over time as more rewards are seen. Each additional reward would require additional memory to store it and additional computation to compute the sum in the numerator.

As you might suspect, this is not really necessary. It is easy to devise incremental formulas for updating averages with small, constant computation required to process each new reward. Given Q_n and the n th reward, R_n , the new average of all n rewards can be computed by

$$\begin{aligned} Q_{n+1} &= \frac{1}{n} \sum_{i=1}^n R_i \\ &= \frac{1}{n} \left(R_n + \sum_{i=1}^{n-1} R_i \right) \\ &= \frac{1}{n} \left(R_n + (n-1) \frac{1}{n-1} \sum_{i=1}^{n-1} R_i \right) \\ &= \frac{1}{n} (R_n + (n-1)Q_n) \\ &= \frac{1}{n} (R_n + nQ_n - Q_n) \\ &= Q_n + \frac{1}{n} [R_n - Q_n], \end{aligned} \tag{2.3}$$

which holds even for $n = 1$, obtaining $Q_2 = R_1$ for arbitrary Q_1 . This implementation requires memory only for Q_n and n , and only the small computation (2.3) for each new reward.

This update rule (2.3) is of a form that occurs frequently throughout this book. The general form is

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize} [\text{Target} - \text{OldEstimate}]. \tag{2.4}$$

The expression $[\text{Target} - \text{OldEstimate}]$ is an *error* in the estimate. It is reduced by taking a step toward the “Target.” The target is presumed to indicate a desirable direction in which to move, though it may be noisy. In the case above, for example, the target is the n th reward.

Note that the step-size parameter (*StepSize*) used in the incremental method (2.3) changes from time step to time step. In processing the n th reward for action a , the

method uses the step-size parameter $\frac{1}{n}$. In this book we denote the step-size parameter by α or, more generally, by $\alpha_t(a)$.

Pseudocode for a complete bandit algorithm using incrementally computed sample averages and ε -greedy action selection is shown in the box below. The function $bandit(A)$ is assumed to take an action and return a corresponding reward.

A simple bandit algorithm

Initialize, for $a = 1$ to k :

$$\begin{aligned} Q(a) &\leftarrow 0 \\ N(a) &\leftarrow 0 \end{aligned}$$

Loop forever:

$$\begin{aligned} A &\leftarrow \begin{cases} \arg \max_a Q(a) & \text{with probability } 1 - \varepsilon \\ \text{a random action} & \text{with probability } \varepsilon \end{cases} \quad (\text{breaking ties randomly}) \\ R &\leftarrow bandit(A) \\ N(A) &\leftarrow N(A) + 1 \\ Q(A) &\leftarrow Q(A) + \frac{1}{N(A)} [R - Q(A)] \end{aligned}$$

2.5 Tracking a Nonstationary Problem

The averaging methods discussed so far are appropriate for stationary bandit problems, that is, for bandit problems in which the reward probabilities do not change over time. As noted earlier, we often encounter reinforcement learning problems that are effectively nonstationary. In such cases it makes sense to give more weight to recent rewards than to long-past rewards. One of the most popular ways of doing this is to use a constant step-size parameter. For example, the incremental update rule (2.3) for updating an average Q_n of the $n - 1$ past rewards is modified to be

$$Q_{n+1} \doteq Q_n + \alpha [R_n - Q_n], \quad (2.5)$$

where the step-size parameter $\alpha \in (0, 1]$ is constant. This results in Q_{n+1} being a weighted average of past rewards and the initial estimate Q_1 :

$$\begin{aligned} Q_{n+1} &= Q_n + \alpha [R_n - Q_n] \\ &= \alpha R_n + (1 - \alpha) Q_n \\ &= \alpha R_n + (1 - \alpha) [\alpha R_{n-1} + (1 - \alpha) Q_{n-1}] \\ &= \alpha R_n + (1 - \alpha) \alpha R_{n-1} + (1 - \alpha)^2 Q_{n-1} \\ &= \alpha R_n + (1 - \alpha) \alpha R_{n-1} + (1 - \alpha)^2 \alpha R_{n-2} + \\ &\quad \cdots + (1 - \alpha)^{n-1} \alpha R_1 + (1 - \alpha)^n Q_1 \\ &= (1 - \alpha)^n Q_1 + \sum_{i=1}^n \alpha (1 - \alpha)^{n-i} R_i. \end{aligned} \quad (2.6)$$

We call this a weighted average because the sum of the weights is $(1 - \alpha)^n + \sum_{i=1}^n \alpha(1 - \alpha)^{n-i} = 1$, as you can check for yourself. Note that the weight, $\alpha(1 - \alpha)^{n-i}$, given to the reward R_i depends on how many rewards ago, $n - i$, it was observed. The quantity $1 - \alpha$ is less than 1, and thus the weight given to R_i decreases as the number of intervening rewards increases. In fact, the weight decays exponentially according to the exponent on $1 - \alpha$. (If $1 - \alpha = 0$, then all the weight goes on the very last reward, R_n , because of the convention that $0^0 = 1$.) Accordingly, this is sometimes called an *exponential recency-weighted average*.

Sometimes it is convenient to vary the step-size parameter from step to step. Let $\alpha_n(a)$ denote the step-size parameter used to process the reward received after the n th selection of action a . As we have noted, the choice $\alpha_n(a) = \frac{1}{n}$ results in the sample-average method, which is guaranteed to converge to the true action values by the law of large numbers. But of course convergence is not guaranteed for all choices of the sequence $\{\alpha_n(a)\}$. A well-known result in stochastic approximation theory gives us the conditions required to assure convergence with probability 1:

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \quad \text{and} \quad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty. \quad (2.7)$$

The first condition is required to guarantee that the steps are large enough to eventually overcome any initial conditions or random fluctuations. The second condition guarantees that eventually the steps become small enough to assure convergence.

Note that both convergence conditions are met for the sample-average case, $\alpha_n(a) = \frac{1}{n}$, but not for the case of constant step-size parameter, $\alpha_n(a) = \alpha$. In the latter case, the second condition is not met, indicating that the estimates never completely converge but continue to vary in response to the most recently received rewards. As we mentioned above, this is actually desirable in a nonstationary environment, and problems that are effectively nonstationary are the most common in reinforcement learning. In addition, sequences of step-size parameters that meet the conditions (2.7) often converge very slowly or need considerable tuning in order to obtain a satisfactory convergence rate. Although sequences of step-size parameters that meet these convergence conditions are often used in theoretical work, they are seldom used in applications and empirical research.

Exercise 2.4 If the step-size parameters, α_n , are not constant, then the estimate Q_n is a weighted average of previously received rewards with a weighting different from that given by (2.6). What is the weighting on each prior reward for the general case, analogous to (2.6), in terms of the sequence of step-size parameters? \square

Exercise 2.5 (programming) Design and conduct an experiment to demonstrate the difficulties that sample-average methods have for nonstationary problems. Use a modified version of the 10-armed testbed in which all the $q_*(a)$ start out equal and then take independent random walks (say by adding a normally distributed increment with mean 0 and standard deviation 0.01 to all the $q_*(a)$ on each step). Prepare plots like Figure 2.2 for an action-value method using sample averages, incrementally computed, and another action-value method using a constant step-size parameter, $\alpha = 0.1$. Use $\varepsilon = 0.1$ and longer runs, say of 10,000 steps. \square

2.6 Optimistic Initial Values

All the methods we have discussed so far are dependent to some extent on the initial action-value estimates, $Q_1(a)$. In the language of statistics, these methods are *biased* by their initial estimates. For the sample-average methods, the bias disappears once all actions have been selected at least once, but for methods with constant α , the bias is permanent, though decreasing over time as given by (2.6). In practice, this kind of bias is usually not a problem and can sometimes be very helpful. The downside is that the initial estimates become, in effect, a set of parameters that must be picked by the user, if only to set them all to zero. The upside is that they provide an easy way to supply some prior knowledge about what level of rewards can be expected.

Initial action values can also be used as a simple way to encourage exploration. Suppose that instead of setting the initial action values to zero, as we did in the 10-armed testbed, we set them all to +5. Recall that the $q_*(a)$ in this problem are selected from a normal distribution with mean 0 and variance 1. An initial estimate of +5 is thus wildly optimistic. But this optimism encourages action-value methods to explore. Whichever actions are initially selected, the reward is less than the starting estimates; the learner switches to other actions, being “disappointed” with the rewards it is receiving. The result is that all actions are tried several times before the value estimates converge. The system does a fair amount of exploration even if greedy actions are selected all the time.

Figure 2.3 shows the performance on the 10-armed bandit testbed of a greedy method using $Q_1(a) = +5$, for all a . For comparison, also shown is an ε -greedy method with $Q_1(a) = 0$. Initially, the optimistic method performs worse because it explores more, but eventually it performs better because its exploration decreases with time. We call this technique for encouraging exploration *optimistic initial values*. We regard it as a simple trick that can be quite effective on stationary problems, but it is far from being a generally useful approach to encouraging exploration. For example, it is not well suited to nonstationary problems because its drive for exploration is inherently

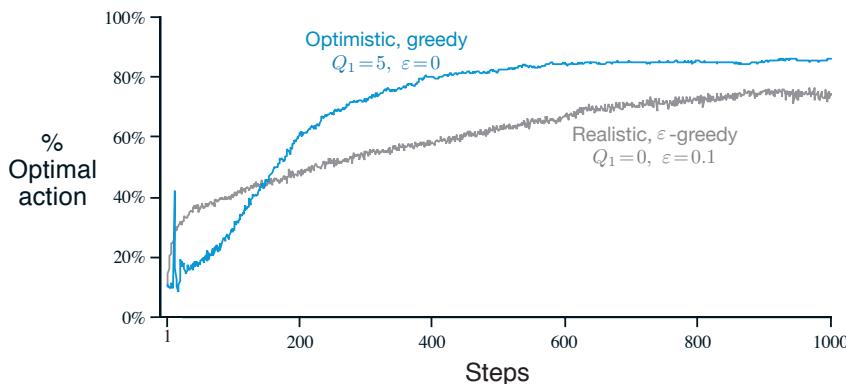


Figure 2.3: The effect of optimistic initial action-value estimates on the 10-armed testbed. Both methods used a constant step-size parameter, $\alpha = 0.1$.

temporary. If the task changes, creating a renewed need for exploration, this method cannot help. Indeed, any method that focuses on the initial conditions in any special way is unlikely to help with the general nonstationary case. The beginning of time occurs only once, and thus we should not focus on it too much. This criticism applies as well to the sample-average methods, which also treat the beginning of time as a special event, averaging all subsequent rewards with equal weights. Nevertheless, all of these methods are very simple, and one of them—or some simple combination of them—is often adequate in practice. In the rest of this book we make frequent use of several of these simple exploration techniques.

Exercise 2.6: Mysterious Spikes The results shown in Figure 2.3 should be quite reliable because they are averages over 2000 individual, randomly chosen 10-armed bandit tasks. Why, then, are there oscillations and spikes in the early part of the curve for the optimistic method? In other words, what might make this method perform particularly better or worse, on average, on particular early steps? \square

Exercise 2.7: Unbiased Constant-Step-Size Trick In most of this chapter we have used sample averages to estimate action values because sample averages do not produce the initial bias that constant step sizes do (see the analysis leading to (2.6)). However, sample averages are not a completely satisfactory solution because they may perform poorly on nonstationary problems. Is it possible to avoid the bias of constant step sizes while retaining their advantages on nonstationary problems? One way is to use a step size of

$$\beta_n \doteq \alpha/\bar{o}_n, \quad (2.8)$$

to process the n th reward for a particular action, where $\alpha > 0$ is a conventional constant step size, and \bar{o}_n is a trace of one that starts at 0:

$$\bar{o}_n \doteq \bar{o}_{n-1} + \alpha(1 - \bar{o}_{n-1}), \quad \text{for } n \geq 0, \quad \text{with } \bar{o}_0 \doteq 0. \quad (2.9)$$

Carry out an analysis like that in (2.6) to show that Q_n is an exponential recency-weighted average *without initial bias*. \square

2.7 Upper-Confidence-Bound Action Selection

Exploration is needed because there is always uncertainty about the accuracy of the action-value estimates. The greedy actions are those that look best at present, but some of the other actions may actually be better. ε -greedy action selection forces the non-greedy actions to be tried, but indiscriminately, with no preference for those that are nearly greedy or particularly uncertain. It would be better to select among the non-greedy actions according to their potential for actually being optimal, taking into account both how close their estimates are to being maximal and the uncertainties in those estimates. One effective way of doing this is to select actions according to

$$A_t \doteq \arg \max_a \left[Q_t(a) + c \sqrt{\frac{\ln t}{N_t(a)}} \right], \quad (2.10)$$

where $\ln t$ denotes the natural logarithm of t (the number that $e \approx 2.71828$ would have to be raised to in order to equal t), $N_t(a)$ denotes the number of times that action a has

been selected prior to time t (the denominator in (2.1)), and the number $c > 0$ controls the degree of exploration. If $N_t(a) = 0$, then a is considered to be a maximizing action.

The idea of this *upper confidence bound* (UCB) action selection is that the square-root term is a measure of the uncertainty or variance in the estimate of a 's value. The quantity being max'ed over is thus a sort of upper bound on the possible true value of action a , with c determining the confidence level. Each time a is selected the uncertainty is presumably reduced: $N_t(a)$ increments, and, as it appears in the denominator, the uncertainty term decreases. On the other hand, each time an action other than a is selected, t increases but $N_t(a)$ does not; because t appears in the numerator, the uncertainty estimate increases. The use of the natural logarithm means that the increases get smaller over time, but are unbounded; all actions will eventually be selected, but actions with lower value estimates, or that have already been selected frequently, will be selected with decreasing frequency over time.

Results with UCB on the 10-armed testbed are shown in Figure 2.4. UCB often performs well, as shown here, but is more difficult than ε -greedy to extend beyond bandits to the more general reinforcement learning settings considered in the rest of this book. One difficulty is in dealing with nonstationary problems; methods more complex than those presented in Section 2.5 would be needed. Another difficulty is dealing with large state spaces, particularly when using function approximation as developed in Part II of this book. In these more advanced settings the idea of UCB action selection is usually not practical.

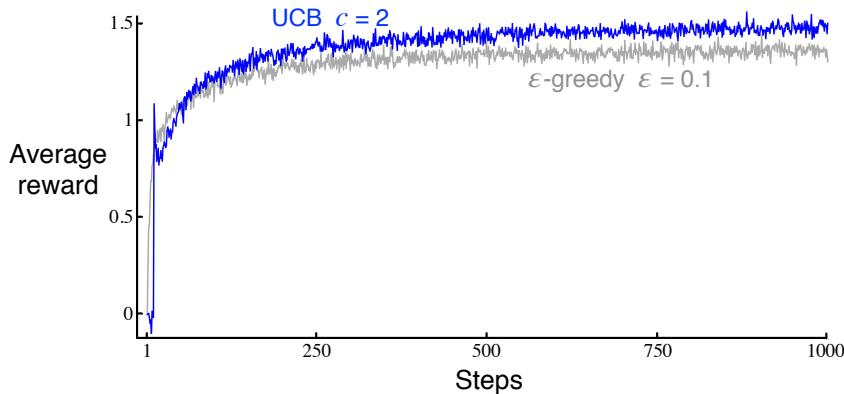


Figure 2.4: Average performance of UCB action selection on the 10-armed testbed. As shown, UCB generally performs better than ε -greedy action selection, except in the first k steps, when it selects randomly among the as-yet-untried actions.

Exercise 2.8: UCB Spikes In Figure 2.4 the UCB algorithm shows a distinct spike in performance on the 11th step. Why is this? Note that for your answer to be fully satisfactory it must explain both why the reward increases on the 11th step and why it decreases on the subsequent steps. Hint: If $c = 1$, then the spike is less prominent. \square

2.8 Gradient Bandit Algorithms

So far in this chapter we have considered methods that estimate action values and use those estimates to select actions. This is often a good approach, but it is not the only one possible. In this section we consider learning a numerical *preference* for each action a , which we denote $H_t(a) \in \mathbb{R}$. The larger the preference, the more often that action is taken, but the preference has no interpretation in terms of reward. Only the relative preference of one action over another is important; if we add 1000 to all the action preferences there is no effect on the action probabilities, which are determined according to a *soft-max distribution* (i.e., Gibbs or Boltzmann distribution) as follows:

$$\Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^k e^{H_t(b)}} \doteq \pi_t(a), \quad (2.11)$$

where here we have also introduced a useful new notation, $\pi_t(a)$, for the probability of taking action a at time t . Initially all action preferences are the same (e.g., $H_1(a) = 0$, for all a) so that all actions have an equal probability of being selected.

Exercise 2.9 Show that in the case of two actions, the soft-max distribution is the same as that given by the logistic, or sigmoid, function often used in statistics and artificial neural networks. \square

There is a natural learning algorithm for soft-max action preferences based on the idea of stochastic gradient ascent. On each step, after selecting action A_t and receiving the reward R_t , the action preferences are updated by:

$$\begin{aligned} H_{t+1}(A_t) &\doteq H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)), && \text{and} \\ H_{t+1}(a) &\doteq H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a), && \text{for all } a \neq A_t, \end{aligned} \quad (2.12)$$

where $\alpha > 0$ is a step-size parameter, and $\bar{R}_t \in \mathbb{R}$ is the average of the rewards up to but not including time t (with $\bar{R}_1 \doteq R_1$), which can be computed incrementally as described in Section 2.4 (or Section 2.5 if the problem is nonstationary).¹ The \bar{R}_t term serves as a baseline with which the reward is compared. If the reward is higher than the baseline, then the probability of taking A_t in the future is increased, and if the reward is below baseline, then the probability is decreased. The non-selected actions move in the opposite direction.

Figure 2.5 shows results with the gradient bandit algorithm on a variant of the 10-armed testbed in which the true expected rewards were selected according to a normal distribution with a mean of +4 instead of zero (and with unit variance as before). This shifting up of all the rewards has absolutely no effect on the gradient bandit algorithm because of the reward baseline term, which instantaneously adapts to the new level. But if the baseline were omitted (that is, if \bar{R}_t was taken to be constant zero in (2.12)), then performance would be significantly degraded, as shown in the figure.

¹In the empirical results in this chapter, the baseline \bar{R}_t also included R_t .

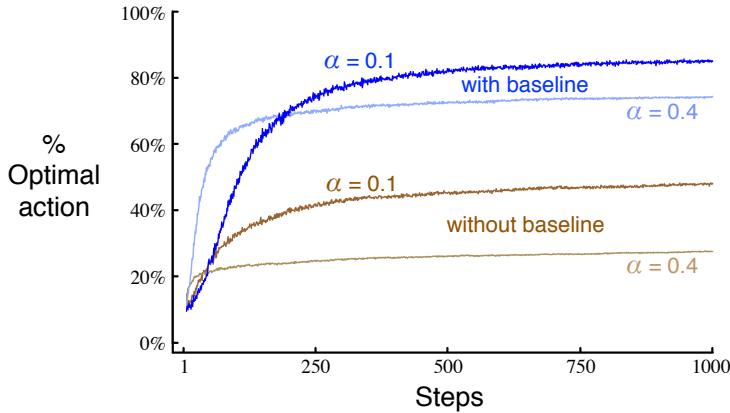


Figure 2.5: Average performance of the gradient bandit algorithm with and without a reward baseline on the 10-armed testbed when the $q_*(a)$ are chosen to be near +4 rather than near zero.

The Bandit Gradient Algorithm as Stochastic Gradient Ascent

One can gain a deeper insight into the gradient bandit algorithm by understanding it as a stochastic approximation to gradient ascent. In exact *gradient ascent*, each action preference $H_t(a)$ would be incremented in proportion to the increment's effect on performance:

$$H_{t+1}(a) \doteq H_t(a) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}, \quad (2.13)$$

where the measure of performance here is the expected reward:

$$\mathbb{E}[R_t] = \sum_x \pi_t(x) q_*(x),$$

and the measure of the increment's effect is the *partial derivative* of this performance measure with respect to the action preference. Of course, it is not possible to implement gradient ascent exactly in our case because by assumption we do not know the $q_*(x)$, but in fact the updates of our algorithm (2.12) are equal to (2.13) in expected value, making the algorithm an instance of *stochastic gradient ascent*. The calculations showing this require only beginning calculus, but take several

steps. First we take a closer look at the exact performance gradient:

$$\begin{aligned}\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)} \left[\sum_x \pi_t(x) q_*(x) \right] \\ &= \sum_x q_*(x) \frac{\partial \pi_t(x)}{\partial H_t(a)} \\ &= \sum_x (q_*(x) - B_t) \frac{\partial \pi_t(x)}{\partial H_t(a)},\end{aligned}$$

where B_t , called the *baseline*, can be any scalar that does not depend on x . We can include a baseline here without changing the equality because the gradient sums to zero over all the actions, $\sum_x \frac{\partial \pi_t(x)}{\partial H_t(a)} = 0$. As $H_t(a)$ is changed, some actions' probabilities go up and some go down, but the sum of the changes must be zero because the sum of the probabilities is always one.

Next we multiply each term of the sum by $\pi_t(x)/\pi_t(x)$:

$$\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} = \sum_x \pi_t(x) (q_*(x) - B_t) \frac{\partial \pi_t(x)}{\partial H_t(a)} / \pi_t(x).$$

The equation is now in the form of an expectation, summing over all possible values x of the random variable A_t , then multiplying by the probability of taking those values. Thus:

$$\begin{aligned}&= \mathbb{E} \left[(q_*(A_t) - B_t) \frac{\partial \pi_t(A_t)}{\partial H_t(a)} / \pi_t(A_t) \right] \\ &= \mathbb{E} \left[(R_t - \bar{R}_t) \frac{\partial \pi_t(A_t)}{\partial H_t(a)} / \pi_t(A_t) \right],\end{aligned}$$

where here we have chosen the baseline $B_t = \bar{R}_t$ and substituted R_t for $q_*(A_t)$, which is permitted because $\mathbb{E}[R_t|A_t] = q_*(A_t)$. Shortly we will establish that $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)(\mathbb{1}_{a=x} - \pi_t(a))$, where $\mathbb{1}_{a=x}$ is defined to be 1 if $a = x$, else 0. Assuming that for now, we have

$$\begin{aligned}&= \mathbb{E} [(R_t - \bar{R}_t) \pi_t(A_t) (\mathbb{1}_{a=A_t} - \pi_t(a)) / \pi_t(A_t)] \\ &= \mathbb{E} [(R_t - \bar{R}_t) (\mathbb{1}_{a=A_t} - \pi_t(a))].\end{aligned}$$

Recall that our plan has been to write the performance gradient as an expectation of something that we can sample on each step, as we have just done, and then update on each step in proportion to the sample. Substituting a sample of the expectation above for the performance gradient in (2.13) yields:

$$H_{t+1}(a) = H_t(a) + \alpha(R_t - \bar{R}_t)(\mathbb{1}_{a=A_t} - \pi_t(a)), \quad \text{for all } a,$$

which you may recognize as being equivalent to our original algorithm (2.12).

Thus it remains only to show that $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)(\mathbb{1}_{a=x} - \pi_t(a))$, as we assumed. Recall the standard quotient rule for derivatives:

$$\frac{\partial}{\partial x} \left[\frac{f(x)}{g(x)} \right] = \frac{\frac{\partial f(x)}{\partial x} g(x) - f(x) \frac{\partial g(x)}{\partial x}}{g(x)^2}.$$

Using this, we can write

$$\begin{aligned} \frac{\partial \pi_t(x)}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)} \pi_t(x) \\ &= \frac{\partial}{\partial H_t(a)} \left[\frac{e^{H_t(x)}}{\sum_{y=1}^k e^{H_t(y)}} \right] \\ &= \frac{\frac{\partial e^{H_t(x)}}{\partial H_t(a)} \sum_{y=1}^k e^{H_t(y)} - e^{H_t(x)} \frac{\partial \sum_{y=1}^k e^{H_t(y)}}{\partial H_t(a)}}{\left(\sum_{y=1}^k e^{H_t(y)} \right)^2} \quad (\text{by the quotient rule}) \\ &= \frac{\mathbb{1}_{a=x} e^{H_t(x)} \sum_{y=1}^k e^{H_t(y)} - e^{H_t(x)} e^{H_t(a)}}{\left(\sum_{y=1}^k e^{H_t(y)} \right)^2} \quad (\text{because } \frac{\partial e^x}{\partial x} = e^x) \\ &= \frac{\mathbb{1}_{a=x} e^{H_t(x)}}{\sum_{y=1}^k e^{H_t(y)}} - \frac{e^{H_t(x)} e^{H_t(a)}}{\left(\sum_{y=1}^k e^{H_t(y)} \right)^2} \\ &= \mathbb{1}_{a=x} \pi_t(x) - \pi_t(x) \pi_t(a) \\ &= \pi_t(x)(\mathbb{1}_{a=x} - \pi_t(a)). \end{aligned}$$

Q.E.D.

We have just shown that the expected update of the gradient bandit algorithm is equal to the gradient of expected reward, and thus that the algorithm is an instance of stochastic gradient ascent. This assures us that the algorithm has robust convergence properties.

Note that we did not require any properties of the reward baseline other than that it does not depend on the selected action. For example, we could have set it to zero, or to 1000, and the algorithm would still be an instance of stochastic gradient ascent. The choice of the baseline does not affect the expected update of the algorithm, but it does affect the variance of the update and thus the rate of convergence (as shown, for example, in Figure 2.5). Choosing it as the average of the rewards may not be the very best, but it is simple and works well in practice.

2.9 Associative Search (Contextual Bandits)

So far in this chapter we have considered only nonassociative tasks, that is, tasks in which there is no need to associate different actions with different situations. In these tasks the learner either tries to find a single best action when the task is stationary, or tries to track the best action as it changes over time when the task is nonstationary. However, in a general reinforcement learning task there is more than one situation, and the goal is to learn a policy: a mapping from situations to the actions that are best in those situations. To set the stage for the full problem, we briefly discuss the simplest way in which nonassociative tasks extend to the associative setting.

As an example, suppose there are several different k -armed bandit tasks, and that on each step you confront one of these chosen at random. Thus, the bandit task changes randomly from step to step. If the probabilities with which each task is selected for you do not change over time, this would appear as a single stationary k -armed bandit task, and you could use one of the methods described in this chapter. Now suppose, however, that when a bandit task is selected for you, you are given some distinctive clue about its identity (but not its action values). Maybe you are facing an actual slot machine that changes the color of its display as it changes its action values. Now you can learn a policy associating each task, signaled by the color you see, with the best action to take when facing that task—for instance, if red, select arm 1; if green, select arm 2. With the right policy you can usually do much better than you could in the absence of any information distinguishing one bandit task from another.

This is an example of an *associative search* task, so called because it involves both trial-and-error learning to *search* for the best actions, and *association* of these actions with the situations in which they are best. Associative search tasks are often now called *contextual bandits* in the literature. Associative search tasks are intermediate between the k -armed bandit problem and the full reinforcement learning problem. They are like the full reinforcement learning problem in that they involve learning a policy, but they are also like our version of the k -armed bandit problem in that each action affects only the immediate reward. If actions are allowed to affect the *next situation* as well as the reward, then we have the full reinforcement learning problem. We present this problem in the next chapter and consider its ramifications throughout the rest of the book.

Exercise 2.10 Suppose you face a 2-armed bandit task whose true action values change randomly from time step to time step. Specifically, suppose that, for any time step, the true values of actions 1 and 2 are respectively 10 and 20 with probability 0.5 (case A), and 90 and 80 with probability 0.5 (case B). If you are not able to tell which case you face at any step, what is the best expected reward you can achieve and how should you behave to achieve it? Now suppose that on each step you are told whether you are facing case A or case B (although you still don't know the true action values). This is an associative search task. What is the best expected reward you can achieve in this task, and how should you behave to achieve it? □

2.10 Summary

We have presented in this chapter several simple ways of balancing exploration and exploitation. The ϵ -greedy methods choose randomly a small fraction of the time, whereas UCB methods choose deterministically but achieve exploration by subtly favoring at each step the actions that have so far received fewer samples. Gradient bandit algorithms estimate not action values, but action preferences, and favor the more preferred actions in a graded, probabilistic manner using a soft-max distribution. The simple expedient of initializing estimates optimistically causes even greedy methods to explore significantly.

It is natural to ask which of these methods is best. Although this is a difficult question to answer in general, we can certainly run them all on the 10-armed testbed that we have used throughout this chapter and compare their performances. A complication is that they all have a parameter; to get a meaningful comparison we have to consider their performance as a function of their parameter. Our graphs so far have shown the course of learning over time for each algorithm and parameter setting, to produce a *learning curve* for that algorithm and parameter setting. If we plotted learning curves for all algorithms and all parameter settings, then the graph would be too complex and crowded to make clear comparisons. Instead we summarize a complete learning curve by its average value over the 1000 steps; this value is proportional to the area under the learning curve. Figure 2.6 shows this measure for the various bandit algorithms from this chapter, each as a function of its own parameter shown on a single scale on the x-axis. This kind of graph is called a *parameter study*. Note that the parameter values are varied by factors of two and presented on a log scale. Note also the characteristic inverted-U shapes of each algorithm's performance; all the algorithms perform best at an intermediate value of their parameter, neither too large nor too small. In assessing

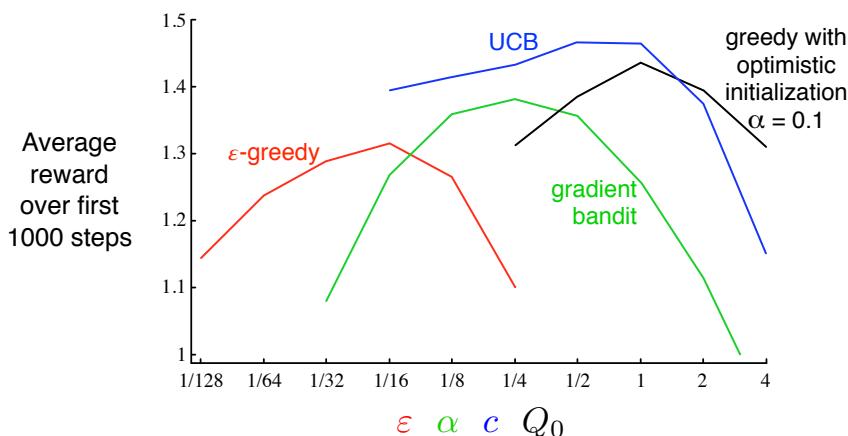


Figure 2.6: A parameter study of the various bandit algorithms presented in this chapter. Each point is the average reward obtained over 1000 steps with a particular algorithm at a particular setting of its parameter.

a method, we should attend not just to how well it does at its best parameter setting, but also to how sensitive it is to its parameter value. All of these algorithms are fairly insensitive, performing well over a range of parameter values varying by about an order of magnitude. Overall, on this problem, UCB seems to perform best.

Despite their simplicity, in our opinion the methods presented in this chapter can fairly be considered the state of the art. There are more sophisticated methods, but their complexity and assumptions make them impractical for the full reinforcement learning problem that is our real focus. Starting in Chapter 5 we present learning methods for solving the full reinforcement learning problem that use in part the simple methods explored in this chapter.

Although the simple methods explored in this chapter may be the best we can do at present, they are far from a fully satisfactory solution to the problem of balancing exploration and exploitation.

One well-studied approach to balancing exploration and exploitation in k -armed bandit problems is to compute a special kind of action value called a *Gittins index*. In certain important special cases, this computation is tractable and leads directly to optimal solutions, although it does require complete knowledge of the prior distribution of possible problems, which we generally assume is not available. In addition, neither the theory nor the computational tractability of this approach appear to generalize to the full reinforcement learning problem that we consider in the rest of the book.

The Gittins-index approach is an instance of *Bayesian* methods, which assume a known initial distribution over the action values and then update the distribution exactly after each step (assuming that the true action values are stationary). In general, the update computations can be very complex, but for certain special distributions (called *conjugate priors*) they are easy. One possibility is to then select actions at each step according to their posterior probability of being the best action. This method, sometimes called *posterior sampling* or *Thompson sampling*, often performs similarly to the best of the distribution-free methods we have presented in this chapter.

In the Bayesian setting it is even conceivable to compute the *optimal* balance between exploration and exploitation. One can compute for any possible action the probability of each possible immediate reward and the resultant posterior distributions over action values. This evolving distribution becomes the *information state* of the problem. Given a horizon, say of 1000 steps, one can consider all possible actions, all possible resulting rewards, all possible next actions, all next rewards, and so on for all 1000 steps. Given the assumptions, the rewards and probabilities of each possible chain of events can be determined; one need only pick the best. But the tree of possibilities grows extremely rapidly; even if there were only two actions and two rewards, the tree would have 2^{2000} leaves. It is generally not feasible to perform this immense computation exactly, but perhaps it could be approximated efficiently. This approach would effectively turn the bandit problem into an instance of the full reinforcement learning problem. In the end, we may be able to use approximate reinforcement learning methods such as those presented in Part II of this book to approach this optimal solution. But that is a topic for research and beyond the scope of this introductory book.

Exercise 2.11 (programming) Make a figure analogous to Figure 2.6 for the nonstationary case outlined in Exercise 2.5. Include the constant-step-size ε -greedy algorithm with $\alpha=0.1$. Use runs of 200,000 steps and, as a performance measure for each algorithm and parameter setting, use the average reward over the last 100,000 steps. \square

Bibliographical and Historical Remarks

- 2.1** Bandit problems have been studied in statistics, engineering, and psychology. In statistics, bandit problems fall under the heading “sequential design of experiments,” introduced by Thompson (1933, 1934) and Robbins (1952), and studied by Bellman (1956). Berry and Fristedt (1985) provide an extensive treatment of bandit problems from the perspective of statistics. Narendra and Thathachar (1989) treat bandit problems from the engineering perspective, providing a good discussion of the various theoretical traditions that have focused on them. In psychology, bandit problems have played roles in statistical learning theory (e.g., Bush and Mosteller, 1955; Estes, 1950).
- The term *greedy* is often used in the heuristic search literature (e.g., Pearl, 1984). The conflict between exploration and exploitation is known in control engineering as the conflict between identification (or estimation) and control (e.g., Witten, 1976b). Feldbaum (1965) called it the *dual control* problem, referring to the need to solve the two problems of identification and control simultaneously when trying to control a system under uncertainty. In discussing aspects of genetic algorithms, Holland (1975) emphasized the importance of this conflict, referring to it as the conflict between the need to exploit and the need for new information.
- 2.2** Action-value methods for our k -armed bandit problem were first proposed by Thathachar and Sastry (1985). These are often called *estimator algorithms* in the learning automata literature. The term *action value* is due to Watkins (1989). The first to use ε -greedy methods may also have been Watkins (1989, p. 187), but the idea is so simple that some earlier use seems likely.
- 2.4–5** This material falls under the general heading of stochastic iterative algorithms, which is well covered by Bertsekas and Tsitsiklis (1996).
- 2.6** Optimistic initialization was used in reinforcement learning by Sutton (1996).
- 2.7** Early work on using estimates of the upper confidence bound to select actions was done by Lai and Robbins (1985), Kaelbling (1993b), and Agrawal (1995). The UCB algorithm we present here is called UCB1 in the literature and was first developed by Auer, Cesa-Bianchi and Fischer (2002).
- 2.8** Gradient bandit algorithms are a special case of the gradient-based reinforcement learning algorithms introduced by Williams (1992) that later developed into the actor-critic and policy-gradient algorithms that we treat later in this book. Our development here was influenced by that by Balaraman Ravindran (personal

communication). Further discussion of the choice of baseline is provided by Greensmith, Bartlett, and Baxter (2002, 2004) and by Dick (2015). Early systematic studies of algorithms like this were done by Sutton (1984).

The term *soft-max* for the action selection rule (2.11) is due to Bridle (1990). This rule appears to have been first proposed by Luce (1959).

- 2.9** The term *associative search* and the corresponding problem were introduced by Barto, Sutton, and Brouwer (1981). The term *associative reinforcement learning* has also been used for associative search (Barto and Anandan, 1985), but we prefer to reserve that term as a synonym for the full reinforcement learning problem (as in Sutton, 1984). (And, as we noted, the modern literature also uses the term “contextual bandits” for this problem.) We note that Thorndike’s Law of Effect (quoted in Chapter 1) describes associative search by referring to the formation of associative links between situations (states) and actions. According to the terminology of operant, or instrumental, conditioning (e.g., Skinner, 1938), a discriminative stimulus is a stimulus that signals the presence of a particular reinforcement contingency. In our terms, different discriminative stimuli correspond to different states.

- 2.10** Bellman (1956) was the first to show how dynamic programming could be used to compute the optimal balance between exploration and exploitation within a Bayesian formulation of the problem. The Gittins index approach is due to Gittins and Jones (1974). Duff (1995) showed how it is possible to learn Gittins indices for bandit problems through reinforcement learning. The survey by Kumar (1985) provides a good discussion of Bayesian and non-Bayesian approaches to these problems. The term *information state* comes from the literature on partially observable MDPs; see, for example, Lovejoy (1991).

Other theoretical research focuses on the efficiency of exploration, usually expressed as how quickly an algorithm can approach an optimal decision-making policy. One way to formalize exploration efficiency is by adapting to reinforcement learning the notion of *sample complexity* for a supervised learning algorithm, which is the number of training examples the algorithm needs to attain a desired degree of accuracy in learning the target function. A definition of the sample complexity of exploration for a reinforcement learning algorithm is the number of time steps in which the algorithm does not select near-optimal actions (Kakade, 2003). Li (2012) discusses this and several other approaches in a survey of theoretical approaches to exploration efficiency in reinforcement learning. A thorough modern treatment of Thompson sampling is provided by Russo et al. (2018).

Chapter 3

Finite Markov Decision Processes

In this chapter we introduce the formal problem of finite Markov decision processes, or finite MDPs, which we try to solve in the rest of the book. This problem involves evaluative feedback, as in bandits, but also an associative aspect—choosing different actions in different situations. MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. Thus MDPs involve delayed reward and the need to trade off immediate and delayed reward. Whereas in bandit problems we estimated the value $q_*(a)$ of each action a , in MDPs we estimate the value $q_*(s, a)$ of each action a in each state s , or we estimate the value $v_*(s)$ of each state given optimal action selections. These state-dependent quantities are essential to accurately assigning credit for long-term consequences to individual action selections.

MDPs are a mathematically idealized form of the reinforcement learning problem for which precise theoretical statements can be made. We introduce key elements of the problem’s mathematical structure, such as returns, value functions, and Bellman equations. We try to convey the wide range of applications that can be formulated as finite MDPs. As in all of artificial intelligence, there is a tension between breadth of applicability and mathematical tractability. In this chapter we introduce this tension and discuss some of the trade-offs and challenges that it implies. Some ways in which reinforcement learning can be taken beyond MDPs are treated in Chapter 17.

3.1 The Agent–Environment Interface

MDPs are meant to be a straightforward framing of the problem of learning from interaction to achieve a goal. The learner and decision maker is called the *agent*. The thing it interacts with, comprising everything outside the agent, is called the *environment*. These interact continually, the agent selecting actions and the environment responding to

these actions and presenting new situations to the agent.¹ The environment also gives rise to rewards, special numerical values that the agent seeks to maximize over time through its choice of actions.

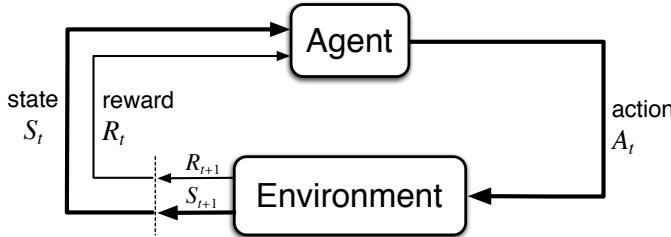


Figure 3.1: The agent–environment interaction in a Markov decision process.

More specifically, the agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, \dots$.² At each time step t , the agent receives some representation of the environment’s *state*, $S_t \in \mathcal{S}$, and on that basis selects an *action*, $A_t \in \mathcal{A}(s)$.³ One time step later, in part as a consequence of its action, the agent receives a numerical *reward*, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, and finds itself in a new state, S_{t+1} .⁴ The MDP and agent together thereby give rise to a sequence or *trajectory* that begins like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \dots \quad (3.1)$$

In a *finite* MDP, the sets of states, actions, and rewards (\mathcal{S} , \mathcal{A} , and \mathcal{R}) all have a finite number of elements. In this case, the random variables R_t and S_t have well defined discrete probability distributions dependent only on the preceding state and action. That is, for particular values of these random variables, $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, there is a probability of those values occurring at time t , given particular values of the preceding state and action:

$$p(s', r | s, a) \doteq \Pr\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\}, \quad (3.2)$$

for all $s', s \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$. The function p defines the *dynamics* of the MDP. The dot over the equals sign in the equation reminds us that it is a definition (in this case of the function p) rather than a fact that follows from previous definitions. The dynamics function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ is an ordinary deterministic function of four arguments. The ‘|’ in the middle of it comes from the notation for conditional probability,

¹We use the terms *agent*, *environment*, and *action* instead of the engineers’ terms *controller*, *controlled system* (or *plant*), and *control signal* because they are meaningful to a wider audience.

²We restrict attention to discrete time to keep things as simple as possible, even though many of the ideas can be extended to the continuous-time case (e.g., see Bertsekas and Tsitsiklis, 1996; Doya, 1996).

³To simplify notation, we sometimes assume the special case in which the action set is the same in all states and write it simply as \mathcal{A} .

⁴We use R_{t+1} instead of R_t to denote the reward due to A_t because it emphasizes that the next reward and next state, R_{t+1} and S_{t+1} , are jointly determined. Unfortunately, both conventions are widely used in the literature.

but here it just reminds us that p specifies a probability distribution for each choice of s and a , that is, that

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a) = 1, \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s). \quad (3.3)$$

In a *Markov* decision process, the probabilities given by p completely characterize the environment’s dynamics. That is, the probability of each possible value for S_t and R_t depends on the immediately preceding state and action, S_{t-1} and A_{t-1} , and, given them, not at all on earlier states and actions. This is best viewed as a restriction not on the decision process, but on the *state*. The state must include information about all aspects of the past agent–environment interaction that make a difference for the future. If it does, then the state is said to have the *Markov property*. We will assume the Markov property throughout this book, though starting in Part II we will consider approximation methods that do not rely on it, and in Chapter 17 we consider how a Markov state can be efficiently learned and constructed from non-Markov observations.

From the four-argument dynamics function, p , one can compute anything else one might want to know about the environment, such as the *state-transition probabilities* (which we denote, with a slight abuse of notation, as a three-argument function $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$),

$$p(s' | s, a) \doteq \Pr\{S_t = s' | S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r | s, a). \quad (3.4)$$

We can also compute the expected rewards for state–action pairs as a two-argument function $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$:

$$r(s, a) \doteq \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r | s, a), \quad (3.5)$$

and the expected rewards for state–action–next-state triples as a three-argument function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$,

$$r(s, a, s') \doteq \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathcal{R}} r \frac{p(s', r | s, a)}{p(s' | s, a)}. \quad (3.6)$$

In this book, we usually use the four-argument p function (3.2), but each of these other notations are also occasionally convenient.

The MDP framework is abstract and flexible and can be applied to many different problems in many different ways. For example, the time steps need not refer to fixed intervals of real time; they can refer to arbitrary successive stages of decision making and acting. The actions can be low-level controls, such as the voltages applied to the motors of a robot arm, or high-level decisions, such as whether or not to have lunch or to go to graduate school. Similarly, the states can take a wide variety of forms. They can be completely determined by low-level sensations, such as direct sensor readings, or they can be more high-level and abstract, such as symbolic descriptions of objects in a room. Some of what makes up a state could be based on memory of past sensations or

even be entirely mental or subjective. For example, an agent could be in the state of not being sure where an object is, or of having just been surprised in some clearly defined sense. Similarly, some actions might be totally mental or computational. For example, some actions might control what an agent chooses to think about, or where it focuses its attention. In general, actions can be any decisions we want to learn how to make, and states can be anything we can know that might be useful in making them.

In particular, the boundary between agent and environment is typically not the same as the physical boundary of a robot's or an animal's body. Usually, the boundary is drawn closer to the agent than that. For example, the motors and mechanical linkages of a robot and its sensing hardware should usually be considered parts of the environment rather than parts of the agent. Similarly, if we apply the MDP framework to a person or animal, the muscles, skeleton, and sensory organs should be considered part of the environment. Rewards, too, presumably are computed inside the physical bodies of natural and artificial learning systems, but are considered external to the agent.

The general rule we follow is that anything that cannot be changed arbitrarily by the agent is considered to be outside of it and thus part of its environment. We do not assume that everything in the environment is unknown to the agent. For example, the agent often knows quite a bit about how its rewards are computed as a function of its actions and the states in which they are taken. But we always consider the reward computation to be external to the agent because it defines the task facing the agent and thus must be beyond its ability to change arbitrarily. In fact, in some cases the agent may know *everything* about how its environment works and still face a difficult reinforcement learning task, just as we may know exactly how a puzzle like Rubik's cube works, but still be unable to solve it. The agent–environment boundary represents the limit of the agent's *absolute control*, not of its knowledge.

The agent–environment boundary can be located at different places for different purposes. In a complicated robot, many different agents may be operating at once, each with its own boundary. For example, one agent may make high-level decisions which form part of the states faced by a lower-level agent that implements the high-level decisions. In practice, the agent–environment boundary is determined once one has selected particular states, actions, and rewards, and thus has identified a specific decision-making task of interest.

The MDP framework is a considerable abstraction of the problem of goal-directed learning from interaction. It proposes that whatever the details of the sensory, memory, and control apparatus, and whatever objective one is trying to achieve, any problem of learning goal-directed behavior can be reduced to three signals passing back and forth between an agent and its environment: one signal to represent the choices made by the agent (the actions), one signal to represent the basis on which the choices are made (the states), and one signal to define the agent's goal (the rewards). This framework may not be sufficient to represent all decision-learning problems usefully, but it has proved to be widely useful and applicable.

Of course, the particular states and actions vary greatly from task to task, and how they are represented can strongly affect performance. In reinforcement learning, as in other kinds of learning, such representational choices are at present more art than science.

In this book we offer some advice and examples regarding good ways of representing states and actions, but our primary focus is on general principles for learning how to behave once the representations have been selected.

Example 3.1: Bioreactor Suppose reinforcement learning is being applied to determine moment-by-moment temperatures and stirring rates for a bioreactor (a large vat of nutrients and bacteria used to produce useful chemicals). The actions in such an application might be target temperatures and target stirring rates that are passed to lower-level control systems that, in turn, directly activate heating elements and motors to attain the targets. The states are likely to be thermocouple and other sensory readings, perhaps filtered and delayed, plus symbolic inputs representing the ingredients in the vat and the target chemical. The rewards might be moment-by-moment measures of the rate at which the useful chemical is produced by the bioreactor. Notice that here each state is a list, or vector, of sensor readings and symbolic inputs, and each action is a vector consisting of a target temperature and a stirring rate. It is typical of reinforcement learning tasks to have states and actions with such structured representations. Rewards, on the other hand, are always single numbers. ■

Example 3.2: Pick-and-Place Robot Consider using reinforcement learning to control the motion of a robot arm in a repetitive pick-and-place task. If we want to learn movements that are fast and smooth, the learning agent will have to control the motors directly and have low-latency information about the current positions and velocities of the mechanical linkages. The actions in this case might be the voltages applied to each motor at each joint, and the states might be the latest readings of joint angles and velocities. The reward might be +1 for each object successfully picked up and placed. To encourage smooth movements, on each time step a small, negative reward could be given as a function of the moment-to-moment jerkiness of the motion. ■

Exercise 3.1 Devise three example tasks of your own that fit into the MDP framework, identifying for each its states, actions, and rewards. Make the three examples as *different* from each other as possible. The framework is abstract and flexible and can be applied in many different ways. Stretch its limits in some way in at least one of your examples. □

Exercise 3.2 Is the MDP framework adequate to usefully represent *all* goal-directed learning tasks? Can you think of any clear exceptions? □

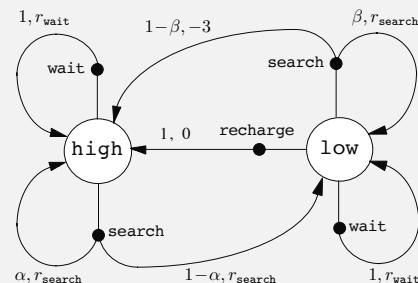
Exercise 3.3 Consider the problem of driving. You could define the actions in terms of the accelerator, steering wheel, and brake, that is, where your body meets the machine. Or you could define them farther out—say, where the rubber meets the road, considering your actions to be tire torques. Or you could define them farther in—say, where your brain meets your body, the actions being muscle twitches to control your limbs. Or you could go to a really high level and say that your actions are your choices of *where* to drive. What is the right level, the right place to draw the line between agent and environment? On what basis is one location of the line to be preferred over another? Is there any fundamental reason for preferring one location over another, or is it a free choice? □

Example 3.3 Recycling Robot

A mobile robot has the job of collecting empty soda cans in an office environment. It has sensors for detecting cans, and an arm and gripper that can pick them up and place them in an onboard bin; it runs on a rechargeable battery. The robot's control system has components for interpreting sensory information, for navigating, and for controlling the arm and gripper. High-level decisions about how to search for cans are made by a reinforcement learning agent based on the current charge level of the battery. To make a simple example, we assume that only two charge levels can be distinguished, comprising a small state set $S = \{\text{high}, \text{low}\}$. In each state, the agent can decide whether to (1) actively **search** for a can for a certain period of time, (2) remain stationary and **wait** for someone to bring it a can, or (3) head back to its home base to **recharge** its battery. When the energy level is **high**, recharging would always be foolish, so we do not include it in the action set for this state. The action sets are then $\mathcal{A}(\text{high}) = \{\text{search}, \text{wait}\}$ and $\mathcal{A}(\text{low}) = \{\text{search}, \text{wait}, \text{recharge}\}$.

The rewards are zero most of the time, but become positive when the robot secures an empty can, or large and negative if the battery runs all the way down. The best way to find cans is to actively search for them, but this runs down the robot's battery, whereas waiting does not. Whenever the robot is searching, the possibility exists that its battery will become depleted. In this case the robot must shut down and wait to be rescued (producing a low reward). If the energy level is **high**, then a period of active search can always be completed without risk of depleting the battery. A period of searching that begins with a **high** energy level leaves the energy level **high** with probability α and reduces it to **low** with probability $1 - \alpha$. On the other hand, a period of searching undertaken when the energy level is **low** leaves it **low** with probability β and depletes the battery with probability $1 - \beta$. In the latter case, the robot must be rescued, and the battery is then recharged back to **high**. Each can collected by the robot counts as a unit reward, whereas a reward of -3 results whenever the robot has to be rescued. Let r_{search} and r_{wait} , with $r_{\text{search}} > r_{\text{wait}}$, denote the expected number of cans the robot will collect (and hence the expected reward) while searching and while waiting respectively. Finally, suppose that no cans can be collected during a run home for recharging, and that no cans can be collected on a step in which the battery is depleted. This system is then a finite MDP, and we can write down the transition probabilities and the expected rewards, with dynamics as indicated in the table on the left:

s	a	s'	$p(s' s, a)$	$r(s, a, s')$
high	search	high	α	r_{search}
high	search	low	$1 - \alpha$	r_{search}
low	search	high	$1 - \beta$	-3
low	search	low	β	r_{search}
high	wait	high	1	r_{wait}
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	r_{wait}
low	recharge	high	1	0
low	recharge	low	0	-



Note that there is a row in the table for each possible combination of current state, s , action, $a \in \mathcal{A}(s)$, and next state, s' . Some transitions have zero probability of occurring, so no expected reward is specified for them. Shown on the right is another useful way of

summarizing the dynamics of a finite MDP, as a *transition graph*. There are two kinds of nodes: *state nodes* and *action nodes*. There is a state node for each possible state (a large open circle labeled by the name of the state), and an action node for each state-action pair (a small solid circle labeled by the name of the action and connected by a line to the state node). Starting in state s and taking action a moves you along the line from state node s to action node (s, a) . Then the environment responds with a transition to the next state's node via one of the arrows leaving action node (s, a) . Each arrow corresponds to a triple (s, s', a) , where s' is the next state, and we label the arrow with the transition probability, $p(s'|s, a)$, and the expected reward for that transition, $r(s, a, s')$. Note that the transition probabilities labeling the arrows leaving an action node always sum to 1.

Exercise 3.4 Give a table analogous to that in Example 3.3, but for $p(s', r|s, a)$. It should have columns for s , a , s' , r , and $p(s', r|s, a)$, and a row for every 4-tuple for which $p(s', r|s, a) > 0$. \square

3.2 Goals and Rewards

In reinforcement learning, the purpose or goal of the agent is formalized in terms of a special signal, called the *reward*, passing from the environment to the agent. At each time step, the reward is a simple number, $R_t \in \mathbb{R}$. Informally, the agent's goal is to maximize the total amount of reward it receives. This means maximizing not immediate reward, but cumulative reward in the long run. We can clearly state this informal idea as the *reward hypothesis*:

That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).

The use of a reward signal to formalize the idea of a goal is one of the most distinctive features of reinforcement learning.

Although formulating goals in terms of reward signals might at first appear limiting, in practice it has proved to be flexible and widely applicable. The best way to see this is to consider examples of how it has been, or could be, used. For example, to make a robot learn to walk, researchers have provided reward on each time step proportional to the robot's forward motion. In making a robot learn how to escape from a maze, the reward is often -1 for every time step that passes prior to escape; this encourages the agent to escape as quickly as possible. To make a robot learn to find and collect empty soda cans for recycling, one might give it a reward of zero most of the time, and then a reward of $+1$ for each can collected. One might also want to give the robot negative rewards when it bumps into things or when somebody yells at it. For an agent to learn to play checkers or chess, the natural rewards are $+1$ for winning, -1 for losing, and 0 for drawing and for all nonterminal positions.

You can see what is happening in all of these examples. The agent always learns to maximize its reward. If we want it to do something for us, we must provide rewards to it in such a way that in maximizing them the agent will also achieve our goals. It

is thus critical that the rewards we set up truly indicate what we want accomplished. In particular, the reward signal is not the place to impart to the agent prior knowledge about *how* to achieve what we want it to do.⁵ For example, a chess-playing agent should be rewarded only for actually winning, not for achieving subgoals such as taking its opponent’s pieces or gaining control of the center of the board. If achieving these sorts of subgoals were rewarded, then the agent might find a way to achieve them without achieving the real goal. For example, it might find a way to take the opponent’s pieces even at the cost of losing the game. The reward signal is your way of communicating to the agent *what* you want achieved, not *how* you want it achieved.⁶

3.3 Returns and Episodes

So far we have discussed informally the objective of learning. We have said that the agent’s goal is to maximize the cumulative reward it receives in the long run. How might this be defined formally? If the sequence of rewards received after time step t is denoted $R_{t+1}, R_{t+2}, R_{t+3}, \dots$, then what precise aspect of this sequence do we wish to maximize? In general, we seek to maximize the *expected return*, where the return, denoted G_t , is defined as some specific function of the reward sequence. In the simplest case the return is the sum of the rewards:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T, \quad (3.7)$$

where T is a final time step. This approach makes sense in applications in which there is a natural notion of final time step, that is, when the agent–environment interaction breaks naturally into subsequences, which we call *episodes*,⁷ such as plays of a game, trips through a maze, or any sort of repeated interaction. Each episode ends in a special state called the *terminal state*, followed by a reset to a standard starting state or to a sample from a standard distribution of starting states. Even if you think of episodes as ending in different ways, such as winning and losing a game, the next episode begins independently of how the previous one ended. Thus the episodes can all be considered to end in the same terminal state, with different rewards for the different outcomes. Tasks with episodes of this kind are called *episodic tasks*. In episodic tasks we sometimes need to distinguish the set of all nonterminal states, denoted \mathcal{S} , from the set of all states plus the terminal state, denoted \mathcal{S}^+ . The time of termination, T , is a random variable that normally varies from episode to episode.

On the other hand, in many cases the agent–environment interaction does not break naturally into identifiable episodes, but goes on continually without limit. For example, this would be the natural way to formulate an on-going process-control task, or an application to a robot with a long life span. We call these *continuing tasks*. The return formulation (3.7) is problematic for continuing tasks because the final time step would be $T = \infty$, and the return, which is what we are trying to maximize, could easily be infinite.

⁵Better places for imparting this kind of prior knowledge are the initial policy or initial value function.

⁶Section 17.4 delves further into the issue of designing effective reward signals.

⁷Episodes are sometimes called “trials” in the literature.

(For example, suppose the agent receives a reward of +1 at each time step.) Thus, in this book we usually use a definition of return that is slightly more complex conceptually but much simpler mathematically.

The additional concept that we need is that of *discounting*. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. In particular, it chooses A_t to maximize the expected *discounted return*:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \quad (3.8)$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate*.

The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. If $\gamma < 1$, the infinite sum in (3.8) has a finite value as long as the reward sequence $\{R_k\}$ is bounded. If $\gamma = 0$, the agent is “myopic” in being concerned only with maximizing immediate rewards: its objective in this case is to learn how to choose A_t so as to maximize only R_{t+1} . If each of the agent’s actions happened to influence only the immediate reward, not future rewards as well, then a myopic agent could maximize (3.8) by separately maximizing each immediate reward. But in general, acting to maximize immediate reward can reduce access to future rewards so that the return is reduced. As γ approaches 1, the return objective takes future rewards into account more strongly; the agent becomes more farsighted.

Returns at successive time steps are related to each other in a way that is important for the theory and algorithms of reinforcement learning:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \quad (3.9)$$

Note that this works for all time steps $t < T$, even if termination occurs at $t+1$, provided we define $G_T = 0$. This often makes it easy to compute returns from reward sequences.

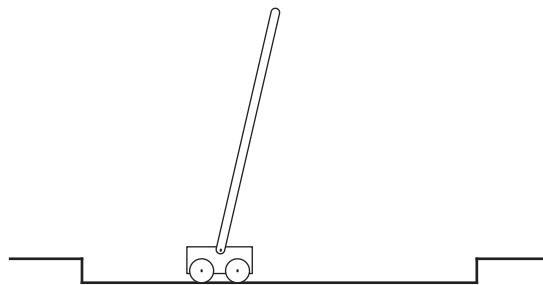
Note that although the return (3.8) is a sum of an infinite number of terms, it is still finite if the reward is nonzero and constant—if $\gamma < 1$. For example, if the reward is a constant +1, then the return is

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}. \quad (3.10)$$

Exercise 3.5 The equations in Section 3.1 are for the continuing case and need to be modified (very slightly) to apply to episodic tasks. Show that you know the modifications needed by giving the modified version of (3.3). \square

Example 3.4: Pole-Balancing

The objective in this task is to apply forces to a cart moving along a track so as to keep a pole hinged to the cart from falling over: A failure is said to occur if the pole falls past a given angle from vertical or if the cart runs off the track. The pole is reset to vertical after each failure. This task could be treated as episodic, where the natural episodes are the repeated attempts to balance the pole. The reward in this case could be +1 for every time step on which failure did not occur, so that the return at each time would be the number of steps until failure. In this case, successful balancing forever would mean a return of infinity. Alternatively, we could treat pole-balancing as a continuing task, using discounting. In this case the reward would be -1 on each failure and zero at all other times. The return at each time would then be related to $-\gamma^{K-1}$, where K is the number of time steps before failure (as well as to the times of later failures). In either case, the return is maximized by keeping the pole balanced for as long as possible. ■



Exercise 3.6 Suppose you treated pole-balancing as an episodic task but also used discounting, with all rewards zero except for -1 upon failure. What then would the return be at each time? How does this return differ from that in the discounted, continuing formulation of this task? □

Exercise 3.7 Imagine that you are designing a robot to run a maze. You decide to give it a reward of +1 for escaping from the maze and a reward of zero at all other times. The task seems to break down naturally into episodes—the successive runs through the maze—so you decide to treat it as an episodic task, where the goal is to maximize expected total reward (3.7). After running the learning agent for a while, you find that it is showing no improvement in escaping from the maze. What is going wrong? Have you effectively communicated to the agent what you want it to achieve? □

Exercise 3.8 Suppose $\gamma = 0.5$ and the following sequence of rewards is received $R_1 = -1$, $R_2 = 2$, $R_3 = 6$, $R_4 = 3$, and $R_5 = 2$, with $T = 5$. What are G_0 , G_1 , ..., G_5 ? Hint: Work backwards. □

Exercise 3.9 Suppose $\gamma = 0.9$ and the reward sequence is $R_1 = 2$ followed by an infinite sequence of 7s. What are G_1 and G_0 ? □

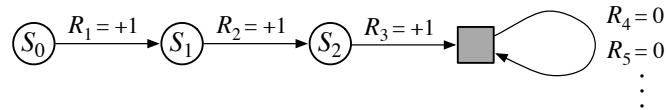
Exercise 3.10 Prove the second equality in (3.10). □

3.4 Unified Notation for Episodic and Continuing Tasks

In the preceding section we described two kinds of reinforcement learning tasks, one in which the agent–environment interaction naturally breaks down into a sequence of separate episodes (episodic tasks), and one in which it does not (continuing tasks). The former case is mathematically easier because each action affects only the finite number of rewards subsequently received during the episode. In this book we consider sometimes one kind of problem and sometimes the other, but often both. It is therefore useful to establish one notation that enables us to talk precisely about both cases simultaneously.

To be precise about episodic tasks requires some additional notation. Rather than one long sequence of time steps, we need to consider a series of episodes, each of which consists of a finite sequence of time steps. We number the time steps of each episode starting anew from zero. Therefore, we have to refer not just to S_t , the state representation at time t , but to $S_{t,i}$, the state representation at time t of episode i (and similarly for $A_{t,i}$, $R_{t,i}$, $\pi_{t,i}$, T_i , etc.). However, it turns out that when we discuss episodic tasks we almost never have to distinguish between different episodes. We are almost always considering a particular episode, or stating something that is true for all episodes. Accordingly, in practice we almost always abuse notation slightly by dropping the explicit reference to episode number. That is, we write S_t to refer to $S_{t,i}$, and so on.

We need one other convention to obtain a single notation that covers both episodic and continuing tasks. We have defined the return as a sum over a finite number of terms in one case (3.7) and as a sum over an infinite number of terms in the other (3.8). These two can be unified by considering episode termination to be the entering of a special *absorbing state* that transitions only to itself and that generates only rewards of zero. For example, consider the state transition diagram:



Here the solid square represents the special absorbing state corresponding to the end of an episode. Starting from S_0 , we get the reward sequence $+1, +1, +1, 0, 0, 0, \dots$. Summing these, we get the same return whether we sum over the first T rewards (here $T = 3$) or over the full infinite sequence. This remains true even if we introduce discounting. Thus, we can define the return, in general, according to (3.8), using the convention of omitting episode numbers when they are not needed, and including the possibility that $\gamma = 1$ if the sum remains defined (e.g., because all episodes terminate). Alternatively, we can write

$$G_t \doteq \sum_{k=t+1}^T \gamma^{k-t-1} R_k, \quad (3.11)$$

including the possibility that $T = \infty$ or $\gamma = 1$ (but not both). We use these conventions throughout the rest of the book to simplify notation and to express the close parallels

between episodic and continuing tasks. (Later, in Chapter 10, we will introduce a formulation that is both continuing and undiscounted.)

3.5 Policies and Value Functions

Almost all reinforcement learning algorithms involve estimating *value functions*—functions of states (or of state-action pairs) that estimate *how good* it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The notion of “how good” here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return. Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular ways of acting, called policies.

Formally, a *policy* is a mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t , then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Like p , π is an ordinary function; the “|” in the middle of $\pi(a|s)$ merely reminds us that it defines a probability distribution over $a \in \mathcal{A}(s)$ for each $s \in \mathcal{S}$. Reinforcement learning methods specify how the agent’s policy is changed as a result of its experience.

Exercise 3.11 If the current state is S_t , and actions are selected according to a stochastic policy π , then what is the expectation of R_{t+1} in terms of π and the four-argument function p (3.2)? \square

The *value function* of a state s under a policy π , denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs, we can define v_π formally by

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right], \text{ for all } s \in \mathcal{S}, \quad (3.12)$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π , and t is any time step. Note that the value of the terminal state, if any, is always zero. We call the function v_π the *state-value function for policy π* .

Similarly, we define the value of taking action a in state s under a policy π , denoted $q_\pi(s, a)$, as the expected return starting from s , taking the action a , and thereafter following policy π :

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right]. \quad (3.13)$$

We call q_π the *action-value function for policy π* .

Exercise 3.12 Give an equation for v_π in terms of q_π and π . \square

Exercise 3.13 Give an equation for q_π in terms of v_π and the four-argument p . \square

The value functions v_π and q_π can be estimated from experience. For example, if an agent follows policy π and maintains an average, for each state encountered, of the actual returns that have followed that state, then the average will converge to the state’s value, $v_\pi(s)$, as the number of times that state is encountered approaches infinity. If separate

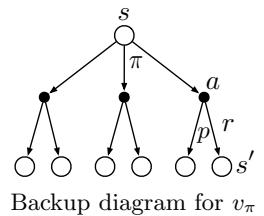
averages are kept for each action taken in each state, then these averages will similarly converge to the action values, $q_\pi(s, a)$. We call estimation methods of this kind *Monte Carlo methods* because they involve averaging over many random samples of actual returns. These kinds of methods are presented in Chapter 5. Of course, if there are very many states, then it may not be practical to keep separate averages for each state individually. Instead, the agent would have to maintain v_π and q_π as parameterized functions (with fewer parameters than states) and adjust the parameters to better match the observed returns. This can also produce accurate estimates, although much depends on the nature of the parameterized function approximator. These possibilities are discussed in Part II of the book.

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy recursive relationships similar to that which we have already established for the return (3.9). For any policy π and any state s , the following consistency condition holds between the value of s and the value of its possible successor states:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \tag{by (3.9)} \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s']] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) [r + \gamma v_\pi(s')], \quad \text{for all } s \in \mathcal{S}, \end{aligned} \quad (3.14)$$

where it is implicit that the actions, a , are taken from the set $\mathcal{A}(s)$, that the next states, s' , are taken from the set \mathcal{S} (or from \mathcal{S}^+ in the case of an episodic problem), and that the rewards, r , are taken from the set \mathcal{R} . Note also how in the last equation we have merged the two sums, one over all the values of s' and the other over all the values of r , into one sum over all the possible values of both. We use this kind of merged sum often to simplify formulas. Note how the final expression can be read easily as an expected value. It is really a sum over all values of the three variables, a , s' , and r . For each triple, we compute its probability, $\pi(a|s)p(s', r | s, a)$, weight the quantity in brackets by that probability, then sum over all possibilities to get an expected value.

Equation (3.14) is the *Bellman equation* for v_π . It expresses a relationship between the value of a state and the values of its successor states. Think of looking ahead from a state to its possible successor states, as suggested by the diagram to the right. Each open circle represents a state and each solid circle represents a state-action pair. Starting from state s , the root node at the top, the agent could take any of some set of actions—three are shown in the diagram—based on its policy π . From each of these, the environment could respond with one of several next states, s' (two are shown in the figure), along with a reward, r , depending on its dynamics given by the function p . The Bellman equation (3.14) averages over all the possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way.



Backup diagram for v_π

The value function v_π is the unique solution to its Bellman equation. We show in subsequent chapters how this Bellman equation forms the basis of a number of ways to compute, approximate, and learn v_π . We call diagrams like that above *backup diagrams* because they diagram relationships that form the basis of the update or *backup* operations that are at the heart of reinforcement learning methods. These operations transfer value information *back* to a state (or a state–action pair) from its successor states (or state–action pairs). We use backup diagrams throughout the book to provide graphical summaries of the algorithms we discuss. (Note that, unlike transition graphs, the state nodes of backup diagrams do not necessarily represent distinct states; for example, a state might be its own successor.)

Example 3.5: Gridworld Figure 3.2 (left) shows a rectangular gridworld representation of a simple finite MDP. The cells of the grid correspond to the states of the environment. At each cell, four actions are possible: **north**, **south**, **east**, and **west**, which deterministically cause the agent to move one cell in the respective direction on the grid. Actions that would take the agent off the grid leave its location unchanged, but also result in a reward of -1 . Other actions result in a reward of 0 , except those that move the agent out of the special states **A** and **B**. From state **A**, all four actions yield a reward of $+10$ and take the agent to **A'**. From state **B**, all actions yield a reward of $+5$ and take the agent to **B'**.

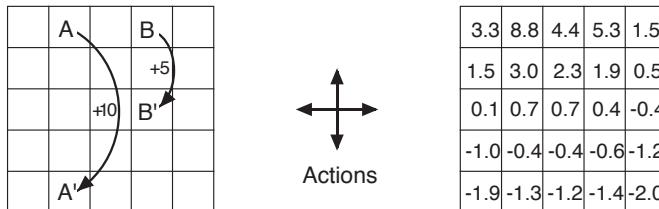


Figure 3.2: Gridworld example: exceptional reward dynamics (left) and state-value function for the equiprobable random policy (right).

Suppose the agent selects all four actions with equal probability in all states. Figure 3.2 (right) shows the value function, v_π , for this policy, for the discounted reward case with $\gamma = 0.9$. This value function was computed by solving the system of linear equations (3.14). Notice the negative values near the lower edge; these are the result of the high probability of hitting the edge of the grid there under the random policy. State **A** is the best state to be in under this policy. Note that **A**'s expected return is *less* than its immediate reward of 10 , because from **A** the agent is taken to state **A'** from which it is likely to run into the edge of the grid. State **B**, on the other hand, is valued *more* than its immediate reward of 5 , because from **B** the agent is taken to **B'** which has a positive value. From **B'** the expected penalty (negative reward) for possibly running into an edge is more than compensated for by the expected gain for possibly stumbling onto **A** or **B**. ■

Exercise 3.14 The Bellman equation (3.14) must hold for each state for the value function v_π shown in Figure 3.2 (right) of Example 3.5. Show numerically that this equation holds for the center state, valued at $+0.7$, with respect to its four neighboring states, valued at $+2.3$, $+0.4$, -0.4 , and $+0.7$. (These numbers are accurate only to one decimal place.) □

Exercise 3.15 In the gridworld example, rewards are positive for goals, negative for running into the edge of the world, and zero the rest of the time. Are the signs of these rewards important, or only the intervals between them? Prove, using (3.8), that adding a constant c to all the rewards adds a constant, v_c , to the values of all states, and thus does not affect the relative values of any states under any policies. What is v_c in terms of c and γ ? \square

Exercise 3.16 Now consider adding a constant c to all the rewards in an episodic task, such as maze running. Would this have any effect, or would it leave the task unchanged as in the continuing task above? Why or why not? Give an example. \square

Example 3.6: Golf To formulate playing a hole of golf as a reinforcement learning task, we count a penalty (negative reward) of -1 for each stroke until we hit the ball into the hole. The state is the location of the ball. The value of a state is the negative of the number of strokes to the hole from that location. Our actions are how we aim and swing at the ball, of course, and which club we select. Let us take the former as given and consider just the choice of club, which we assume is either a putter or a driver. The upper part of Figure 3.3 shows a possible state-value function, $v_{\text{putt}}(s)$, for the policy that always uses the putter. The terminal state *in-the-hole* has a value of 0 . From anywhere on the green we assume we can make a putt; these states have value -1 . Off the green we cannot reach the hole by putting, and the value is lower. If we can reach the green from a state by putting, then that state must have value one less than the green's value, that is, -2 . For simplicity, let us assume we can putt very precisely and deterministically, but with a limited range. This gives us the sharp contour line labeled -2 in the figure; all locations between that line and the green require exactly two strokes to complete the hole. Similarly, any location within putting range of the -2 contour line must have a value of -3 , and so on to get all the contour lines shown in the figure. Putting doesn't get us out of sand traps, so they have a value of $-\infty$. Overall, it takes us six strokes to get from the tee to the hole by putting.

Exercise 3.17 What is the Bellman equation for action values, that is, for q_π ? It must give the action value $q_\pi(s, a)$ in terms of the action values, $q_\pi(s', a')$, of possible successors to the state-action pair (s, a) . Hint: The backup diagram to the right corresponds to this equation. Show the sequence of equations analogous to (3.14), but for action values. \square

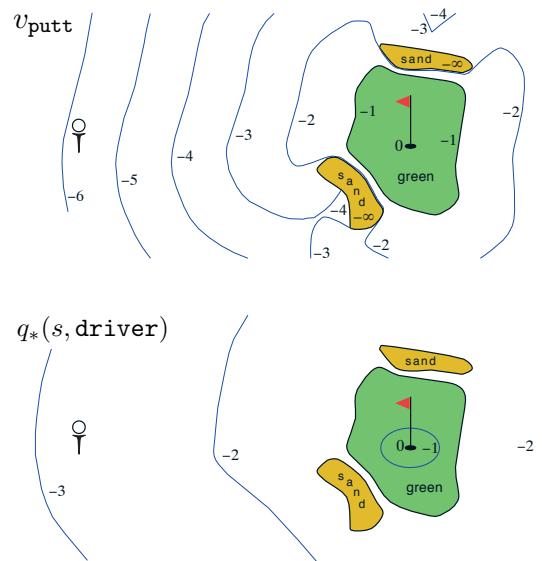
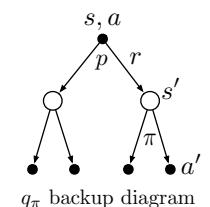
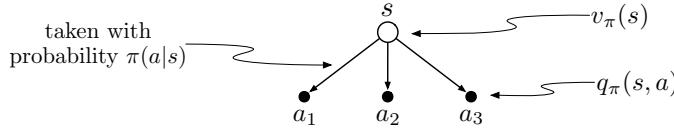


Figure 3.3: A golf example: the state-value function for putting (upper) and the optimal action-value function for using the driver (lower). \blacksquare

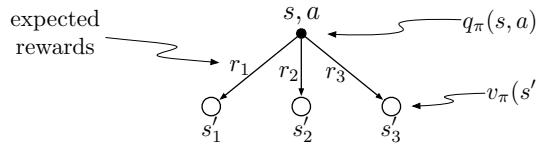


Exercise 3.18 The value of a state depends on the values of the actions possible in that state and on how likely each action is to be taken under the current policy. We can think of this in terms of a small backup diagram rooted at the state and considering each possible action:



Give the equation corresponding to this intuition and diagram for the value at the root node, $v_\pi(s)$, in terms of the value at the expected leaf node, $q_\pi(s, a)$, given $S_t = s$. This equation should include an expectation conditioned on following the policy, π . Then give a second equation in which the expected value is written out explicitly in terms of $\pi(a|s)$ such that no expected value notation appears in the equation. \square

Exercise 3.19 The value of an action, $q_\pi(s, a)$, depends on the expected next reward and the expected sum of the remaining rewards. Again we can think of this in terms of a small backup diagram, this one rooted at an action (state-action pair) and branching to the possible next states:



Give the equation corresponding to this intuition and diagram for the action value, $q_\pi(s, a)$, in terms of the expected next reward, R_{t+1} , and the expected next state value, $v_\pi(S_{t+1})$, given that $S_t = s$ and $A_t = a$. This equation should include an expectation but *not* one conditioned on following the policy. Then give a second equation, writing out the expected value explicitly in terms of $p(s', r | s, a)$ defined by (3.2), such that no expected value notation appears in the equation. \square

3.6 Optimal Policies and Optimal Value Functions

Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run. For finite MDPs, we can precisely define an optimal policy in the following way. Value functions define a partial ordering over policies. A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies. This is an *optimal policy*. Although there may be more than one, we denote all the optimal policies by π_* . They share the same state-value function, called the *optimal state-value function*, denoted v_* , and defined as

$$v_*(s) \doteq \max_{\pi} v_{\pi}(s), \tag{3.15}$$

for all $s \in \mathcal{S}$.

Optimal policies also share the same *optimal action-value function*, denoted q_* , and defined as

$$q_*(s, a) \doteq \max_{\pi} q_{\pi}(s, a), \quad (3.16)$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. For the state-action pair (s, a) , this function gives the expected return for taking action a in state s and thereafter following an optimal policy. Thus, we can write q_* in terms of v_* as follows:

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]. \quad (3.17)$$

Example 3.7: Optimal Value Functions for Golf The lower part of Figure 3.3 shows the contours of a possible optimal action-value function $q_*(s, \text{driver})$. These are the values of each state if we first play a stroke with the driver and afterward select either the driver or the putter, whichever is better. The driver enables us to hit the ball farther, but with less accuracy. We can reach the hole in one shot using the driver only if we are already very close; thus the -1 contour for $q_*(s, \text{driver})$ covers only a small portion of the green. If we have two strokes, however, then we can reach the hole from much farther away, as shown by the -2 contour. In this case we don't have to drive all the way to within the small -1 contour, but only to anywhere on the green; from there we can use the putter. The optimal action-value function gives the values after committing to a particular *first* action, in this case, to the driver, but afterward using whichever actions are best. The -3 contour is still farther out and includes the starting tee. From the tee, the best sequence of actions is two drives and one putt, sinking the ball in three strokes. ■

Because v_* is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values (3.14). Because it is the optimal value function, however, v_* 's consistency condition can be written in a special form without reference to any specific policy. This is the Bellman equation for v_* , or the *Bellman optimality equation*. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \quad (\text{by (3.9)}) \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \end{aligned} \quad (3.18)$$

$$= \max_a \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_*(s')]. \quad (3.19)$$

The last two equations are two forms of the Bellman optimality equation for v_* . The Bellman optimality equation for q_* is

$$\begin{aligned} q_*(s, a) &= \mathbb{E} \left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a \right] \\ &= \sum_{s', r} p(s', r | s, a) \left[r + \gamma \max_{a'} q_*(s', a') \right]. \end{aligned} \quad (3.20)$$

The backup diagrams in the figure below show graphically the spans of future states and actions considered in the Bellman optimality equations for v_* and q_* . These are the same as the backup diagrams for v_π and q_π presented earlier except that arcs have been added at the agent's choice points to represent that the maximum over that choice is taken rather than the expected value given some policy. The backup diagram on the left graphically represents the Bellman optimality equation (3.19) and the backup diagram on the right graphically represents (3.20).

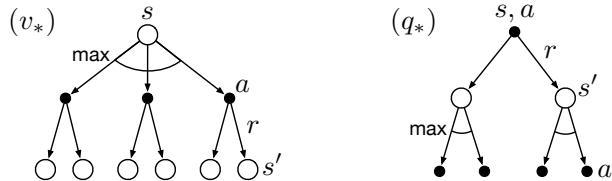


Figure 3.4: Backup diagrams for v_* and q_*

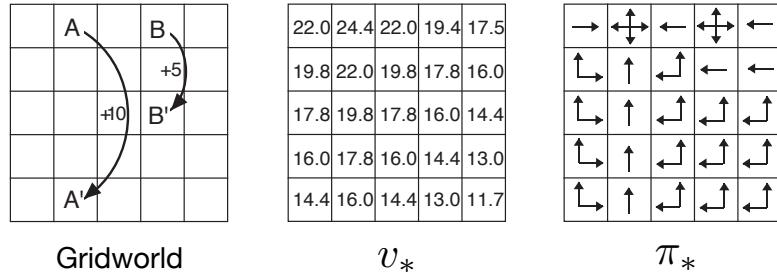
For finite MDPs, the Bellman optimality equation for v_* (3.19) has a unique solution. The Bellman optimality equation is actually a system of equations, one for each state, so if there are n states, then there are n equations in n unknowns. If the dynamics p of the environment are known, then in principle one can solve this system of equations for v_* using any one of a variety of methods for solving systems of nonlinear equations. One can solve a related set of equations for q_* .

Once one has v_* , it is relatively easy to determine an optimal policy. For each state s , there will be one or more actions at which the maximum is obtained in the Bellman optimality equation. Any policy that assigns nonzero probability only to these actions is an optimal policy. You can think of this as a one-step search. If you have the optimal value function, v_* , then the actions that appear best after a one-step search will be optimal actions. Another way of saying this is that any policy that is *greedy* with respect to the optimal evaluation function v_* is an optimal policy. The term *greedy* is used in computer science to describe any search or decision procedure that selects alternatives based only on local or immediate considerations, without considering the possibility that such a selection may prevent future access to even better alternatives. Consequently, it describes policies that select actions based only on their short-term consequences. The beauty of v_* is that if one uses it to evaluate the short-term consequences of actions—specifically, the one-step consequences—then a greedy policy is actually optimal in the long-term sense in which we are interested because v_* already takes into account the reward consequences of all possible future behavior. By means of v_* , the optimal expected long-term return is

turned into a quantity that is locally and immediately available for each state. Hence, a one-step-ahead search yields the long-term optimal actions.

Having q_* makes choosing optimal actions even easier. With q_* , the agent does not even have to do a one-step-ahead search: for any state s , it can simply find any action that maximizes $q_*(s, a)$. The action-value function effectively caches the results of all one-step-ahead searches. It provides the optimal expected long-term return as a value that is locally and immediately available for each state-action pair. Hence, at the cost of representing a function of state-action pairs, instead of just of states, the optimal action-value function allows optimal actions to be selected without having to know anything about possible successor states and their values, that is, without having to know anything about the environment's dynamics.

Example 3.8: Solving the Gridworld Suppose we solve the Bellman equation for v_* for the simple grid task introduced in Example 3.5 and shown again in Figure 3.5 (left). Recall that state A is followed by a reward of +10 and transition to state A', while state B is followed by a reward of +5 and transition to state B'. Figure 3.5 (middle) shows the optimal value function, and Figure 3.5 (right) shows the corresponding optimal policies. Where there are multiple arrows in a cell, all of the corresponding actions are optimal.



Gridworld

 v_* π_*

Figure 3.5: Optimal solutions to the gridworld example. ■

Example 3.9: Bellman Optimality Equations for the Recycling Robot Using (3.19), we can explicitly give the Bellman optimality equation for the recycling robot example. To make things more compact, we abbreviate the states `high` and `low`, and the actions `search`, `wait`, and `recharge` respectively by h , l , s , w , and re . Because there are only two states, the Bellman optimality equation consists of two equations. The equation for $v_*(h)$ can be written as follows:

$$\begin{aligned}
 v_*(h) &= \max \left\{ \begin{array}{l} p(h|h, s)[r(h, s, h) + \gamma v_*(h)] + p(l|h, s)[r(h, s, l) + \gamma v_*(l)], \\ p(h|w, h)[r(h, w, h) + \gamma v_*(h)] + p(l|w, h)[r(h, w, l) + \gamma v_*(l)] \end{array} \right\} \\
 &= \max \left\{ \begin{array}{l} \alpha[r_s + \gamma v_*(h)] + (1 - \alpha)[r_s + \gamma v_*(l)], \\ 1[r_w + \gamma v_*(h)] + 0[r_w + \gamma v_*(l)] \end{array} \right\} \\
 &= \max \left\{ \begin{array}{l} r_s + \gamma[\alpha v_*(h) + (1 - \alpha)v_*(l)], \\ r_w + \gamma v_*(h) \end{array} \right\}.
 \end{aligned}$$

Following the same procedure for $v_*(1)$ yields the equation

$$v_*(1) = \max \left\{ \begin{array}{l} \beta r_s - 3(1-\beta) + \gamma[(1-\beta)v_*(h) + \beta v_*(1)], \\ r_w + \gamma v_*(1), \\ \gamma v_*(h) \end{array} \right\}.$$

For any choice of r_s , r_w , α , β , and γ , with $0 \leq \gamma < 1$, $0 \leq \alpha, \beta \leq 1$, there is exactly one pair of numbers, $v_*(h)$ and $v_*(1)$, that simultaneously satisfy these two nonlinear equations. ■

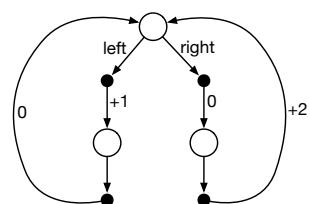
Explicitly solving the Bellman optimality equation provides one route to finding an optimal policy, and thus to solving the reinforcement learning problem. However, this solution is rarely directly useful. It is akin to an exhaustive search, looking ahead at all possibilities, computing their probabilities of occurrence and their desirabilities in terms of expected rewards. This solution relies on at least three assumptions that are rarely true in practice: (1) the dynamics of the environment are accurately known; (2) computational resources are sufficient to complete the calculation; and (3) the states have the Markov property. For the kinds of tasks in which we are interested, one is generally not able to implement this solution exactly because various combinations of these assumptions are violated. For example, although the first and third assumptions present no problems for the game of backgammon, the second is a major impediment. Because the game has about 10^{20} states, it would take thousands of years on today's fastest computers to solve the Bellman equation for v_* , and the same is true for finding q_* . In reinforcement learning one typically has to settle for approximate solutions.

Many different decision-making methods can be viewed as ways of approximately solving the Bellman optimality equation. For example, heuristic search methods can be viewed as expanding the right-hand side of (3.19) several times, up to some depth, forming a “tree” of possibilities, and then using a heuristic evaluation function to approximate v_* at the “leaf” nodes. (Heuristic search methods such as A* are almost always based on the episodic case.) The methods of dynamic programming can be related even more closely to the Bellman optimality equation. Many reinforcement learning methods can be clearly understood as approximately solving the Bellman optimality equation, using actual experienced transitions in place of knowledge of the expected transitions. We consider a variety of such methods in the following chapters.

Exercise 3.20 Draw or describe the optimal state-value function for the golf example. □

Exercise 3.21 Draw or describe the contours of the optimal action-value function for putting, $q_*(s, \text{putter})$, for the golf example. □

Exercise 3.22 Consider the continuing MDP shown to the right. The only decision to be made is that in the top state, where two actions are available, left and right. The numbers show the rewards that are received deterministically after each action. There are exactly two deterministic policies, π_{left} and π_{right} . What policy is optimal if $\gamma = 0$? If $\gamma = 0.9$? If $\gamma = 0.5$? □



Exercise 3.23 Give the Bellman equation for q_* for the recycling robot. □

Exercise 3.24 Figure 3.5 gives the optimal value of the best state of the gridworld as 24.4, to one decimal place. Use your knowledge of the optimal policy and (3.8) to express this value symbolically, and then to compute it to three decimal places. □

Exercise 3.25 Give an equation for v_* in terms of q_* . □

Exercise 3.26 Give an equation for q_* in terms of v_* and the four-argument p . □

Exercise 3.27 Give an equation for π_* in terms of q_* . □

Exercise 3.28 Give an equation for π_* in terms of v_* and the four-argument p . □

Exercise 3.29 Rewrite the four Bellman equations for the four value functions (v_π , v_* , q_π , and q_*) in terms of the three argument function p (3.4) and the two-argument function r (3.5). □

3.7 Optimality and Approximation

We have defined optimal value functions and optimal policies. Clearly, an agent that learns an optimal policy has done very well, but in practice this rarely happens. For the kinds of tasks in which we are interested, optimal policies can be generated only with extreme computational cost. A well-defined notion of optimality organizes the approach to learning we describe in this book and provides a way to understand the theoretical properties of various learning algorithms, but it is an ideal that agents can only approximate. As we discussed above, even if we have a complete and accurate model of the environment's dynamics, it is usually not possible to simply compute an optimal policy by solving the Bellman optimality equation. For example, board games such as chess are a tiny fraction of human experience, yet large, custom-designed computers still cannot compute the optimal moves. A critical aspect of the problem facing the agent is always the computational power available to it, in particular, the amount of computation it can perform in a single time step.

The memory available is also an important constraint. A large amount of memory is often required to build up approximations of value functions, policies, and models. In tasks with small, finite state sets, it is possible to form these approximations using arrays or tables with one entry for each state (or state-action pair). This we call the *tabular* case, and the corresponding methods we call tabular methods. In many cases of practical interest, however, there are far more states than could possibly be entries in a table. In these cases the functions must be approximated, using some sort of more compact parameterized function representation.

Our framing of the reinforcement learning problem forces us to settle for approximations. However, it also presents us with some unique opportunities for achieving useful approximations. For example, in approximating optimal behavior, there may be many states that the agent faces with such a low probability that selecting suboptimal actions for them has little impact on the amount of reward the agent receives. Tesauro's backgammon player, for example, plays with exceptional skill even though it might make

very bad decisions on board configurations that never occur in games against experts. In fact, it is possible that TD-Gammon makes bad decisions for a large fraction of the game's state set. The online nature of reinforcement learning makes it possible to approximate optimal policies in ways that put more effort into learning to make good decisions for frequently encountered states, at the expense of less effort for infrequently encountered states. This is one key property that distinguishes reinforcement learning from other approaches to approximately solving MDPs.

3.8 Summary

Let us summarize the elements of the reinforcement learning problem that we have presented in this chapter. Reinforcement learning is about learning from interaction how to behave in order to achieve a goal. The reinforcement learning *agent* and its *environment* interact over a sequence of discrete time steps. The specification of their interface defines a particular task: the *actions* are the choices made by the agent; the *states* are the basis for making the choices; and the *rewards* are the basis for evaluating the choices. Everything inside the agent is known and controllable. Its environment, on the other hand, is incompletely controllable and may or may not be completely known. A *policy* is a stochastic rule by which the agent selects actions as a function of states. The agent's objective is to maximize the amount of reward it receives over time.

When the reinforcement learning setup described above is formulated with well defined transition probabilities it constitutes a *Markov decision process* (MDP). A *finite MDP* is an MDP with finite state, action, and (as we formulate it here) reward sets. Much of the current theory of reinforcement learning is restricted to finite MDPs, but the methods and ideas apply more generally.

The *return* is the function of future rewards that the agent seeks to maximize (in expected value). It has several different definitions depending upon the nature of the task and whether one wishes to *discount* delayed reward. The undiscounted formulation is appropriate for *episodic tasks*, in which the agent–environment interaction breaks naturally into *episodes*; the discounted formulation is appropriate for *continuing tasks*, in which the interaction does not naturally break into episodes but continues without limit. We try to define the returns for the two kinds of tasks such that one set of equations can apply to both the episodic and continuing cases.

A policy's *value functions* (v_π and q_π) assign to each state, or state–action pair, the expected return from that state, or state–action pair, given that the agent uses the policy. The *optimal value functions* (v_* and q_*) assign to each state, or state–action pair, the largest expected return achievable by any policy. A policy whose value functions are optimal is an *optimal policy*. Whereas the optimal value functions for states and state–action pairs are unique for a given MDP, there can be many optimal policies. Any policy that is *greedy* with respect to the optimal value functions must be an optimal policy. The *Bellman optimality equations* are special consistency conditions that the optimal value functions must satisfy and that can, in principle, be solved for the optimal value functions, from which an optimal policy can be determined with relative ease.

A reinforcement learning problem can be posed in a variety of different ways depending on assumptions about the level of knowledge initially available to the agent. In problems of *complete knowledge*, the agent has a complete and accurate model of the environment's dynamics. If the environment is an MDP, then such a model consists of the complete four-argument dynamics function p (3.2). In problems of *incomplete knowledge*, a complete and perfect model of the environment is not available.

Even if the agent had a complete and accurate environment model, the agent would typically be unable to fully use it because of limitations on its memory and computation per time step. In particular, extensive memory may be required to build up accurate approximations of value functions, policies, and models. In most cases of practical interest there are far more states than could possibly be entries in a table, and approximations must be made.

A well-defined notion of optimality organizes the approach to learning we describe in this book and provides a way to understand the theoretical properties of various learning algorithms, but it is an ideal that reinforcement learning agents can only approximate to varying degrees. In reinforcement learning we are very much concerned with cases in which optimal solutions cannot be found but must be approximated in some way.

Bibliographical and Historical Remarks

The reinforcement learning problem is deeply indebted to the idea of Markov decision processes (MDPs) from the field of optimal control. These historical influences and other major influences from psychology are described in the brief history given in Chapter 1. Reinforcement learning adds to MDPs a focus on approximation and incomplete information for realistically large problems. MDPs and the reinforcement learning problem are only weakly linked to traditional learning and decision-making problems in artificial intelligence. However, artificial intelligence is now vigorously exploring MDP formulations for planning and decision making from a variety of perspectives. MDPs are more general than previous formulations used in artificial intelligence in that they permit more general kinds of goals and uncertainty.

The theory of MDPs is treated by, for example, Bertsekas (2005), White (1969), Whittle (1982, 1983), and Puterman (1994). A particularly compact treatment of the finite case is given by Ross (1983). MDPs are also studied under the heading of stochastic optimal control, where *adaptive* optimal control methods are most closely related to reinforcement learning (e.g., Kumar, 1985; Kumar and Varaiya, 1986).

The theory of MDPs evolved from efforts to understand the problem of making sequences of decisions under uncertainty, where each decision can depend on the previous decisions and their outcomes. It is sometimes called the theory of multistage decision processes, or sequential decision processes, and has roots in the statistical literature on sequential sampling beginning with the papers by Thompson (1933, 1934) and Robbins (1952) that we cited in Chapter 2 in connection with bandit problems (which are prototypical MDPs if formulated as multiple-situation problems).

The earliest instance (that we are aware of) in which reinforcement learning was discussed using the MDP formalism is Andreae's (1969) description of a unified view of

learning machines. Witten and Corbin (1973) experimented with a reinforcement learning system later analyzed by Witten (1977, 1976a) using the MDP formalism. Although he did not explicitly mention MDPs, Werbos (1977) suggested approximate solution methods for stochastic optimal control problems that are related to modern reinforcement learning methods (see also Werbos, 1982, 1987, 1988, 1989, 1992). Although Werbos's ideas were not widely recognized at the time, they were prescient in emphasizing the importance of approximately solving optimal control problems in a variety of domains, including artificial intelligence. The most influential integration of reinforcement learning and MDPs is due to Watkins (1989).

- 3.1** Our characterization of the dynamics of an MDP in terms of $p(s', r|s, a)$ is slightly unusual. It is more common in the MDP literature to describe the dynamics in terms of the state transition probabilities $p(s'|s, a)$ and expected next rewards $r(s, a)$. In reinforcement learning, however, we more often have to refer to individual actual or sample rewards (rather than just their expected values). Our notation also makes it plainer that S_t and R_t are in general jointly determined, and thus must have the same time index. In teaching reinforcement learning, we have found our notation to be more straightforward conceptually and easier to understand.

For a good intuitive discussion of the system-theoretic concept of state, see Minsky (1967).

The bioreactor example is based on the work of Ungar (1990) and Miller and Williams (1992). The recycling robot example was inspired by the can-collecting robot built by Jonathan Connell (1989). Kober and Peters (2012) present a collection of robotics applications of reinforcement learning.

- 3.2** An explicit statement of the reward hypothesis was suggested by Michael Littman (personal communication).

- 3.3–4** The terminology of *episodic* and *continuing* tasks is different from that usually used in the MDP literature. In that literature it is common to distinguish three types of tasks: (1) finite-horizon tasks, in which interaction terminates after a particular *fixed* number of time steps; (2) indefinite-horizon tasks, in which interaction can last arbitrarily long but must eventually terminate; and (3) infinite-horizon tasks, in which interaction does not terminate. Our episodic and continuing tasks are similar to indefinite-horizon and infinite-horizon tasks, respectively, but we prefer to emphasize the difference in the nature of the interaction. This difference seems more fundamental than the difference in the objective functions emphasized by the usual terms. Often episodic tasks use an indefinite-horizon objective function and continuing tasks an infinite-horizon objective function, but we see this as a common coincidence rather than a fundamental difference.

The pole-balancing example is from Michie and Chambers (1968) and Barto, Sutton, and Anderson (1983).

3.5–6 Assigning value on the basis of what is good or bad in the long run has ancient roots. In control theory, mapping states to numerical values representing the long-term consequences of control decisions is a key part of optimal control theory, which was developed in the 1950s by extending nineteenth century state-function theories of classical mechanics (see, for example, Schultz and Melsa, 1967). In describing how a computer could be programmed to play chess, Shannon (1950) suggested using an evaluation function that took into account the long-term advantages and disadvantages of chess positions.

Watkins's (1989) Q-learning algorithm for estimating q_* (Chapter 6) made action-value functions an important part of reinforcement learning, and consequently these functions are often called “Q-functions.” But the idea of an action-value function is much older than this. Shannon (1950) suggested that a function $h(P, M)$ could be used by a chess-playing program to decide whether a move M in position P is worth exploring. Michie's (1961, 1963) MENACE system and Michie and Chambers's (1968) BOXES system can be understood as estimating action-value functions. In classical physics, Hamilton's principal function is an action-value function; Newtonian dynamics are greedy with respect to this function (e.g., Goldstein, 1957). Action-value functions also played a central role in Denardo's (1967) theoretical treatment of dynamic programming in terms of contraction mappings.

The Bellman optimality equation (for v_*) was popularized by Richard Bellman (1957a), who called it the “basic functional equation.” The counterpart of the Bellman optimality equation for continuous time and state problems is known as the Hamilton–Jacobi–Bellman equation (or often just the Hamilton–Jacobi equation), indicating its roots in classical physics (e.g., Schultz and Melsa, 1967).

The golf example was suggested by Chris Watkins.

Chapter 4

Dynamic Programming

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically. DP provides an essential foundation for the understanding of the methods presented in the rest of this book. In fact, all of these methods can be viewed as attempts to achieve much the same effect as DP, only with less computation and without assuming a perfect model of the environment.

Starting with this chapter, we usually assume that the environment is a finite MDP. That is, we assume that its state, action, and reward sets, \mathcal{S} , \mathcal{A} , and \mathcal{R} , are finite, and that its dynamics are given by a set of probabilities $p(s', r | s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, $r \in \mathcal{R}$, and $s' \in \mathcal{S}^+$ (\mathcal{S}^+ is \mathcal{S} plus a terminal state if the problem is episodic). Although DP ideas can be applied to problems with continuous state and action spaces, exact solutions are possible only in special cases. A common way of obtaining approximate solutions for tasks with continuous states and actions is to quantize the state and action spaces and then apply finite-state DP methods. The methods we explore in Part II are applicable to continuous problems and are a significant extension of that approach.

The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies. In this chapter we show how DP can be used to compute the value functions defined in Chapter 3. As discussed there, we can easily obtain optimal policies once we have found the optimal value functions, v_* or q_* , which satisfy the Bellman optimality equations:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_*(s')], \end{aligned} \tag{4.1}$$

or

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a\right] \\ &= \sum_{s', r} p(s', r | s, a) [r + \gamma \max_{a'} q_*(s', a')], \end{aligned} \tag{4.2}$$

for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, and $s' \in \mathcal{S}^+$. As we shall see, DP algorithms are obtained by turning Bellman equations such as these into assignments, that is, into update rules for improving approximations of the desired value functions.

4.1 Policy Evaluation (Prediction)

First we consider how to compute the state-value function v_π for an arbitrary policy π . This is called *policy evaluation* in the DP literature. We also refer to it as the *prediction problem*. Recall from Chapter 3 that, for all $s \in \mathcal{S}$,

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \end{aligned} \tag{from (3.9)}$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \tag{4.3}$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_\pi(s')], \tag{4.4}$$

where $\pi(a|s)$ is the probability of taking action a in state s under policy π , and the expectations are subscripted by π to indicate that they are conditional on π being followed. The existence and uniqueness of v_π are guaranteed as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under the policy π .

If the environment's dynamics are completely known, then (4.4) is a system of $|\mathcal{S}|$ simultaneous linear equations in $|\mathcal{S}|$ unknowns (the $v_\pi(s)$, $s \in \mathcal{S}$). In principle, its solution is a straightforward, if tedious, computation. For our purposes, iterative solution methods are most suitable. Consider a sequence of approximate value functions v_0, v_1, v_2, \dots , each mapping \mathcal{S}^+ to \mathbb{R} (the real numbers). The initial approximation, v_0 , is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation for v_π (4.4) as an update rule:

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')], \end{aligned} \tag{4.5}$$

for all $s \in \mathcal{S}$. Clearly, $v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation for v_π assures us of equality in this case. Indeed, the sequence $\{v_k\}$ can be shown in general to converge to v_π as $k \rightarrow \infty$ under the same conditions that guarantee the existence of v_π . This algorithm is called *iterative policy evaluation*.

To produce each successive approximation, v_{k+1} from v_k , iterative policy evaluation applies the same operation to each state s : it replaces the old value of s with a new value obtained from the old values of the successor states of s , and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated. We call this kind of operation an *expected update*. Each iteration of iterative policy evaluation updates the value of every state once to produce the new approximate value function

v_{k+1} . There are several different kinds of expected updates, depending on whether a state (as here) or a state-action pair is being updated, and depending on the precise way the estimated values of the successor states are combined. All the updates done in DP algorithms are called *expected* updates because they are based on an expectation over all possible next states rather than on a sample next state. The nature of an update can be expressed in an equation, as above, or in a backup diagram like those introduced in Chapter 3. For example, the backup diagram corresponding to the expected update used in iterative policy evaluation is shown on page 59.

To write a sequential computer program to implement iterative policy evaluation as given by (4.5) you would have to use two arrays, one for the old values, $v_k(s)$, and one for the new values, $v_{k+1}(s)$. With two arrays, the new values can be computed one by one from the old values without the old values being changed. Alternatively, you could use one array and update the values “in place,” that is, with each new value immediately overwriting the old one. Then, depending on the order in which the states are updated, sometimes new values are used instead of old ones on the right-hand side of (4.5). This in-place algorithm also converges to v_π ; in fact, it usually converges faster than the two-array version, as you might expect, because it uses new data as soon as they are available. We think of the updates as being done in a *sweep* through the state space. For the in-place algorithm, the order in which states have their values updated during the sweep has a significant influence on the rate of convergence. We usually have the in-place version in mind when we think of DP algorithms.

A complete in-place version of iterative policy evaluation is shown in pseudocode in the box below. Note how it handles termination. Formally, iterative policy evaluation converges only in the limit, but in practice it must be halted short of this. The pseudocode tests the quantity $\max_{s \in \mathcal{S}} |v_{k+1}(s) - v_k(s)|$ after each sweep and stops when it is sufficiently small.

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π , the policy to be evaluated

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
 Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop:

$$\Delta \leftarrow 0$$

Loop for each $s \in \mathcal{S}$:

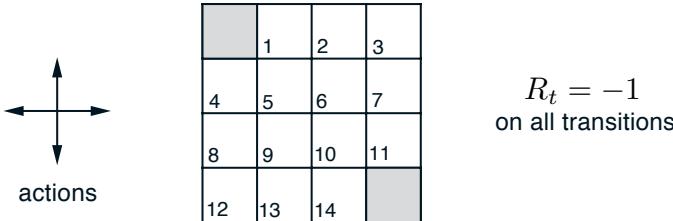
$$v \leftarrow V(s)$$

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a) [r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until $\Delta < \theta$

Example 4.1 Consider the 4×4 gridworld shown below.



The nonterminal states are $\mathcal{S} = \{1, 2, \dots, 14\}$. There are four actions possible in each state, $\mathcal{A} = \{\text{up}, \text{down}, \text{right}, \text{left}\}$, which deterministically cause the corresponding state transitions, except that actions that would take the agent off the grid in fact leave the state unchanged. Thus, for instance, $p(6, -1|5, \text{right}) = 1$, $p(7, -1|7, \text{right}) = 1$, and $p(10, r|5, \text{right}) = 0$ for all $r \in \mathcal{R}$. This is an undiscounted, episodic task. The reward is -1 on all transitions until the terminal state is reached. The terminal state is shaded in the figure (although it is shown in two places, it is formally one state). The expected reward function is thus $r(s, a, s') = -1$ for all states s, s' and actions a . Suppose the agent follows the equiprobable random policy (all actions equally likely). The left side of Figure 4.1 shows the sequence of value functions $\{v_k\}$ computed by iterative policy evaluation. The final estimate is in fact v_π , which in this case gives for each state the negation of the expected number of steps from that state until termination. ■

Exercise 4.1 In Example 4.1, if π is the equiprobable random policy, what is $q_\pi(11, \text{down})$? What is $q_\pi(7, \text{down})$? □

Exercise 4.2 In Example 4.1, suppose a new state 15 is added to the gridworld just below state 13, and its actions, `left`, `up`, `right`, and `down`, take the agent to states 12, 13, 14, and 15, respectively. Assume that the transitions *from* the original states are unchanged. What, then, is $v_\pi(15)$ for the equiprobable random policy? Now suppose the dynamics of state 13 are also changed, such that action `down` from state 13 takes the agent to the new state 15. What is $v_\pi(15)$ for the equiprobable random policy in this case? □

Exercise 4.3 What are the equations analogous to (4.3), (4.4), and (4.5), but for *action-value* functions instead of state-value functions? □

4.2 Policy Improvement

Our reason for computing the value function for a policy is to help find better policies. Suppose we have determined the value function v_π for an arbitrary deterministic policy π . For some state s we would like to know whether or not we should change the policy to deterministically choose an action $a \neq \pi(s)$. We know how good it is to follow the current policy from s —that is $v_\pi(s)$ —but would it be better or worse to change to the new policy? One way to answer this question is to consider selecting a in s and thereafter

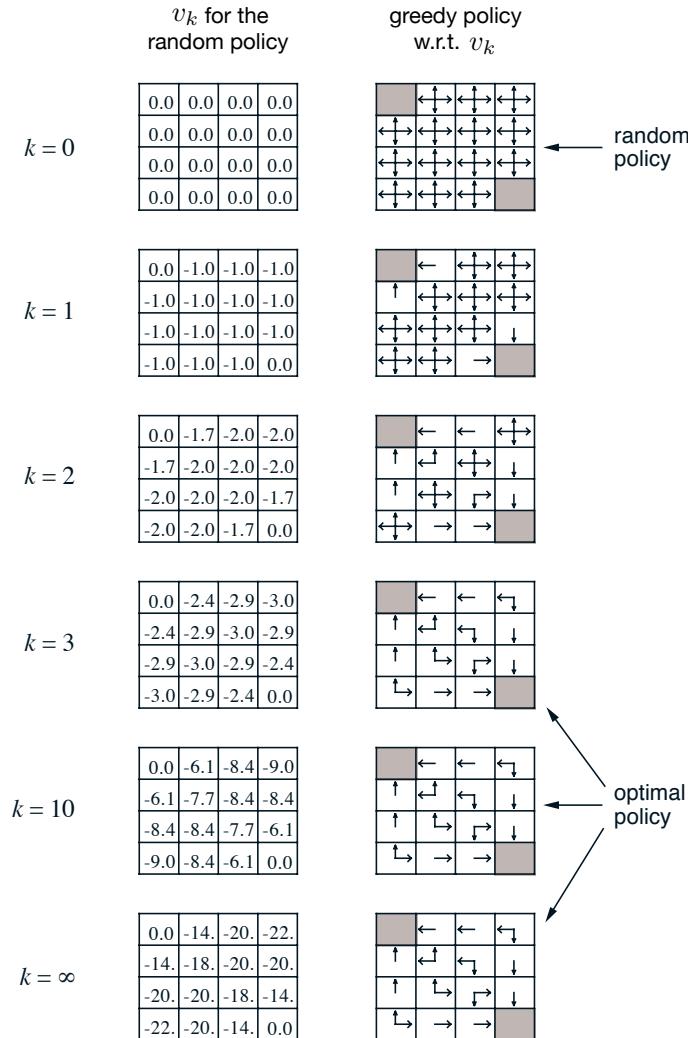


Figure 4.1: Convergence of iterative policy evaluation on a small gridworld. The left column is the sequence of approximations of the state-value function for the random policy (all actions equally likely). The right column is the sequence of greedy policies corresponding to the value function estimates (arrows are shown for all actions achieving the maximum, and the numbers shown are rounded to two significant digits). The last policy is guaranteed only to be an improvement over the random policy, but in this case it, and all policies after the third iteration, are optimal.

following the existing policy, π . The value of this way of behaving is

$$\begin{aligned} q_\pi(s, a) &\doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_\pi(s')]. \end{aligned} \tag{4.6}$$

The key criterion is whether this is greater than or less than $v_\pi(s)$. If it is greater—that is, if it is better to select a once in s and thereafter follow π than it would be to follow π all the time—then one would expect it to be better still to select a every time s is encountered, and that the new policy would in fact be a better one overall.

That this is true is a special case of a general result called the *policy improvement theorem*. Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$,

$$q_\pi(s, \pi'(s)) \geq v_\pi(s). \tag{4.7}$$

Then the policy π' must be as good as, or better than, π . That is, it must obtain greater or equal expected return from all states $s \in \mathcal{S}$:

$$v_{\pi'}(s) \geq v_\pi(s). \tag{4.8}$$

Moreover, if there is strict inequality of (4.7) at any state, then there must be strict inequality of (4.8) at that state.

The policy improvement theorem applies to the two policies that we considered at the beginning of this section: an original deterministic policy, π , and a changed policy, π' , that is identical to π except that $\pi'(s) = a \neq \pi(s)$. For states other than s , (4.7) holds because the two sides are equal. Thus, if $q_\pi(s, a) > v_\pi(s)$, then the changed policy is indeed better than π .

The idea behind the proof of the policy improvement theorem is easy to understand. Starting from (4.7), we keep expanding the q_π side with (4.6) and reapplying (4.7) until we get $v_{\pi'}(s)$:

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\ &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = \pi'(s)] \quad (\text{by (4.6)}) \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s] \quad (\text{by (4.7)}) \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}[R_{t+2} + \gamma v_\pi(S_{t+2}) \mid S_{t+1}, A_{t+1} = \pi'(S_{t+1})] \mid S_t = s] \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) \mid S_t = s] \\ &\vdots \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \mid S_t = s] \\ &= v_{\pi'}(s). \end{aligned}$$

So far we have seen how, given a policy and its value function, we can easily evaluate a change in the policy at a single state. It is a natural extension to consider changes at

all states, selecting at each state the action that appears best according to $q_\pi(s, a)$. In other words, to consider the new *greedy* policy, π' , given by

$$\begin{aligned}\pi'(s) &\doteq \arg\max_a q_\pi(s, a) \\ &= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \arg\max_a \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_\pi(s')],\end{aligned}\tag{4.9}$$

where $\arg\max_a$ denotes the value of a at which the expression that follows is maximized (with ties broken arbitrarily). The greedy policy takes the action that looks best in the short term—after one step of lookahead—according to v_π . By construction, the greedy policy meets the conditions of the policy improvement theorem (4.7), so we know that it is as good as, or better than, the original policy. The process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called *policy improvement*.

Suppose the new greedy policy, π' , is as good as, but not better than, the old policy π . Then $v_\pi = v_{\pi'}$, and from (4.9) it follows that for all $s \in \mathcal{S}$:

$$\begin{aligned}v_{\pi'}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_{\pi'}(s')].\end{aligned}$$

But this is the same as the Bellman optimality equation (4.1), and therefore, $v_{\pi'}$ must be v_* , and both π and π' must be optimal policies. Policy improvement thus must give us a strictly better policy except when the original policy is already optimal.

So far in this section we have considered the special case of deterministic policies. In the general case, a stochastic policy π specifies probabilities, $\pi(a|s)$, for taking each action, a , in each state, s . We will not go through the details, but in fact all the ideas of this section extend easily to stochastic policies. In particular, the policy improvement theorem carries through as stated for the stochastic case. In addition, if there are ties in policy improvement steps such as (4.9)—that is, if there are several actions at which the maximum is achieved—then in the stochastic case we need not select a single action from among them. Instead, each maximizing action can be given a portion of the probability of being selected in the new greedy policy. Any apportioning scheme is allowed as long as all submaximal actions are given zero probability.

The last row of Figure 4.1 shows an example of policy improvement for stochastic policies. Here the original policy, π , is the equiprobable random policy, and the new policy, π' , is greedy with respect to v_π . The value function v_π is shown in the bottom-left diagram and the set of possible π' is shown in the bottom-right diagram. The states with multiple arrows in the π' diagram are those in which several actions achieve the maximum in (4.9); any apportionment of probability among these actions is permitted. For any such policy, its state values $v_{\pi'}(s)$ can be seen by inspection to be either -1 , -2 , or -3 , for all states $s \in \mathcal{S}$, whereas $v_\pi(s)$ is at most -14 . Thus, $v_{\pi'}(s) \geq v_\pi(s)$, for all

$s \in \mathcal{S}$, illustrating policy improvement. Although in this case the new policy π' happens to be optimal, in general only an improvement is guaranteed.

4.3 Policy Iteration

Once a policy, π , has been improved using v_π to yield a better policy, π' , we can then compute $v_{\pi'}$ and improve it again to yield an even better π'' . We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

where $\xrightarrow{\text{E}}$ denotes a policy *evaluation* and $\xrightarrow{\text{I}}$ denotes a policy *improvement*. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and the optimal value function in a finite number of iterations.

This way of finding an optimal policy is called *policy iteration*. A complete algorithm is given in the box below. Note that each policy evaluation, itself an iterative computation, is started with the value function for the previous policy. This typically results in a great increase in the speed of convergence of policy evaluation (presumably because the value function changes little from one policy to the next).

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
 $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$
2. Policy Evaluation
 Loop:
 $\Delta \leftarrow 0$
 Loop for each $s \in \mathcal{S}$:
 $v \leftarrow V(s)$
 $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s)) [r + \gamma V(s')]$
 $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
 until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)
3. Policy Improvement
 $policy-stable \leftarrow true$
 For each $s \in \mathcal{S}$:
 $old-action \leftarrow \pi(s)$
 $\pi(s) \leftarrow \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$
 If $old-action \neq \pi(s)$, then $policy-stable \leftarrow false$
 If $policy-stable$, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Example 4.2: Jack’s Car Rental Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited \$10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of \$2 per car moved. We assume that the number of cars requested and returned at each location are Poisson random variables, meaning that the probability that the number is n is $\frac{\lambda^n}{n!} e^{-\lambda}$, where λ is the expected number. Suppose λ is 3 and 4 for rental requests at the first and second locations and 3 and 2 for returns. To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of five cars can be moved from one location to the other in one night. We take the discount rate to be $\gamma = 0.9$ and formulate this as a continuing finite MDP, where the time steps are days, the state is the number of cars at each location at the end of the day, and the actions are the net numbers of cars moved between the two locations overnight. Figure 4.2 shows the sequence of policies found by policy iteration starting from the policy that never moves any cars.

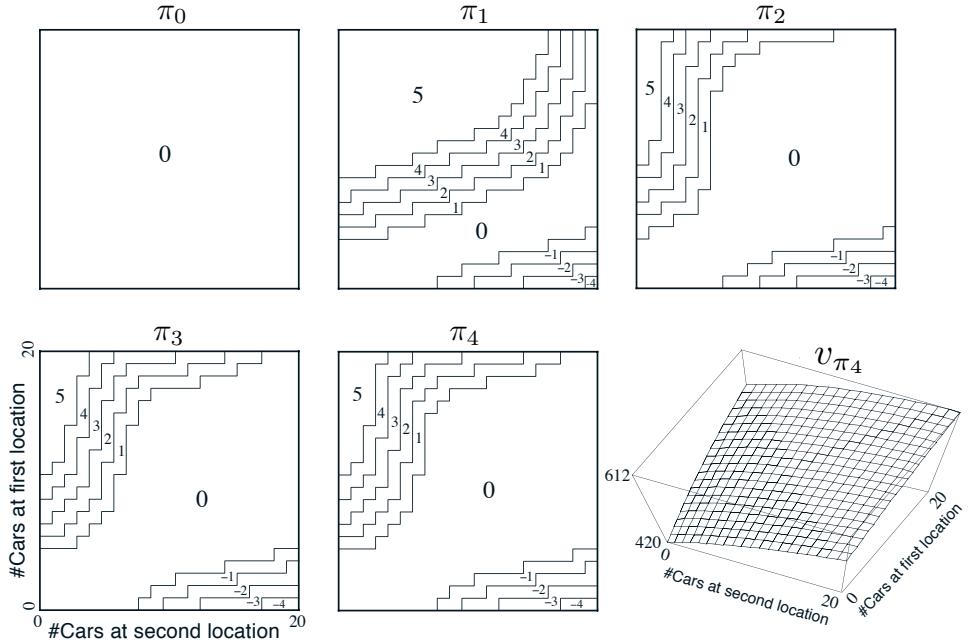


Figure 4.2: The sequence of policies found by policy iteration on Jack’s car rental problem, and the final state-value function. The first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second (negative numbers indicate transfers from the second location to the first). Each successive policy is a strict improvement over the previous policy, and the last policy is optimal. ■

Policy iteration often converges in surprisingly few iterations, as illustrated in the example of Jack’s car rental and in the example in Figure 4.1. The bottom-left diagram of Figure 4.1 shows the value function for the equiprobable random policy, and the bottom-right diagram shows a greedy policy for this value function. The policy improvement theorem assures us that these policies are better than the original random policy. In this case, however, these policies are not just better, but optimal, proceeding to the terminal states in the minimum number of steps. In this example, policy iteration would find the optimal policy after just one iteration.

Exercise 4.4 The policy iteration algorithm on page 80 has a subtle bug in that it may never terminate if the policy continually switches between two or more policies that are equally good. This is okay for pedagogy, but not for actual use. Modify the pseudocode so that convergence is guaranteed. \square

Exercise 4.5 How would policy iteration be defined for action values? Give a complete algorithm for computing q_* , analogous to that on page 80 for computing v_* . Please pay special attention to this exercise, because the ideas involved will be used throughout the rest of the book. \square

Exercise 4.6 Suppose you are restricted to considering only policies that are ε -soft, meaning that the probability of selecting each action in each state, s , is at least $\varepsilon/|\mathcal{A}(s)|$. Describe qualitatively the changes that would be required in each of the steps 3, 2, and 1, in that order, of the policy iteration algorithm for v_* on page 80. \square

Exercise 4.7 (programming) Write a program for policy iteration and re-solve Jack’s car rental problem with the following changes. One of Jack’s employees at the first location rides a bus home each night and lives near the second location. She is happy to shuttle one car to the second location for free. Each additional car still costs \$2, as do all cars moved in the other direction. In addition, Jack has limited parking space at each location. If more than 10 cars are kept overnight at a location (after any moving of cars), then an additional cost of \$4 must be incurred to use a second parking lot (independent of how many cars are kept there). These sorts of nonlinearities and arbitrary dynamics often occur in real problems and cannot easily be handled by optimization methods other than dynamic programming. To check your program, first replicate the results given for the original problem. \square

4.4 Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. If policy evaluation is done iteratively, then convergence exactly to v_π occurs only in the limit. Must we wait for exact convergence, or can we stop short of that? The example in Figure 4.1 certainly suggests that it may be possible to truncate policy evaluation. In that example, policy evaluation iterations beyond the first three have no effect on the corresponding greedy policy.

In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special

case is when policy evaluation is stopped after just one sweep (one update of each state). This algorithm is called *value iteration*. It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps:

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')], \end{aligned} \quad (4.10)$$

for all $s \in \mathcal{S}$. For arbitrary v_0 , the sequence $\{v_k\}$ can be shown to converge to v_* under the same conditions that guarantee the existence of v_* .

Another way of understanding value iteration is by reference to the Bellman optimality equation (4.1). Note that value iteration is obtained simply by turning the Bellman optimality equation into an update rule. Also note how the value iteration update is identical to the policy evaluation update (4.5) except that it requires the maximum to be taken over all actions. Another way of seeing this close relationship is to compare the backup diagrams for these algorithms on page 59 (policy evaluation) and on the left of Figure 3.4 (value iteration). These two are the natural backup operations for computing v_π and v_* .

Finally, let us consider how value iteration terminates. Like policy evaluation, value iteration formally requires an infinite number of iterations to converge exactly to v_* . In practice, we stop once the value function changes by only a small amount in a sweep. The box below shows a complete algorithm with this kind of termination condition.

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
 Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop:

```

  Δ ← 0
  Loop for each  $s \in \mathcal{S}$ :
     $v \leftarrow V(s)$ 
     $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$ 
     $\Delta \leftarrow \max(\Delta, |v - V(s)|)$ 
  until  $\Delta < \theta$ 

```

Output a deterministic policy, $\pi \approx \pi_*$, such that

$$\pi(s) = \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep. In general, the entire class of truncated policy iteration algorithms can be thought of as sequences of sweeps, some of which use policy evaluation updates and some of which use value iteration updates. Because the max operation in (4.10) is the only difference between

these updates, this just means that the max operation is added to some sweeps of policy evaluation. All of these algorithms converge to an optimal policy for discounted finite MDPs.

Example 4.3: Gambler's Problem A gambler has the opportunity to make bets on the outcomes of a sequence of coin flips. If the coin comes up heads, he wins as many dollars as he has staked on that flip; if it is tails, he loses his stake. The game ends when the gambler wins by reaching his goal of \$100, or loses by running out of money. On each flip, the gambler must decide what portion of his capital to stake, in integer numbers of dollars. This problem can be formulated as an undiscounted, episodic, finite MDP. The state is the gambler's capital, $s \in \{1, 2, \dots, 99\}$ and the actions are stakes, $a \in \{0, 1, \dots, \min(s, 100 - s)\}$. The reward is zero on all transitions except those on which the gambler reaches his goal, when it is +1. The state-value function then gives the probability of winning from each state. A policy is a mapping from levels of capital to stakes. The optimal policy maximizes the probability of reaching the goal. Let p_h denote the probability of the coin coming up heads. If p_h is known, then the entire problem is known and it can be solved, for instance, by value iteration. Figure 4.3 shows the change in the value function over successive sweeps of value iteration, and the final policy found, for the case of $p_h = 0.4$. This policy is optimal, but not unique. In fact, there is a whole family of optimal policies, all corresponding to ties for the argmax action selection with respect to the optimal value function. Can you guess what the entire family looks like?

Exercise 4.8 Why does the optimal policy for the gambler's problem have such a curious form? In particular, for capital of 50 it bets it all on one flip, but for capital of 51 it does not. Why is this a good policy? \square

Exercise 4.9 (programming) Implement value iteration for the gambler's problem and solve it for $p_h = 0.25$ and $p_h = 0.55$. In programming, you may find it convenient to introduce two dummy states corresponding to termination with capital of 0 and 100, giving them values of 0 and 1 respectively. Show your results graphically, as in Figure 4.3. Are your results stable as $\theta \rightarrow 0$? \square

Exercise 4.10 What is the analog of the value iteration update (4.10) for action values, $q_{k+1}(s, a)$? \square

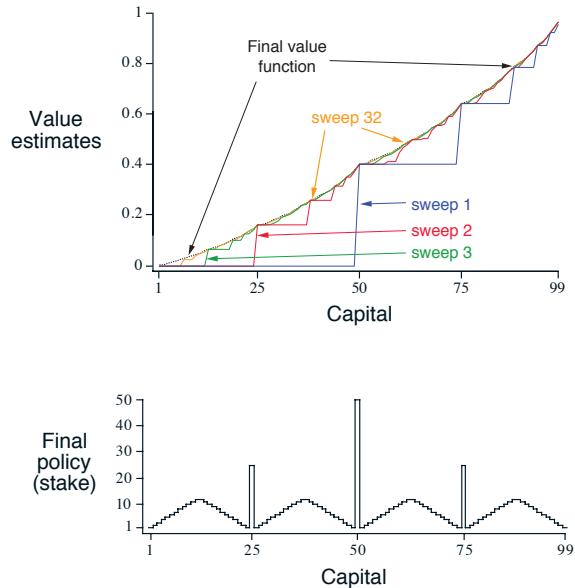


Figure 4.3: The solution to the gambler's problem for $p_h = 0.4$. The upper graph shows the value function found by successive sweeps of value iteration. The lower graph shows the final policy. \blacksquare

4.5 Asynchronous Dynamic Programming

A major drawback to the DP methods that we have discussed so far is that they involve operations over the entire state set of the MDP, that is, they require sweeps of the state set. If the state set is very large, then even a single sweep can be prohibitively expensive. For example, the game of backgammon has over 10^{20} states. Even if we could perform the value iteration update on a million states per second, it would take over a thousand years to complete a single sweep.

Asynchronous DP algorithms are in-place iterative DP algorithms that are not organized in terms of systematic sweeps of the state set. These algorithms update the values of states in any order whatsoever, using whatever values of other states happen to be available. The values of some states may be updated several times before the values of others are updated once. To converge correctly, however, an asynchronous algorithm must continue to update the values of all the states: it can't ignore any state after some point in the computation. Asynchronous DP algorithms allow great flexibility in selecting states to update.

For example, one version of asynchronous value iteration updates the value, in place, of only one state, s_k , on each step, k , using the value iteration update (4.10). If $0 \leq \gamma < 1$, asymptotic convergence to v_* is guaranteed given only that all states occur in the sequence $\{s_k\}$ an infinite number of times (the sequence could even be random).¹ Similarly, it is possible to intermix policy evaluation and value iteration updates to produce a kind of asynchronous truncated policy iteration. Although the details of this and other more unusual DP algorithms are beyond the scope of this book, it is clear that a few different updates form building blocks that can be used flexibly in a wide variety of sweepless DP algorithms.

Of course, avoiding sweeps does not necessarily mean that we can get away with less computation. It just means that an algorithm does not need to get locked into any hopelessly long sweep before it can make progress improving a policy. We can try to take advantage of this flexibility by selecting the states to which we apply updates so as to improve the algorithm's rate of progress. We can try to order the updates to let value information propagate from state to state in an efficient way. Some states may not need their values updated as often as others. We might even try to skip updating some states entirely if they are not relevant to optimal behavior. Some ideas for doing this are discussed in Chapter 8.

Asynchronous algorithms also make it easier to intermix computation with real-time interaction. To solve a given MDP, we can run an iterative DP algorithm *at the same time that an agent is actually experiencing the MDP*. The agent's experience can be used to determine the states to which the DP algorithm applies its updates. At the same time, the latest value and policy information from the DP algorithm can guide the agent's decision making. For example, we can apply updates to states as the agent visits them. This makes it possible to *focus* the DP algorithm's updates onto parts of the state set

¹In the undiscounted episodic case, it is possible that there are some orderings of updates that do not result in convergence, but it is relatively easy to avoid these.

that are most relevant to the agent. This kind of focusing is a repeated theme in reinforcement learning.

4.6 Generalized Policy Iteration

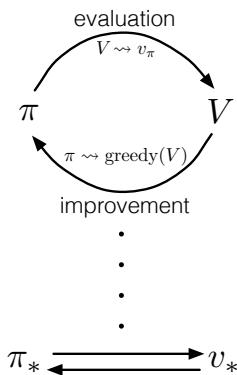
Policy iteration consists of two simultaneous, interacting processes, one making the value function consistent with the current policy (policy evaluation), and the other making the policy greedy with respect to the current value function (policy improvement). In policy iteration, these two processes alternate, each completing before the other begins, but this is not really necessary. In value iteration, for example, only a single iteration of policy evaluation is performed in between each policy improvement. In asynchronous DP methods, the evaluation and improvement processes are interleaved at an even finer grain. In some cases a single state is updated in one process before returning to the other. As long as both processes continue to update all states, the ultimate result is typically the same—convergence to the optimal value function and an optimal policy.

We use the term *generalized policy iteration* (GPI) to refer to the general idea of letting policy-evaluation and policy-improvement processes interact, independent of the granularity and other details of the two processes. Almost all reinforcement learning methods are well described as GPI. That is, all have identifiable policies and value functions, with the policy always being improved with respect to the value function and the value function always being driven toward the value function for the policy, as suggested by the diagram to the right. If both the evaluation process and the improvement process stabilize, that is, no longer produce changes, then the value function and policy must be optimal. The value function stabilizes only when it is consistent with the current policy, and the policy stabilizes only when it is greedy with respect to the current value function.

Thus, both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function. This implies that the Bellman optimality equation (4.1) holds, and thus that the policy and the value function are optimal.

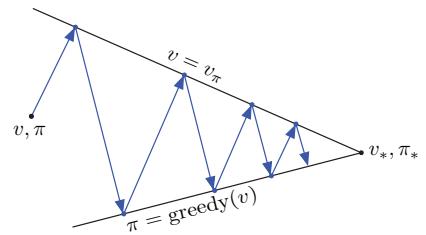
The evaluation and improvement processes in GPI can be viewed as both competing and cooperating. They compete in the sense that they pull in opposing directions. Making the policy greedy with respect to the value function typically makes the value function incorrect for the changed policy, and making the value function consistent with the policy typically causes that policy no longer to be greedy. In the long run, however, these two processes interact to find a single joint solution: the optimal value function and an optimal policy.

One might also think of the interaction between the evaluation and improvement processes in GPI in terms of two constraints or goals—for example, as two lines in



a two-dimensional space as suggested by the diagram to the right. Although the real geometry is much more complicated than this, the diagram suggests what happens in the real case. Each process drives the value function or policy toward one of the lines representing a solution to one of the two goals. The goals interact because the two lines are not orthogonal. Driving directly toward one goal causes some movement away from the other goal.

Inevitably, however, the joint process is brought closer to the overall goal of optimality. The arrows in this diagram correspond to the behavior of policy iteration in that each takes the system all the way to achieving one of the two goals completely. In GPI one could also take smaller, incomplete steps toward each goal. In either case, the two processes together achieve the overall goal of optimality even though neither is attempting to achieve it directly.



4.7 Efficiency of Dynamic Programming

DP may not be practical for very large problems, but compared with other methods for solving MDPs, DP methods are actually quite efficient. If we ignore a few technical details, then, in the worst case, the time that DP methods take to find an optimal policy is polynomial in the number of states and actions. If n and k denote the number of states and actions, this means that a DP method takes a number of computational operations that is less than some polynomial function of n and k . A DP method is guaranteed to find an optimal policy in polynomial time even though the total number of (deterministic) policies is k^n . In this sense, DP is exponentially faster than any direct search in policy space could be, because direct search would have to exhaustively examine each policy to provide the same guarantee. Linear programming methods can also be used to solve MDPs, and in some cases their worst-case convergence guarantees are better than those of DP methods. But linear programming methods become impractical at a much smaller number of states than do DP methods (by a factor of about 100). For the largest problems, only DP methods are feasible.

DP is sometimes thought to be of limited applicability because of the *curse of dimensionality*, the fact that the number of states often grows exponentially with the number of state variables. Large state sets do create difficulties, but these are inherent difficulties of the problem, not of DP as a solution method. In fact, DP is comparatively better suited to handling large state spaces than competing methods such as direct search and linear programming.

In practice, DP methods can be used with today's computers to solve MDPs with millions of states. Both policy iteration and value iteration are widely used, and it is not clear which, if either, is better in general. In practice, these methods usually converge much faster than their theoretical worst-case run times, particularly if they are started with good initial value functions or policies.

On problems with large state spaces, *asynchronous* DP methods are often preferred. To complete even one sweep of a synchronous method requires computation and memory for every state. For some problems, even this much memory and computation is impractical, yet the problem is still potentially solvable because relatively few states occur along optimal solution trajectories. Asynchronous methods and other variations of GPI can be applied in such cases and may find good or optimal policies much faster than synchronous methods can.

4.8 Summary

In this chapter we have become familiar with the basic ideas and algorithms of dynamic programming as they relate to solving finite MDPs. *Policy evaluation* refers to the (typically) iterative computation of the value function for a given policy. *Policy improvement* refers to the computation of an improved policy given the value function for that policy. Putting these two computations together, we obtain *policy iteration* and *value iteration*, the two most popular DP methods. Either of these can be used to reliably compute optimal policies and value functions for finite MDPs given complete knowledge of the MDP.

Classical DP methods operate in sweeps through the state set, performing an *expected update* operation on each state. Each such operation updates the value of one state based on the values of all possible successor states and their probabilities of occurring. Expected updates are closely related to Bellman equations: they are little more than these equations turned into assignment statements. When the updates no longer result in any changes in value, convergence has occurred to values that satisfy the corresponding Bellman equation. Just as there are four primary value functions (v_π , v_* , q_π , and q_*), there are four corresponding Bellman equations and four corresponding expected updates. An intuitive view of the operation of DP updates is given by their *backup diagrams*.

Insight into DP methods and, in fact, into almost all reinforcement learning methods, can be gained by viewing them as *generalized policy iteration* (GPI). GPI is the general idea of two interacting processes revolving around an approximate policy and an approximate value function. One process takes the policy as given and performs some form of policy evaluation, changing the value function to be more like the true value function for the policy. The other process takes the value function as given and performs some form of policy improvement, changing the policy to make it better, assuming that the value function is its value function. Although each process changes the basis for the other, overall they work together to find a joint solution: a policy and value function that are unchanged by either process and, consequently, are optimal. In some cases, GPI can be proved to converge, most notably for the classical DP methods that we have presented in this chapter. In other cases convergence has not been proved, but still the idea of GPI improves our understanding of the methods.

It is not necessary to perform DP methods in complete sweeps through the state set. *Asynchronous DP* methods are in-place iterative methods that update states in an arbitrary order, perhaps stochastically determined and using out-of-date information. Many of these methods can be viewed as fine-grained forms of GPI.

Finally, we note one last special property of DP methods. All of them update estimates of the values of states based on estimates of the values of successor states. That is, they update estimates on the basis of other estimates. We call this general idea *bootstrapping*. Many reinforcement learning methods perform bootstrapping, even those that do not require, as DP requires, a complete and accurate model of the environment. In the next chapter we explore reinforcement learning methods that do not require a model and do not bootstrap. In the chapter after that we explore methods that do not require a model but do bootstrap. These key features and properties are separable, yet can be mixed in interesting combinations.

Bibliographical and Historical Remarks

The term “dynamic programming” is due to Bellman (1957a), who showed how these methods could be applied to a wide range of problems. Extensive treatments of DP can be found in many texts, including Bertsekas (2005, 2012), Bertsekas and Tsitsiklis (1996), Dreyfus and Law (1977), Ross (1983), White (1969), and Whittle (1982, 1983). Our interest in DP is restricted to its use in solving MDPs, but DP also applies to other types of problems. Kumar and Kanal (1988) provide a more general look at DP.

To the best of our knowledge, the first connection between DP and reinforcement learning was made by Minsky (1961) in commenting on Samuel’s checkers player. In a footnote, Minsky mentioned that it is possible to apply DP to problems in which Samuel’s backing-up process can be handled in closed analytic form. This remark may have misled artificial intelligence researchers into believing that DP was restricted to analytically tractable problems and therefore largely irrelevant to artificial intelligence. Andreae (1969) mentioned DP in the context of reinforcement learning. Werbos (1977) suggested an approach to approximating DP called “heuristic dynamic programming” that emphasizes gradient-descent methods for continuous-state problems (Werbos, 1982, 1987, 1988, 1989, 1992). These methods are closely related to the reinforcement learning algorithms that we discuss in this book. Watkins (1989) was explicit in connecting reinforcement learning to DP, characterizing a class of reinforcement learning methods as “incremental dynamic programming.”

4.1–4 These sections describe well-established DP algorithms that are covered in any of the general DP references cited above. The policy improvement theorem and the policy iteration algorithm are due to Bellman (1957a) and Howard (1960). Our presentation was influenced by the local view of policy improvement taken by Watkins (1989). Our discussion of value iteration as a form of truncated policy iteration is based on the approach of Puterman and Shin (1978), who presented a class of algorithms called *modified policy iteration*, which includes policy iteration and value iteration as special cases. An analysis showing how value iteration can be made to find an optimal policy in finite time is given by Bertsekas (1987).

Iterative policy evaluation is an example of a classical successive approximation algorithm for solving a system of linear equations. The version of the algorithm

that uses two arrays, one holding the old values while the other is updated, is often called a *Jacobi-style* algorithm, after Jacobi's classical use of this method. It is also sometimes called a *synchronous* algorithm because the effect is as if all the values are updated at the same time. The second array is needed to simulate this parallel computation sequentially. The in-place version of the algorithm is often called a *Gauss–Seidel-style* algorithm after the classical Gauss–Seidel algorithm for solving systems of linear equations. In addition to iterative policy evaluation, other DP algorithms can be implemented in these different versions. Bertsekas and Tsitsiklis (1989) provide excellent coverage of these variations and their performance differences.

- 4.5** Asynchronous DP algorithms are due to Bertsekas (1982, 1983), who also called them distributed DP algorithms. The original motivation for asynchronous DP was its implementation on a multiprocessor system with communication delays between processors and no global synchronizing clock. These algorithms are extensively discussed by Bertsekas and Tsitsiklis (1989). Jacobi-style and Gauss–Seidel-style DP algorithms are special cases of the asynchronous version. Williams and Baird (1990) presented DP algorithms that are asynchronous at a finer grain than the ones we have discussed: the update operations themselves are broken into steps that can be performed asynchronously.
- 4.7** This section, written with the help of Michael Littman, is based on Littman, Dean, and Kaelbling (1995). The phrase “curse of dimensionality” is due to Bellman (1957a).
Foundational work on the linear programming approach to reinforcement learning was done by Daniela de Farias (de Farias, 2002; de Farias and Van Roy, 2003).

Chapter 5

Monte Carlo Methods

In this chapter we consider our first learning methods for estimating value functions and discovering optimal policies. Unlike the previous chapter, here we do not assume complete knowledge of the environment. Monte Carlo methods require only *experience*—sample sequences of states, actions, and rewards from actual or simulated interaction with an environment. Learning from *actual* experience is striking because it requires no prior knowledge of the environment’s dynamics, yet can still attain optimal behavior. Learning from *simulated* experience is also powerful. Although a model is required, the model need only generate sample transitions, not the complete probability distributions of all possible transitions that is required for dynamic programming (DP). In surprisingly many cases it is easy to generate experience sampled according to the desired probability distributions, but infeasible to obtain the distributions in explicit form.

Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns. To ensure that well-defined returns are available, here we define Monte Carlo methods only for episodic tasks. That is, we assume experience is divided into episodes, and that all episodes eventually terminate no matter what actions are selected. Only on the completion of an episode are value estimates and policies changed. Monte Carlo methods can thus be incremental in an episode-by-episode sense, but not in a step-by-step (online) sense. The term “Monte Carlo” is often used more broadly for any estimation method whose operation involves a significant random component. Here we use it specifically for methods based on averaging complete returns (as opposed to methods that learn from partial returns, considered in the next chapter).

Monte Carlo methods sample and average *returns* for each state–action pair much like the bandit methods we explored in Chapter 2 sample and average *rewards* for each action. The main difference is that now there are multiple states, each acting like a different bandit problem (like an associative-search or contextual bandit) and the different bandit problems are interrelated. That is, the return after taking an action in one state depends on the actions taken in later states in the same episode. Because all the action selections are undergoing learning, the problem becomes nonstationary from the point of view of the earlier state.

To handle the nonstationarity, we adapt the idea of general policy iteration (GPI) developed in Chapter 4 for DP. Whereas there we *computed* value functions from knowledge of the MDP, here we *learn* value functions from sample returns with the MDP. The value functions and corresponding policies still interact to attain optimality in essentially the same way (GPI). As in the DP chapter, first we consider the prediction problem (the computation of v_π and q_π for a fixed arbitrary policy π) then policy improvement, and, finally, the control problem and its solution by GPI. Each of these ideas taken from DP is extended to the Monte Carlo case in which only sample experience is available.

5.1 Monte Carlo Prediction

We begin by considering Monte Carlo methods for learning the state-value function for a given policy. Recall that the value of a state is the expected return—expected cumulative future discounted reward—starting from that state. An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value. This idea underlies all Monte Carlo methods.

In particular, suppose we wish to estimate $v_\pi(s)$, the value of a state s under policy π , given a set of episodes obtained by following π and passing through s . Each occurrence of state s in an episode is called a *visit* to s . Of course, s may be visited multiple times in the same episode; let us call the first time it is visited in an episode the *first visit* to s . The *first-visit MC method* estimates $v_\pi(s)$ as the average of the returns following first visits to s , whereas the *every-visit MC method* averages the returns following all visits to s . These two Monte Carlo (MC) methods are very similar but have slightly different theoretical properties. First-visit MC has been most widely studied, dating back to the 1940s, and is the one we focus on in this chapter. Every-visit MC extends more naturally to function approximation and eligibility traces, as discussed in Chapters 9 and 12. First-visit MC is shown in procedural form in the box. Every-visit MC would be the same except without the check for S_t having occurred earlier in the episode.

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π : $S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless S_t appears in S_0, S_1, \dots, S_{t-1} :

Append G to $Returns(S_t)$

$V(S_t) \leftarrow \text{average}(Returns(S_t))$

Both first-visit MC and every-visit MC converge to $v_\pi(s)$ as the number of visits (or first visits) to s goes to infinity. This is easy to see for the case of first-visit MC. In this case each return is an independent, identically distributed estimate of $v_\pi(s)$ with finite variance. By the law of large numbers the sequence of averages of these estimates converges to their expected value. Each average is itself an unbiased estimate, and the standard deviation of its error falls as $1/\sqrt{n}$, where n is the number of returns averaged. Every-visit MC is less straightforward, but its estimates also converge quadratically to $v_\pi(s)$ (Singh and Sutton, 1996).

The use of Monte Carlo methods is best illustrated through an example.

Example 5.1: Blackjack The object of the popular casino card game of *blackjack* is to obtain cards the sum of whose numerical values is as great as possible without exceeding 21. All face cards count as 10, and an ace can count either as 1 or as 11. We consider the version in which each player competes independently against the dealer. The game begins with two cards dealt to both dealer and player. One of the dealer's cards is face up and the other is face down. If the player has 21 immediately (an ace and a 10-card), it is called a *natural*. He then wins unless the dealer also has a natural, in which case the game is a draw. If the player does not have a natural, then he can request additional cards, one by one (*hits*), until he either stops (*sticks*) or exceeds 21 (*goes bust*). If he goes bust, he loses; if he sticks, then it becomes the dealer's turn. The dealer hits or sticks according to a fixed strategy without choice: he sticks on any sum of 17 or greater, and hits otherwise. If the dealer goes bust, then the player wins; otherwise, the outcome—win, lose, or draw—is determined by whose final sum is closer to 21.

Playing blackjack is naturally formulated as an episodic finite MDP. Each game of blackjack is an episode. Rewards of +1, -1, and 0 are given for winning, losing, and drawing, respectively. All rewards within a game are zero, and we do not discount ($\gamma = 1$); therefore these terminal rewards are also the returns. The player's actions are to hit or to stick. The states depend on the player's cards and the dealer's showing card. We assume that cards are dealt from an infinite deck (i.e., with replacement) so that there is no advantage to keeping track of the cards already dealt. If the player holds an ace that he could count as 11 without going bust, then the ace is said to be *usable*. In this case it is always counted as 11 because counting it as 1 would make the sum 11 or less, in which case there is no decision to be made because, obviously, the player should always hit. Thus, the player makes decisions on the basis of three variables: his current sum (12–21), the dealer's one showing card (ace–10), and whether or not he holds a usable ace. This makes for a total of 200 states.

Consider the policy that sticks if the player's sum is 20 or 21, and otherwise hits. To find the state-value function for this policy by a Monte Carlo approach, one simulates many blackjack games using the policy and averages the returns following each state. In this way, we obtained the estimates of the state-value function shown in Figure 5.1. The estimates for states with a usable ace are less certain and less regular because these states are less common. In any event, after 500,000 games the value function is very well approximated.

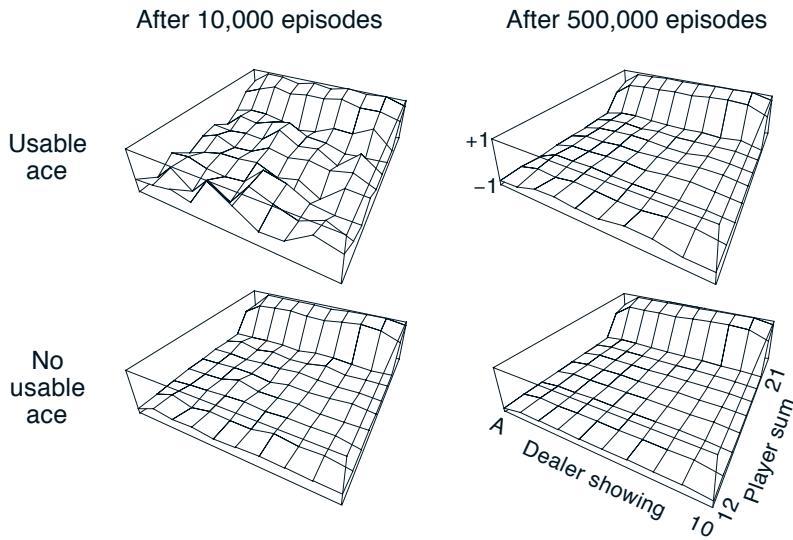


Figure 5.1: Approximate state-value functions for the blackjack policy that sticks only on 20 or 21, computed by Monte Carlo policy evaluation. ■

Exercise 5.1 Consider the diagrams on the right in Figure 5.1. Why does the estimated value function jump up for the last two rows in the rear? Why does it drop off for the whole last row on the left? Why are the frontmost values higher in the upper diagrams than in the lower? □

Exercise 5.2 Suppose every-visit MC was used instead of first-visit MC on the blackjack task. Would you expect the results to be very different? Why or why not? □

Although we have complete knowledge of the environment in the blackjack task, it would not be easy to apply DP methods to compute the value function. DP methods require the distribution of next events—in particular, they require the environments dynamics as given by the four-argument function p —and it is not easy to determine this for blackjack. For example, suppose the player’s sum is 14 and he chooses to stick. What is his probability of terminating with a reward of +1 as a function of the dealer’s showing card? All of the probabilities must be computed *before* DP can be applied, and such computations are often complex and error-prone. In contrast, generating the sample games required by Monte Carlo methods is easy. This is the case surprisingly often; the ability of Monte Carlo methods to work with sample episodes alone can be a significant advantage even when one has complete knowledge of the environment’s dynamics.

Can we generalize the idea of backup diagrams to Monte Carlo algorithms? The general idea of a backup diagram is to show at the top the root node to be updated and to show below all the transitions and leaf nodes whose rewards and estimated values contribute to the update. For Monte Carlo estimation of v_π , the root is a state node, and below it is the entire trajectory of transitions along a particular single episode, ending

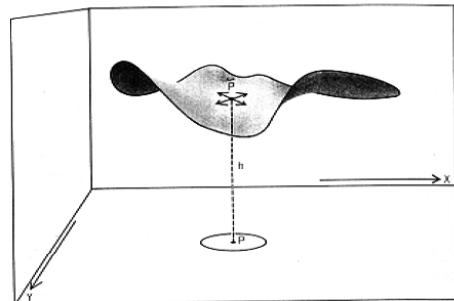
at the terminal state, as shown to the right. Whereas the DP diagram (page 59) shows all possible transitions, the Monte Carlo diagram shows only those sampled on the one episode. Whereas the DP diagram includes only one-step transitions, the Monte Carlo diagram goes all the way to the end of the episode. These differences in the diagrams accurately reflect the fundamental differences between the algorithms.

An important fact about Monte Carlo methods is that the estimates for each state are independent. The estimate for one state does not build upon the estimate of any other state, as is the case in DP. In other words, Monte Carlo methods do not *bootstrap* as we defined it in the previous chapter.

In particular, note that the computational expense of estimating the value of a single state is independent of the number of states. This can make Monte Carlo methods particularly attractive when one requires the value of only one or a subset of states. One can generate many sample episodes starting from the states of interest, averaging returns from only these states, ignoring all others. This is a third advantage Monte Carlo methods can have over DP methods (after the ability to learn from actual experience and from simulated experience).

Example 5.2: Soap Bubble Suppose a wire frame forming a closed loop is dunked in soapy water to form a soap surface or bubble conforming at its edges to the wire frame. If the geometry of the wire frame is irregular but known, how can you compute the shape of the surface? The shape has the property that the total force on each point exerted by neighboring points is zero (or else the shape would change). This means that the surface's height at any point is the average of its heights at points in a small circle around that point. In addition, the surface must meet at its boundaries with the wire frame. The usual approach to problems of this kind is to put a grid over the area covered by the surface and solve for its height at the grid points by an iterative computation. Grid points at the boundary are forced to the wire frame, and all others are adjusted toward the average of the heights of their four nearest neighbors. This process then iterates, much like DP's iterative policy evaluation, and ultimately converges to a close approximation to the desired surface.

This is similar to the kind of problem for which Monte Carlo methods were originally designed. Instead of the iterative computation described above, imagine standing on the surface and taking a random walk, stepping randomly from grid point to neighboring grid point, with equal probability, until you reach the boundary. It turns out that the expected value of the height at the boundary is a close approximation to the height of the desired surface at the starting point (in fact, it is exactly the value computed by the iterative method described above). Thus, one can closely approximate the height of the



A bubble on a wire loop.

From Hersh and Griego (1969). Reproduced with permission. ©1969 Scientific American, a division of Nature America, Inc. All rights reserved.

surface at a point by simply averaging the boundary heights of many walks started at the point. If one is interested in only the value at one point, or any fixed small set of points, then this Monte Carlo method can be far more efficient than the iterative method based on local consistency. ■

5.2 Monte Carlo Estimation of Action Values

If a model is not available, then it is particularly useful to estimate *action* values (the values of state–action pairs) rather than *state* values. With a model, state values alone are sufficient to determine a policy; one simply looks ahead one step and chooses whichever action leads to the best combination of reward and next state, as we did in the chapter on DP. Without a model, however, state values alone are not sufficient. One must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy. Thus, one of our primary goals for Monte Carlo methods is to estimate q_* . To achieve this, we first consider the policy evaluation problem for action values.

The policy evaluation problem for action values is to estimate $q_\pi(s, a)$, the expected return when starting in state s , taking action a , and thereafter following policy π . The Monte Carlo methods for this are essentially the same as just presented for state values, except now we talk about visits to a state–action pair rather than to a state. A state–action pair s, a is said to be visited in an episode if ever the state s is visited and action a is taken in it. The every-visit MC method estimates the value of a state–action pair as the average of the returns that have followed all the visits to it. The first-visit MC method averages the returns following the first time in each episode that the state was visited and the action was selected. These methods converge quadratically, as before, to the true expected values as the number of visits to each state–action pair approaches infinity.

The only complication is that many state–action pairs may never be visited. If π is a deterministic policy, then in following π one will observe returns only for one of the actions from each state. With no returns to average, the Monte Carlo estimates of the other actions will not improve with experience. This is a serious problem because the purpose of learning action values is to help in choosing among the actions available in each state. To compare alternatives we need to estimate the value of *all* the actions from each state, not just the one we currently favor.

This is the general problem of *maintaining exploration*, as discussed in the context of the k -armed bandit problem in Chapter 2. For policy evaluation to work for action values, we must assure continual exploration. One way to do this is by specifying that the episodes *start in a state–action pair*, and that every pair has a nonzero probability of being selected as the start. This guarantees that all state–action pairs will be visited an infinite number of times in the limit of an infinite number of episodes. We call this the assumption of *exploring starts*.

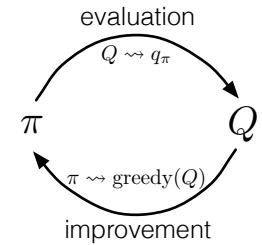
The assumption of exploring starts is sometimes useful, but of course it cannot be relied upon in general, particularly when learning directly from actual interaction with an environment. In that case the starting conditions are unlikely to be so helpful. The most common alternative approach to assuring that all state–action pairs are encountered is

to consider only policies that are stochastic with a nonzero probability of selecting all actions in each state. We discuss two important variants of this approach in later sections. For now, we retain the assumption of exploring starts and complete the presentation of a full Monte Carlo control method.

Exercise 5.3 What is the backup diagram for Monte Carlo estimation of q_π ? \square

5.3 Monte Carlo Control

We are now ready to consider how Monte Carlo estimation can be used in control, that is, to approximate optimal policies. The overall idea is to proceed according to the same pattern as in the DP chapter, that is, according to the idea of generalized policy iteration (GPI). In GPI one maintains both an approximate policy and an approximate value function. The value function is repeatedly altered to more closely approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function, as suggested by the diagram to the right. These two kinds of changes work against each other to some extent, as each creates a moving target for the other, but together they cause both policy and value function to approach optimality.



To begin, let us consider a Monte Carlo version of classical policy iteration. In this method, we perform alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal policy and optimal action-value function:

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} q_*,$$

where \xrightarrow{E} denotes a complete policy evaluation and \xrightarrow{I} denotes a complete policy improvement. Policy evaluation is done exactly as described in the preceding section. Many episodes are experienced, with the approximate action-value function approaching the true function asymptotically. For the moment, let us assume that we do indeed observe an infinite number of episodes and that, in addition, the episodes are generated with exploring starts. Under these assumptions, the Monte Carlo methods will compute each q_{π_k} exactly, for arbitrary π_k .

Policy improvement is done by making the policy greedy with respect to the current value function. In this case we have an *action*-value function, and therefore no model is needed to construct the greedy policy. For any action-value function q , the corresponding greedy policy is the one that, for each $s \in \mathcal{S}$, deterministically chooses an action with maximal action-value:

$$\pi(s) \doteq \arg \max_a q(s, a). \quad (5.1)$$

Policy improvement then can be done by constructing each π_{k+1} as the greedy policy with respect to q_{π_k} . The policy improvement theorem (Section 4.2) then applies to π_k

and π_{k+1} because, for all $s \in \mathcal{S}$,

$$\begin{aligned} q_{\pi_k}(s, \pi_{k+1}(s)) &= q_{\pi_k}(s, \arg \max_a q_{\pi_k}(s, a)) \\ &= \max_a q_{\pi_k}(s, a) \\ &\geq q_{\pi_k}(s, \pi_k(s)) \\ &\geq v_{\pi_k}(s). \end{aligned}$$

As we discussed in the previous chapter, the theorem assures us that each π_{k+1} is uniformly better than π_k , or just as good as π_k , in which case they are both optimal policies. This in turn assures us that the overall process converges to the optimal policy and optimal value function. In this way Monte Carlo methods can be used to find optimal policies given only sample episodes and no other knowledge of the environment's dynamics.

We made two unlikely assumptions above in order to easily obtain this guarantee of convergence for the Monte Carlo method. One was that the episodes have exploring starts, and the other was that policy evaluation could be done with an infinite number of episodes. To obtain a practical algorithm we will have to remove both assumptions. We postpone consideration of the first assumption until later in this chapter.

For now we focus on the assumption that policy evaluation operates on an infinite number of episodes. This assumption is relatively easy to remove. In fact, the same issue arises even in classical DP methods such as iterative policy evaluation, which also converge only asymptotically to the true value function. In both DP and Monte Carlo cases there are two ways to solve the problem. One is to hold firm to the idea of approximating q_{π_k} in each policy evaluation. Measurements and assumptions are made to obtain bounds on the magnitude and probability of error in the estimates, and then sufficient steps are taken during each policy evaluation to assure that these bounds are sufficiently small. This approach can probably be made completely satisfactory in the sense of guaranteeing correct convergence up to some level of approximation. However, it is also likely to require far too many episodes to be useful in practice on any but the smallest problems.

There is a second approach to avoiding the infinite number of episodes nominally required for policy evaluation, in which we give up trying to complete policy evaluation before returning to policy improvement. On each evaluation step we move the value function *toward* q_{π_k} , but we do not expect to actually get close except over many steps. We used this idea when we first introduced the idea of GPI in Section 4.6. One extreme form of the idea is value iteration, in which only one iteration of iterative policy evaluation is performed between each step of policy improvement. The in-place version of value iteration is even more extreme; there we alternate between improvement and evaluation steps for single states.

For Monte Carlo policy iteration it is natural to alternate between evaluation and improvement on an episode-by-episode basis. After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode. A complete simple algorithm along these lines, which we call *Monte Carlo ES*, for Monte Carlo with Exploring Starts, is given in pseudocode in the box on the next page.

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:

$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
 $Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
 $Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

Loop forever (for each episode):

Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0

Generate an episode from S_0, A_0 , following π : $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$\pi(S_t) \leftarrow \arg \max_a Q(S_t, a)$

Exercise 5.4 The pseudocode for Monte Carlo ES is inefficient because, for each state-action pair, it maintains a list of all returns and repeatedly calculates their mean. It would be more efficient to use techniques similar to those explained in Section 2.4 to maintain just the mean and a count (for each state-action pair) and update them incrementally. Describe how the pseudocode would be altered to achieve this. \square

In Monte Carlo ES, all the returns for each state-action pair are accumulated and averaged, irrespective of what policy was in force when they were observed. It is easy to see that Monte Carlo ES cannot converge to any suboptimal policy. If it did, then the value function would eventually converge to the value function for that policy, and that in turn would cause the policy to change. Stability is achieved only when both the policy and the value function are optimal. Convergence to this optimal fixed point seems inevitable as the changes to the action-value function decrease over time, but has not yet been formally proved. In our opinion, this is one of the most fundamental open theoretical questions in reinforcement learning (for a partial solution, see Tsitsiklis, 2002).

Example 5.3: Solving Blackjack It is straightforward to apply Monte Carlo ES to blackjack. Because the episodes are all simulated games, it is easy to arrange for exploring starts that include all possibilities. In this case one simply picks the dealer's cards, the player's sum, and whether or not the player has a usable ace, all at random with equal probability. As the initial policy we use the policy evaluated in the previous blackjack example, that which sticks only on 20 or 21. The initial action-value function can be zero for all state-action pairs. Figure 5.2 shows the optimal policy for blackjack found by Monte Carlo ES. This policy is the same as the "basic" strategy of Thorp (1966) with the sole exception of the leftmost notch in the policy for a usable ace, which is not present in Thorp's strategy. We are uncertain of the reason for this discrepancy, but confident that what is shown here is indeed the optimal policy for the version of blackjack we have described.

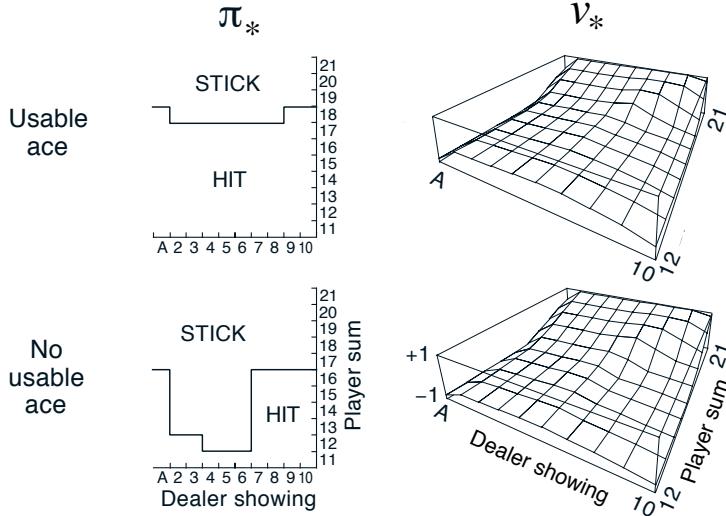


Figure 5.2: The optimal policy and state-value function for blackjack, found by Monte Carlo ES. The state-value function shown was computed from the action-value function found by Monte Carlo ES. ■

5.4 Monte Carlo Control without Exploring Starts

How can we avoid the unlikely assumption of exploring starts? The only general way to ensure that all actions are selected infinitely often is for the agent to continue to select them. There are two approaches to ensuring this, resulting in what we call *on-policy* methods and *off-policy* methods. On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data. The Monte Carlo ES method developed above is an example of an on-policy method. In this section we show how an on-policy Monte Carlo control method can be designed that does not use the unrealistic assumption of exploring starts. Off-policy methods are considered in the next section.

In on-policy control methods the policy is generally *soft*, meaning that $\pi(a|s) > 0$ for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}(s)$, but gradually shifted closer and closer to a deterministic optimal policy. Many of the methods discussed in Chapter 2 provide mechanisms for this. The on-policy method we present in this section uses ε -*greedy* policies, meaning that most of the time they choose an action that has maximal estimated action value, but with probability ε they instead select an action at random. That is, all nongreedy actions are given the minimal probability of selection, $\frac{\varepsilon}{|\mathcal{A}(s)|}$, and the remaining bulk of the probability, $1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|}$, is given to the greedy action. The ε -greedy policies are examples of ε -*soft* policies, defined as policies for which $\pi(a|s) \geq \frac{\varepsilon}{|\mathcal{A}(s)|}$ for all states and actions, for some $\varepsilon > 0$. Among ε -soft policies, ε -greedy policies are in some sense those that are closest to greedy.

The overall idea of on-policy Monte Carlo control is still that of GPI. As in Monte Carlo ES, we use first-visit MC methods to estimate the action-value function for the current policy. Without the assumption of exploring starts, however, we cannot simply improve the policy by making it greedy with respect to the current value function, because that would prevent further exploration of nongreedy actions. Fortunately, GPI does not require that the policy be taken all the way to a greedy policy, only that it be moved *toward* a greedy policy. In our on-policy method we will move it only to an ε -greedy policy. For any ε -soft policy, π , any ε -greedy policy with respect to q_π is guaranteed to be better than or equal to π . The complete algorithm is given in the box below.

On-policy first-visit MC control (for ε -soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε -soft policy

$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π : $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \arg \max_a Q(S_t, a)$ (with ties broken arbitrarily)

For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon / |\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon / |\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

That any ε -greedy policy with respect to q_π is an improvement over any ε -soft policy π is assured by the policy improvement theorem. Let π' be the ε -greedy policy. The conditions of the policy improvement theorem apply because for any $s \in \mathcal{S}$:

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_a \pi'(a|s) q_\pi(s, a) \\ &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s, a) + (1 - \varepsilon) \max_a q_\pi(s, a) \\ &\geq \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s, a) + (1 - \varepsilon) \sum_a \frac{\pi(a|s) - \frac{\varepsilon}{|\mathcal{A}(s)|}}{1 - \varepsilon} q_\pi(s, a) \end{aligned} \tag{5.2}$$

(the sum is a weighted average with nonnegative weights summing to 1, and as such it must be less than or equal to the largest number averaged)

$$\begin{aligned} &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s, a) - \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s, a) + \sum_a \pi(a|s) q_\pi(s, a) \\ &= v_\pi(s). \end{aligned}$$

Thus, by the policy improvement theorem, $\pi' \geq \pi$ (i.e., $v_{\pi'}(s) \geq v_\pi(s)$, for all $s \in \mathcal{S}$). We now prove that equality can hold only when both π' and π are optimal among the ε -soft policies, that is, when they are better than or equal to all other ε -soft policies.

Consider a new environment that is just like the original environment, except with the requirement that policies be ε -soft “moved inside” the environment. The new environment has the same action and state set as the original and behaves as follows. If in state s and taking action a , then with probability $1 - \varepsilon$ the new environment behaves exactly like the old environment. With probability ε it repicks the action at random, with equal probabilities, and then behaves like the old environment with the new, random action. The best one can do in this new environment with general policies is the same as the best one could do in the original environment with ε -soft policies. Let \tilde{v}_* and \tilde{q}_* denote the optimal value functions for the new environment. Then a policy π is optimal among ε -soft policies if and only if $v_\pi = \tilde{v}_*$. From the definition of \tilde{v}_* we know that it is the unique solution to

$$\begin{aligned} \tilde{v}_*(s) &= (1 - \varepsilon) \max_a \tilde{q}_*(s, a) + \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a \tilde{q}_*(s, a) \\ &= (1 - \varepsilon) \max_a \sum_{s', r} p(s', r|s, a) [r + \gamma \tilde{v}_*(s')] \\ &\quad + \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a \sum_{s', r} p(s', r|s, a) [r + \gamma \tilde{v}_*(s')]. \end{aligned}$$

When equality holds and the ε -soft policy π is no longer improved, then we also know, from (5.2), that

$$\begin{aligned} v_\pi(s) &= (1 - \varepsilon) \max_a q_\pi(s, a) + \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s, a) \\ &= (1 - \varepsilon) \max_a \sum_{s', r} p(s', r|s, a) [r + \gamma v_\pi(s')] \\ &\quad + \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a \sum_{s', r} p(s', r|s, a) [r + \gamma v_\pi(s')]. \end{aligned}$$

However, this equation is the same as the previous one, except for the substitution of v_π for \tilde{v}_* . Because \tilde{v}_* is the unique solution, it must be that $v_\pi = \tilde{v}_*$.

In essence, we have shown in the last few pages that policy iteration works for ε -soft policies. Using the natural notion of greedy policy for ε -soft policies, one is assured of improvement on every step, except when the best policy has been found among the ε -soft policies. This analysis is independent of how the action-value functions are determined

at each stage, but it does assume that they are computed exactly. This brings us to roughly the same point as in the previous section. Now we only achieve the best policy among the ε -soft policies, but on the other hand, we have eliminated the assumption of exploring starts.

5.5 Off-policy Prediction via Importance Sampling

All learning control methods face a dilemma: They seek to learn action values conditional on subsequent *optimal* behavior, but they need to behave non-optimally in order to explore all actions (to *find* the optimal actions). How can they learn about the optimal policy while behaving according to an exploratory policy? The on-policy approach in the preceding section is actually a compromise—it learns action values not for the optimal policy, but for a near-optimal policy that still explores. A more straightforward approach is to use two policies, one that is learned about and that becomes the optimal policy, and one that is more exploratory and is used to generate behavior. The policy being learned about is called the *target policy*, and the policy used to generate behavior is called the *behavior policy*. In this case we say that learning is from data “off” the target policy, and the overall process is termed *off-policy learning*.

Throughout the rest of this book we consider both on-policy and off-policy methods. On-policy methods are generally simpler and are considered first. Off-policy methods require additional concepts and notation, and because the data is due to a different policy, off-policy methods are often of greater variance and are slower to converge. On the other hand, off-policy methods are more powerful and general. They include on-policy methods as the special case in which the target and behavior policies are the same. Off-policy methods also have a variety of additional uses in applications. For example, they can often be applied to learn from data generated by a conventional non-learning controller, or from a human expert. Off-policy learning is also seen by some as key to learning multi-step predictive models of the world’s dynamics (see Section 17.2; Sutton, 2009; Sutton et al., 2011).

In this section we begin the study of off-policy methods by considering the *prediction* problem, in which both target and behavior policies are fixed. That is, suppose we wish to estimate v_π or q_π , but all we have are episodes following another policy b , where $b \neq \pi$. In this case, π is the target policy, b is the behavior policy, and both policies are considered fixed and given.

In order to use episodes from b to estimate values for π , we require that every action taken under π is also taken, at least occasionally, under b . That is, we require that $\pi(a|s) > 0$ implies $b(a|s) > 0$. This is called the assumption of *coverage*. It follows from coverage that b must be stochastic in states where it is not identical to π . The target policy π , on the other hand, may be deterministic, and, in fact, this is a case of particular interest in control applications. In control, the target policy is typically the deterministic greedy policy with respect to the current estimate of the action-value function. This policy becomes a deterministic optimal policy while the behavior policy remains stochastic and more exploratory, for example, an ε -greedy policy. In this section, however, we consider the prediction problem, in which π is unchanging and given.

Almost all off-policy methods utilize *importance sampling*, a general technique for estimating expected values under one distribution given samples from another. We apply importance sampling to off-policy learning by weighting returns according to the relative probability of their trajectories occurring under the target and behavior policies, called the *importance-sampling ratio*. Given a starting state S_t , the probability of the subsequent state-action trajectory, $A_t, S_{t+1}, A_{t+1}, \dots, S_T$, occurring under any policy π is

$$\begin{aligned} & \Pr\{A_t, S_{t+1}, A_{t+1}, \dots, S_T \mid S_t, A_{t:T-1} \sim \pi\} \\ &= \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1}) \cdots p(S_T|S_{T-1}, A_{T-1}) \\ &= \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k), \end{aligned}$$

where p here is the state-transition probability function defined by (3.4). Thus, the relative probability of the trajectory under the target and behavior policies (the importance-sampling ratio) is

$$\rho_{t:T-1} \doteq \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \quad (5.3)$$

Although the trajectory probabilities depend on the MDP's transition probabilities, which are generally unknown, they appear identically in both the numerator and denominator, and thus cancel. The importance sampling ratio ends up depending only on the two policies and the sequence, not on the MDP.

Recall that we wish to estimate the expected returns (values) under the target policy, but all we have are returns G_t due to the behavior policy. These returns have the wrong expectation $\mathbb{E}[G_t|S_t=s] = v_b(s)$ and so cannot be averaged to obtain v_π . This is where importance sampling comes in. The ratio $\rho_{t:T-1}$ transforms the returns to have the right expected value:

$$\mathbb{E}[\rho_{t:T-1} G_t \mid S_t=s] = v_\pi(s). \quad (5.4)$$

Now we are ready to give a Monte Carlo algorithm that averages returns from a batch of observed episodes following policy b to estimate $v_\pi(s)$. It is convenient here to number time steps in a way that increases across episode boundaries. That is, if the first episode of the batch ends in a terminal state at time 100, then the next episode begins at time $t = 101$. This enables us to use time-step numbers to refer to particular steps in particular episodes. In particular, we can define the set of all time steps in which state s is visited, denoted $\mathcal{T}(s)$. This is for an every-visit method; for a first-visit method, $\mathcal{T}(s)$ would only include time steps that were first visits to s within their episodes. Also, let $T(t)$ denote the first time of termination following time t , and G_t denote the return after t up through $T(t)$. Then $\{G_t\}_{t \in \mathcal{T}(s)}$ are the returns that pertain to state s , and $\{\rho_{t:T(t)-1}\}_{t \in \mathcal{T}(s)}$ are the corresponding importance-sampling ratios. To estimate $v_\pi(s)$, we simply scale the returns by the ratios and average the results:

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|}. \quad (5.5)$$

When importance sampling is done as a simple average in this way it is called *ordinary importance sampling*.

An important alternative is *weighted importance sampling*, which uses a *weighted* average, defined as

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1}}, \quad (5.6)$$

or zero if the denominator is zero. To understand these two varieties of importance sampling, consider the estimates of their first-visit methods after observing a single return from state s . In the weighted-average estimate, the ratio $\rho_{t:T(t)-1}$ for the single return cancels in the numerator and denominator, so that the estimate is equal to the observed return independent of the ratio (assuming the ratio is nonzero). Given that this return was the only one observed, this is a reasonable estimate, but its expectation is $v_b(s)$ rather than $v_\pi(s)$, and in this statistical sense it is biased. In contrast, the first-visit version of the ordinary importance-sampling estimator (5.5) is always $v_\pi(s)$ in expectation (it is unbiased), but it can be extreme. Suppose the ratio were ten, indicating that the trajectory observed is ten times as likely under the target policy as under the behavior policy. In this case the ordinary importance-sampling estimate would be *ten times* the observed return. That is, it would be quite far from the observed return even though the episode's trajectory is considered very representative of the target policy.

Formally, the difference between the first-visit methods of the two kinds of importance sampling is expressed in their biases and variances. Ordinary importance sampling is unbiased whereas weighted importance sampling is biased (though the bias converges asymptotically to zero). On the other hand, the variance of ordinary importance sampling is in general unbounded because the variance of the ratios can be unbounded, whereas in the weighted estimator the largest weight on any single return is one. In fact, assuming bounded returns, the variance of the weighted importance-sampling estimator converges to zero even if the variance of the ratios themselves is infinite (Precup, Sutton, and Dasgupta 2001). In practice, the weighted estimator usually has dramatically lower variance and is strongly preferred. Nevertheless, we will not totally abandon ordinary importance sampling as it is easier to extend to the approximate methods using function approximation that we explore in the second part of this book.

The every-visit methods for ordinary and weighed importance sampling are both biased, though, again, the bias falls asymptotically to zero as the number of samples increases. In practice, every-visit methods are often preferred because they remove the need to keep track of which states have been visited and because they are much easier to extend to approximations. A complete every-visit MC algorithm for off-policy policy evaluation using weighted importance sampling is given in the next section on page 110.

Exercise 5.5 Consider an MDP with a single nonterminal state and a single action that transitions back to the nonterminal state with probability p and transitions to the terminal state with probability $1-p$. Let the reward be +1 on all transitions, and let $\gamma=1$. Suppose you observe one episode that lasts 10 steps, with a return of 10. What are the first-visit and every-visit estimators of the value of the nonterminal state? \square

Example 5.4: Off-policy Estimation of a Blackjack State Value We applied both ordinary and weighted importance-sampling methods to estimate the value of a single blackjack state (Example 5.1) from off-policy data. Recall that one of the advantages of Monte Carlo methods is that they can be used to evaluate a single state without forming estimates for any other states. In this example, we evaluated the state in which the dealer is showing a deuce, the sum of the player’s cards is 13, and the player has a usable ace (that is, the player holds an ace and a deuce, or equivalently three aces). The data was generated by starting in this state then choosing to hit or stick at random with equal probability (the behavior policy). The target policy was to stick only on a sum of 20 or 21, as in Example 5.1. The value of this state under the target policy is approximately -0.27726 (this was determined by separately generating one-hundred million episodes using the target policy and averaging their returns). Both off-policy methods closely approximated this value after 1000 off-policy episodes using the random policy. To make sure they did this reliably, we performed 100 independent runs, each starting from estimates of zero and learning for 10,000 episodes. Figure 5.3 shows the resultant learning curves—the squared error of the estimates of each method as a function of number of episodes, averaged over the 100 runs. The error approaches zero for both algorithms, but the weighted importance-sampling method has much lower error at the beginning, as is typical in practice.

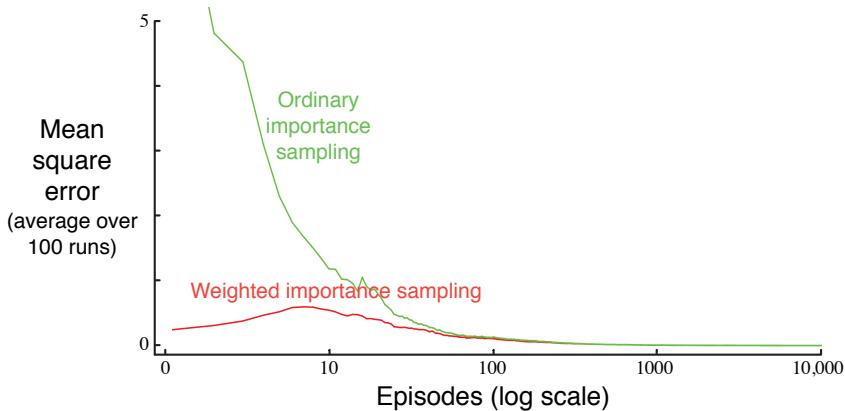


Figure 5.3: Weighted importance sampling produces lower error estimates of the value of a single blackjack state from off-policy episodes. ■

Example 5.5: Infinite Variance The estimates of ordinary importance sampling will typically have infinite variance, and thus unsatisfactory convergence properties, whenever the scaled returns have infinite variance—and this can easily happen in off-policy learning when trajectories contain loops. A simple example is shown inset in Figure 5.4. There is only one nonterminal state s and two actions, right and left. The right action causes a deterministic transition to termination, whereas the left action transitions, with probability 0.9, back to s or, with probability 0.1, on to termination. The rewards are +1 on the latter transition and otherwise zero. Consider the target policy that always selects left. All episodes under this policy consist of some number (possibly zero) of transitions back

to s followed by termination with a reward and return of +1. Thus the value of s under the target policy is 1 ($\gamma = 1$). Suppose we are estimating this value from off-policy data using the behavior policy that selects `right` and `left` with equal probability.

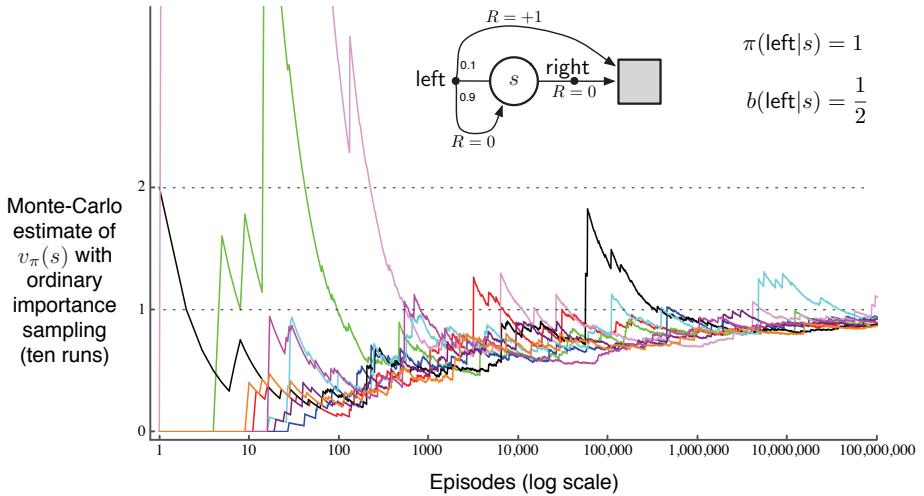


Figure 5.4: Ordinary importance sampling produces surprisingly unstable estimates on the one-state MDP shown inset (Example 5.5). The correct estimate here is 1 ($\gamma = 1$), and, even though this is the expected value of a sample return (after importance sampling), the variance of the samples is infinite, and the estimates do not converge to this value. These results are for off-policy first-visit MC.

The lower part of Figure 5.4 shows ten independent runs of the first-visit MC algorithm using ordinary importance sampling. Even after millions of episodes, the estimates fail to converge to the correct value of 1. In contrast, the weighted importance-sampling algorithm would give an estimate of exactly 1 forever after the first episode that ended with the `left` action. All returns not equal to 1 (that is, ending with the `right` action) would be inconsistent with the target policy and thus would have a $\rho_{t:T(t)-1}$ of zero and contribute neither to the numerator nor denominator of (5.6). The weighted importance-sampling algorithm produces a weighted average of only the returns consistent with the target policy, and all of these would be exactly 1.

We can verify that the variance of the importance-sampling-scaled returns is infinite in this example by a simple calculation. The variance of any random variable X is the expected value of the deviation from its mean \bar{X} , which can be written

$$\text{Var}[X] \doteq \mathbb{E}[(X - \bar{X})^2] = \mathbb{E}[X^2 - 2X\bar{X} + \bar{X}^2] = \mathbb{E}[X^2] - \bar{X}^2.$$

Thus, if the mean is finite, as it is in our case, the variance is infinite if and only if the expectation of the square of the random variable is infinite. Thus, we need only show

that the expected square of the importance-sampling-scaled return is infinite:

$$\mathbb{E} \left[\left(\prod_{t=0}^{T-1} \frac{\pi(A_t|S_t)}{b(A_t|S_t)} G_0 \right)^2 \right].$$

To compute this expectation, we break it down into cases based on episode length and termination. First note that, for any episode ending with the right action, the importance sampling ratio is zero, because the target policy would never take this action; these episodes thus contribute nothing to the expectation (the quantity in parenthesis will be zero) and can be ignored. We need only consider episodes that involve some number (possibly zero) of left actions that transition back to the nonterminal state, followed by a left action transitioning to termination. All of these episodes have a return of 1, so the G_0 factor can be ignored. To get the expected square we need only consider each length of episode, multiplying the probability of the episode's occurrence by the square of its importance-sampling ratio, and add these up:

$$\begin{aligned} &= \frac{1}{2} \cdot 0.1 \left(\frac{1}{0.5} \right)^2 && \text{(the length 1 episode)} \\ &+ \frac{1}{2} \cdot 0.9 \cdot \frac{1}{2} \cdot 0.1 \left(\frac{1}{0.5} \frac{1}{0.5} \right)^2 && \text{(the length 2 episode)} \\ &+ \frac{1}{2} \cdot 0.9 \cdot \frac{1}{2} \cdot 0.9 \cdot \frac{1}{2} \cdot 0.1 \left(\frac{1}{0.5} \frac{1}{0.5} \frac{1}{0.5} \right)^2 && \text{(the length 3 episode)} \\ &+ \dots \\ &= 0.1 \sum_{k=0}^{\infty} 0.9^k \cdot 2^k \cdot 2 = 0.2 \sum_{k=0}^{\infty} 1.8^k = \infty. \end{aligned}$$
■

Exercise 5.6 What is the equation analogous to (5.6) for *action* values $Q(s, a)$ instead of state values $V(s)$, again given returns generated using b ? □

Exercise 5.7 In learning curves such as those shown in Figure 5.3 error generally decreases with training, as indeed happened for the ordinary importance-sampling method. But for the weighted importance-sampling method error first increased and then decreased. Why do you think this happened? □

Exercise 5.8 The results with Example 5.5 and shown in Figure 5.4 used a first-visit MC method. Suppose that instead an every-visit MC method was used on the same problem. Would the variance of the estimator still be infinite? Why or why not? □

5.6 Incremental Implementation

Monte Carlo prediction methods can be implemented incrementally, on an episode-by-episode basis, using extensions of the techniques described in Chapter 2 (Section 2.4). Whereas in Chapter 2 we averaged *rewards*, in Monte Carlo methods we average *returns*. In all other respects exactly the same methods as used in Chapter 2 can be used for *on-policy* Monte Carlo methods. For *off-policy* Monte Carlo methods, we need to separately consider those that use *ordinary* importance sampling and those that use *weighted* importance sampling.

In ordinary importance sampling, the returns are scaled by the importance sampling ratio $\rho_{t:T(t)-1}$ (5.3), then simply averaged, as in (5.5). For these methods we can again use the incremental methods of Chapter 2, but using the scaled returns in place of the rewards of that chapter. This leaves the case of off-policy methods using *weighted* importance sampling. Here we have to form a weighted average of the returns, and a slightly different incremental algorithm is required.

Suppose we have a sequence of returns G_1, G_2, \dots, G_{n-1} , all starting in the same state and each with a corresponding random weight W_i (e.g., $W_i = \rho_{t_i:T(t_i)-1}$). We wish to form the estimate

$$V_n \doteq \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}, \quad n \geq 2, \tag{5.7}$$

and keep it up-to-date as we obtain a single additional return G_n . In addition to keeping track of V_n , we must maintain for each state the cumulative sum C_n of the weights given to the first n returns. The update rule for V_n is

$$V_{n+1} \doteq V_n + \frac{W_n}{C_n} [G_n - V_n], \quad n \geq 1, \tag{5.8}$$

and

$$C_{n+1} \doteq C_n + W_{n+1},$$

where $C_0 \doteq 0$ (and V_1 is arbitrary and thus need not be specified). The box on the next page contains a complete episode-by-episode incremental algorithm for Monte Carlo policy evaluation. The algorithm is nominally for the off-policy case, using weighted importance sampling, but applies as well to the on-policy case just by choosing the target and behavior policies as the same (in which case ($\pi = b$), W is always 1). The approximation Q converges to q_π (for all encountered state-action pairs) while actions are selected according to a potentially different policy, b .

Exercise 5.9 Modify the algorithm for first-visit MC policy evaluation (Section 5.1) to use the incremental implementation for sample averages described in Section 2.4. \square

Exercise 5.10 Derive the weighted-average update rule (5.8) from (5.7). Follow the pattern of the derivation of the unweighted rule (2.3). \square

Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

Input: an arbitrary target policy π

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

$$Q(s, a) \in \mathbb{R} \text{ (arbitrarily)}$$

$$C(s, a) \leftarrow 0$$

Loop forever (for each episode):

$$b \leftarrow \text{any policy with coverage of } \pi$$

Generate an episode following b : $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$$G \leftarrow 0$$

$$W \leftarrow 1$$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$, while $W \neq 0$:

$$G \leftarrow \gamma G + R_{t+1}$$

$$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$$

$$W \leftarrow W \frac{\pi(A_t | S_t)}{b(A_t | S_t)}$$

5.7 Off-policy Monte Carlo Control

We are now ready to present an example of the second class of learning control methods we consider in this book: off-policy methods. Recall that the distinguishing feature of on-policy methods is that they estimate the value of a policy while using it for control. In off-policy methods these two functions are separated. The policy used to generate behavior, called the *behavior* policy, may in fact be unrelated to the policy that is evaluated and improved, called the *target* policy. An advantage of this separation is that the target policy may be deterministic (e.g., greedy), while the behavior policy can continue to sample all possible actions.

Off-policy Monte Carlo control methods use one of the techniques presented in the preceding two sections. They follow the behavior policy while learning about and improving the target policy. These techniques require that the behavior policy has a nonzero probability of selecting all actions that might be selected by the target policy (coverage). To explore all possibilities, we require that the behavior policy be soft (i.e., that it select all actions in all states with nonzero probability).

The box on the next page shows an off-policy Monte Carlo control method, based on GPI and weighted importance sampling, for estimating π_* and q_* . The target policy $\pi \approx \pi_*$ is the greedy policy with respect to Q , which is an estimate of q_π . The behavior policy b can be anything, but in order to assure convergence of π to the optimal policy, an infinite number of returns must be obtained for each pair of state and action. This can be assured by choosing b to be ε -soft. The policy π converges to optimal at all encountered states even though actions are selected according to a different soft policy b , which may change between or even within episodes.

Off-policy MC control, for estimating $\pi \approx \pi_*$

```

Initialize, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$ :
 $Q(s, a) \in \mathbb{R}$  (arbitrarily)
 $C(s, a) \leftarrow 0$ 
 $\pi(s) \leftarrow \operatorname{argmax}_a Q(s, a)$  (with ties broken consistently)

Loop forever (for each episode):
 $b \leftarrow$  any soft policy
Generate an episode using  $b$ :  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ 
 $G \leftarrow 0$ 
 $W \leftarrow 1$ 
Loop for each step of episode,  $t = T-1, T-2, \dots, 0$ :
 $G \leftarrow \gamma G + R_{t+1}$ 
 $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$ 
 $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$ 
 $\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$  (with ties broken consistently)
If  $A_t \neq \pi(S_t)$  then exit inner Loop (proceed to next episode)
 $W \leftarrow W \frac{1}{b(A_t|S_t)}$ 

```

A potential problem is that this method learns only from the tails of episodes, when all of the remaining actions in the episode are greedy. If nongreedy actions are common, then learning will be slow, particularly for states appearing in the early portions of long episodes. Potentially, this could greatly slow learning. There has been insufficient experience with off-policy Monte Carlo methods to assess how serious this problem is. If it is serious, the most important way to address it is probably by incorporating temporal-difference learning, the algorithmic idea developed in the next chapter. Alternatively, if γ is less than 1, then the idea developed in the next section may also help significantly.

Exercise 5.11 In the boxed algorithm for off-policy MC control, you may have been expecting the W update to have involved the importance-sampling ratio $\frac{\pi(A_t|S_t)}{b(A_t|S_t)}$, but instead it involves $\frac{1}{b(A_t|S_t)}$. Why is this nevertheless correct? \square

Exercise 5.12: Racetrack (programming) Consider driving a race car around a turn like those shown in Figure 5.5. You want to go as fast as possible, but not so fast as to run off the track. In our simplified racetrack, the car is at one of a discrete set of grid positions, the cells in the diagram. The velocity is also discrete, a number of grid cells moved horizontally and vertically per time step. The actions are increments to the velocity components. Each may be changed by $+1$, -1 , or 0 in each step, for a total of nine (3×3) actions. Both velocity components are restricted to be nonnegative and less than 5, and they cannot both be zero except at the starting line. Each episode begins in one of the randomly selected start states with both velocity components zero and ends when the car crosses the finish line. The rewards are -1 for each step until the car crosses the finish line. If the car hits the track boundary, it is moved back to a random position on the starting line, both velocity components are reduced to zero, and the

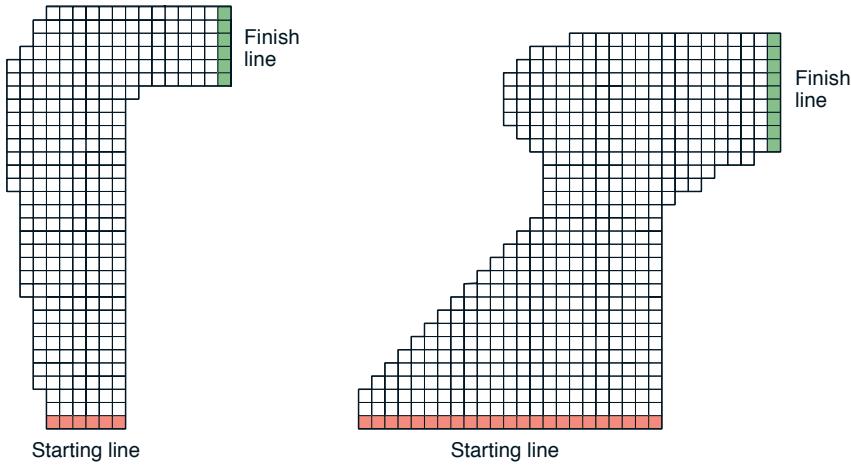


Figure 5.5: A couple of right turns for the racetrack task.

episode continues. Before updating the car’s location at each time step, check to see if the projected path of the car intersects the track boundary. If it intersects the finish line, the episode ends; if it intersects anywhere else, the car is considered to have hit the track boundary and is sent back to the starting line. To make the task more challenging, with probability 0.1 at each time step the velocity increments are both zero, independently of the intended increments. Apply a Monte Carlo control method to this task to compute the optimal policy from each starting state. Exhibit several trajectories following the optimal policy (but turn the noise off for these trajectories). \square

5.8 *Discounting-aware Importance Sampling

The off-policy methods that we have considered so far are based on forming importance-sampling weights for returns considered as unitary wholes, without taking into account the returns’ internal structures as sums of discounted rewards. We now briefly consider cutting-edge research ideas for using this structure to significantly reduce the variance of off-policy estimators.

For example, consider the case where episodes are long and γ is significantly less than 1. For concreteness, say that episodes last 100 steps and that $\gamma = 0$. The return from time 0 will then be just $G_0 = R_1$, but its importance sampling ratio will be a product of 100 factors, $\frac{\pi(A_0|S_0)}{b(A_0|S_0)} \frac{\pi(A_1|S_1)}{b(A_1|S_1)} \dots \frac{\pi(A_{99}|S_{99})}{b(A_{99}|S_{99})}$. In ordinary importance sampling, the return will be scaled by the entire product, but it is really only necessary to scale by the first factor, by $\frac{\pi(A_0|S_0)}{b(A_0|S_0)}$. The other 99 factors $\frac{\pi(A_1|S_1)}{b(A_1|S_1)} \dots \frac{\pi(A_{99}|S_{99})}{b(A_{99}|S_{99})}$ are irrelevant because after the first reward the return has already been determined. These later factors are all independent of the return and of expected value 1; they do not change the expected update, but they add enormously to its variance. In some cases they could even make the variance infinite. Let us now consider an idea for avoiding this large extraneous variance.

The essence of the idea is to think of discounting as determining a probability of termination or, equivalently, a *degree* of partial termination. For any $\gamma \in [0, 1)$, we can think of the return G_0 as partly terminating in one step, to the degree $1 - \gamma$, producing a return of just the first reward, R_1 , and as partly terminating after two steps, to the degree $(1 - \gamma)\gamma$, producing a return of $R_1 + R_2$, and so on. The latter degree corresponds to terminating on the second step, $1 - \gamma$, and not having already terminated on the first step, γ . The degree of termination on the third step is thus $(1 - \gamma)\gamma^2$, with the γ^2 reflecting that termination did not occur on either of the first two steps. The partial returns here are called *flat partial returns*:

$$\bar{G}_{t:h} \doteq R_{t+1} + R_{t+2} + \cdots + R_h, \quad 0 \leq t < h \leq T,$$

where “flat” denotes the absence of discounting, and “partial” denotes that these returns do not extend all the way to termination but instead stop at h , called the *horizon* (and T is the time of termination of the episode). The conventional full return G_t can be viewed as a sum of flat partial returns as suggested above as follows:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T \\ &= (1 - \gamma)R_{t+1} \\ &\quad + (1 - \gamma)\gamma(R_{t+1} + R_{t+2}) \\ &\quad + (1 - \gamma)\gamma^2(R_{t+1} + R_{t+2} + R_{t+3}) \\ &\quad \vdots \\ &\quad + (1 - \gamma)\gamma^{T-t-2}(R_{t+1} + R_{t+2} + \cdots + R_{T-1}) \\ &\quad + \gamma^{T-t-1}(R_{t+1} + R_{t+2} + \cdots + R_T) \\ &= (1 - \gamma) \sum_{h=t+1}^{T-1} \gamma^{h-t-1} \bar{G}_{t:h} + \gamma^{T-t-1} \bar{G}_{t:T}. \end{aligned}$$

Now we need to scale the flat partial returns by an importance sampling ratio that is similarly truncated. As $\bar{G}_{t:h}$ only involves rewards up to a horizon h , we only need the ratio of the probabilities up to $h - 1$. We define an ordinary importance-sampling estimator, analogous to (5.5), as

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \left((1 - \gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} \bar{G}_{t:h} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \bar{G}_{t:T(t)} \right)}{|\mathcal{T}(s)|}, \quad (5.9)$$

and a weighted importance-sampling estimator, analogous to (5.6), as

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \left((1 - \gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} \bar{G}_{t:h} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \bar{G}_{t:T(t)} \right)}{\sum_{t \in \mathcal{T}(s)} \left((1 - \gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \right)}. \quad (5.10)$$

We call these two estimators *discounting-aware* importance sampling estimators. They take into account the discount rate but have no effect (are the same as the off-policy estimators from Section 5.5) if $\gamma = 1$.

5.9 *Per-decision Importance Sampling

There is one more way in which the structure of the return as a sum of rewards can be taken into account in off-policy importance sampling, a way that may be able to reduce variance even in the absence of discounting (that is, even if $\gamma = 1$). In the off-policy estimators (5.5) and (5.6), each term of the sum in the numerator is itself a sum:

$$\begin{aligned}\rho_{t:T-1}G_t &= \rho_{t:T-1}(R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1}R_T) \\ &= \rho_{t:T-1}R_{t+1} + \gamma\rho_{t:T-1}R_{t+2} + \cdots + \gamma^{T-t-1}\rho_{t:T-1}R_T.\end{aligned}\quad (5.11)$$

The off-policy estimators rely on the expected values of these terms, which can be written in a simpler way. Note that each sub-term of (5.11) is a product of a random reward and a random importance-sampling ratio. For example, the first sub-term can be written, using (5.3), as

$$\rho_{t:T-1}R_{t+1} = \frac{\pi(A_t|S_t)}{b(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})} \frac{\pi(A_{t+2}|S_{t+2})}{b(A_{t+2}|S_{t+2})} \cdots \frac{\pi(A_{T-1}|S_{T-1})}{b(A_{T-1}|S_{T-1})} R_{t+1}. \quad (5.12)$$

Of all these factors, one might suspect that only the first and the last (the reward) are related; all the others are for events that occurred after the reward. Moreover, the expected value of all these other factors is one:

$$\mathbb{E}\left[\frac{\pi(A_k|S_k)}{b(A_k|S_k)}\right] \doteq \sum_a b(a|S_k) \frac{\pi(a|S_k)}{b(a|S_k)} = \sum_a \pi(a|S_k) = 1. \quad (5.13)$$

With a few more steps, one can show that, as suspected, all of these other factors have no effect in expectation, in other words, that

$$\mathbb{E}[\rho_{t:T-1}R_{t+1}] = \mathbb{E}[\rho_{t:t}R_{t+1}]. \quad (5.14)$$

If we repeat this process for the k th sub-term of (5.11), we get

$$\mathbb{E}[\rho_{t:T-1}R_{t+k}] = \mathbb{E}[\rho_{t:t+k-1}R_{t+k}].$$

It follows then that the expectation of our original term (5.11) can be written

$$\mathbb{E}[\rho_{t:T-1}G_t] = \mathbb{E}\left[\tilde{G}_t\right],$$

where

$$\tilde{G}_t = \rho_{t:t}R_{t+1} + \gamma\rho_{t:t+1}R_{t+2} + \gamma^2\rho_{t:t+2}R_{t+3} + \cdots + \gamma^{T-t-1}\rho_{t:T-1}R_T.$$

We call this idea *per-decision* importance sampling.

It follows immediately that there is an alternate importance-sampling estimator, with the same unbiased expectation (in the first-visit case) as the ordinary-importance-sampling estimator (5.5), using \tilde{G}_t :

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \tilde{G}_t}{|\mathcal{T}(s)|}, \quad (5.15)$$

which we might expect to sometimes be of lower variance.

Is there a per-decision version of *weighted* importance sampling? This is less clear. So far, all the estimators that have been proposed for this that we know of are not consistent (that is, they do not converge to the true value with infinite data).

*Exercise 5.13 Show the steps to derive (5.14) from (5.12). \square

*Exercise 5.14 Modify the algorithm for off-policy Monte Carlo control (page 111) to use the idea of the truncated weighted-average estimator (5.10). Note that you will first need to convert this equation to action values. \square

5.10 Summary

The Monte Carlo methods presented in this chapter learn value functions and optimal policies from experience in the form of *sample episodes*. This gives them at least three kinds of advantages over DP methods. First, they can be used to learn optimal behavior directly from interaction with the environment, with no model of the environment's dynamics. Second, they can be used with simulation or *sample models*. For surprisingly many applications it is easy to simulate sample episodes even though it is difficult to construct the kind of explicit model of transition probabilities required by DP methods. Third, it is easy and efficient to *focus* Monte Carlo methods on a small subset of the states. A region of special interest can be accurately evaluated without going to the expense of accurately evaluating the rest of the state set (we explore this further in Chapter 8).

A fourth advantage of Monte Carlo methods, which we discuss later in the book, is that they may be less harmed by violations of the Markov property. This is because they do not update their value estimates on the basis of the value estimates of successor states. In other words, it is because they do not bootstrap.

In designing Monte Carlo control methods we have followed the overall schema of *generalized policy iteration* (GPI) introduced in Chapter 4. GPI involves interacting processes of policy evaluation and policy improvement. Monte Carlo methods provide an alternative policy evaluation process. Rather than use a model to compute the value of each state, they simply average many returns that start in the state. Because a state's value is the expected return, this average can become a good approximation to the value. In control methods we are particularly interested in approximating action-value functions, because these can be used to improve the policy without requiring a model of the environment's transition dynamics. Monte Carlo methods intermix policy evaluation and policy improvement steps on an episode-by-episode basis, and can be incrementally implemented on an episode-by-episode basis.

Maintaining *sufficient exploration* is an issue in Monte Carlo control methods. It is not enough just to select the actions currently estimated to be best, because then no returns will be obtained for alternative actions, and it may never be learned that they are actually better. One approach is to ignore this problem by assuming that episodes begin with state-action pairs randomly selected to cover all possibilities. Such *exploring starts* can sometimes be arranged in applications with simulated episodes, but are unlikely in learning from real experience. In *on-policy* methods, the agent commits to always exploring and tries to find the best policy that still explores. In *off-policy* methods, the agent also explores, but learns a deterministic optimal policy that may be unrelated to the policy followed.

Off-policy prediction refers to learning the value function of a *target policy* from data generated by a different *behavior policy*. Such learning methods are based on some form of *importance sampling*, that is, on weighting returns by the ratio of the probabilities of taking the observed actions under the two policies, thereby transforming their expectations from the behavior policy to the target policy. *Ordinary importance sampling* uses a simple average of the weighted returns, whereas *weighted importance sampling* uses a weighted average. Ordinary importance sampling produces unbiased estimates, but has larger, possibly infinite, variance, whereas weighted importance sampling always has finite variance and is preferred in practice. Despite their conceptual simplicity, off-policy Monte Carlo methods for both prediction and control remain unsettled and are a subject of ongoing research.

The Monte Carlo methods treated in this chapter differ from the DP methods treated in the previous chapter in two major ways. First, they operate on sample experience, and thus can be used for direct learning without a model. Second, they do not bootstrap. That is, they do not update their value estimates on the basis of other value estimates. These two differences are not tightly linked, and can be separated. In the next chapter we consider methods that learn from experience, like Monte Carlo methods, but also bootstrap, like DP methods.

Bibliographical and Historical Remarks

The term “Monte Carlo” dates from the 1940s, when physicists at Los Alamos devised games of chance that they could study to help understand complex physical phenomena relating to the atom bomb. Coverage of Monte Carlo methods in this sense can be found in several textbooks (e.g., Kalos and Whitlock, 1986; Rubinstein, 1981).

5.1–2 Singh and Sutton (1996) distinguished between every-visit and first-visit MC methods and proved results relating these methods to reinforcement learning algorithms. The blackjack example is based on an example used by Widrow, Gupta, and Maitra (1973). The soap bubble example is a classical Dirichlet problem whose Monte Carlo solution was first proposed by Kakutani (1945; see Hersh and Griego, 1969; Doyle and Snell, 1984).

Barto and Duff (1994) discussed policy evaluation in the context of classical Monte Carlo algorithms for solving systems of linear equations. They used the

analysis of Curtiss (1954) to point out the computational advantages of Monte Carlo policy evaluation for large problems.

- 5.3–4** Monte Carlo ES was introduced in the 1998 edition of this book. That may have been the first explicit connection between Monte Carlo estimation and control methods based on policy iteration. An early use of Monte Carlo methods to estimate action values in a reinforcement learning context was by Michie and Chambers (1968). In pole balancing (page 56), they used averages of episode durations to assess the worth (expected balancing “life”) of each possible action in each state, and then used these assessments to control action selections. Their method is similar in spirit to Monte Carlo ES with every-visit MC estimates. Narendra and Wheeler (1986) studied a Monte Carlo method for ergodic finite Markov chains that used the return accumulated between successive visits to the same state as a reward for adjusting a learning automaton’s action probabilities.
- 5.5** Efficient off-policy learning has become recognized as an important challenge that arises in several fields. For example, it is closely related to the idea of “interventions” and “counterfactuals” in probabilistic graphical (Bayesian) models (e.g., Pearl, 1995; Balke and Pearl, 1994). Off-policy methods using importance sampling have a long history and yet still are not well understood. Weighted importance sampling, which is also sometimes called normalized importance sampling (e.g., Koller and Friedman, 2009), is discussed by Rubinstein (1981), Hesterberg (1988), Shelton (2001), and Liu (2001) among others. The target policy in off-policy learning is sometimes referred to in the literature as the “estimation” policy, as it was in the first edition of this book.
- 5.7** The racetrack exercise is adapted from Barto, Bradtke, and Singh (1995), and from Gardner (1973).
- 5.8** Our treatment of the idea of discounting-aware importance sampling is based on the analysis of Sutton, Mahmood, Precup, and van Hasselt (2014). It has been worked out most fully to date by Mahmood (2017; Mahmood, van Hasselt, and Sutton, 2014).
- 5.9** Per-decision importance sampling was introduced by Precup, Sutton, and Singh (2000). They also combined off-policy learning with temporal-difference learning, eligibility traces, and approximation methods, introducing subtle issues that we consider in later chapters.

Chapter 6

Temporal-Difference Learning

If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be *temporal-difference* (TD) learning. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap). The relationship between TD, DP, and Monte Carlo methods is a recurring theme in the theory of reinforcement learning; this chapter is the beginning of our exploration of it. Before we are done, we will see that these ideas and methods blend into each other and can be combined in many ways. In particular, in Chapter 7 we introduce n -step algorithms, which provide a bridge from TD to Monte Carlo methods, and in Chapter 12 we introduce the $\text{TD}(\lambda)$ algorithm, which seamlessly unifies them.

As usual, we start by focusing on the policy evaluation or *prediction* problem, the problem of estimating the value function v_π for a given policy π . For the *control* problem (finding an optimal policy), DP, TD, and Monte Carlo methods all use some variation of generalized policy iteration (GPI). The differences in the methods are primarily differences in their approaches to the prediction problem.

6.1 TD Prediction

Both TD and Monte Carlo methods use experience to solve the prediction problem. Given some experience following a policy π , both methods update their estimate V of v_π for the nonterminal states S_t occurring in that experience. Roughly speaking, Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$. A simple every-visit Monte Carlo method suitable for nonstationary environments is

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)], \quad (6.1)$$

where G_t is the actual return following time t , and α is a constant step-size parameter (c.f., Equation 2.4). Let us call this method *constant- α MC*. Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(S_t)$ (only then is G_t known), TD methods need to wait only until the next time step. At time $t+1$ they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$. The simplest TD method makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \quad (6.2)$$

immediately on transition to S_{t+1} and receiving R_{t+1} . In effect, the target for the Monte Carlo update is G_t , whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. This TD method is called *TD(0)*, or *one-step TD*, because it is a special case of the $\text{TD}(\lambda)$ and n -step TD methods developed in Chapter 12 and Chapter 7. The box below specifies TD(0) completely in procedural form.

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated

Algorithm parameter: step size $\alpha \in (0, 1]$

Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop for each episode:

 Initialize S

 Loop for each step of episode:

$A \leftarrow$ action given by π for S

 Take action A , observe R, S'

$V(S) \leftarrow V(S) + \alpha [R + \gamma V(S') - V(S)]$

$S \leftarrow S'$

 until S is terminal

Because TD(0) bases its update in part on an existing estimate, we say that it is a *bootstrapping* method, like DP. We know from Chapter 3 that

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] \quad (6.3)$$

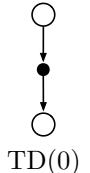
$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \quad (\text{from (3.9)})$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]. \quad (6.4)$$

Roughly speaking, Monte Carlo methods use an estimate of (6.3) as a target, whereas DP methods use an estimate of (6.4) as a target. The Monte Carlo target is an estimate because the expected value in (6.3) is not known; a sample return is used in place of the real expected return. The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because $v_\pi(S_{t+1})$ is not known and the current estimate, $V(S_{t+1})$, is used instead. The TD target is an estimate for both reasons: it samples the expected values in (6.4) *and* it uses the current estimate V instead of the true v_π . Thus, TD methods combine the sampling of

Monte Carlo with the bootstrapping of DP. As we shall see, with care and imagination this can take us a long way toward obtaining the advantages of both Monte Carlo and DP methods.

Shown to the right is the backup diagram for tabular TD(0). The value estimate for the state node at the top of the backup diagram is updated on the basis of the one sample transition from it to the immediately following state. We refer to TD and Monte Carlo updates as *sample updates* because they involve looking ahead to a sample successor state (or state-action pair), using the value of the successor and the reward along the way to compute a backed-up value, and then updating the value of the original state (or state-action pair) accordingly. *Sample* updates differ from the *expected* updates of DP methods in that they are based on a single sample successor rather than on a complete distribution of all possible successors.



Finally, note that the quantity in brackets in the TD(0) update is a sort of error, measuring the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$. This quantity, called the *TD error*, arises in various forms throughout reinforcement learning:

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \quad (6.5)$$

Notice that the TD error at each time is the error in the estimate *made at that time*. Because the TD error depends on the next state and next reward, it is not actually available until one time step later. That is, δ_t is the error in $V(S_t)$, available at time $t+1$. Also note that if the array V does not change during the episode (as it does not in Monte Carlo methods), then the Monte Carlo error can be written as a sum of TD errors:

$$\begin{aligned} G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) && \text{(from (3.9))} \\ &= \delta_t + \gamma(G_{t+1} - V(S_{t+1})) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2(G_{t+2} - V(S_{t+2})) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(G_T - V(S_T)) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(0 - 0) \\ &= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k. \end{aligned} \quad (6.6)$$

This identity is not exact if V is updated during the episode (as it is in TD(0)), but if the step size is small then it may still hold approximately. Generalizations of this identity play an important role in the theory and algorithms of temporal-difference learning.

Exercise 6.1 If V changes during the episode, then (6.6) only holds approximately; what would the difference be between the two sides? Let V_t denote the array of state values used at time t in the TD error (6.5) and in the TD update (6.2). Redo the derivation above to determine the additional amount that must be added to the sum of TD errors in order to equal the Monte Carlo error. \square

Example 6.1: Driving Home Each day as you drive home from work, you try to predict how long it will take to get home. When you leave your office, you note the time, the day of week, the weather, and anything else that might be relevant. Say on this Friday you are leaving at exactly 6 o'clock, and you estimate that it will take 30 minutes to get home. As you reach your car it is 6:05, and you notice it is starting to rain. Traffic is often slower in the rain, so you reestimate that it will take 35 minutes from then, or a total of 40 minutes. Fifteen minutes later you have completed the highway portion of your journey in good time. As you exit onto a secondary road you cut your estimate of total travel time to 35 minutes. Unfortunately, at this point you get stuck behind a slow truck, and the road is too narrow to pass. You end up having to follow the truck until you turn onto the side street where you live at 6:40. Three minutes later you are home. The sequence of states, times, and predictions is thus as follows:

State	Elapsed Time (minutes)	Predicted Time to Go	Predicted Total Time
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

The rewards in this example are the elapsed times on each leg of the journey.¹ We are not discounting ($\gamma = 1$), and thus the return for each state is the actual time to go from that state. The value of each state is the *expected* time to go. The second column of numbers gives the current estimated value for each state encountered.

A simple way to view the operation of Monte Carlo methods is to plot the predicted total time (the last column) over the sequence, as in Figure 6.1 (left). The red arrows show the changes in predictions recommended by the constant- α MC method (6.1), for $\alpha = 1$. These are exactly the errors between the estimated value (predicted time to go) in each state and the actual return (actual time to go). For example, when you exited the highway you thought it would take only 15 minutes more to get home, but in fact it took 23 minutes. Equation 6.1 applies at this point and determines an increment in the estimate of time to go after exiting the highway. The error, $G_t - V(S_t)$, at this time is eight minutes. Suppose the step-size parameter, α , is 1/2. Then the predicted time to go after exiting the highway would be revised upward by four minutes as a result of this experience. This is probably too large a change in this case; the truck was probably just an unlucky break. In any event, the change can only be made off-line, that is, after you have reached home. Only at this point do you know any of the actual returns.

Is it necessary to wait until the final outcome is known before learning can begin? Suppose on another day you again estimate when leaving your office that it will take 30 minutes to drive home, but then you become stuck in a massive traffic jam. Twenty-five minutes after leaving the office you are still bumper-to-bumper on the highway. You now

¹If this were a control problem with the objective of minimizing travel time, then we would of course make the rewards the *negative* of the elapsed time. But because we are concerned here only with prediction (policy evaluation), we can keep things simple by using positive numbers.

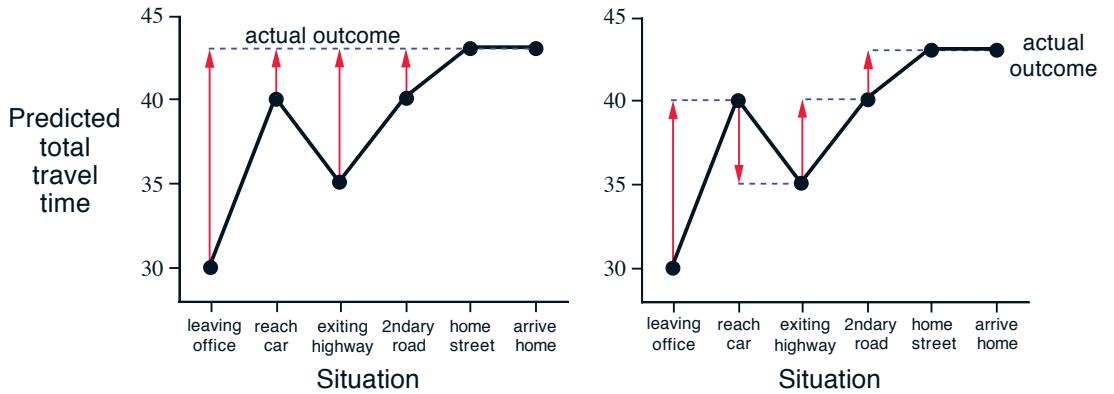


Figure 6.1: Changes recommended in the driving home example by Monte Carlo methods (left) and TD methods (right).

estimate that it will take another 25 minutes to get home, for a total of 50 minutes. As you wait in traffic, you already know that your initial estimate of 30 minutes was too optimistic. Must you wait until you get home before increasing your estimate for the initial state? According to the Monte Carlo approach you must, because you don't yet know the true return.

According to a TD approach, on the other hand, you would learn immediately, shifting your initial estimate from 30 minutes toward 50. In fact, each estimate would be shifted toward the estimate that immediately follows it. Returning to our first day of driving, Figure 6.1 (right) shows the changes in the predictions recommended by the TD rule (6.2) (these are the changes made by the rule if $\alpha = 1$). Each error is proportional to the change over time of the prediction, that is, to the *temporal differences* in predictions.

Besides giving you something to do while waiting in traffic, there are several computational reasons why it is advantageous to learn based on your current predictions rather than waiting until termination when you know the actual return. We briefly discuss some of these in the next section. ■

Exercise 6.2 This is an exercise to help develop your intuition about why TD methods are often more efficient than Monte Carlo methods. Consider the driving home example and how it is addressed by TD and Monte Carlo methods. Can you imagine a scenario in which a TD update would be better on average than a Monte Carlo update? Give an example scenario—a description of past experience and a current state—in which you would expect the TD update to be better. Here's a hint: Suppose you have lots of experience driving home from work. Then you move to a new building and a new parking lot (but you still enter the highway at the same place). Now you are starting to learn predictions for the new building. Can you see why TD updates are likely to be much better, at least initially, in this case? Might the same sort of thing happen in the original scenario? □

6.2 Advantages of TD Prediction Methods

TD methods update their estimates based in part on other estimates. They learn a guess from a guess—they *bootstrap*. Is this a good thing to do? What advantages do TD methods have over Monte Carlo and DP methods? Developing and answering such questions will take the rest of this book and more. In this section we briefly anticipate some of the answers.

Obviously, TD methods have an advantage over DP methods in that they do not require a model of the environment, of its reward and next-state probability distributions.

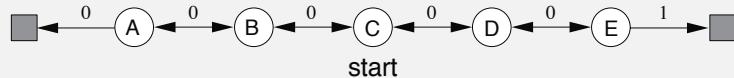
The next most obvious advantage of TD methods over Monte Carlo methods is that they are naturally implemented in an online, fully incremental fashion. With Monte Carlo methods one must wait until the end of an episode, because only then is the return known, whereas with TD methods one need wait only one time step. Surprisingly often this turns out to be a critical consideration. Some applications have very long episodes, so that delaying all learning until the end of the episode is too slow. Other applications are continuing tasks and have no episodes at all. Finally, as we noted in the previous chapter, some Monte Carlo methods must ignore or discount episodes on which experimental actions are taken, which can greatly slow learning. TD methods are much less susceptible to these problems because they learn from each transition regardless of what subsequent actions are taken.

But are TD methods sound? Certainly it is convenient to learn one guess from the next, without waiting for an actual outcome, but can we still guarantee convergence to the correct answer? Happily, the answer is yes. For any fixed policy π , TD(0) has been proved to converge to v_π , in the mean for a constant step-size parameter if it is sufficiently small, and with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions (2.7). Most convergence proofs apply only to the table-based case of the algorithm presented above (6.2), but some also apply to the case of general linear function approximation. These results are discussed in a more general setting in Section 9.4.

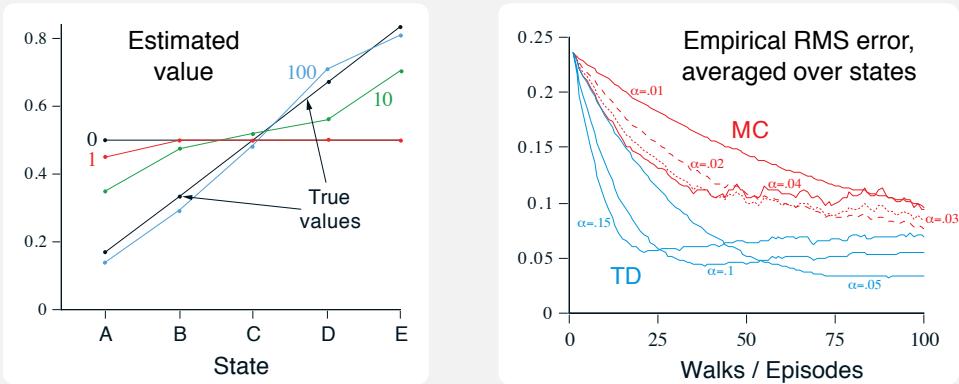
If both TD and Monte Carlo methods converge asymptotically to the correct predictions, then a natural next question is “Which gets there first?” In other words, which method learns faster? Which makes the more efficient use of limited data? At the current time this is an open question in the sense that no one has been able to prove mathematically that one method converges faster than the other. In fact, it is not even clear what is the most appropriate formal way to phrase this question! In practice, however, TD methods have usually been found to converge faster than constant- α MC methods on stochastic tasks, as illustrated in Example 6.2.

Example 6.2 Random Walk

In this example we empirically compare the prediction abilities of TD(0) and constant- α MC when applied to the following Markov reward process:



A *Markov reward process*, or MRP, is a Markov decision process without actions. We will often use MRPs when focusing on the prediction problem, in which there is no need to distinguish the dynamics due to the environment from those due to the agent. In this MRP, all episodes start in the center state, C, then proceed either left or right by one state on each step, with equal probability. Episodes terminate either on the extreme left or the extreme right. When an episode terminates on the right, a reward of +1 occurs; all other rewards are zero. For example, a typical episode might consist of the following state-and-reward sequence: C, 0, B, 0, C, 0, D, 0, E, 1. Because this task is undiscounted, the true value of each state is the probability of terminating on the right if starting from that state. Thus, the true value of the center state is $v_\pi(C) = 0.5$. The true values of all the states, A through E, are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}$, and $\frac{5}{6}$.



The left graph above shows the values learned after various numbers of episodes on a single run of TD(0). The estimates after 100 episodes are about as close as they ever come to the true values—with a constant step-size parameter ($\alpha = 0.1$ in this example), the values fluctuate indefinitely in response to the outcomes of the most recent episodes. The right graph shows learning curves for the two methods for various values of α . The performance measure shown is the root mean square (RMS) error between the value function learned and the true value function, averaged over the five states, then averaged over 100 runs. In all cases the approximate value function was initialized to the intermediate value $V(s) = 0.5$, for all s . The TD method was consistently better than the MC method on this task.

Exercise 6.3 From the results shown in the left graph of the random walk example it appears that the first episode results in a change in only $V(A)$. What does this tell you about what happened on the first episode? Why was only the estimate for this one state changed? By exactly how much was it changed? \square

Exercise 6.4 The specific results shown in the right graph of the random walk example are dependent on the value of the step-size parameter, α . Do you think the conclusions about which algorithm is better would be affected if a wider range of α values were used? Is there a different, fixed value of α at which either algorithm would have performed significantly better than shown? Why or why not? \square

**Exercise 6.5* In the right graph of the random walk example, the RMS error of the TD method seems to go down and then up again, particularly at high α 's. What could have caused this? Do you think this always occurs, or might it be a function of how the approximate value function was initialized? \square

Exercise 6.6 In Example 6.2 we stated that the true values for the random walk example are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}$, and $\frac{5}{6}$, for states A through E. Describe at least two different ways that these could have been computed. Which would you guess we actually used? Why? \square

6.3 Optimality of TD(0)

Suppose there is available only a finite amount of experience, say 10 episodes or 100 time steps. In this case, a common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer. Given an approximate value function, V , the increments specified by (6.1) or (6.2) are computed for every time step t at which a nonterminal state is visited, but the value function is changed only once, by the sum of all the increments. Then all the available experience is processed again with the new value function to produce a new overall increment, and so on, until the value function converges. We call this *batch updating* because updates are made only after processing each complete *batch* of training data.

Under batch updating, TD(0) converges deterministically to a single answer independent of the step-size parameter, α , as long as α is chosen to be sufficiently small. The constant- α MC method also converges deterministically under the same conditions, but to a different answer. Understanding these two answers will help us understand the difference between the two methods. Under normal updating the methods do not move all the way to their respective batch answers, but in some sense they take steps in these directions. Before trying to understand the two answers in general, for all possible tasks, we first look at a few examples.

Example 6.3: Random walk under batch updating Batch-updating versions of TD(0) and constant- α MC were applied as follows to the random walk prediction example (Example 6.2). After each new episode, all episodes seen so far were treated as a batch. They were repeatedly presented to the algorithm, either TD(0) or constant- α MC, with α sufficiently small that the value function converged. The resulting value function was then compared with v_π , and the average root mean square error across the five states (and across 100 independent repetitions of the whole experiment) was plotted to obtain

the learning curves shown in Figure 6.2. Note that the batch TD method was consistently better than the batch Monte Carlo method.

Under batch training, constant- α MC converges to values, $V(s)$, that are sample averages of the actual returns experienced after visiting each state s . These are optimal estimates in the sense that they minimize the mean square error from the actual returns in the training set. In this sense it is surprising that the batch TD method was able to perform better according to the root mean square error measure shown in the figure to the right. How is it that batch TD was able to perform better than this optimal method? The answer is that the Monte Carlo method is optimal only in a limited way, and that TD is optimal in a way that is more relevant to predicting returns. ■

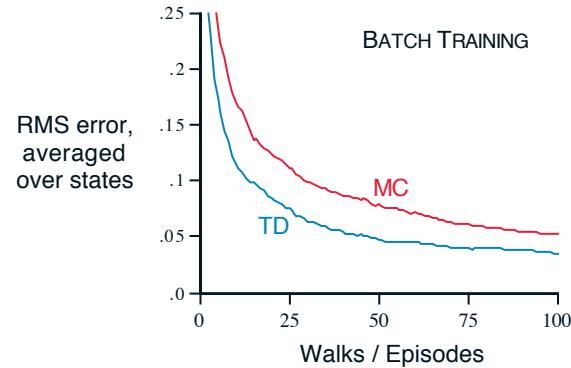


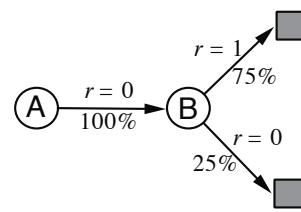
Figure 6.2: Performance of TD(0) and constant- α MC under batch training on the random walk task.

Example 6.4: You are the Predictor Place yourself now in the role of the predictor of returns for an unknown Markov reward process. Suppose you observe the following eight episodes:

A, 0, B, 0	B, 1
B, 1	B, 1
B, 1	B, 1
B, 1	B, 0

This means that the first episode started in state A, transitioned to B with a reward of 0, and then terminated from B with a reward of 0. The other seven episodes were even shorter, starting from B and terminating immediately. Given this batch of data, what would you say are the optimal predictions, the best values for the estimates $V(A)$ and $V(B)$? Everyone would probably agree that the optimal value for $V(B)$ is $\frac{3}{4}$, because six out of the eight times in state B the process terminated immediately with a return of 1, and the other two times in B the process terminated immediately with a return of 0.

But what is the optimal value for the estimate $V(A)$ given this data? Here there are two reasonable answers. One is to observe that 100% of the times the process was in state A it traversed immediately to B (with a reward of 0); and because we have already decided that B has value $\frac{3}{4}$, therefore A must have value $\frac{3}{4}$ as well. One way of viewing this answer is that it is based on first modeling the Markov process, in this case as shown to the right, and then computing the correct estimates given the model, which indeed in this case gives $V(A) = \frac{3}{4}$. This is also the answer that batch TD(0) gives.



The other reasonable answer is simply to observe that we have seen A once and the return that followed it was 0; we therefore estimate $V(A)$ as 0. This is the answer that batch Monte Carlo methods give. Notice that it is also the answer that gives minimum squared error on the training data. In fact, it gives zero error on the data. But still we expect the first answer to be better. If the process is Markov, we expect that the first answer will produce lower error on *future* data, even though the Monte Carlo answer is better on the existing data. ■

Example 6.4 illustrates a general difference between the estimates found by batch TD(0) and batch Monte Carlo methods. Batch Monte Carlo methods always find the estimates that minimize mean square error on the training set, whereas batch TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process. In general, the *maximum-likelihood estimate* of a parameter is the parameter value whose probability of generating the data is greatest. In this case, the maximum-likelihood estimate is the model of the Markov process formed in the obvious way from the observed episodes: the estimated transition probability from i to j is the fraction of observed transitions from i that went to j , and the associated expected reward is the average of the rewards observed on those transitions. Given this model, we can compute the estimate of the value function that would be exactly correct if the model were exactly correct. This is called the *certainty-equivalence estimate* because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated. In general, batch TD(0) converges to the certainty-equivalence estimate.

This helps explain why TD methods converge more quickly than Monte Carlo methods. In batch form, TD(0) is faster than Monte Carlo methods because it computes the true certainty-equivalence estimate. This explains the advantage of TD(0) shown in the batch results on the random walk task (Figure 6.2). The relationship to the certainty-equivalence estimate may also explain in part the speed advantage of nonbatch TD(0) (e.g., Example 6.2, page 125, right graph). Although the nonbatch methods do not achieve either the certainty-equivalence or the minimum squared error estimates, they can be understood as moving roughly in these directions. Nonbatch TD(0) may be faster than constant- α MC because it is moving toward a better estimate, even though it is not getting all the way there. At the current time nothing more definite can be said about the relative efficiency of online TD and Monte Carlo methods.

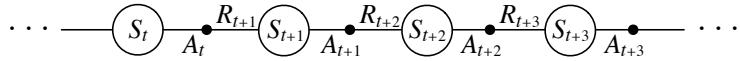
Finally, it is worth noting that although the certainty-equivalence estimate is in some sense an optimal solution, it is almost never feasible to compute it directly. If $n = |\mathcal{S}|$ is the number of states, then just forming the maximum-likelihood estimate of the process may require on the order of n^2 memory, and computing the corresponding value function requires on the order of n^3 computational steps if done conventionally. In these terms it is indeed striking that TD methods can approximate the same solution using memory no more than order n and repeated computations over the training set. On tasks with large state spaces, TD methods may be the only feasible way of approximating the certainty-equivalence solution.

*Exercise 6.7 Design an off-policy version of the TD(0) update that can be used with arbitrary target policy π and covering behavior policy b , using at each step t the importance sampling ratio $\rho_{t:t}$ (5.3). □

6.4 Sarsa: On-policy TD Control

We turn now to the use of TD prediction methods for the control problem. As usual, we follow the pattern of generalized policy iteration (GPI), only this time using TD methods for the evaluation or prediction part. As with Monte Carlo methods, we face the need to trade off exploration and exploitation, and again approaches fall into two main classes: on-policy and off-policy. In this section we present an on-policy TD control method.

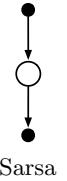
The first step is to learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $q_\pi(s, a)$ for the current behavior policy π and for all states s and actions a . This can be done using essentially the same TD method described above for learning v_π . Recall that an episode consists of an alternating sequence of states and state-action pairs:



In the previous section we considered transitions from state to state and learned the values of states. Now we consider transitions from state-action pair to state-action pair, and learn the values of state-action pairs. Formally these cases are identical: they are both Markov chains with a reward process. The theorems assuring the convergence of state values under TD(0) also apply to the corresponding algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \quad (6.7)$$

This update is done after every transition from a nonterminal state S_t . If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state-action pair to the next. This quintuple gives rise to the name *Sarsa* for the algorithm. The backup diagram for Sarsa is as shown to the right.



It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method. As in all on-policy methods, we continually estimate q_π for the behavior policy π , and at the same time change π toward greediness with respect to q_π . The general form of the Sarsa control algorithm is given in the box on the next page.

The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q . For example, one could use ε -greedy or ε -soft policies. Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be arranged, for example, with ε -greedy policies by setting $\varepsilon = 1/t$).

Exercise 6.8 Show that an action-value version of (6.6) holds for the action-value form of the TD error $\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$, again assuming that the values don't change from step to step. \square

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Choose A from S using policy derived from Q (e.g., ε -greedy)

 Loop for each step of episode:

 Take action A , observe R, S'

 Choose A' from S' using policy derived from Q (e.g., ε -greedy)

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$

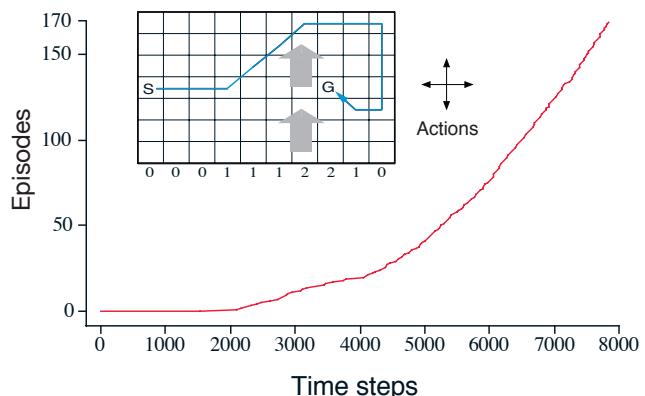
$S \leftarrow S'; A \leftarrow A';$

 until S is terminal

Example 6.5: Windy Gridworld Shown inset below is a standard gridworld, with start and goal states, but with one difference: there is a crosswind running upward through the middle of the grid. The actions are the standard four—**up**, **down**, **right**, and **left**—but in the middle region the resultant next states are shifted upward by a “wind,” the strength of which varies from column to column. The strength of the wind is given below each column, in number of cells shifted upward. For example, if you are one cell to the right of the goal, then the action **left** takes you to the cell just above the goal. This is an undiscounted episodic task, with constant rewards of -1 until the goal state is reached.

The graph to the right shows the results of applying ε -greedy Sarsa to this task, with $\varepsilon = 0.1$, $\alpha = 0.5$, and the initial values $Q(s, a) = 0$ for all s, a . The increasing slope of the graph shows that the goal was reached more quickly over time. By

8000 time steps, the greedy policy was long since optimal (a trajectory from it is shown inset); continued ε -greedy exploration kept the average episode length at about 17 steps, two more than the minimum of 15. Note that Monte Carlo methods cannot easily be used here because termination is not guaranteed for all policies. If a policy was ever found that caused the agent to stay in the same state, then the next episode would never end. Online learning methods such as Sarsa do not have this problem because they quickly learn *during the episode* that such policies are poor, and switch to something else. ■



Exercise 6.9: Windy Gridworld with King's Moves (programming) Re-solve the windy gridworld assuming eight possible actions, including the diagonal moves, rather than four. How much better can you do with the extra actions? Can you do even better by including a ninth action that causes no movement at all other than that caused by the wind? \square

Exercise 6.10: Stochastic Wind (programming) Re-solve the windy gridworld task with King's moves, assuming that the effect of the wind, if there is any, is stochastic, sometimes varying by 1 from the mean values given for each column. That is, a third of the time you move exactly according to these values, as in the previous exercise, but also a third of the time you move one cell above that, and another third of the time you move one cell below that. For example, if you are one cell to the right of the goal and you move `left`, then one-third of the time you move one cell above the goal, one-third of the time you move two cells above the goal, and one-third of the time you move to the goal. \square

6.5 Q-learning: Off-policy TD Control

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as *Q-learning* (Watkins, 1989), defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]. \quad (6.8)$$

In this case, the learned action-value function, Q , directly approximates q_* , the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state-action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. As we observed in Chapter 5, this is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to q_* . The Q-learning algorithm is shown below in procedural form.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Loop for each step of episode:

 Choose A from S using policy derived from Q (e.g., ε -greedy)

 Take action A , observe R, S'

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

$S \leftarrow S'$

 until S is terminal

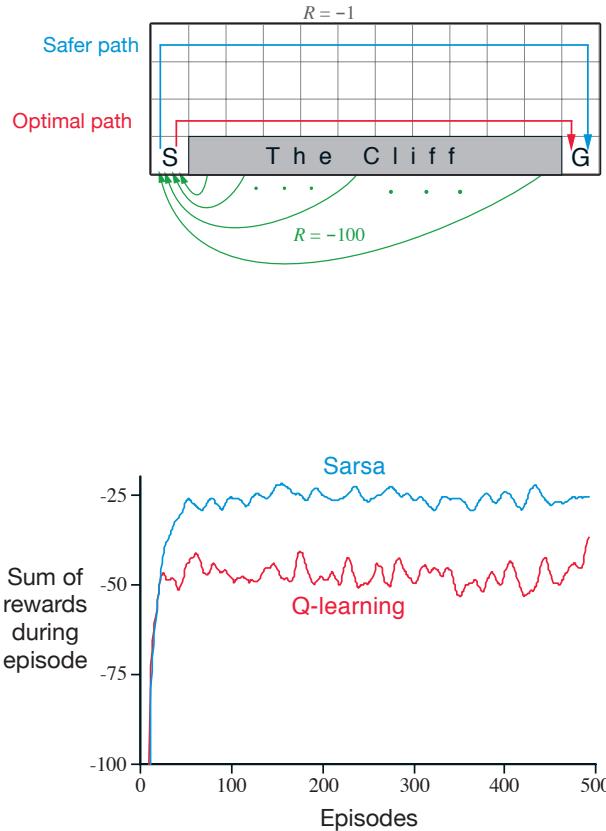
What is the backup diagram for Q-learning? The rule (6.8) updates a state-action pair, so the top node, the root of the update, must be a small, filled action node. The update is also *from* action nodes, maximizing over all those actions possible in the next state. Thus the bottom nodes of the backup diagram should be all these action nodes. Finally, remember that we indicate taking the maximum of these “next action” nodes with an arc across them (Figure 3.4-right). Can you guess now what the diagram is? If so, please do make a guess before turning to the answer in Figure 6.4 on page 134.

Example 6.6: Cliff Walking This gridworld example compares Sarsa and Q-learning, highlighting the difference between on-policy (Sarsa) and off-policy (Q-learning) methods. Consider the gridworld shown to the right. This is a standard undiscounted, episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Reward is -1 on all transitions except those into the region marked “The Cliff.” Stepping into this region incurs a reward of -100 and sends the agent instantly back to the start.

The graph to the right shows the performance of the Sarsa and Q-learning methods with ε -greedy action selection, $\varepsilon = 0.1$. After an initial transient, Q-learning learns values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the ε -greedy action selection. Sarsa, on the other hand, takes the action selection into account and learns the longer but safer path through the upper part of the grid. Although Q-learning actually learns the values of the optimal policy, its online performance is worse than that of Sarsa, which learns the roundabout policy. Of course, if ε were gradually reduced, then both methods would asymptotically converge to the optimal policy. ■

Exercise 6.11 Why is Q-learning considered an *off-policy* control method? □

Exercise 6.12 Suppose action selection is greedy. Is Q-learning then exactly the same algorithm as Sarsa? Will they make exactly the same action selections and weight updates? □



6.6 Expected Sarsa

Consider the learning algorithm that is just like Q-learning except that instead of the maximum over next state-action pairs it uses the expected value, taking into account how likely each action is under the current policy. That is, consider the algorithm with the update rule

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E}_\pi [Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t) \right] \\ &\leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right], \end{aligned} \quad (6.9)$$

but that otherwise follows the schema of Q-learning. Given the next state, S_{t+1} , this algorithm moves *deterministically* in the same direction as Sarsa moves *in expectation*, and accordingly it is called *Expected Sarsa*. Its backup diagram is shown on the right in Figure 6.4.

Expected Sarsa is more complex computationally than Sarsa but, in return, it eliminates the variance due to the random selection of A_{t+1} . Given the same amount of experience we might expect it to perform slightly better than Sarsa, and indeed it generally does. Figure 6.3 shows summary results on the cliff-walking task with Expected Sarsa compared to Sarsa and Q-learning. Expected Sarsa retains the significant advantage of Sarsa over Q-learning on this problem. In addition, Expected Sarsa shows a significant improvement

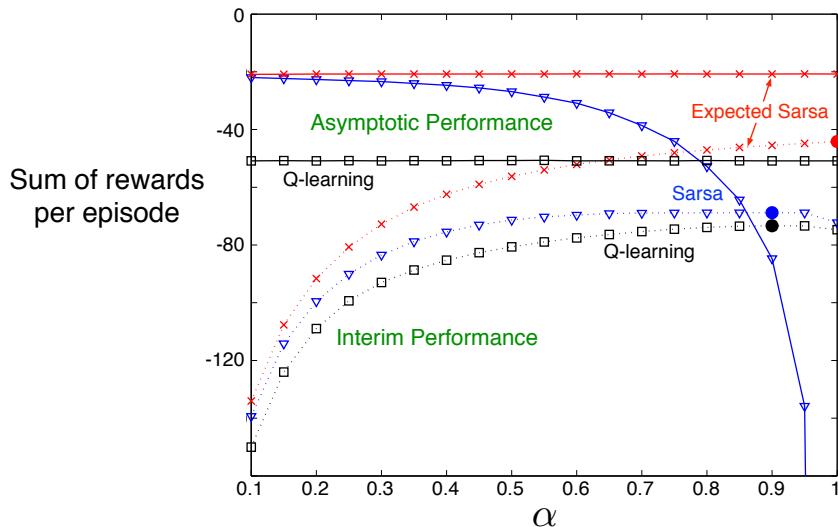


Figure 6.3: Interim and asymptotic performance of TD control methods on the cliff-walking task as a function of α . All algorithms used an ε -greedy policy with $\varepsilon = 0.1$. Asymptotic performance is an average over 100,000 episodes whereas interim performance is an average over the first 100 episodes. These data are averages of over 50,000 and 10 runs for the interim and asymptotic cases respectively. The solid circles mark the best interim performance of each method. Adapted from van Seijen et al. (2009).



Figure 6.4: The backup diagrams for Q-learning and Expected Sarsa.

over Sarsa over a wide range of values for the step-size parameter α . In cliff walking the state transitions are all deterministic and all randomness comes from the policy. In such cases, Expected Sarsa can safely set $\alpha = 1$ without suffering any degradation of asymptotic performance, whereas Sarsa can only perform well in the long run at a small value of α , at which short-term performance is poor. In this and other examples there is a consistent empirical advantage of Expected Sarsa over Sarsa.

In these cliff walking results Expected Sarsa was used on-policy, but in general it might use a policy different from the target policy π to generate behavior, in which case it becomes an off-policy algorithm. For example, suppose π is the greedy policy while behavior is more exploratory; then Expected Sarsa is exactly Q-learning. In this sense Expected Sarsa subsumes and generalizes Sarsa while reliably improving over Sarsa. Except for the small additional computational cost, Expected Sarsa may completely dominate both of the other more-well-known TD control algorithms.

6.7 Maximization Bias and Double Learning

All the control algorithms that we have discussed so far involve maximization in the construction of their target policies. For example, in Q-learning the target policy is the greedy policy given the current action values, which is defined with a max, and in Sarsa the policy is often ε -greedy, which also involves a maximization operation. In these algorithms, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias. To see why, consider a single state s where there are many actions a whose true values, $q(s, a)$, are all zero but whose estimated values, $Q(s, a)$, are uncertain and thus distributed some above and some below zero. The maximum of the true values is zero, but the maximum of the estimates is positive, a positive bias. We call this *maximization bias*.

Example 6.7: Maximization Bias Example The small MDP shown inset in Figure 6.5 provides a simple example of how maximization bias can harm the performance of TD control algorithms. The MDP has two non-terminal states A and B. Episodes always start in A with a choice between two actions, left and right. The right action transitions immediately to the terminal state with a reward and return of zero. The left action transitions to B, also with a reward of zero, from which there are many possible actions all of which cause immediate termination with a reward drawn from a normal distribution with mean -0.1 and variance 1.0. Thus, the expected return for any trajectory starting with left is -0.1 , and thus taking left in state A is always a

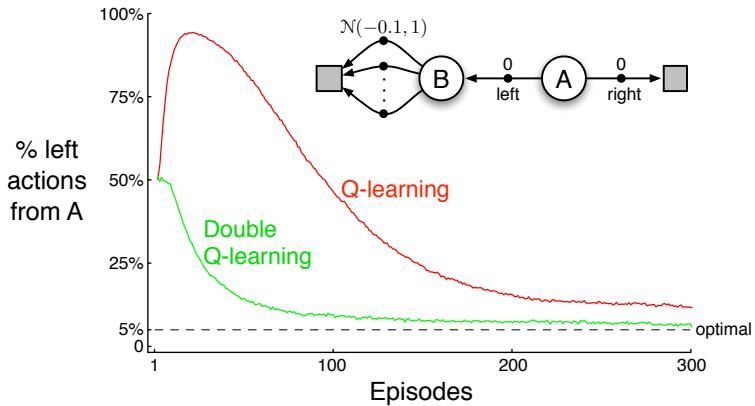


Figure 6.5: Comparison of Q-learning and Double Q-learning on a simple episodic MDP (shown inset). Q-learning initially learns to take the left action much more often than the right action, and always takes it significantly more often than the 5% minimum probability enforced by ε -greedy action selection with $\varepsilon = 0.1$. In contrast, Double Q-learning is essentially unaffected by maximization bias. These data are averaged over 10,000 runs. The initial action-value estimates were zero. Any ties in ε -greedy action selection were broken randomly. ■

mistake. Nevertheless, our control methods may favor left because of maximization bias making B appear to have a positive value. Figure 6.5 shows that Q-learning with ε -greedy action selection initially learns to strongly favor the left action on this example. Even at asymptote, Q-learning takes the left action about 5% more often than is optimal at our parameter settings ($\varepsilon = 0.1$, $\alpha = 0.1$, and $\gamma = 1$). ■

Are there algorithms that avoid maximization bias? To start, consider a bandit case in which we have noisy estimates of the value of each of many actions, obtained as sample averages of the rewards received on all the plays with each action. As we discussed above, there will be a positive maximization bias if we use the maximum of the estimates as an estimate of the maximum of the true values. One way to view the problem is that it is due to using the same samples (plays) both to determine the maximizing action and to estimate its value. Suppose we divided the plays in two sets and used them to learn two independent estimates, call them $Q_1(a)$ and $Q_2(a)$, each an estimate of the true value $q(a)$, for all $a \in \mathcal{A}$. We could then use one estimate, say Q_1 , to determine the maximizing action $A^* = \operatorname{argmax}_a Q_1(a)$, and the other, Q_2 , to provide the estimate of its value, $Q_2(A^*) = Q_2(\operatorname{argmax}_a Q_1(a))$. This estimate will then be unbiased in the sense that $\mathbb{E}[Q_2(A^*)] = q(A^*)$. We can also repeat the process with the role of the two estimates reversed to yield a second unbiased estimate $Q_1(\operatorname{argmax}_a Q_2(a))$. This is the idea of *double learning*. Note that although we learn two estimates, only one estimate is updated on each play; double learning doubles the memory requirements, but does not increase the amount of computation per step.

The idea of double learning extends naturally to algorithms for full MDPs. For example, the double learning algorithm analogous to Q-learning, called Double Q-learning, divides the time steps in two, perhaps by flipping a coin on each step. If the coin comes up heads, the update is

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_2(S_{t+1}, \arg \max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t) \right]. \quad (6.10)$$

If the coin comes up tails, then the same update is done with Q_1 and Q_2 switched, so that Q_2 is updated. The two approximate value functions are treated completely symmetrically. The behavior policy can use both action-value estimates. For example, an ε -greedy policy for Double Q-learning could be based on the average (or sum) of the two action-value estimates. A complete algorithm for Double Q-learning is given in the box below. This is the algorithm used to produce the results in Figure 6.5. In that example, double learning seems to eliminate the harm caused by maximization bias. Of course there are also double versions of Sarsa and Expected Sarsa.

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Loop for each step of episode:

 Choose A from S using the policy ε -greedy in $Q_1 + Q_2$

 Take action A , observe R, S'

 With 0.5 probability:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha \left(R + \gamma Q_2(S', \arg \max_a Q_1(S', a)) - Q_1(S, A) \right)$$

 else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha \left(R + \gamma Q_1(S', \arg \max_a Q_2(S', a)) - Q_2(S, A) \right)$$

$S \leftarrow S'$

 until S is terminal

*Exercise 6.13 What are the update equations for Double Expected Sarsa with an ε -greedy target policy? \square

6.8 Games, Afterstates, and Other Special Cases

In this book we try to present a uniform approach to a wide class of tasks, but of course there are always exceptional tasks that are better treated in a specialized way. For example, our general approach involves learning an *action*-value function, but in Chapter 1 we presented a TD method for learning to play tic-tac-toe that learned something much more like a *state*-value function. If we look closely at that example, it becomes apparent that the function learned there is neither an action-value function nor a state-value function in the usual sense. A conventional state-value function evaluates states in which the agent has the option of selecting an action, but the state-value function used in

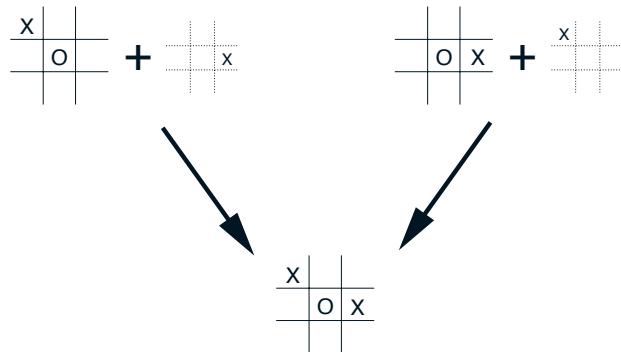
tic-tac-toe evaluates board positions *after* the agent has made its move. Let us call these *afterstates*, and value functions over these, *afterstate value functions*. Afterstates are useful when we have knowledge of an initial part of the environment's dynamics but not necessarily of the full dynamics. For example, in games we typically know the immediate effects of our moves. We know for each possible chess move what the resulting position will be, but not how our opponent will reply. Afterstate value functions are a natural way to take advantage of this kind of knowledge and thereby produce a more efficient learning method.

The reason it is more efficient to design algorithms in terms of afterstates is apparent from the tic-tac-toe example. A conventional action-value function would map from positions *and* moves to an estimate of the value. But many position–move pairs produce the same resulting position, as in the example below: In such cases the position–move pairs are different but produce the same “afterposition,” and thus must have the same value. A conventional action-value function would have to separately assess both pairs, whereas an afterstate value function would immediately assess both equally. Any learning about the position–move pair on the left would immediately transfer to the pair on the right.

Afterstates arise in many tasks, not just games. For example, in queuing tasks there are actions such as assigning customers to servers, rejecting customers, or discarding information. In such cases the actions are in fact defined in terms of their immediate effects, which are completely known.

It is impossible to describe all the possible kinds of specialized problems and corresponding specialized learning algorithms. However, the principles developed in this book should apply widely. For example, afterstate methods are still aptly described in terms of generalized policy iteration, with a policy and (afterstate) value function interacting in essentially the same way. In many cases one will still face the choice between on-policy and off-policy methods for managing the need for persistent exploration.

Exercise 6.14 Describe how the task of Jack’s Car Rental (Example 4.2) could be reformulated in terms of afterstates. Why, in terms of this specific task, would such a reformulation be likely to speed convergence? □



6.9 Summary

In this chapter we introduced a new kind of learning method, temporal-difference (TD) learning, and showed how it can be applied to the reinforcement learning problem. As usual, we divided the overall problem into a prediction problem and a control problem. TD methods are alternatives to Monte Carlo methods for solving the prediction problem. In both cases, the extension to the control problem is via the idea of generalized policy iteration (GPI) that we abstracted from dynamic programming. This is the idea that approximate policy and value functions should interact in such a way that they both move toward their optimal values.

One of the two processes making up GPI drives the value function to accurately predict returns for the current policy; this is the prediction problem. The other process drives the policy to improve locally (e.g., to be ε -greedy) with respect to the current value function. When the first process is based on experience, a complication arises concerning maintaining sufficient exploration. We can classify TD control methods according to whether they deal with this complication by using an on-policy or off-policy approach. Sarsa is an on-policy method, and Q-learning is an off-policy method. Expected Sarsa is also an off-policy method as we present it here. There is a third way in which TD methods can be extended to control which we did not include in this chapter, called actor-critic methods. These methods are covered in full in Chapter 13.

The methods presented in this chapter are today the most widely used reinforcement learning methods. This is probably due to their great simplicity: they can be applied online, with a minimal amount of computation, to experience generated from interaction with an environment; they can be expressed nearly completely by single equations that can be implemented with small computer programs. In the next few chapters we extend these algorithms, making them slightly more complicated and significantly more powerful. All the new algorithms will retain the essence of those introduced here: they will be able to process experience online, with relatively little computation, and they will be driven by TD errors. The special cases of TD methods introduced in the present chapter should rightly be called *one-step, tabular, model-free* TD methods. In the next two chapters we extend them to n -step forms (a link to Monte Carlo methods) and forms that include a model of the environment (a link to planning and dynamic programming). Then, in the second part of the book we extend them to various forms of function approximation rather than tables (a link to deep learning and artificial neural networks).

Finally, in this chapter we have discussed TD methods entirely within the context of reinforcement learning problems, but TD methods are actually more general than this. They are general methods for learning to make long-term predictions about dynamical systems. For example, TD methods may be relevant to predicting financial data, life spans, election outcomes, weather patterns, animal behavior, demands on power stations, or customer purchases. It was only when TD methods were analyzed as pure prediction methods, independent of their use in reinforcement learning, that their theoretical properties first came to be well understood. Even so, these other potential applications of TD learning methods have not yet been extensively explored.

Bibliographical and Historical Remarks

As we outlined in Chapter 1, the idea of TD learning has its early roots in animal learning psychology and artificial intelligence, most notably the work of Samuel (1959) and Klopff (1972). Samuel’s work is described as a case study in Section 16.2. Also related to TD learning are Holland’s (1975, 1976) early ideas about consistency among value predictions. These influenced one of the authors (Barto), who was a graduate student from 1970 to 1975 at the University of Michigan, where Holland was teaching. Holland’s ideas led to a number of TD-related systems, including the work of Booker (1982) and the bucket brigade of Holland (1986), which is related to Sarsa as discussed below.

- 6.1–2** Most of the specific material from these sections is from Sutton (1988), including the TD(0) algorithm, the random walk example, and the term “temporal-difference learning.” The characterization of the relationship to dynamic programming and Monte Carlo methods was influenced by Watkins (1989), Werbos (1987), and others. The use of backup diagrams was new to the first edition of this book.

Tabular TD(0) was proved to converge in the mean by Sutton (1988) and with probability 1 by Dayan (1992), based on the work of Watkins and Dayan (1992). These results were extended and strengthened by Jaakkola, Jordan, and Singh (1994) and Tsitsiklis (1994) by using extensions of the powerful existing theory of stochastic approximation. Other extensions and generalizations are covered in later chapters.

- 6.3** The optimality of the TD algorithm under batch training was established by Sutton (1988). Illuminating this result is Barnard’s (1993) derivation of the TD algorithm as a combination of one step of an incremental method for learning a model of the Markov chain and one step of a method for computing predictions from the model. The term *certainty equivalence* is from the adaptive control literature (e.g., Goodwin and Sin, 1984).

- 6.4** The Sarsa algorithm was introduced by Rummery and Niranjan (1994). They explored it in conjunction with artificial neural networks and called it “Modified Connectionist Q-learning”. The name “Sarsa” was introduced by Sutton (1996). The convergence of one-step tabular Sarsa (the form treated in this chapter) has been proved by Singh, Jaakkola, Littman, and Szepesvári (2000). The “windy gridworld” example was suggested by Tom Kalt.

Holland’s (1986) bucket brigade idea evolved into an algorithm closely related to Sarsa. The original idea of the bucket brigade involved chains of rules triggering each other; it focused on passing credit back from the current rule to the rules that triggered it. Over time, the bucket brigade came to be more like TD learning in passing credit back to any temporally preceding rule, not just to the ones that triggered the current rule. The modern form of the bucket brigade, when simplified in various natural ways, is nearly identical to one-step Sarsa, as detailed by Wilson (1994).

- 6.5** Q-learning was introduced by Watkins (1989), whose outline of a convergence proof was made rigorous by Watkins and Dayan (1992). More general convergence results were proved by Jaakkola, Jordan, and Singh (1994) and Tsitsiklis (1994).
- 6.6** The Expected Sarsa algorithm was introduced by George John (1994), who called it “ \bar{Q} -learning” and stressed its advantages over Q-learning as an off-policy algorithm. John’s work was not known to us when we presented Expected Sarsa in the first edition of this book as an exercise, or to van Seijen, van Hasselt, Whiteson, and Weiring (2009) when they established Expected Sarsa’s convergence properties and conditions under which it will outperform regular Sarsa and Q-learning. Our Figure 6.3 is adapted from their results. Van Seijen et al. defined “Expected Sarsa” to be an on-policy method exclusively (as we did in the first edition), whereas now we use this name for the general algorithm in which the target and behavior policies may differ. The general off-policy view of Expected Sarsa was noted by van Hasselt (2011), who called it “General Q-learning.”
- 6.7** Maximization bias and double learning were introduced and extensively investigated by van Hasselt (2010, 2011). The example MDP in Figure 6.5 was adapted from that in his Figure 4.1 (van Hasselt, 2011).
- 6.8** The notion of an afterstate is the same as that of a “post-decision state” (Van Roy, Bertsekas, Lee, and Tsitsiklis, 1997; Powell, 2011).

Chapter 7

n-step Bootstrapping

In this chapter we unify the Monte Carlo (MC) methods and the one-step temporal-difference (TD) methods presented in the previous two chapters. Neither MC methods nor one-step TD methods are always the best. In this chapter we present *n-step TD methods* that generalize both methods so that one can shift from one to the other smoothly as needed to meet the demands of a particular task. *n*-step methods span a spectrum with MC methods at one end and one-step TD methods at the other. The best methods are often intermediate between the two extremes.

Another way of looking at the benefits of *n*-step methods is that they free you from the tyranny of the time step. With one-step TD methods the same time step determines how often the action can be changed and the time interval over which bootstrapping is done. In many applications one wants to be able to update the action very fast to take into account anything that has changed, but bootstrapping works best if it is over a length of time in which a significant and recognizable state change has occurred. With one-step TD methods, these time intervals are the same, and so a compromise must be made. *n*-step methods enable bootstrapping to occur over multiple steps, freeing us from the tyranny of the single time step.

The idea of *n*-step methods is usually used as an introduction to the algorithmic idea of *eligibility traces* (Chapter 12), which enable bootstrapping over multiple time intervals simultaneously. Here we instead consider the *n*-step bootstrapping idea on its own, postponing the treatment of eligibility-trace mechanisms until later. This allows us to separate the issues better, dealing with as many of them as possible in the simpler *n*-step setting.

As usual, we first consider the prediction problem and then the control problem. That is, we first consider how *n*-step methods can help in predicting returns as a function of state for a fixed policy (i.e., in estimating v_π). Then we extend the ideas to action values and control methods.

7.1 n -step TD Prediction

What is the space of methods lying between Monte Carlo and TD methods? Consider estimating v_π from sample episodes generated using π . Monte Carlo methods perform an update for each state based on the entire sequence of observed rewards from that state until the end of the episode. The update of one-step TD methods, on the other hand, is based on just the one next reward, bootstrapping from the value of the state one step later as a proxy for the remaining rewards. One kind of intermediate method, then, would perform an update based on an intermediate number of rewards: more than one, but less than all of them until termination. For example, a two-step update would be based on the first two rewards and the estimated value of the state two steps later. Similarly, we could have three-step updates, four-step updates, and so on. Figure 7.1 shows the backup diagrams of the spectrum of n -step updates for v_π , with the one-step TD update on the left and the up-until-termination Monte Carlo update on the right.

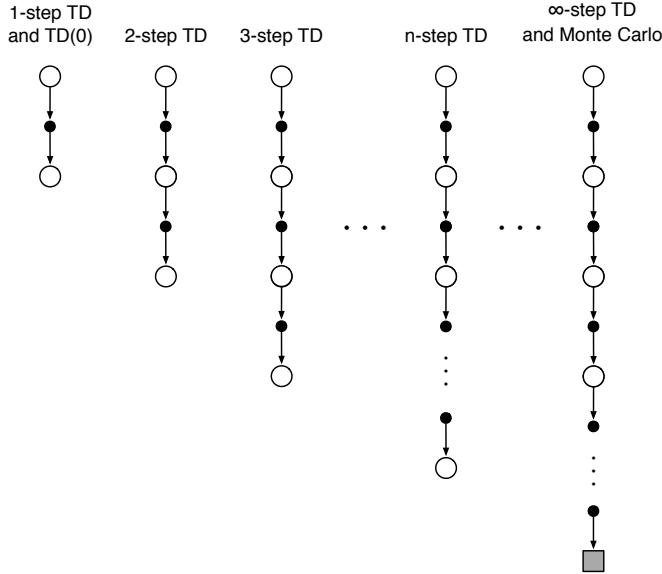


Figure 7.1: The backup diagrams of n -step methods. These methods form a spectrum ranging from one-step TD methods to Monte Carlo methods.

The methods that use n -step updates are still TD methods because they still change an earlier estimate based on how it differs from a later estimate. Now the later estimate is not one step later, but n steps later. Methods in which the temporal difference extends over n steps are called *n -step TD methods*. The TD methods introduced in the previous chapter all used one-step updates, which is why we called them one-step TD methods.

More formally, consider the update of the estimated value of state S_t as a result of the state-reward sequence, $S_t, R_{t+1}, S_{t+1}, R_{t+2}, \dots, R_T, S_T$ (omitting the actions). We know that in Monte Carlo updates the estimate of $v_\pi(S_t)$ is updated in the direction of the

complete return:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T,$$

where T is the last time step of the episode. Let us call this quantity the *target* of the update. Whereas in Monte Carlo updates the target is the return, in one-step updates the target is the first reward plus the discounted estimated value of the next state, which we call the *one-step return*:

$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1}),$$

where $V_t : \mathcal{S} \rightarrow \mathbb{R}$ here is the estimate at time t of v_π . The subscripts on $G_{t:t+1}$ indicate that it is a truncated return for time t using rewards up until time $t+1$, with the discounted estimate $\gamma V_t(S_{t+1})$ taking the place of the other terms $\gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$ of the full return, as discussed in the previous chapter. Our point now is that this idea makes just as much sense after two steps as it does after one. The target for a two-step update is the *two-step return*:

$$G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2}),$$

where now $\gamma^2 V_{t+1}(S_{t+2})$ corrects for the absence of the terms $\gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots + \gamma^{T-t-1} R_T$. Similarly, the target for an arbitrary n -step update is the *n-step return*:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}), \quad (7.1)$$

for all n, t such that $n \geq 1$ and $0 \leq t < T - n$. All n -step returns can be considered approximations to the full return, truncated after n steps and then corrected for the remaining missing terms by $V_{t+n-1}(S_{t+n})$. If $t + n \geq T$ (if the n -step return extends to or beyond termination), then all the missing terms are taken as zero, and the n -step return defined to be equal to the ordinary full return ($G_{t:t+n} \doteq G_t$ if $t + n \geq T$).

Note that n -step returns for $n > 1$ involve future rewards and states that are not available at the time of transition from t to $t + 1$. No real algorithm can use the n -step return until after it has seen R_{t+n} and computed V_{t+n-1} . The first time these are available is $t + n$. The natural state-value learning algorithm for using n -step returns is thus

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha [G_{t:t+n} - V_{t+n-1}(S_t)], \quad 0 \leq t < T, \quad (7.2)$$

while the values of all other states remain unchanged: $V_{t+n}(s) = V_{t+n-1}(s)$, for all $s \neq S_t$. We call this algorithm *n-step TD*. Note that no changes at all are made during the first $n - 1$ steps of each episode. To make up for that, an equal number of additional updates are made at the end of the episode, after termination and before starting the next episode. Complete pseudocode is given in the box on the next page.

Exercise 7.1 In Chapter 6 we noted that the Monte Carlo error can be written as the sum of TD errors (6.6) if the value estimates don't change from step to step. Show that the n -step error used in (7.2) can also be written as a sum TD errors (again if the value estimates don't change) generalizing the earlier result. \square

Exercise 7.2 (programming) With an n -step method, the value estimates *do* change from step to step, so an algorithm that used the sum of TD errors (see previous exercise) in

***n*-step TD for estimating $V \approx v_\pi$**

Input: a policy π

Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n

Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}$

All store and access operations (for S_t and R_t) can take their index mod $n + 1$

Loop for each episode:

 Initialize and store $S_0 \neq$ terminal

$T \leftarrow \infty$

 Loop for $t = 0, 1, 2, \dots$:

 If $t < T$, then:

 Take an action according to $\pi(\cdot | S_t)$

 Observe and store the next reward as R_{t+1} and the next state as S_{t+1}

 If S_{t+1} is terminal, then $T \leftarrow t + 1$

$\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)

 If $\tau \geq 0$:

$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$

 If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_{\tau:\tau+n})$

$V(S_\tau) \leftarrow V(S_\tau) + \alpha [G - V(S_\tau)]$

 Until $\tau = T - 1$

place of the error in (7.2) would actually be a slightly different algorithm. Would it be a better algorithm or a worse one? Devise and program a small experiment to answer this question empirically. \square

The n -step return uses the value function V_{t+n-1} to correct for the missing rewards beyond R_{t+n} . An important property of n -step returns is that their expectation is guaranteed to be a better estimate of v_π than V_{t+n-1} is, in a worst-state sense. That is, the worst error of the expected n -step return is guaranteed to be less than or equal to γ^n times the worst error under V_{t+n-1} :

$$\max_s |\mathbb{E}_\pi[G_{t:t+n} | S_t = s] - v_\pi(s)| \leq \gamma^n \max_s |V_{t+n-1}(s) - v_\pi(s)|, \quad (7.3)$$

for all $n \geq 1$. This is called the *error reduction property* of n -step returns. Because of the error reduction property, one can show formally that all n -step TD methods converge to the correct predictions under appropriate technical conditions. The n -step TD methods thus form a family of sound methods, with one-step TD methods and Monte Carlo methods as extreme members.

Example 7.1: *n*-step TD Methods on the Random Walk Consider using n -step TD methods on the 5-state random walk task described in Example 6.2 (page 125). Suppose the first episode progressed directly from the center state, C, to the right, through D and E, and then terminated on the right with a return of 1. Recall that the estimated values of all the states started at an intermediate value, $V(s) = 0.5$. As a result of this experience, a one-step method would change only the estimate for the last state,

$V(E)$, which would be incremented toward 1, the observed return. A two-step method, on the other hand, would increment the values of the two states preceding termination: $V(D)$ and $V(E)$ both would be incremented toward 1. A three-step method, or any n -step method for $n > 2$, would increment the values of all three of the visited states toward 1, all by the same amount.

Which value of n is better? Figure 7.2 shows the results of a simple empirical test for a larger random walk process, with 19 states instead of 5 (and with a -1 outcome on the left, all values initialized to 0), which we use as a running example in this chapter. Results are shown for n -step TD methods with a range of values for n and α . The performance measure for each parameter setting, shown on the vertical axis, is the square-root of the average squared error between the predictions at the end of the episode for the 19 states and their true values, then averaged over the first 10 episodes and 100 repetitions of the whole experiment (the same sets of walks were used for all parameter settings). Note that methods with an intermediate value of n worked best. This illustrates how the generalization of TD and Monte Carlo methods to n -step methods can potentially perform better than either of the two extreme methods.

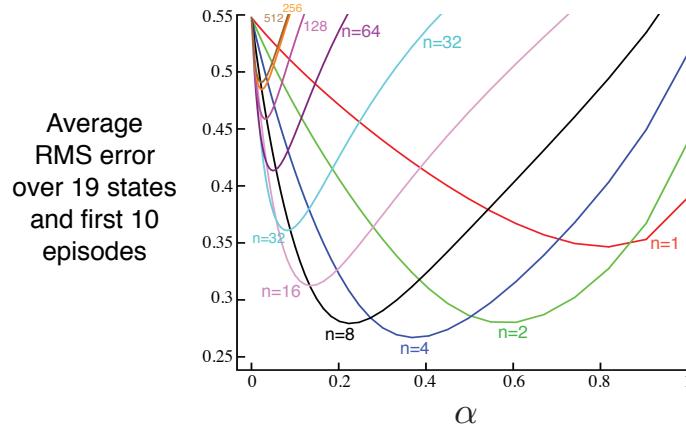


Figure 7.2: Performance of n -step TD methods as a function of α , for various values of n , on a 19-state random walk task (Example 7.1). ■

Exercise 7.3 Why do you think a larger random walk task (19 states instead of 5) was used in the examples of this chapter? Would a smaller walk have shifted the advantage to a different value of n ? How about the change in left-side outcome from 0 to -1 made in the larger walk? Do you think that made any difference in the best value of n ? □

7.2 *n*-step Sarsa

How can n -step methods be used not just for prediction, but for control? In this section we show how n -step methods can be combined with Sarsa in a straightforward way to

produce an on-policy TD control method. The n -step version of Sarsa we call n -step Sarsa, and the original version presented in the previous chapter we henceforth call *one-step Sarsa*, or *Sarsa(0)*.

The main idea is to simply switch states for actions (state-action pairs) and then use an ε -greedy policy. The backup diagrams for n -step Sarsa (shown in Figure 7.3), like those of n -step TD (Figure 7.1), are strings of alternating states and actions, except that the Sarsa ones all start and end with an action rather than a state. We redefine n -step returns (update targets) in terms of estimated action values:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}), \quad n \geq 1, 0 \leq t < T-n, \quad (7.4)$$

with $G_{t:t+n} \doteq G_t$ if $t+n \geq T$. The natural algorithm is then

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], \quad 0 \leq t < T, \quad (7.5)$$

while the values of all other states remain unchanged: $Q_{t+n}(s, a) = Q_{t+n-1}(s, a)$, for all s, a such that $s \neq S_t$ or $a \neq A_t$. This is the algorithm we call n -step Sarsa. Pseudocode is shown in the box on the next page, and an example of why it can speed up learning compared to one-step methods is given in Figure 7.4.

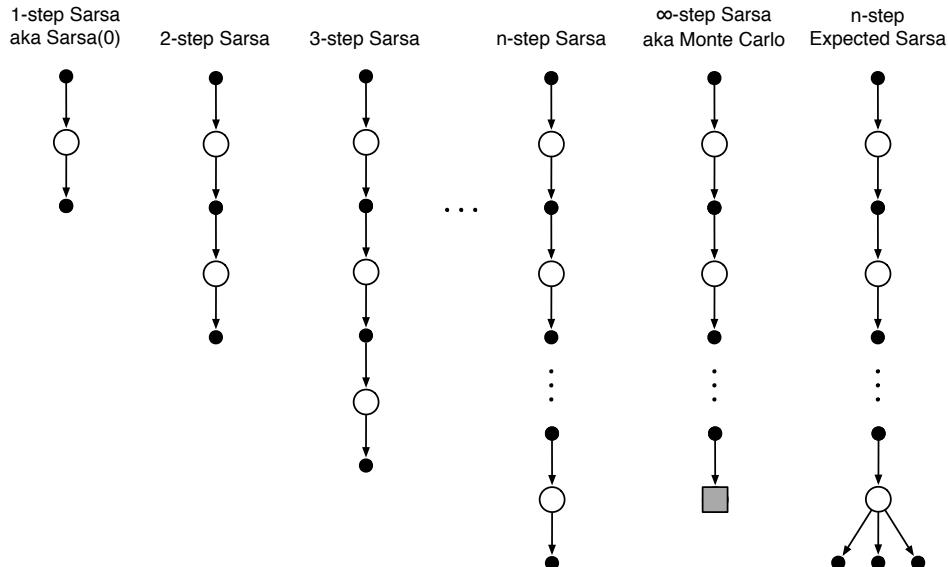


Figure 7.3: The backup diagrams for the spectrum of n -step methods for state-action values. They range from the one-step update of Sarsa(0) to the up-until-termination update of the Monte Carlo method. In between are the n -step updates, based on n steps of real rewards and the estimated value of the n th next state-action pair, all appropriately discounted. On the far right is the backup diagram for n -step Expected Sarsa.

***n*-step Sarsa for estimating $Q \approx q_*$ or q_π**

Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$

Initialize π to be ε -greedy with respect to Q , or to a fixed given policy

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$, a positive integer n

All store and access operations (for S_t , A_t , and R_t) can take their index mod $n + 1$

Loop for each episode:

Initialize and store $S_0 \neq \text{terminal}$

Select and store an action $A_0 \sim \pi(\cdot | S_0)$

$$T \leftarrow \infty$$

Loop for $t = 0, 1, 2, \dots$:

If $t \leq T$, then:

Take action A_t

Observe and store the ne

S_{t+1} is ter

7

Select and store an action $A_{t+1} \sim \pi(\cdot | S_{t+1})$

$$\tau \leftarrow t -$$

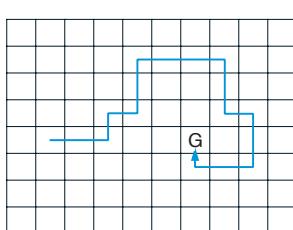
$$\tau \geq 0: \\ G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$$

If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$

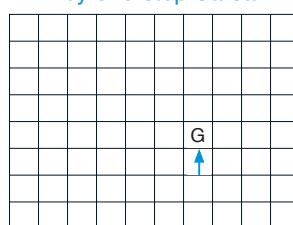
If π is being learned, then ensure that $\pi(\cdot | S_t)$ is ϵ -greedy wrt Q

Until $\pi = T - 1$

Path taken



Action values increased by one-step Sarsa



Action values increased by 10-step Sarsa

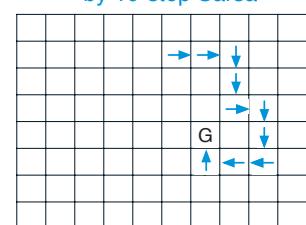


Figure 7.4: Gridworld example of the speedup of policy learning due to the use of n -step methods. The first panel shows the path taken by an agent in a single episode, ending at a location of high reward, marked by the G. In this example the values were all initially 0, and all rewards were zero except for a positive reward at G. The arrows in the other two panels show which action values were strengthened as a result of this path by one-step and n -step Sarsa methods. The one-step method strengthens only the last action of the sequence of actions that led to the high reward, whereas the n -step method strengthens the last n actions of the sequence, so that much more is learned from the one episode.

Exercise 7.4 Prove that the *n*-step return of Sarsa (7.4) can be written exactly in terms of a novel TD error, as

$$G_{t:t+n} = Q_{t-1}(S_t, A_t) + \sum_{k=t}^{\min(t+n, T)-1} \gamma^{k-t} [R_{k+1} + \gamma Q_k(S_{k+1}, A_{k+1}) - Q_{k-1}(S_k, A_k)]. \quad (7.6)$$

□

What about Expected Sarsa? The backup diagram for the *n*-step version of Expected Sarsa is shown on the far right in Figure 7.3. It consists of a linear string of sample actions and states, just as in *n*-step Sarsa, except that its last element is a branch over all action possibilities weighted, as always, by their probability under π . This algorithm can be described by the same equation as *n*-step Sarsa (above) except with the *n*-step return redefined as

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \bar{V}_{t+n-1}(S_{t+n}), \quad t+n < T, \quad (7.7)$$

(with $G_{t:t+n} \doteq G_t$ for $t+n \geq T$) where $\bar{V}_t(s)$ is the *expected approximate value* of state s , using the estimated action values at time t , under the target policy:

$$\bar{V}_t(s) \doteq \sum_a \pi(a|s) Q_t(s, a), \quad \text{for all } s \in \mathcal{S}. \quad (7.8)$$

Expected approximate values are used in developing many of the action-value methods in the rest of this book. If s is terminal, then its expected approximate value is defined to be 0.

7.3 *n*-step Off-policy Learning

Recall that off-policy learning is learning the value function for one policy, π , while following another policy, b . Often, π is the greedy policy for the current action-value-function estimate, and b is a more exploratory policy, perhaps ε -greedy. In order to use the data from b we must take into account the difference between the two policies, using their relative probability of taking the actions that were taken (see Section 5.5). In *n*-step methods, returns are constructed over *n* steps, so we are interested in the relative probability of just those *n* actions. For example, to make a simple off-policy version of *n*-step TD, the update for time t (actually made at time $t+n$) can simply be weighted by $\rho_{t:t+n-1}$:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha \rho_{t:t+n-1} [G_{t:t+n} - V_{t+n-1}(S_t)], \quad 0 \leq t < T, \quad (7.9)$$

where $\rho_{t:t+n-1}$, called the *importance sampling ratio*, is the relative probability under the two policies of taking the *n* actions from A_t to A_{t+n-1} (cf. Eq. 5.3):

$$\rho_{t:h} \doteq \prod_{k=t}^{\min(h, T)-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \quad (7.10)$$

For example, if any one of the actions would never be taken by π (i.e., $\pi(A_k|S_k) = 0$) then the n -step return should be given zero weight and be totally ignored. On the other hand, if by chance an action is taken that π would take with much greater probability than b does, then this will increase the weight that would otherwise be given to the return. This makes sense because that action is characteristic of π (and therefore we want to learn about it) but is selected only rarely by b and thus rarely appears in the data. To make up for this we have to over-weight it when it does occur. Note that if the two policies are actually the same (the on-policy case) then the importance sampling ratio is always 1. Thus our new update (7.9) generalizes and can completely replace our earlier n -step TD update. Similarly, our previous n -step Sarsa update can be completely replaced by a simple off-policy form:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha \rho_{t+1:t+n} [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], \quad (7.11)$$

for $0 \leq t < T$. Note that the importance sampling ratio here starts and ends one step later than for n -step TD (7.9). This is because here we are updating a state-action pair. We do not have to care how likely we were to select the action; now that we have selected it we want to learn fully from what happens, with importance sampling only for subsequent actions. Pseudocode for the full algorithm is shown in the box below.

Off-policy n -step Sarsa for estimating $Q \approx q_*$ or q_π

```

Input: an arbitrary behavior policy  $b$  such that  $b(a|s) > 0$ , for all  $s \in \mathcal{S}, a \in \mathcal{A}$ 
Initialize  $Q(s, a)$  arbitrarily, for all  $s \in \mathcal{S}, a \in \mathcal{A}$ 
Initialize  $\pi$  to be greedy with respect to  $Q$ , or as a fixed given policy
Algorithm parameters: step size  $\alpha \in (0, 1]$ , a positive integer  $n$ 
All store and access operations (for  $S_t$ ,  $A_t$ , and  $R_t$ ) can take their index mod  $n + 1$ 

Loop for each episode:
  Initialize and store  $S_0 \neq$  terminal
  Select and store an action  $A_0 \sim b(\cdot|S_0)$ 
   $T \leftarrow \infty$ 
  Loop for  $t = 0, 1, 2, \dots$  :
    If  $t < T$ , then:
      Take action  $A_t$ 
      Observe and store the next reward as  $R_{t+1}$  and the next state as  $S_{t+1}$ 
      If  $S_{t+1}$  is terminal, then:
         $T \leftarrow t + 1$ 
      else:
        Select and store an action  $A_{t+1} \sim b(\cdot|S_{t+1})$ 
       $\tau \leftarrow t - n + 1$    ( $\tau$  is the time whose estimate is being updated)
      If  $\tau \geq 0$ :
         $\rho \leftarrow \prod_{i=\tau+1}^{\min(\tau+n, T-1)} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$                                 $(\rho_{\tau+1:t+n})$ 
         $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$ 
        If  $\tau + n < T$ , then:  $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$                           $(G_{\tau:\tau+n})$ 
         $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha \rho [G - Q(S_\tau, A_\tau)]$ 
        If  $\pi$  is being learned, then ensure that  $\pi(\cdot|S_\tau)$  is greedy wrt  $Q$ 
    Until  $\tau = T - 1$ 

```

The off-policy version of *n*-step Expected Sarsa would use the same update as above for *n*-step Sarsa except that the importance sampling ratio would have one less factor in it. That is, the above equation would use $\rho_{t+1:t+n-1}$ instead of $\rho_{t+1:t+n}$, and of course it would use the Expected Sarsa version of the *n*-step return (7.7). This is because in Expected Sarsa all possible actions are taken into account in the last state; the one actually taken has no effect and does not have to be corrected for.

7.4 *Per-decision Methods with Control Variates

The multi-step off-policy methods presented in the previous section are simple and conceptually clear, but are probably not the most efficient. A more sophisticated approach would use per-decision importance sampling ideas such as were introduced in Section 5.9. To understand this approach, first note that the ordinary *n*-step return (7.1), like all returns, can be written recursively. For the *n* steps ending at horizon *h*, the *n*-step return can be written

$$G_{t:h} = R_{t+1} + \gamma G_{t+1:h}, \quad t < h < T, \quad (7.12)$$

where $G_{h:h} \doteq V_{h-1}(S_h)$. (Recall that this return is used at time *h*, previously denoted *t* + *n*.) Now consider the effect of following a behavior policy *b* that is not the same as the target policy π . All of the resulting experience, including the first reward R_{t+1} and the next state S_{t+1} , must be weighted by the importance sampling ratio for time *t*, $\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$. One might be tempted to simply weight the righthand side of the above equation, but one can do better. Suppose the action at time *t* would never be selected by π , so that ρ_t is zero. Then a simple weighting would result in the *n*-step return being zero, which could result in high variance when it was used as a target. Instead, in this more sophisticated approach, one uses an alternate, *off-policy* definition of the *n*-step return ending at horizon *h*, as

$$G_{t:h} \doteq \rho_t (R_{t+1} + \gamma G_{t+1:h}) + (1 - \rho_t) V_{h-1}(S_t), \quad t < h < T, \quad (7.13)$$

where again $G_{h:h} \doteq V_{h-1}(S_h)$. In this approach, if ρ_t is zero, then instead of the target being zero and causing the estimate to shrink, the target is the same as the estimate and causes no change. The importance sampling ratio being zero means we should ignore the sample, so leaving the estimate unchanged seems appropriate. The second, additional term in (7.13) is called a *control variate* (for obscure reasons). Notice that the control variate does not change the expected update; the importance sampling ratio has expected value one (Section 5.9) and is uncorrelated with the estimate, so the expected value of the control variate is zero. Also note that the off-policy definition (7.13) is a strict generalization of the earlier on-policy definition of the *n*-step return (7.1), as the two are identical in the on-policy case, in which ρ_t is always 1.

For a conventional *n*-step method, the learning rule to use in conjunction with (7.13) is the *n*-step TD update (7.2), which has no explicit importance sampling ratios other than those embedded in the return.

Exercise 7.5 Write the pseudocode for the off-policy state-value prediction algorithm described above. \square

For action values, the off-policy definition of the n -step return is a little different because the first action does not play a role in the importance sampling. That first action is the one being learned; it does not matter if it was unlikely or even impossible under the target policy—it has been taken and now full unit weight must be given to the reward and state that follows it. Importance sampling will apply only to the actions that follow it.

First note that for action values the n -step *on-policy* return ending at horizon h , expectation form (7.7), can be written recursively just as in (7.12), except that for action values the recursion ends with $G_{h:h} \doteq \bar{V}_{h-1}(S_h)$ as in (7.8). An off-policy form with control variates is

$$\begin{aligned} G_{t:h} &\doteq R_{t+1} + \gamma \left(\rho_{t+1} G_{t+1:h} + \bar{V}_{h-1}(S_{t+1}) - \rho_{t+1} Q_{h-1}(S_{t+1}, A_{t+1}) \right), \\ &= R_{t+1} + \gamma \rho_{t+1} \left(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1}) \right) + \gamma \bar{V}_{h-1}(S_{t+1}), \quad t < h \leq T. \end{aligned} \tag{7.14}$$

If $h < T$, then the recursion ends with $G_{h:h} \doteq Q_{h-1}(S_h, A_h)$, whereas, if $h \geq T$, the recursion ends with $G_{T-1:h} \doteq R_T$. The resultant prediction algorithm (after combining with (7.5)) is analogous to Expected Sarsa.

Exercise 7.6 Prove that the control variate in the above equations does not change the expected value of the return. \square

**Exercise 7.7* Write the pseudocode for the off-policy action-value prediction algorithm described immediately above. Pay particular attention to the termination conditions for the recursion upon hitting the horizon or the end of episode. \square

Exercise 7.8 Show that the general (off-policy) version of the n -step return (7.13) can still be written exactly and compactly as the sum of state-based TD errors (6.5) if the approximate state value function does not change. \square

Exercise 7.9 Repeat the above exercise for the action version of the off-policy n -step return (7.14) and the Expected Sarsa TD error (the quantity in brackets in Equation 6.9). \square

Exercise 7.10 (programming) Devise a small off-policy prediction problem and use it to show that the off-policy learning algorithm using (7.13) and (7.2) is more data efficient than the simpler algorithm using (7.1) and (7.9). \square

The importance sampling that we have used in this section, the previous section, and in Chapter 5, enables sound off-policy learning, but also results in high variance updates, forcing the use of a small step-size parameter and thereby causing learning to be slow. It is probably inevitable that off-policy training is slower than on-policy training—after all, the data is less relevant to what is being learned. However, it is probably also true that these methods can be improved on. The control variates are one way of reducing the variance. Another is to rapidly adapt the step sizes to the observed variance, as in the Autostep method (Mahmood, Sutton, Degris and Pilarski, 2012). Yet another promising approach is the invariant updates of Karampatziakis and Langford (2010) as extended to TD by Tian (in preparation). The usage technique of Mahmood (2017; Mahmood

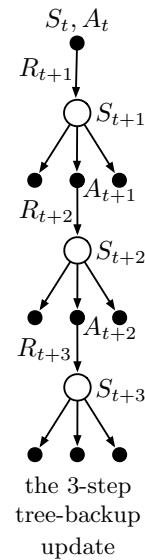
and Sutton, 2015) may also be part of the solution. In the next section we consider an off-policy learning method that does not use importance sampling.

7.5 Off-policy Learning Without Importance Sampling: The *n*-step Tree Backup Algorithm

Is off-policy learning possible without importance sampling? Q-learning and Expected Sarsa from Chapter 6 do this for the one-step case, but is there a corresponding multi-step algorithm? In this section we present just such an *n*-step method, called the *tree-backup algorithm*.

The idea of the algorithm is suggested by the 3-step tree-backup backup diagram shown to the right. Down the central spine and labeled in the diagram are three sample states and rewards, and two sample actions. These are the random variables representing the events occurring after the initial state-action pair S_t, A_t . Hanging off to the sides of each state are the actions that were *not* selected. (For the last state, all the actions are considered to have not (yet) been selected.) Because we have no sample data for the unselected actions, we bootstrap and use the estimates of their values in forming the target for the update. This slightly extends the idea of a backup diagram. So far we have always updated the estimated value of the node at the top of the diagram toward a target combining the rewards along the way (appropriately discounted) and the estimated values of the nodes at the bottom. In the tree-backup update, the target includes all these things *plus* the estimated values of the dangling action nodes hanging off the sides, at all levels. This is why it is called a *tree-backup* update; it is an update from the entire tree of estimated action values.

More precisely, the update is from the estimated action values of the *leaf nodes* of the tree. The action nodes in the interior, corresponding to the actual actions taken, do not participate. Each leaf node contributes to the target with a weight proportional to its probability of occurring under the target policy π . Thus each first-level action a contributes with a weight of $\pi(a|S_{t+1})$, except that the action actually taken, A_{t+1} , does not contribute at all. Its probability, $\pi(A_{t+1}|S_{t+1})$, is used to weight all the second-level action values. Thus, each non-selected second-level action a' contributes with weight $\pi(A_{t+1}|S_{t+1})\pi(a'|S_{t+2})$. Each third-level action contributes with weight $\pi(A_{t+1}|S_{t+1})\pi(A_{t+2}|S_{t+2})\pi(a''|S_{t+3})$, and so on. It is as if each arrow to an action node in the diagram is weighted by the action's probability of being selected under the target policy and, if there is a tree below the action, then that weight applies to all the leaf nodes in the tree.



We can think of the 3-step tree-backup update as consisting of 6 half-steps, alternating between sample half-steps from an action to a subsequent state, and expected half-steps considering from that state all possible actions with their probabilities of occurring under the policy.

Now let us develop the detailed equations for the n -step tree-backup algorithm. The one-step return (target) is the same as that of Expected Sarsa,

$$G_{t:t+1} \doteq R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q_t(S_{t+1}, a), \quad (7.15)$$

for $t < T - 1$, and the two-step tree-backup return is

$$\begin{aligned} G_{t:t+2} &\doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+1}(S_{t+1}, a) \\ &\quad + \gamma \pi(A_{t+1}|S_{t+1}) \left(R_{t+2} + \gamma \sum_a \pi(a|S_{t+2}) Q_{t+1}(S_{t+2}, a) \right) \\ &= R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+2}, \end{aligned}$$

for $t < T - 2$. The latter form suggests the general recursive definition of the tree-backup n -step return:

$$G_{t:t+n} \doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+n-1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n}, \quad (7.16)$$

for $t < T - 1, n \geq 2$, with the $n = 1$ case handled by (7.15) except for $G_{T-1:t+n} \doteq R_T$. This target is then used with the usual action-value update rule from n -step Sarsa:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)],$$

for $0 \leq t < T$, while the values of all other state-action pairs remain unchanged: $Q_{t+n}(s, a) = Q_{t+n-1}(s, a)$, for all s, a such that $s \neq S_t$ or $a \neq A_t$. Pseudocode for this algorithm is shown in the box on the next page.

Exercise 7.11 Show that if the approximate action values are unchanging, then the tree-backup return (7.16) can be written as a sum of expectation-based TD errors:

$$G_{t:t+n} = Q(S_t, A_t) + \sum_{k=t}^{\min(t+n-1, T-1)} \delta_k \prod_{i=t+1}^k \gamma \pi(A_i|S_i),$$

where $\delta_t \doteq R_{t+1} + \gamma \bar{V}_t(S_{t+1}) - Q(S_t, A_t)$ and \bar{V}_t is given by (7.8). \square

***n*-step Tree Backup for estimating $Q \approx q_*$ or q_π**

```

Initialize  $Q(s, a)$  arbitrarily, for all  $s \in \mathcal{S}, a \in \mathcal{A}$ 
Initialize  $\pi$  to be greedy with respect to  $Q$ , or as a fixed given policy
Algorithm parameters: step size  $\alpha \in (0, 1]$ , a positive integer  $n$ 
All store and access operations can take their index mod  $n + 1$ 

Loop for each episode:
  Initialize and store  $S_0 \neq$  terminal
  Choose an action  $A_0$  arbitrarily as a function of  $S_0$ ; Store  $A_0$ 
   $T \leftarrow \infty$ 
  Loop for  $t = 0, 1, 2, \dots$  :
    If  $t < T$ :
      Take action  $A_t$ ; observe and store the next reward and state as  $R_{t+1}, S_{t+1}$ 
      If  $S_{t+1}$  is terminal:
         $T \leftarrow t + 1$ 
      else:
        Choose an action  $A_{t+1}$  arbitrarily as a function of  $S_{t+1}$ ; Store  $A_{t+1}$ 
         $\tau \leftarrow t + 1 - n$  ( $\tau$  is the time whose estimate is being updated)
        If  $\tau \geq 0$ :
          If  $t + 1 \geq T$ :
             $G \leftarrow R_T$ 
          else
             $G \leftarrow R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, a)$ 
        Loop for  $k = \min(t, T - 1)$  down through  $\tau + 1$ :
           $G \leftarrow R_k + \gamma \sum_{a \neq A_k} \pi(a|S_k)Q(S_k, a) + \gamma \pi(A_k|S_k)G$ 
           $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha [G - Q(S_\tau, A_\tau)]$ 
          If  $\pi$  is being learned, then ensure that  $\pi(\cdot|S_\tau)$  is greedy wrt  $Q$ 
    Until  $\tau = T - 1$ 

```

7.6 *A Unifying Algorithm: *n*-step $Q(\sigma)$

So far in this chapter we have considered three different kinds of action-value algorithms, corresponding to the first three backup diagrams shown in Figure 7.5. *n*-step Sarsa has all sample transitions, the tree-backup algorithm has all state-to-action transitions fully branched without sampling, and *n*-step Expected Sarsa has all sample transitions except for the last state-to-action one, which is fully branched with an expected value. To what extent can these algorithms be unified?

One idea for unification is suggested by the fourth backup diagram in Figure 7.5. This is the idea that one might decide on a step-by-step basis whether one wanted to take the action as a sample, as in Sarsa, or consider the expectation over all actions instead, as in the tree-backup update. Then, if one chose always to sample, one would obtain Sarsa, whereas if one chose never to sample, one would get the tree-backup algorithm. Expected Sarsa would be the case where one chose to sample for all steps except for the last one.

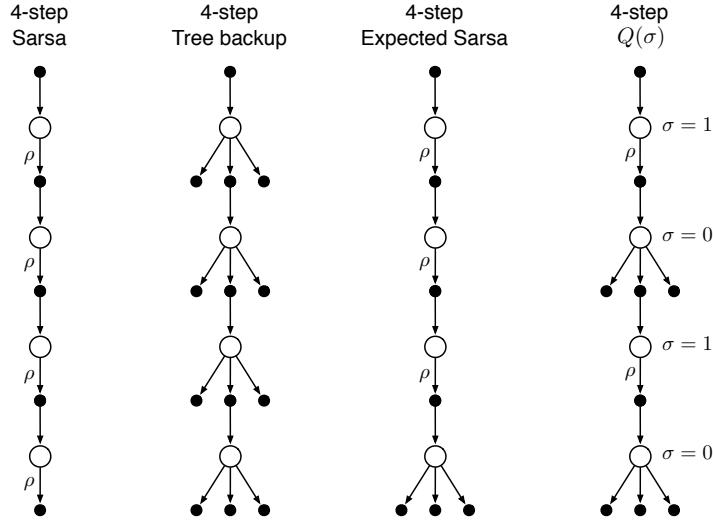


Figure 7.5: The backup diagrams of the three kinds of n -step action-value updates considered so far in this chapter (4-step case) plus the backup diagram of a fourth kind of update that unifies them all. The label ‘ ρ ’ indicates half transitions on which importance sampling is required in the off-policy case. The fourth kind of update unifies all the others by choosing on a state-by-state basis whether to sample ($\sigma_t = 1$) or not ($\sigma_t = 0$).

And of course there would be many other possibilities, as suggested by the last diagram in the figure. To increase the possibilities even further we can consider a continuous variation between sampling and expectation. Let $\sigma_t \in [0, 1]$ denote the degree of sampling on step t , with $\sigma = 1$ denoting full sampling and $\sigma = 0$ denoting a pure expectation with no sampling. The random variable σ_t might be set as a function of the state, action, or state-action pair at time t . We call this proposed new algorithm n -step $Q(\sigma)$.

Now let us develop the equations of n -step $Q(\sigma)$. First we write the tree-backup n -step return (7.16) in terms of the horizon $h = t + n$ and then in terms of the expected approximate value \bar{V} (7.8):

$$\begin{aligned} G_{t:h} &= R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{h-1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:h} \\ &= R_{t+1} + \gamma \bar{V}_{h-1}(S_{t+1}) - \gamma \pi(A_{t+1}|S_{t+1}) Q_{h-1}(S_{t+1}, A_{t+1}) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:h} \\ &= R_{t+1} + \gamma \pi(A_{t+1}|S_{t+1}) (G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})) + \gamma \bar{V}_{h-1}(S_{t+1}), \end{aligned}$$

after which it is exactly like the n -step return for Sarsa with control variates (7.14) except with the action probability $\pi(A_{t+1}|S_{t+1})$ substituted for the importance-sampling ratio ρ_{t+1} . For $Q(\sigma)$, we slide linearly between these two cases:

$$\begin{aligned} G_{t:h} &\doteq R_{t+1} + \gamma (\sigma_{t+1} \rho_{t+1} + (1 - \sigma_{t+1}) \pi(A_{t+1}|S_{t+1})) (G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})) \\ &\quad + \gamma \bar{V}_{h-1}(S_{t+1}), \end{aligned} \tag{7.17}$$

for $t < h \leq T$. The recursion ends with $G_{h:h} \doteq Q_{h-1}(S_h, A_h)$ if $h < T$, or with $G_{T-1:T} \doteq R_T$ if $h = T$. Then we use the earlier update for n-step Sarsa without importance-sampling ratios (7.5) instead of (7.11), because now the ratios are incorporated in the n-step return. A complete algorithm is given in the box.

Off-policy n-step $Q(\sigma)$ for estimating $Q \approx q_*$ or q_π

Input: an arbitrary behavior policy b such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$

Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$

Initialize π to be greedy with respect to Q , or else it is a fixed given policy

Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n

All store and access operations can take their index mod $n + 1$

Loop for each episode:

 Initialize and store $S_0 \neq$ terminal

 Choose and store an action $A_0 \sim b(\cdot|S_0)$

$T \leftarrow \infty$

 Loop for $t = 0, 1, 2, \dots$:

 If $t < T$:

 Take action A_t ; observe and store the next reward and state as R_{t+1}, S_{t+1}

 If S_{t+1} is terminal:

$T \leftarrow t + 1$

 else:

 Choose and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$

 Select and store σ_{t+1}

 Store $\frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})}$ as ρ_{t+1}

$\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)

 If $\tau \geq 0$:

 If $t + 1 < T$:

$G \leftarrow Q(S_{t+1}, A_{t+1})$

 Loop for $k = \min(t + 1, T)$ down through $\tau + 1$:

 if $k = T$:

$G \leftarrow R_T$

 else:

$\bar{V} \leftarrow \sum_a \pi(a|S_k)Q(S_k, a)$

$G \leftarrow R_k + \gamma(\sigma_k \rho_k + (1 - \sigma_k)\pi(A_k|S_k))(G - Q(S_k, A_k)) + \gamma\bar{V}$

$Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[G - Q(S_\tau, A_\tau)]$

 If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q

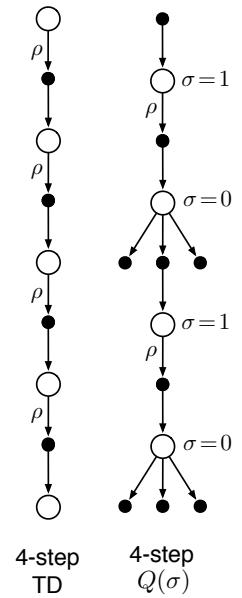
 Until $\tau = T - 1$

7.7 Summary

In this chapter we have developed a range of temporal-difference learning methods that lie in between the one-step TD methods of the previous chapter and the Monte Carlo methods of the chapter before. Methods that involve an intermediate amount of bootstrapping are important because they will typically perform better than either extreme.

Our focus in this chapter has been on n -step methods, which look ahead to the next n rewards, states, and actions. The two 4-step backup diagrams to the right together summarize most of the methods introduced. The state-value update shown is for n -step TD with importance sampling, and the action-value update is for n -step $Q(\sigma)$, which generalizes Expected Sarsa and Q-learning. All n -step methods involve a delay of n time steps before updating, as only then are all the required future events known. A further drawback is that they involve more computation per time step than previous methods. Compared to one-step methods, n -step methods also require more memory to record the states, actions, rewards, and sometimes other variables over the last n time steps. Eventually, in Chapter 12, we will see how multi-step TD methods can be implemented with minimal memory and computational complexity using eligibility traces, but there will always be some additional computation beyond one-step methods. Such costs can be well worth paying to escape the tyranny of the single time step.

Although n -step methods are more complex than those using eligibility traces, they have the great benefit of being conceptually clear. We have sought to take advantage of this by developing two approaches to off-policy learning in the n -step case. One, based on importance sampling is conceptually simple but can be of high variance. If the target and behavior policies are very different it probably needs some new algorithmic ideas before it can be efficient and practical. The other, based on tree-backup updates, is the natural extension of Q-learning to the multi-step case with stochastic target policies. It involves no importance sampling but, again if the target and behavior policies are substantially different, the bootstrapping may span only a few steps even if n is large.



Bibliographical and Historical Remarks

The notion of *n*-step returns is due to Watkins (1989), who also first discussed their error reduction property. *n*-step algorithms were explored in the first edition of this book, in which they were treated as of conceptual interest, but not feasible in practice. The work of Cichosz (1995) and particularly van Seijen (2016) showed that they are actually completely practical algorithms. Given this, and their conceptual clarity and simplicity, we have chosen to highlight them here in the second edition. In particular, we now postpone all discussion of the backward view and of eligibility traces until Chapter 12.

7.1–2 The results in the random walk examples were made for this text based on work of Sutton (1988) and Singh and Sutton (1996). The use of backup diagrams to describe these and other algorithms in this chapter is new.

7.3–5 The developments in these sections are based on the work of Precup, Sutton, and Singh (2000), Precup, Sutton, and Dasgupta (2001), and Sutton, Mahmood, Precup, and van Hasselt (2014).

The tree-backup algorithm is due to Precup, Sutton, and Singh (2000), but the presentation of it here is new.

7.6 The $Q(\sigma)$ algorithm is new to this text, but closely related algorithms have been explored further by De Asis, Hernandez-Garcia, Holland, and Sutton (2017).

Chapter 8

Planning and Learning with Tabular Methods

In this chapter we develop a unified view of reinforcement learning methods that require a model of the environment, such as dynamic programming and heuristic search, and methods that can be used without a model, such as Monte Carlo and temporal-difference methods. These are respectively called *model-based* and *model-free* reinforcement learning methods. Model-based methods rely on *planning* as their primary component, while model-free methods primarily rely on *learning*. Although there are real differences between these two kinds of methods, there are also great similarities. In particular, the heart of both kinds of methods is the computation of value functions. Moreover, all the methods are based on looking ahead to future events, computing a backed-up value, and then using it as an update target for an approximate value function. Earlier in this book we presented Monte Carlo and temporal-difference methods as distinct alternatives, then showed how they can be unified by n -step methods. Our goal in this chapter is a similar integration of model-based and model-free methods. Having established these as distinct in earlier chapters, we now explore the extent to which they can be intermixed.

8.1 Models and Planning

By a *model* of the environment we mean anything that an agent can use to predict how the environment will respond to its actions. Given a state and an action, a model produces a prediction of the resultant next state and next reward. If the model is stochastic, then there are several possible next states and next rewards, each with some probability of occurring. Some models produce a description of all possibilities and their probabilities; these we call *distribution models*. Other models produce just one of the possibilities, sampled according to the probabilities; these we call *sample models*. For example, consider modeling the sum of a dozen dice. A distribution model would produce all possible sums and their probabilities of occurring, whereas a sample model would produce an individual

sum drawn according to this probability distribution. The kind of model assumed in dynamic programming—estimates of the MDP’s dynamics, $p(s', r | s, a)$ —is a distribution model. The kind of model used in the blackjack example in Chapter 5 is a sample model. Distribution models are stronger than sample models in that they can always be used to produce samples. However, in many applications it is much easier to obtain sample models than distribution models. The dozen dice are a simple example of this. It would be easy to write a computer program to simulate the dice rolls and return the sum, but harder and more error-prone to figure out all the possible sums and their probabilities.

Models can be used to mimic or simulate experience. Given a starting state and action, a sample model produces a possible transition, and a distribution model generates all possible transitions weighted by their probabilities of occurring. Given a starting state and a policy, a sample model could produce an entire episode, and a distribution model could generate all possible episodes and their probabilities. In either case, we say the model is used to *simulate* the environment and produce *simulated experience*.

The word *planning* is used in several different ways in different fields. We use the term to refer to any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment:

$$\text{model} \xrightarrow{\text{planning}} \text{policy}$$

In artificial intelligence, there are two distinct approaches to planning according to our definition. *State-space planning*, which includes the approach we take in this book, is viewed primarily as a search through the state space for an optimal policy or an optimal path to a goal. Actions cause transitions from state to state, and value functions are computed over states. In what we call *plan-space planning*, planning is instead a search through the space of plans. Operators transform one plan into another, and value functions, if any, are defined over the space of plans. Plan-space planning includes evolutionary methods and “partial-order planning,” a common kind of planning in artificial intelligence in which the ordering of steps is not completely determined at all stages of planning. Plan-space methods are difficult to apply efficiently to the stochastic sequential decision problems that are the focus in reinforcement learning, and we do not consider them further (but see, e.g., Russell and Norvig, 2010).

The unified view we present in this chapter is that all state-space planning methods share a common structure, a structure that is also present in the learning methods presented in this book. It takes the rest of the chapter to develop this view, but there are two basic ideas: (1) all state-space planning methods involve computing value functions as a key intermediate step toward improving the policy, and (2) they compute value functions by updates or backup operations applied to simulated experience. This common structure can be diagrammed as follows:

$$\text{model} \longrightarrow \text{simulated experience} \xrightarrow{\text{backups}} \text{values} \longrightarrow \text{policy}$$

Dynamic programming methods clearly fit this structure: they make sweeps through the space of states, generating for each state the distribution of possible transitions. Each distribution is then used to compute a backed-up value (update target) and update the

state's estimated value. In this chapter we argue that various other state-space planning methods also fit this structure, with individual methods differing only in the kinds of updates they do, the order in which they do them, and in how long the backed-up information is retained.

Viewing planning methods in this way emphasizes their relationship to the learning methods that we have described in this book. The heart of both learning and planning methods is the estimation of value functions by backing-up update operations. The difference is that whereas planning uses simulated experience generated by a model, learning methods use real experience generated by the environment. Of course this difference leads to a number of other differences, for example, in how performance is assessed and in how flexibly experience can be generated. But the common structure means that many ideas and algorithms can be transferred between planning and learning. In particular, in many cases a learning algorithm can be substituted for the key update step of a planning method. Learning methods require only experience as input, and in many cases they can be applied to simulated experience just as well as to real experience. The box below shows a simple example of a planning method based on one-step tabular Q-learning and on random samples from a sample model. This method, which we call *random-sample one-step tabular Q-planning*, converges to the optimal policy for the model under the same conditions that one-step tabular Q-learning converges to the optimal policy for the real environment (each state-action pair must be selected an infinite number of times in Step 1, and α must decrease appropriately over time).

Random-sample one-step tabular Q-planning

Loop forever:

1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(S)$, at random
2. Send S, A to a sample model, and obtain
a sample next reward, R , and a sample next state, S'
3. Apply one-step tabular Q-learning to S, A, R, S' :

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

In addition to the unified view of planning and learning methods, a second theme in this chapter is the benefits of planning in small, incremental steps. This enables planning to be interrupted or redirected at any time with little wasted computation, which appears to be a key requirement for efficiently intermixing planning with acting and with learning of the model. Planning in very small steps may be the most efficient approach even on pure planning problems if the problem is too large to be solved exactly.

8.2 Dyna: Integrated Planning, Acting, and Learning

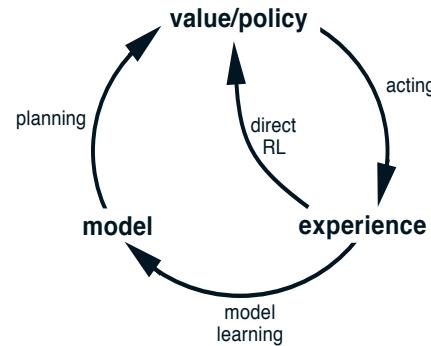
When planning is done online, while interacting with the environment, a number of interesting issues arise. New information gained from the interaction may change the model and thereby interact with planning. It may be desirable to customize the planning process in some way to the states or decisions currently under consideration, or expected

in the near future. If decision making and model learning are both computation-intensive processes, then the available computational resources may need to be divided between them. To begin exploring these issues, in this section we present Dyna-Q, a simple architecture integrating the major functions needed in an online planning agent. Each function appears in Dyna-Q in a simple, almost trivial, form. In subsequent sections we elaborate some of the alternate ways of achieving each function and the trade-offs between them. For now, we seek merely to illustrate the ideas and stimulate your intuition.

Within a planning agent, there are at least two roles for real experience: it can be used to improve the model (to make it more accurately match the real environment) and it can be used to directly improve the value function and policy using the kinds of reinforcement learning methods we have discussed in previous chapters. The former we call *model-learning*, and the latter we call *direct reinforcement learning* (direct RL). The possible relationships between experience, model, values, and policy are summarized in the diagram to the right. Each arrow shows a relationship of influence and presumed improvement. Note how experience can improve value functions and policies either directly or indirectly via the model. It is the latter, which is sometimes called *indirect reinforcement learning*, that is involved in planning.

Both direct and indirect methods have advantages and disadvantages. Indirect methods often make fuller use of a limited amount of experience and thus achieve a better policy with fewer environmental interactions. On the other hand, direct methods are much simpler and are not affected by biases in the design of the model. Some have argued that indirect methods are always superior to direct ones, while others have argued that direct methods are responsible for most human and animal learning. Related debates in psychology and artificial intelligence concern the relative importance of cognition as opposed to trial-and-error learning, and of deliberative planning as opposed to reactive decision making (see Chapter 14 for discussion of some of these issues from the perspective of psychology). Our view is that the contrast between the alternatives in all these debates has been exaggerated, that more insight can be gained by recognizing the similarities between these two sides than by opposing them. For example, in this book we have emphasized the deep similarities between dynamic programming and temporal-difference methods, even though one was designed for planning and the other for model-free learning.

Dyna-Q includes all of the processes shown in the diagram above—planning, acting, model-learning, and direct RL—all occurring continually. The planning method is the random-sample one-step tabular Q-planning method on page 161. The direct RL method is one-step tabular Q-learning. The model-learning method is also table-based and assumes the environment is deterministic. After each transition $S_t, A_t \rightarrow R_{t+1}, S_{t+1}$, the model records in its table entry for S_t, A_t the prediction that R_{t+1}, S_{t+1} will deterministically follow. Thus, if the model is queried with a state-action pair that has been experienced before, it simply returns the last-observed next state and next reward as its prediction.



During planning, the Q-planning algorithm randomly samples only from state-action pairs that have previously been experienced (in Step 1), so the model is never queried with a pair about which it has no information.

The overall architecture of Dyna agents, of which the Dyna-Q algorithm is one example, is shown in Figure 8.1. The central column represents the basic interaction between agent and environment, giving rise to a trajectory of real experience. The arrow on the left of the figure represents direct reinforcement learning operating on real experience to improve the value function and the policy. On the right are model-based processes. The model is learned from real experience and gives rise to simulated experience. We use the term *search control* to refer to the process that selects the starting states and actions for the simulated experiences generated by the model. Finally, planning is achieved by applying reinforcement learning methods to the simulated experiences just as if they had really happened. Typically, as in Dyna-Q, the same reinforcement learning method is used both for learning from real experience and for planning from simulated experience. The reinforcement learning method is thus the “final common path” for both learning and planning. Learning and planning are deeply integrated in the sense that they share almost all the same machinery, differing only in the source of their experience.

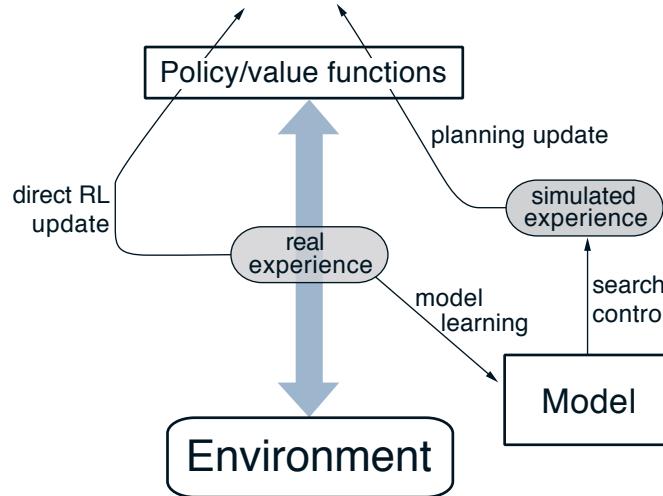


Figure 8.1: The general Dyna Architecture. Real experience, passing back and forth between the environment and the policy, affects policy and value functions in much the same way as does simulated experience generated by the model of the environment.

Conceptually, planning, acting, model-learning, and direct RL occur simultaneously and in parallel in Dyna agents. For concreteness and implementation on a serial computer, however, we fully specify the order in which they occur within a time step. In Dyna-Q, the acting, model-learning, and direct RL processes require little computation, and we assume they consume just a fraction of the time. The remaining time in each step can be devoted to the planning process, which is inherently computation-intensive. Let us assume that there is time in each step, after acting, model-learning, and direct RL, to complete

n iterations (Steps 1–3) of the Q-planning algorithm. In the pseudocode algorithm for Dyna-Q in the box below, $Model(s, a)$ denotes the contents of the (predicted next state and reward) for state-action pair (s, a) . Direct reinforcement learning, model-learning, and planning are implemented by steps (d), (e), and (f), respectively. If (e) and (f) were omitted, the remaining algorithm would be one-step tabular Q-learning.

Tabular Dyna-Q

```

Initialize  $Q(s, a)$  and  $Model(s, a)$  for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$ 
Loop forever:
  (a)  $S \leftarrow$  current (nonterminal) state
  (b)  $A \leftarrow \varepsilon\text{-greedy}(S, Q)$ 
  (c) Take action  $A$ ; observe resultant reward,  $R$ , and state,  $S'$ 
  (d)  $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$ 
  (e)  $Model(S, A) \leftarrow R, S'$  (assuming deterministic environment)
  (f) Loop repeat  $n$  times:
     $S \leftarrow$  random previously observed state
     $A \leftarrow$  random action previously taken in  $S$ 
     $R, S' \leftarrow Model(S, A)$ 
     $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$ 
```

Example 8.1: Dyna Maze Consider the simple maze shown inset in Figure 8.2. In each of the 47 states there are four actions, **up**, **down**, **right**, and **left**, which take the agent deterministically to the corresponding neighboring states, except when movement is blocked by an obstacle or the edge of the maze, in which case the agent remains where it is. Reward is zero on all transitions, except those into the goal state, on which it is +1. After reaching the goal state (G), the agent returns to the start state (S) to begin a new episode. This is a discounted, episodic task with $\gamma = 0.95$.

The main part of Figure 8.2 shows average learning curves from an experiment in which Dyna-Q agents were applied to the maze task. The initial action values were zero, the step-size parameter was $\alpha = 0.1$, and the exploration parameter was $\varepsilon = 0.1$. When selecting greedily among actions, ties were broken randomly. The agents varied in the number of planning steps, n , they performed per real step. For each n , the curves show the number of steps taken by the agent to reach the goal in each episode, averaged over 30 repetitions of the experiment. In each repetition, the initial seed for the random number generator was held constant across algorithms. Because of this, the first episode was exactly the same (about 1700 steps) for all values of n , and its data are not shown in the figure. After the first episode, performance improved for all values of n , but much more rapidly for larger values. Recall that the $n = 0$ agent is a nonplanning agent, using only direct reinforcement learning (one-step tabular Q-learning). This was by far the slowest agent on this problem, despite the fact that the parameter values (α and ε) were optimized for it. The nonplanning agent took about 25 episodes to reach (ε -)optimal performance, whereas the $n = 5$ agent took about five episodes, and the $n = 50$ agent took only three episodes.

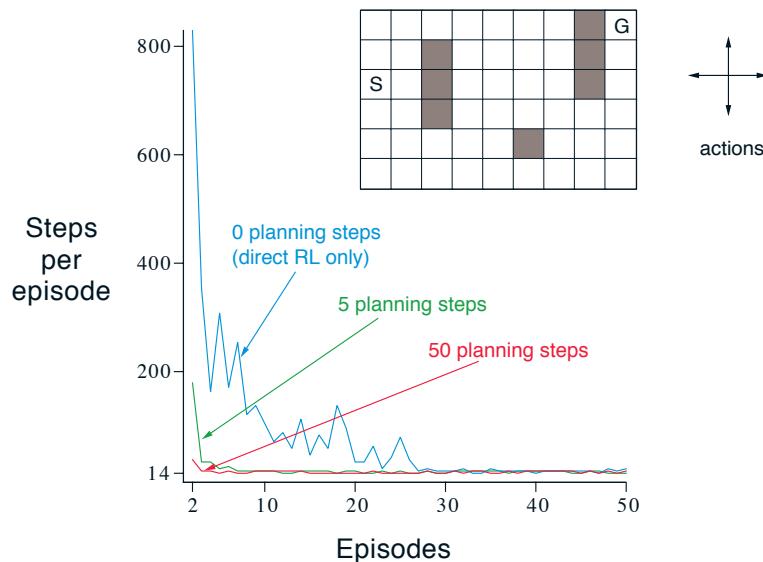


Figure 8.2: A simple maze (inset) and the average learning curves for Dyna-Q agents varying in their number of planning steps (n) per real step. The task is to travel from S to G as quickly as possible.

Figure 8.3 shows why the planning agents found the solution so much faster than the nonplanning agent. Shown are the policies found by the $n = 0$ and $n = 50$ agents halfway through the second episode. Without planning ($n = 0$), each episode adds only one additional step to the policy, and so only one step (the last) has been learned so far. With planning, again only one step is learned during the first episode, but here during the second episode an extensive policy has been developed that by the end of the episode will reach almost back to the start state. This policy is built by the planning process while the agent is still wandering near the start state. By the end of the third episode a complete optimal policy will have been found and perfect performance attained.

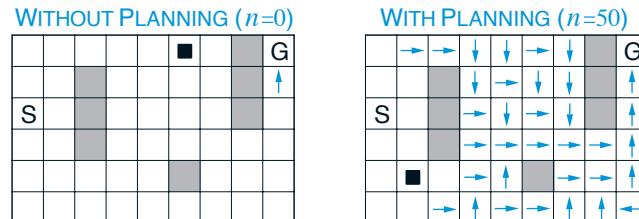


Figure 8.3: Policies found by planning and nonplanning Dyna-Q agents halfway through the second episode. The arrows indicate the greedy action in each state; if no arrow is shown for a state, then all of its action values were equal. The black square indicates the location of the agent. ■

In Dyna-Q, learning and planning are accomplished by exactly the same algorithm, operating on real experience for learning and on simulated experience for planning. Because planning proceeds incrementally, it is trivial to intermix planning and acting. Both proceed as fast as they can. The agent is always reactive and always deliberative, responding instantly to the latest sensory information and yet always planning in the background. Also ongoing in the background is the model-learning process. As new information is gained, the model is updated to better match reality. As the model changes, the ongoing planning process will gradually compute a different way of behaving to match the new model.

Exercise 8.1 The nonplanning method looks particularly poor in Figure 8.3 because it is a one-step method; a method using multi-step bootstrapping would do better. Do you think one of the multi-step bootstrapping methods from Chapter 7 could do as well as the Dyna method? Explain why or why not. \square

8.3 When the Model Is Wrong

In the maze example presented in the previous section, the changes in the model were relatively modest. The model started out empty, and was then filled only with exactly correct information. In general, we cannot expect to be so fortunate. Models may be incorrect because the environment is stochastic and only a limited number of samples have been observed, or because the model was learned using function approximation that has generalized imperfectly, or simply because the environment has changed and its new behavior has not yet been observed. When the model is incorrect, the planning process is likely to compute a suboptimal policy.

In some cases, the suboptimal policy computed by planning quickly leads to the discovery and correction of the modeling error. This tends to happen when the model is optimistic in the sense of predicting greater reward or better state transitions than are actually possible. The planned policy attempts to exploit these opportunities and in doing so discovers that they do not exist.

Example 8.2: Blocking Maze A maze example illustrating this relatively minor kind of modeling error and recovery from it is shown in Figure 8.4. Initially, there is a short path from start to goal, to the right of the barrier, as shown in the upper left of the figure. After 1000 time steps, the short path is “blocked,” and a longer path is opened up along the left-hand side of the barrier, as shown in upper right of the figure. The graph shows average cumulative reward for a Dyna-Q agent and an enhanced Dyna-Q+ agent to be described shortly. The first part of the graph shows that both Dyna agents found the short path within 1000 steps. When the environment changed, the graphs become flat, indicating a period during which the agents obtained no reward because they were wandering around behind the barrier. After a while, however, they were able to find the new opening and the new optimal behavior.

Greater difficulties arise when the environment changes to become *better* than it was before, and yet the formerly correct policy does not reveal the improvement. In these cases the modeling error may not be detected for a long time, if ever.

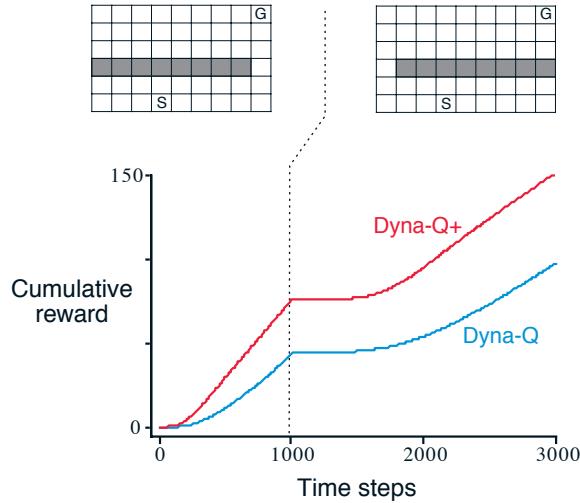


Figure 8.4: Average performance of Dyna agents on a blocking task. The left environment was used for the first 1000 steps, the right environment for the rest. Dyna-Q+ is Dyna-Q with an exploration bonus that encourages exploration. ■

Example 8.3: Shortcut Maze

The problem caused by this kind of environmental change is illustrated by the maze example shown in Figure 8.5. Initially, the optimal path is to go around the left side of the barrier (upper left). After 3000 steps, however, a shorter path is opened up along the right side, without disturbing the longer path (upper right). The graph shows that the regular Dyna-Q agent never switched to the shortcut. In fact, it never realized that it existed. Its model said that there was no shortcut, so the more it planned, the less likely it was to step to the right and discover it. Even with an ϵ -greedy policy, it is very unlikely that an agent will take so many exploratory actions as to discover the shortcut.

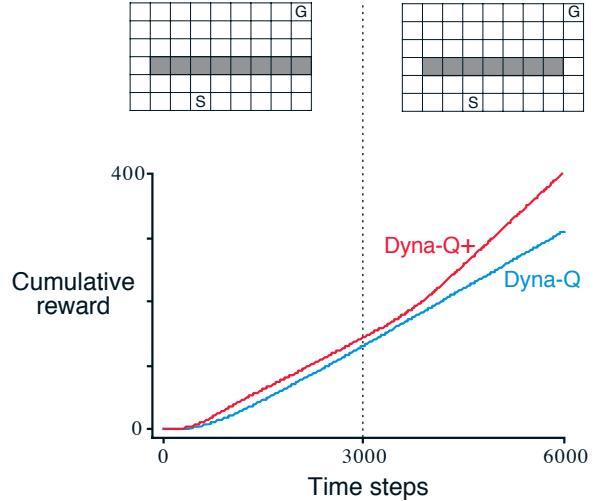


Figure 8.5: Average performance of Dyna agents on a shortcut task. The left environment was used for the first 3000 steps, the right environment for the rest. ■

The general problem here is another version of the conflict between exploration and exploitation. In a planning context, exploration means trying actions that improve the model, whereas exploitation means behaving in the optimal way given the current model.

We want the agent to explore to find changes in the environment, but not so much that performance is greatly degraded. As in the earlier exploration/exploitation conflict, there probably is no solution that is both perfect and practical, but simple heuristics are often effective.

The Dyna-Q+ agent that did solve the shortcut maze uses one such heuristic. This agent keeps track for each state-action pair of how many time steps have elapsed since the pair was last tried in a real interaction with the environment. The more time that has elapsed, the greater (we might presume) the chance that the dynamics of this pair has changed and that the model of it is incorrect. To encourage behavior that tests long-untried actions, a special “bonus reward” is given on simulated experiences involving these actions. In particular, if the modeled reward for a transition is r , and the transition has not been tried in τ time steps, then planning updates are done as if that transition produced a reward of $r + \kappa\sqrt{\tau}$, for some small κ . This encourages the agent to keep testing all accessible state transitions and even to find long sequences of actions in order to carry out such tests.¹ Of course all this testing has its cost, but in many cases, as in the shortcut maze, this kind of computational curiosity is well worth the extra exploration.

Exercise 8.2 Why did the Dyna agent with exploration bonus, Dyna-Q+, perform better in the first phase as well as in the second phase of the blocking and shortcut experiments? \square

Exercise 8.3 Careful inspection of Figure 8.5 reveals that the difference between Dyna-Q+ and Dyna-Q narrowed slightly over the first part of the experiment. What is the reason for this? \square

Exercise 8.4 (programming) The exploration bonus described above actually changes the estimated values of states and actions. Is this necessary? Suppose the bonus $\kappa\sqrt{\tau}$ was used not in updates, but solely in action selection. That is, suppose the action selected was always that for which $Q(S_t, a) + \kappa\sqrt{\tau(S_t, a)}$ was maximal. Carry out a gridworld experiment that tests and illustrates the strengths and weaknesses of this alternate approach. \square

Exercise 8.5 How might the tabular Dyna-Q algorithm shown on page 164 be modified to handle stochastic environments? How might this modification perform poorly on changing environments such as considered in this section? How could the algorithm be modified to handle stochastic environments *and* changing environments? \square

8.4 Prioritized Sweeping

In the Dyna agents presented in the preceding sections, simulated transitions are started in state-action pairs selected uniformly at random from all previously experienced pairs. But a uniform selection is usually not the best; planning can be much more efficient if simulated transitions and updates are focused on particular state-action pairs. For example, consider

¹The Dyna-Q+ agent was changed in two other ways as well. First, actions that had never been tried before from a state were allowed to be considered in the planning step (f) of the Tabular Dyna-Q algorithm in the box above. Second, the initial model for such actions was that they would lead back to the same state with a reward of zero.

what happens during the second episode of the first maze task (Figure 8.3). At the beginning of the second episode, only the state-action pair leading directly into the goal has a positive value; the values of all other pairs are still zero. This means that it is pointless to perform updates along almost all transitions, because they take the agent from one zero-valued state to another, and thus the updates would have no effect. Only an update along a transition into the state just prior to the goal, or from it, will change any values. If simulated transitions are generated uniformly, then many wasteful updates will be made before stumbling onto one of these useful ones. As planning progresses, the region of useful updates grows, but planning is still far less efficient than it would be if focused where it would do the most good. In the much larger problems that are our real objective, the number of states is so large that an unfocused search would be extremely inefficient.

This example suggests that search might be usefully focused by working *backward* from goal states. Of course, we do not really want to use any methods specific to the idea of “goal state.” We want methods that work for general reward functions. Goal states are just a special case, convenient for stimulating intuition. In general, we want to work back not just from goal states but from any state whose value has changed. Suppose that the values are initially correct given the model, as they were in the maze example prior to discovering the goal. Suppose now that the agent discovers a change in the environment and changes its estimated value of one state, either up or down. Typically, this will imply that the values of many other states should also be changed, but the only useful one-step updates are those of actions that lead directly into the one state whose value has been changed. If the values of these actions are updated, then the values of the predecessor states may change in turn. If so, then actions leading into them need to be updated, and then *their* predecessor states may have changed. In this way one can work backward from arbitrary states that have changed in value, either performing useful updates or terminating the propagation. This general idea might be termed *backward focusing* of planning computations.

As the frontier of useful updates propagates backward, it often grows rapidly, producing many state-action pairs that could usefully be updated. But not all of these will be equally useful. The values of some states may have changed a lot, whereas others may have changed little. The predecessor pairs of those that have changed a lot are more likely to also change a lot. In a stochastic environment, variations in estimated transition probabilities also contribute to variations in the sizes of changes and in the urgency with which pairs need to be updated. It is natural to prioritize the updates according to a measure of their urgency, and perform them in order of priority. This is the idea behind *prioritized sweeping*. A queue is maintained of every state-action pair whose estimated value would change nontrivially if updated, prioritized by the size of the change. When the top pair in the queue is updated, the effect on each of its predecessor pairs is computed. If the effect is greater than some small threshold, then the pair is inserted in the queue with the new priority (if there is a previous entry of the pair in the queue, then insertion results in only the higher priority entry remaining in the queue). In this way the effects of changes are efficiently propagated backward until quiescence. The full algorithm for the case of deterministic environments is given in the box on the next page.

Prioritized sweeping for a deterministic environment

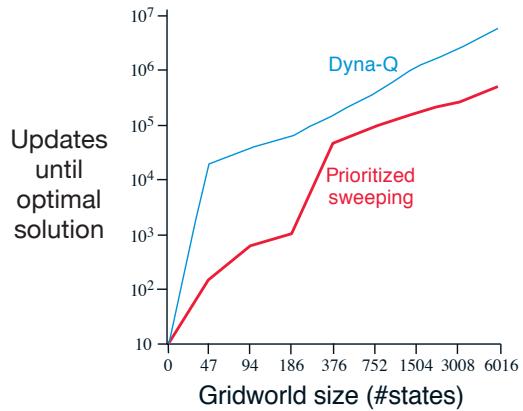
Initialize $Q(s, a)$, $Model(s, a)$, for all s, a , and $PQueue$ to empty

Loop forever:

- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow policy(S, Q)$
- (c) Take action A ; observe resultant reward, R , and state, S'
- (d) $Model(S, A) \leftarrow R, S'$
- (e) $P \leftarrow |R + \gamma \max_a Q(S', a) - Q(S, A)|$.
- (f) if $P > \theta$, then insert S, A into $PQueue$ with priority P
- (g) Loop repeat n times, while $PQueue$ is not empty:
 $S, A \leftarrow first(PQueue)$
 $R, S' \leftarrow Model(S, A)$
 $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
Loop for all \bar{S}, \bar{A} predicted to lead to S :
 $\bar{R} \leftarrow$ predicted reward for \bar{S}, \bar{A}, S
 $P \leftarrow |\bar{R} + \gamma \max_a Q(S, a) - Q(\bar{S}, \bar{A})|$.
if $P > \theta$ then insert \bar{S}, \bar{A} into $PQueue$ with priority P

Example 8.4: Prioritized Sweeping

on Mazes Prioritized sweeping has been found to dramatically increase the speed at which optimal solutions are found in maze tasks, often by a factor of 5 to 10. A typical example is shown to the right. These data are for a sequence of maze tasks of exactly the same structure as the one shown in Figure 8.2, except that they vary in the grid resolution. Prioritized sweeping maintained a decisive advantage over unprioritized Dyna-Q. Both systems made at most $n = 5$ updates per environmental interaction. Adapted from Peng and Williams (1993).

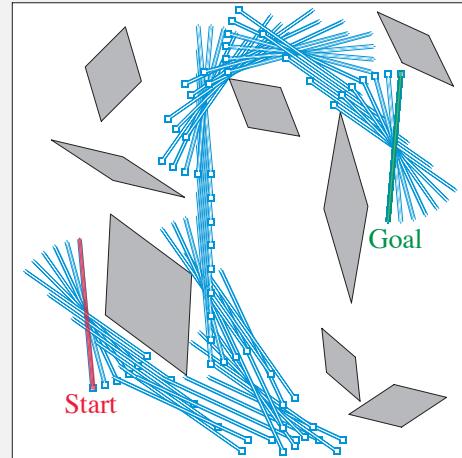


Extensions of prioritized sweeping to stochastic environments are straightforward. The model is maintained by keeping counts of the number of times each state-action pair has been experienced and of what the next states were. It is natural then to update each pair not with a sample update, as we have been using so far, but with an expected update, taking into account all possible next states and their probabilities of occurring.

Prioritized sweeping is just one way of distributing computations to improve planning efficiency, and probably not the best way. One of prioritized sweeping's limitations is that it uses *expected* updates, which in stochastic environments may waste lots of computation on low-probability transitions. As we show in the following section, sample updates

Example 8.5 Prioritized Sweeping for Rod Maneuvering

The objective in this task is to maneuver a rod around some awkwardly placed obstacles within a limited rectangular work space to a goal position in the fewest number of steps. The rod can be translated along its long axis or perpendicular to that axis, or it can be rotated in either direction around its center. The distance of each movement is approximately $1/20$ of the work space, and the rotation increment is 10 degrees. Translations are deterministic and quantized to one of 20×20 positions. To the right is shown the obstacles and the shortest solution from start to goal, found by prioritized sweeping. This problem is deterministic, but has four actions and 14,400 potential states (some of these are unreachable because of the obstacles). This problem is probably too large to be solved with unprioritized methods. Figure reprinted from Moore and Atkeson (1993).



can in many cases get closer to the true value function with less computation despite the variance introduced by sampling. Sample updates can win because they break the overall backing-up computation into smaller pieces—those corresponding to individual transitions—which then enables it to be focused more narrowly on the pieces that will have the largest impact. This idea was taken to what may be its logical limit in the “small backups” introduced by van Seijen and Sutton (2013). These are updates along a single transition, like a sample update, but based on the probability of the transition without sampling, as in an expected update. By selecting the order in which small updates are done it is possible to greatly improve planning efficiency beyond that possible with prioritized sweeping.

We have suggested in this chapter that all kinds of state-space planning can be viewed as sequences of value updates, varying only in the type of update, expected or sample, large or small, and in the order in which the updates are done. In this section we have emphasized backward focusing, but this is just one strategy. For example, another would be to focus on states according to how easily they can be reached from the states that are visited frequently under the current policy, which might be called *forward focusing*. Peng and Williams (1993) and Barto, Bradtke and Singh (1995) have explored versions of forward focusing, and the methods introduced in the next few sections take it to an extreme form.

8.5 Expected vs. Sample Updates

The examples in the previous sections give some idea of the range of possibilities for combining methods of learning and planning. In the rest of this chapter, we analyze some of the component ideas involved, starting with the relative advantages of expected and sample updates.

Much of this book has been about different kinds of value-function updates, and we have considered a great many varieties. Focusing for the moment on one-step updates, they vary primarily along three binary dimensions. The first two dimensions are whether they update state values or action values and whether they estimate the value for the optimal policy or for an arbitrary given policy. These two dimensions give rise to four classes of updates for approximating the four value functions, q_* , v_* , q_π , and v_π . The other binary dimension is whether the updates are *expected* updates, considering all possible events that might happen, or *sample* updates, considering a single sample of what might happen. These three binary dimensions give rise to eight cases, seven of which correspond to specific algorithms, as shown in the figure to the right. (The eighth case does not seem to correspond to any useful update.) Any of these one-step updates can be used in planning methods. The Dyna-Q agents discussed earlier use q_* sample updates, but they could just as well use q_* expected updates, or either expected or sample q_π updates. The Dyna-AC system uses v_π sample updates together with a learning policy structure (as in Chapter 13). For stochastic problems, prioritized sweeping is always done using one of the expected updates.

When we introduced one-step sample updates in Chapter 6, we presented them as substitutes for expected updates. In the absence of a distribution model, expected updates are not possible, but sample updates can be done using sample transitions from the environment or a sample model. Implicit in that point of view is that expected updates, if possible, are preferable to sample updates. But are they? Expected

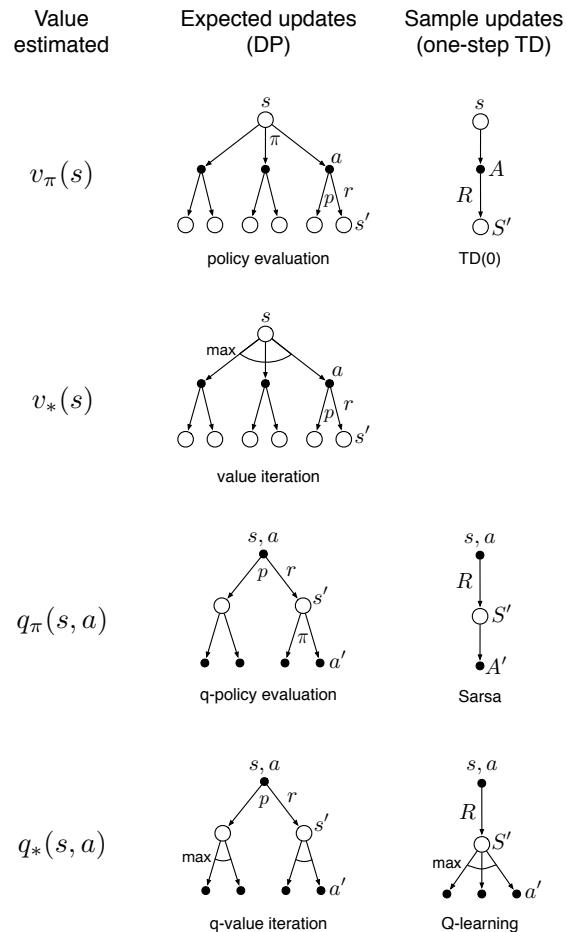


Figure 8.6: Backup diagrams for all the one-step updates considered in this book.

updates certainly yield a better estimate because they are uncorrupted by sampling error, but they also require more computation, and computation is often the limiting resource in planning. To properly assess the relative merits of expected and sample updates for planning we must control for their different computational requirements.

For concreteness, consider the expected and sample updates for approximating q_* , and the special case of discrete states and actions, a table-lookup representation of the approximate value function, Q , and a model in the form of estimated dynamics, $\hat{p}(s', r | s, a)$. The expected update for a state-action pair, s, a , is:

$$Q(s, a) \leftarrow \sum_{s', r} \hat{p}(s', r | s, a) \left[r + \gamma \max_{a'} Q(s', a') \right]. \quad (8.1)$$

The corresponding sample update for s, a , given a sample next state and reward, S' and R (from the model), is the Q-learning-like update:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[R + \gamma \max_{a'} Q(S', a') - Q(s, a) \right], \quad (8.2)$$

where α is the usual positive step-size parameter.

The difference between these expected and sample updates is significant to the extent that the environment is stochastic, specifically, to the extent that, given a state and action, many possible next states may occur with various probabilities. If only one next state is possible, then the expected and sample updates given above are identical (taking $\alpha = 1$). If there are many possible next states, then there may be significant differences. In favor of the expected update is that it is an exact computation, resulting in a new $Q(s, a)$ whose correctness is limited only by the correctness of the $Q(s', a')$ at successor states. The sample update is in addition affected by sampling error. On the other hand, the sample update is cheaper computationally because it considers only one next state, not all possible next states. In practice, the computation required by update operations is usually dominated by the number of state-action pairs at which Q is evaluated. For a particular starting pair, s, a , let b be the *branching factor* (i.e., the number of possible next states, s' , for which $\hat{p}(s' | s, a) > 0$). Then an expected update of this pair requires roughly b times as much computation as a sample update.

If there is enough time to complete an expected update, then the resulting estimate is generally better than that of b sample updates because of the absence of sampling error. But if there is insufficient time to complete an expected update, then sample updates are always preferable because they at least make some improvement in the value estimate with fewer than b updates. In a large problem with many state-action pairs, we are often in the latter situation. With so many state-action pairs, expected updates of all of them would take a very long time. Before that we may be much better off with a few sample updates at many state-action pairs than with expected updates at a few pairs. Given a unit of computational effort, is it better devoted to a few expected updates or to b times as many sample updates?

Figure 8.7 shows the results of an analysis that suggests an answer to this question. It shows the estimation error as a function of computation time for expected and sample updates for a variety of branching factors, b . The case considered is that in which all

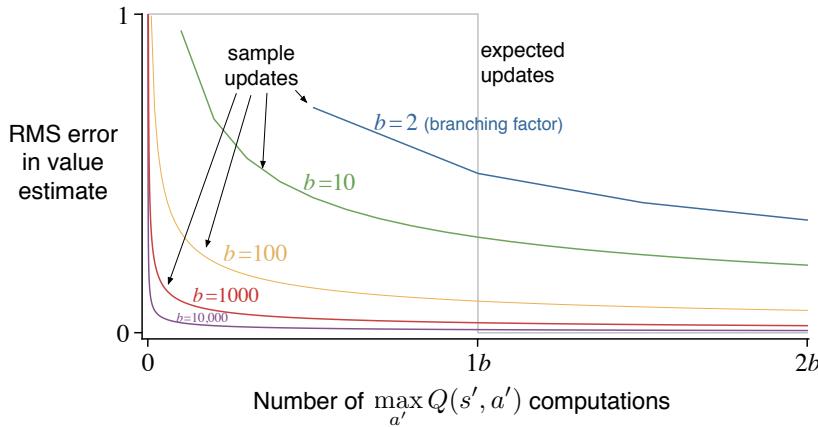


Figure 8.7: Comparison of efficiency of expected and sample updates.

b successor states are equally likely and in which the error in the initial estimate is 1. The values at the next states are assumed correct, so the expected update reduces the error to zero upon its completion. In this case, sample updates reduce the error according to $\sqrt{\frac{b-1}{bt}}$ where t is the number of sample updates that have been performed (assuming sample averages, i.e., $\alpha = 1/t$). The key observation is that for moderately large b the error falls dramatically with a tiny fraction of b updates. For these cases, many state-action pairs could have their values improved dramatically, to within a few percent of the effect of an expected update, in the same time that a single state-action pair could undergo an expected update.

The advantage of sample updates shown in Figure 8.7 is probably an underestimate of the real effect. In a real problem, the values of the successor states would be estimates that are themselves updated. By causing estimates to be more accurate sooner, sample updates will have a second advantage in that the values backed up from the successor states will be more accurate. These results suggest that sample updates are likely to be superior to expected updates on problems with large stochastic branching factors and too many states to be solved exactly.

Exercise 8.6 The analysis above assumed that all of the b possible next states were equally likely to occur. Suppose instead that the distribution was highly skewed, that some of the b states were much more likely to occur than most. Would this strengthen or weaken the case for sample updates over expected updates? Support your answer. \square

8.6 Trajectory Sampling

In this section we compare two ways of distributing updates. The classical approach, from dynamic programming, is to perform sweeps through the entire state (or state-action) space, updating each state (or state-action pair) once per sweep. This is problematic

on large tasks because there may not be time to complete even one sweep. In many tasks the vast majority of the states are irrelevant because they are visited only under very poor policies or with very low probability. Exhaustive sweeps implicitly devote equal time to all parts of the state space rather than focusing where it is needed. As we discussed in Chapter 4, exhaustive sweeps and the equal treatment of all states that they imply are not necessary properties of dynamic programming. In principle, updates can be distributed any way one likes (to assure convergence, all states or state-action pairs must be visited in the limit an infinite number of times; although an exception to this is discussed in Section 8.7 below), but in practice exhaustive sweeps are often used.

The second approach is to sample from the state or state-action space according to some distribution. One could sample uniformly, as in the Dyna-Q agent, but this would suffer from some of the same problems as exhaustive sweeps. More appealing is to distribute updates according to the on-policy distribution, that is, according to the distribution observed when following the current policy. One advantage of this distribution is that it is easily generated; one simply interacts with the model, following the current policy. In an episodic task, one starts in a start state (or according to the starting-state distribution) and simulates until the terminal state. In a continuing task, one starts anywhere and just keeps simulating. In either case, sample state transitions and rewards are given by the model, and sample actions are given by the current policy. In other words, one simulates explicit individual trajectories and performs updates at the state or state-action pairs encountered along the way. We call this way of generating experience and updates *trajectory sampling*.

It is hard to imagine any efficient way of distributing updates according to the on-policy distribution other than by trajectory sampling. If one had an explicit representation of the on-policy distribution, then one could sweep through all states, weighting the update of each according to the on-policy distribution, but this leaves us again with all the computational costs of exhaustive sweeps. Possibly one could sample and update individual state-action pairs from the distribution, but even if this could be done efficiently, what benefit would this provide over simulating trajectories? Even knowing the on-policy distribution in an explicit form is unlikely. The distribution changes whenever the policy changes, and computing the distribution requires computation comparable to a complete policy evaluation. Consideration of such other possibilities makes trajectory sampling seem both efficient and elegant.

Is the on-policy distribution of updates a good one? Intuitively it seems like a good choice, at least better than the uniform distribution. For example, if you are learning to play chess, you study positions that might arise in real games, not random positions of chess pieces. The latter may be valid states, but to be able to accurately value them is a different skill from evaluating positions in real games. We will also see in Part II that the on-policy distribution has significant advantages when function approximation is used. Whether or not function approximation is used, one might expect on-policy focusing to significantly improve the speed of planning.

Focusing on the on-policy distribution could be beneficial because it causes vast, uninteresting parts of the space to be ignored, or it could be detrimental because it causes the same old parts of the space to be updated over and over. We conducted a small

experiment to assess the effect empirically. To isolate the effect of the update distribution, we used entirely one-step expected tabular updates, as defined by (8.1). In the *uniform* case, we cycled through all state–action pairs, updating each in place, and in the *on-policy* case we simulated episodes, all starting in the same state, updating each state–action pair that occurred under the current ε -greedy policy ($\varepsilon=0.1$). The tasks were undiscounted episodic tasks, generated randomly as follows. From each of the $|\mathcal{S}|$ states, two actions were possible, each of which resulted in one of b next states, all equally likely, with a different random selection of b states for each state–action pair. The branching factor, b , was the same for all state–action pairs. In addition, on all transitions there was a 0.1 probability of transition to the terminal state, ending the episode. The expected reward on each transition was selected from a Gaussian distribution with mean 0 and variance 1.

At any point in the planning process one can stop and exhaustively compute $v_{\tilde{\pi}}(s_0)$, the true value of the start state under the greedy policy, $\tilde{\pi}$, given the current action-value function Q , as an indication of how well the agent would do on a new episode on which it acted greedily (all the while assuming the model is correct).

The upper part of the figure to the right shows results averaged over 200 sample tasks with 1000 states and branching factors of 1, 3, and 10. The quality of the policies found is plotted as a function of the number of expected updates completed. In all cases, sampling according to the on-policy distribution resulted in faster planning initially and retarded planning in the long run. The effect was stronger, and the initial period of faster planning was longer, at smaller branching factors. In other experiments, we found that these effects also became stronger as the number of states increased. For example, the lower part of the figure shows results for a branching factor of 1 for tasks with 10,000 states. In this case the advantage of on-policy focusing is large and long-lasting.

All of these results make sense. In the short term, sampling according to the on-policy distribution helps by focusing on states that are near descendants of

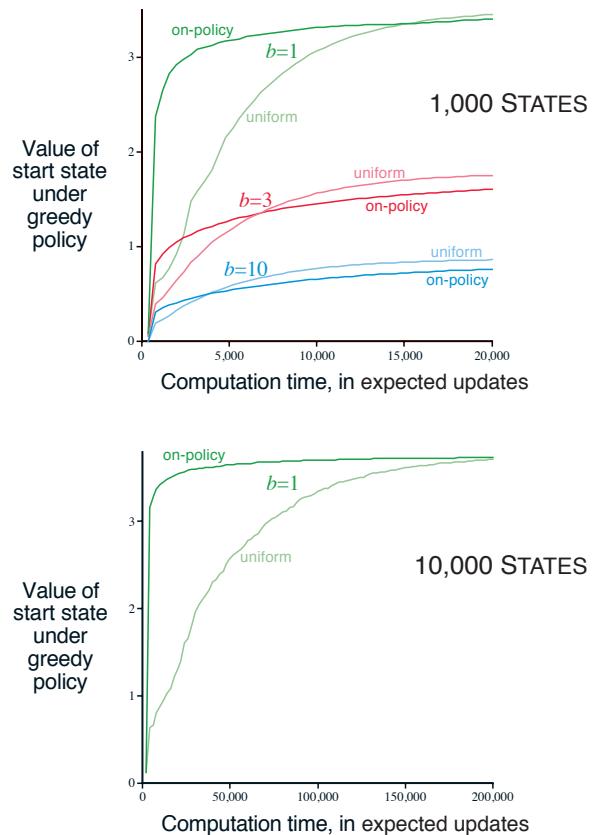


Figure 8.8: Relative efficiency of updates distributed uniformly across the state space versus focused on simulated on-policy trajectories, each starting in the same state. Results are for randomly generated tasks of two sizes and various branching factors, b .

the start state. If there are many states and a small branching factor, this effect will be large and long-lasting. In the long run, focusing on the on-policy distribution may hurt because the commonly occurring states all already have their correct values. Sampling them is useless, whereas sampling other states may actually perform some useful work. This presumably is why the exhaustive, unfocused approach does better in the long run, at least for small problems. These results are not conclusive because they are only for problems generated in a particular, random way, but they do suggest that sampling according to the on-policy distribution can be a great advantage for large problems, in particular for problems in which a small subset of the state-action space is visited under the on-policy distribution.

Exercise 8.7 Some of the graphs in Figure 8.8 seem to be scalloped in their early portions, particularly the upper graph for $b = 1$ and the uniform distribution. Why do you think this is? What aspects of the data shown support your hypothesis? \square

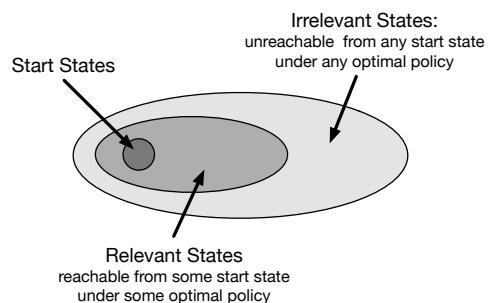
Exercise 8.8 (programming) Replicate the experiment whose results are shown in the lower part of Figure 8.8, then try the same experiment but with $b = 3$. Discuss the meaning of your results. \square

8.7 Real-time Dynamic Programming

Real-time dynamic programming, or RTDP, is an on-policy trajectory-sampling version of the value-iteration algorithm of dynamic programming (DP). Because it is closely related to conventional sweep-based policy iteration, RTDP illustrates in a particularly clear way some of the advantages that on-policy trajectory sampling can provide. RTDP updates the values of states visited in actual or simulated trajectories by means of expected tabular value-iteration updates as defined by (4.10). It is basically the algorithm that produced the on-policy results shown in Figure 8.8.

The close connection between RTDP and conventional DP makes it possible to derive some theoretical results by adapting existing theory. RTDP is an example of an *asynchronous* DP algorithm as described in Section 4.5. Asynchronous DP algorithms are not organized in terms of systematic sweeps of the state set; they update state values in any order whatsoever, using whatever values of other states happen to be available. In RTDP, the update order is dictated by the order states are visited in real or simulated trajectories.

If trajectories can start only from a designated set of start states, and if you are interested in the prediction problem for a given policy, then on-policy trajectory sampling allows the algorithm to completely skip states that cannot be reached by the given policy from any of the start states: such states are *irrelevant* to the prediction problem. For a control problem, where the goal is to find an optimal policy instead of evaluating a given policy, there might well be states that cannot be



reached by any optimal policy from any of the start states, and there is no need to specify optimal actions for these irrelevant states. What is needed is an *optimal partial policy*, meaning a policy that is optimal for the relevant states but can specify arbitrary actions, or even be undefined, for the irrelevant states.

But *finding* such an optimal partial policy with an on-policy trajectory-sampling control method, such as Sarsa (Section 6.4), in general requires visiting all state-action pairs—even those that will turn out to be irrelevant—an infinite number of times. This can be done, for example, by using exploring starts (Section 5.3). This is true for RTDP as well: for episodic tasks with exploring starts, RTDP is an asynchronous value-iteration algorithm that converges to optimal policies for discounted finite MDPs (and for the undiscounted case under certain conditions). Unlike the situation for a prediction problem, it is generally not possible to stop updating any state or state-action pair if convergence to an optimal policy is important.

The most interesting result for RTDP is that for certain types of problems satisfying reasonable conditions, RTDP is guaranteed to find a policy that is optimal on the relevant states without visiting every state infinitely often, or even without visiting some states at all. Indeed, in some problems, only a small fraction of the states need to be visited. This can be a great advantage for problems with very large state sets, where even a single sweep may not be feasible.

The tasks for which this result holds are undiscounted episodic tasks for MDPs with absorbing goal states that generate zero rewards, as described in Section 3.4. At every step of a real or simulated trajectory, RTDP selects a greedy action (breaking ties randomly) and applies the expected value-iteration update operation to the current state. It can also update the values of an arbitrary collection of other states at each step; for example, it can update the values of states visited in a limited-horizon look-ahead search from the current state.

For these problems, with each episode beginning in a state randomly chosen from the set of start states and ending at a goal state, RTDP converges with probability one to a policy that is optimal for all the relevant states provided: (1) the initial value of every goal state is zero, (2) there exists at least one policy that guarantees that a goal state will be reached with probability one from any start state, (3) all rewards for transitions from non-goal states are strictly negative, and (4) all the initial values are equal to, or greater than, their optimal values (which can be satisfied by simply setting the initial values of all states to zero). This result was proved by Barto, Bradtke, and Singh (1995) by combining results for asynchronous DP with results about a heuristic search algorithm known as *learning real-time A** due to Korf (1990).

Tasks having these properties are examples of *stochastic optimal path problems*, which are usually stated in terms of cost minimization instead of as reward maximization as we do here. Maximizing the negative returns in our version is equivalent to minimizing the costs of paths from a start state to a goal state. Examples of this kind of task are minimum-time control tasks, where each time step required to reach a goal produces a reward of -1 , or problems like the Golf example in Section 3.5, whose objective is to hit the hole with the fewest strokes.

Example 8.6: RTDP on the Racetrack The racetrack problem of Exercise 5.12 (page 111) is a stochastic optimal path problem. Comparing RTDP and the conventional DP value iteration algorithm on an example racetrack problem illustrates some of the advantages of on-policy trajectory sampling.

Recall from the exercise that an agent has to learn how to drive a car around a turn like those shown in Figure 5.5 and cross the finish line as quickly as possible while staying on the track. Start states are all the zero-speed states on the starting line; the goal states are all the states that can be reached in one time step by crossing the finish line from inside the track. Unlike Exercise 5.12, here there is no limit on the car’s speed, so the state set is potentially infinite. However, the set of states that can be reached from the set of start states via any policy is finite and can be considered to be the state set of the problem. Each episode begins in a randomly selected start state and ends when the car crosses the finish line. The rewards are -1 for each step until the car crosses the finish line. If the car hits the track boundary, it is moved back to a random start state, and the episode continues.

A racetrack similar to the small racetrack on the left of Figure 5.5 has 9,115 states reachable from start states by any policy, only 599 of which are relevant, meaning that they are reachable from some start state via some optimal policy. (The number of relevant states was estimated by counting the states visited while executing optimal actions for 10^7 episodes.)

The table below compares solving this task by conventional DP and by RTDP. These results are averages over 25 runs, each begun with a different random number seed. Conventional DP in this case is value iteration using exhaustive sweeps of the state set, with values updated one state at a time in place, meaning that the update for each state uses the most recent values of the other states (This is the Gauss-Seidel version of value iteration, which was found to be approximately twice as fast as the Jacobi version on this problem. See Section 4.8.) No special attention was paid to the ordering of the updates; other orderings could have produced faster convergence. Initial values were all zero for each run of both methods. DP was judged to have converged when the maximum change in a state value over a sweep was less than 10^{-4} , and RTDP was judged to have converged when the average time to cross the finish line over 20 episodes appeared to stabilize at an asymptotic number of steps. This version of RTDP updated only the value of the current state on each step.

	DP	RTDP
Average computation to convergence	28 sweeps	4000 episodes
Average number of updates to convergence	252,784	127,600
Average number of updates per episode	—	31.9
% of states updated ≤ 100 times	—	98.45
% of states updated ≤ 10 times	—	80.51
% of states updated 0 times	—	3.18

Both methods produced policies averaging between 14 and 15 steps to cross the finish line, but RTDP required only roughly half of the updates that DP did. This is the result of RTDP’s on-policy trajectory sampling. Whereas the value of every state was updated

in each sweep of DP, RTDP focused updates on fewer states. In an average run, RTDP updated the values of 98.45% of the states no more than 100 times and 80.51% of the states no more than 10 times; the values of about 290 states were not updated at all in an average run. ■

Another advantage of RTDP is that as the value function approaches the optimal value function v_* , the policy used by the agent to generate trajectories approaches an optimal policy because it is always greedy with respect to the current value function. This is in contrast to the situation in conventional value iteration. In practice, value iteration terminates when the value function changes by only a small amount in a sweep, which is how we terminated it to obtain the results in the table above. At this point, the value function closely approximates v_* , and a greedy policy is close to an optimal policy. However, it is possible that policies that are greedy with respect to the latest value function were optimal, or nearly so, long before value iteration terminates. (Recall from Chapter 4 that optimal policies can be greedy with respect to many different value functions, not just v_* .) Checking for the emergence of an optimal policy before value iteration converges is not a part of the conventional DP algorithm and requires considerable additional computation.

In the racetrack example, by running many test episodes after each DP sweep, with actions selected greedily according to the result of that sweep, it was possible to estimate the earliest point in the DP computation at which the approximated optimal evaluation function was good enough so that the corresponding greedy policy was nearly optimal. For this racetrack, a close-to-optimal policy emerged after 15 sweeps of value iteration, or after 136,725 value-iteration updates. This is considerably less than the 252,784 updates DP needed to converge to v_* , but still more than the 127,600 updates RTDP required.

Although these simulations are certainly not definitive comparisons of the RTDP with conventional sweep-based value iteration, they illustrate some of advantages of on-policy trajectory sampling. Whereas conventional value iteration continued to update the value of all the states, RTDP strongly focused on subsets of the states that were relevant to the problem’s objective. This focus became increasingly narrow as learning continued. Because the convergence theorem for RTDP applies to the simulations, we know that RTDP eventually would have focused only on relevant states, i.e., on states making up optimal paths. RTDP achieved nearly optimal control with about 50% of the computation required by sweep-based value iteration.

8.8 Planning at Decision Time

Planning can be used in at least two ways. The one we have considered so far in this chapter, typified by dynamic programming and Dyna, is to use planning to gradually improve a policy or value function on the basis of simulated experience obtained from a model (either a sample or a distribution model). Selecting actions is then a matter of comparing the current state’s action values obtained from a table in the tabular case we have thus far considered, or by evaluating a mathematical expression in the approximate methods we consider in Part II below. Well before an action is selected for any current state S_t , planning has played a part in improving the table entries, or the function

approximation parameters, needed to select actions for many states, including S_t . Used this way, planning is not focused on the current state. We call planning used in this way *background planning*.

The other way to use planning is to begin and complete it *after* encountering each new state S_t , as a computation whose output is the selection of a single action A_t ; on the next step planning begins anew with S_{t+1} to produce A_{t+1} , and so on. The simplest, and almost degenerate, example of this use of planning is when only state values are available, and an action is selected by comparing the values of model-predicted next states for each action (or by comparing the values of afterstates as in the tic-tac-toe example in Chapter 1). More generally, planning used in this way can look much deeper than one-step-ahead and evaluate action choices leading to many different predicted state and reward trajectories. Unlike the first use of planning, here planning focuses on a particular state. We call this *decision-time planning*.

These two ways of thinking about planning—using simulated experience to gradually improve a policy or value function, or using simulated experience to select an action for the current state—can blend together in natural and interesting ways, but they have tended to be studied separately, and that is a good way to first understand them. Let us now take a closer look at decision-time planning.

Even when planning is only done at decision time, we can still view it, as we did in Section 8.1, as proceeding from simulated experience to updates and values, and ultimately to a policy. It is just that now the values and policy are specific to the current state and the action choices available there, so much so that the values and policy created by the planning process are typically discarded after being used to select the current action. In many applications this is not a great loss because there are very many states and we are unlikely to return to the same state for a long time. In general, one may want to do a mix of both: focus planning on the current state *and* store the results of planning so as to be that much farther along should one return to the same state later. Decision-time planning is most useful in applications in which fast responses are not required. In chess playing programs, for example, one may be permitted seconds or minutes of computation for each move, and strong programs may plan dozens of moves ahead within this time. On the other hand, if low latency action selection is the priority, then one is generally better off doing planning in the background to compute a policy that can then be rapidly applied to each newly encountered state.

8.9 Heuristic Search

The classical state-space planning methods in artificial intelligence are decision-time planning methods collectively known as *heuristic search*. In heuristic search, for each state encountered, a large tree of possible continuations is considered. The approximate value function is applied to the leaf nodes and then backed up toward the current state at the root. The backing up within the search tree is just the same as in the expected updates with maxes (those for v_* and q_*) discussed throughout this book. The backing up stops at the state-action nodes for the current state. Once the backed-up values of these nodes are computed, the best of them is chosen as the current action, and then all backed-up values are discarded.

In conventional heuristic search no effort is made to save the backed-up values by changing the approximate value function. In fact, the value function is generally designed by people and never changed as a result of search. However, it is natural to consider allowing the value function to be improved over time, using either the backed-up values computed during heuristic search or any of the other methods presented throughout this book. In a sense we have taken this approach all along. Our greedy, ϵ -greedy, and UCB (Section 2.7) action-selection methods are not unlike heuristic search, albeit on a smaller scale. For example, to compute the greedy action given a model and a state-value function, we must look ahead from each possible action to each possible next state, take into account the rewards and estimated values, and then pick the best action. Just as in conventional heuristic search, this process computes backed-up values of the possible actions, but does not attempt to save them. Thus, heuristic search can be viewed as an extension of the idea of a greedy policy beyond a single step.

The point of searching deeper than one step is to obtain better action selections. If one has a perfect model and an imperfect action-value function, then in fact deeper search will usually yield better policies.² Certainly, if the search is all the way to the end of the episode, then the effect of the imperfect value function is eliminated, and the action determined in this way must be optimal. If the search is of sufficient depth k such that γ^k is very small, then the actions will be correspondingly near optimal. On the other hand, the deeper the search, the more computation is required, usually resulting in a slower response time. A good example is provided by Tesauro's grandmaster-level backgammon player, TD-Gammon (Section 16.1). This system used TD learning to learn an afterstate value function through many games of self-play, using a form of heuristic search to make its moves. As a model, TD-Gammon used a priori knowledge of the probabilities of dice rolls and the assumption that the opponent always selected the actions that TD-Gammon rated as best for it. Tesauro found that the deeper the heuristic search, the better the moves made by TD-Gammon, but the longer it took to make each move. Backgammon has a large branching factor, yet moves must be made within a few seconds. It was only feasible to search ahead selectively a few steps, but even so the search resulted in significantly better action selections.

We should not overlook the most obvious way in which heuristic search focuses updates: on the current state. Much of the effectiveness of heuristic search is due to its search tree being tightly focused on the states and actions that might immediately follow the current state. You may spend more of your life playing chess than checkers, but when you play checkers, it pays to think about checkers and about your particular checkers position, your likely next moves, and successor positions. No matter how you select actions, it is these states and actions that are of highest priority for updates and where you most urgently want your approximate value function to be accurate. Not only should your computation be preferentially devoted to imminent events, but so should your limited memory resources. In chess, for example, there are far too many possible positions to store distinct value estimates for each of them, but chess programs based on heuristic search can easily store distinct estimates for the millions of positions they encounter

²There are interesting exceptions to this (see Pearl, 1984).

looking ahead from a single position. This great focusing of memory and computational resources on the current decision is presumably the reason why heuristic search can be so effective.

The distribution of updates can be altered in similar ways to focus on the current state and its likely successors. As a limiting case we might use exactly the methods of heuristic search to construct a search tree, and then perform the individual, one-step updates from bottom up, as suggested by Figure 8.9. If the updates are ordered in this way and a tabular representation is used, then exactly the same overall update would be achieved as in depth-first heuristic search. Any state-space search can be viewed in this way as the piecing together of a large number of individual one-step updates. Thus, the performance improvement observed with deeper searches is not due to the use of multistep updates as such. Instead, it is due to the focus and concentration of updates on states and actions immediately downstream from the current state. By devoting a large amount of computation specifically relevant to the candidate actions, decision-time planning can produce better decisions than can be produced by relying on unfocused updates.

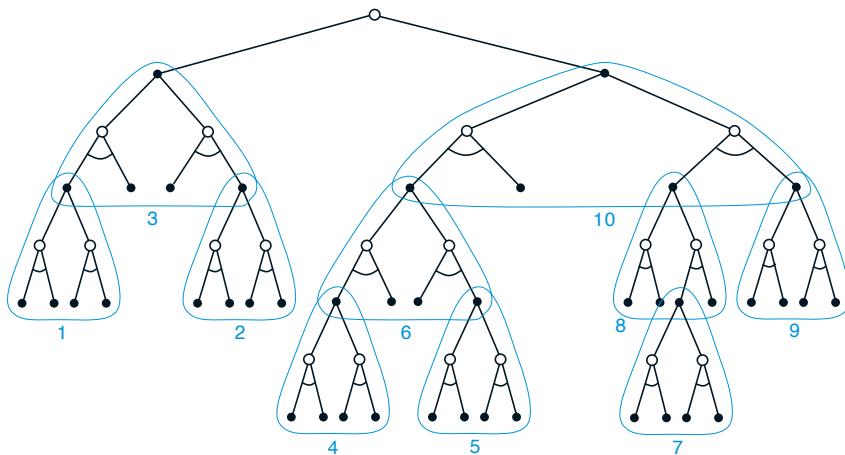


Figure 8.9: Heuristic search can be implemented as a sequence of one-step updates (shown here outlined in blue) backing up values from the leaf nodes toward the root. The ordering shown here is for a selective depth-first search.

8.10 Rollout Algorithms

Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories that all begin at the current environment state. They estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action and then follow the given policy. When the action-value estimates are considered to be accurate enough, the action (or one of the

actions) having the highest estimated value is executed, after which the process is carried out anew from the resulting next state. As explained by Tesauro and Galperin (1997), who experimented with rollout algorithms for playing backgammon, the term “rollout” comes from estimating the value of a backgammon position by playing out, i.e., “rolling out,” the position many times to the game’s end with randomly generated sequences of dice rolls, where the moves of both players are made by some fixed policy.

Unlike the Monte Carlo control algorithms described in Chapter 5, the goal of a rollout algorithm is not to estimate a complete optimal action-value function, q_* , or a complete action-value function, q_π , for a given policy π . Instead, they produce Monte Carlo estimates of action values only for each current state and for a given policy usually called the *rollout policy*. As decision-time planning algorithms, rollout algorithms make immediate use of these action-value estimates, then discard them. This makes rollout algorithms relatively simple to implement because there is no need to sample outcomes for every state-action pair, and there is no need to approximate a function over either the state space or the state-action space.

What then do rollout algorithms accomplish? The policy improvement theorem described in Section 4.2 tells us that given any two policies π and π' that are identical except that $\pi'(s) = a \neq \pi(s)$ for some state s , if $q_\pi(s, a) \geq v_\pi(s)$, then policy π' is as good as, or better, than π . Moreover, if the inequality is strict, then π' is in fact better than π . This applies to rollout algorithms where s is the current state and π is the rollout policy. Averaging the returns of the simulated trajectories produces estimates of $q_\pi(s, a')$ for each action $a' \in \mathcal{A}(s)$. Then the policy that selects an action in s that maximizes these estimates and thereafter follows π is a good candidate for a policy that improves over π . The result is like one step of the policy-iteration algorithm of dynamic programming discussed in Section 4.3 (though it is more like one step of *asynchronous* value iteration described in Section 4.5 because it changes the action for just the current state).

In other words, the aim of a rollout algorithm is to improve upon the rollout policy; not to find an optimal policy. Experience has shown that rollout algorithms can be surprisingly effective. For example, Tesauro and Galperin (1997) were surprised by the dramatic improvements in backgammon playing ability produced by the rollout method. In some applications, a rollout algorithm can produce good performance even if the rollout policy is completely random. But the performance of the improved policy depends on properties of the rollout policy and the ranking of actions produced by the Monte Carlo value estimates. Intuition suggests that the better the rollout policy and the more accurate the value estimates, the better the policy produced by a rollout algorithm is likely be (but see Gelly and Silver, 2007).

This involves important tradeoffs because better rollout policies typically mean that more time is needed to simulate enough trajectories to obtain good value estimates. As decision-time planning methods, rollout algorithms usually have to meet strict time constraints. The computation time needed by a rollout algorithm depends on the number of actions that have to be evaluated for each decision, the number of time steps in the simulated trajectories needed to obtain useful sample returns, the time it takes the rollout policy to make decisions, and the number of simulated trajectories needed to obtain good Monte Carlo action-value estimates.

Balancing these factors is important in any application of rollout methods, though there are several ways to ease the challenge. Because the Monte Carlo trials are independent of one another, it is possible to run many trials in parallel on separate processors. Another approach is to truncate the simulated trajectories short of complete episodes, correcting the truncated returns by means of a stored evaluation function (which brings into play all that we have said about truncated returns and updates in the preceding chapters). It is also possible, as Tesauro and Galperin (1997) suggest, to monitor the Monte Carlo simulations and prune away candidate actions that are unlikely to turn out to be the best, or whose values are close enough to that of the current best that choosing them instead would make no real difference (though Tesauro and Galperin point out that this would complicate a parallel implementation).

We do not ordinarily think of rollout algorithms as *learning* algorithms because they do not maintain long-term memories of values or policies. However, these algorithms take advantage of some of the features of reinforcement learning that we have emphasized in this book. As instances of Monte Carlo control, they estimate action values by averaging the returns of a collection of sample trajectories, in this case trajectories of simulated interactions with a sample model of the environment. In this way they are like reinforcement learning algorithms in avoiding the exhaustive sweeps of dynamic programming by trajectory sampling, and in avoiding the need for distribution models by relying on sample, instead of expected, updates. Finally, rollout algorithms take advantage of the policy improvement property by acting greedily with respect to the estimated action values.

8.11 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a recent and strikingly successful example of decision-time planning. At its base, MCTS is a rollout algorithm as described above, but enhanced by the addition of a means for accumulating value estimates obtained from the Monte Carlo simulations in order to successively direct simulations toward more highly-rewarding trajectories. MCTS is largely responsible for the improvement in computer Go from a weak amateur level in 2005 to a grandmaster level (6 dan or more) in 2015. Many variations of the basic algorithm have been developed, including a variant that we discuss in Section 16.6 that was critical for the stunning 2016 victories of the program AlphaGo over an 18-time world champion Go player. MCTS has proved to be effective in a wide variety of competitive settings, including general game playing (e.g., see Finnsson and Björnsson, 2008; Genesereth and Thielscher, 2014), but it is not limited to games; it can be effective for single-agent sequential decision problems if there is an environment model simple enough for fast multistep simulation.

MCTS is executed after encountering each new state to select the agent’s action for that state; it is executed again to select the action for the next state, and so on. As in a rollout algorithm, each execution is an iterative process that simulates many trajectories starting from the current state and running to a terminal state (or until discounting makes any further reward negligible as a contribution to the return). The core idea of MCTS is to successively focus multiple simulations starting at the current state by

extending the initial portions of trajectories that have received high evaluations from earlier simulations. MCTS does not have to retain approximate value functions or policies from one action selection to the next, though in many implementations it retains selected action values likely to be useful for its next execution.

For the most part, the actions in the simulated trajectories are generated using a simple policy, usually called a rollout policy as it is for simpler rollout algorithms. When both the rollout policy and the model do not require a lot of computation, many simulated trajectories can be generated in a short period of time. As in any tabular Monte Carlo method, the value of a state-action pair is estimated as the average of the (simulated) returns from that pair. Monte Carlo value estimates are maintained only for the subset of state-action pairs that are most likely to be reached in a few steps, which form a tree rooted at the current state, as illustrated in Figure 8.10. MCTS incrementally extends the tree by adding nodes representing states that look promising based on the results of the simulated trajectories. Any simulated trajectory will pass through the tree and then exit it at some leaf node. Outside the tree and at the leaf nodes the rollout policy is used for action selections, but at the states inside the tree something better is possible. For these states we have value estimates for at least some of the actions, so we can pick among them using an informed policy, called the *tree policy*, that balances exploration

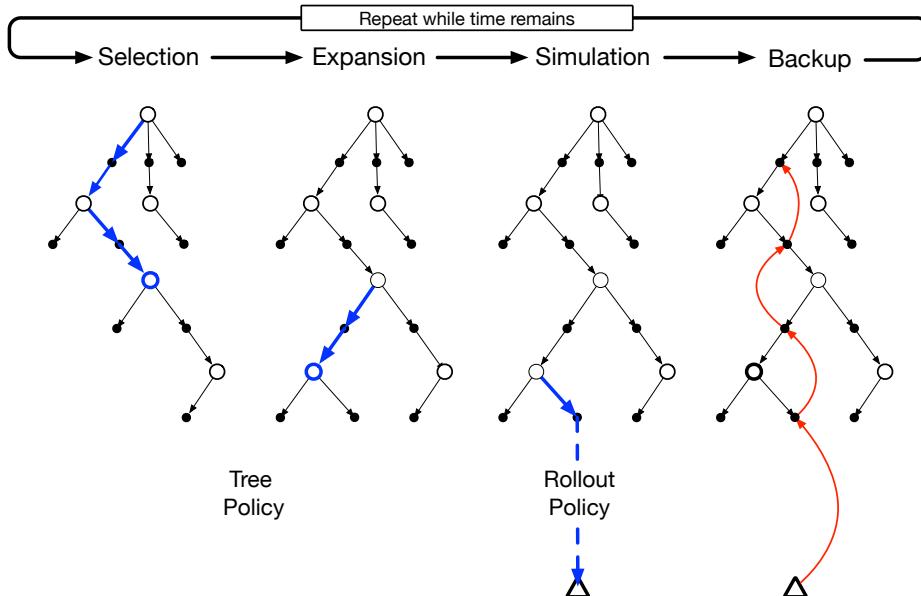


Figure 8.10: Monte Carlo Tree Search. When the environment changes to a new state, MCTS executes as many iterations as possible before an action needs to be selected, incrementally building a tree whose root node represents the current state. Each iteration consists of the four operations **Selection**, **Expansion** (though possibly skipped on some iterations), **Simulation**, and **Backup**, as explained in the text and illustrated by the bold arrows in the trees. Adapted from Chaslot, Bakkes, Szita, and Spronck (2008).

and exploitation. For example, the tree policy could select actions using an ε -greedy or UCB selection rule (Chapter 2).

In more detail, each iteration of a basic version of MCTS consists of the following four steps as illustrated in Figure 8.10:

1. **Selection.** Starting at the root node, a *tree policy* based on the action values attached to the edges of the tree traverses the tree to select a leaf node.
2. **Expansion.** On some iterations (depending on details of the application), the tree is expanded from the selected leaf node by adding one or more child nodes reached from the selected node via unexplored actions.
3. **Simulation.** From the selected node, or from one of its newly-added child nodes (if any), simulation of a complete episode is run with actions selected by the rollout policy. The result is a Monte Carlo trial with actions selected first by the tree policy and beyond the tree by the rollout policy.
4. **Backup.** The return generated by the simulated episode is backed up to update, or to initialize, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS. No values are saved for the states and actions visited by the rollout policy beyond the tree. Figure 8.10 illustrates this by showing a backup from the terminal state of the simulated trajectory directly to the state-action node in the tree where the rollout policy began (though in general, the entire return over the simulated trajectory is backed up to this state-action node).

MCTS continues executing these four steps, starting each time at the tree’s root node, until no more time is left, or some other computational resource is exhausted. Then, finally, an action from the root node (which still represents the current state of the environment) is selected according to some mechanism that depends on the accumulated statistics in the tree; for example, it may be an action having the largest action value of all the actions available from the root state, or perhaps the action with the largest visit count to avoid selecting outliers. This is the action MCTS actually selects. After the environment transitions to a new state, MCTS is run again, sometimes starting with a tree of a single root node representing the new state, but often starting with a tree containing any descendants of this node left over from the tree constructed by the previous execution of MCTS; all the remaining nodes are discarded, along with the action values associated with them.

MCTS was first proposed to select moves in programs playing two-person competitive games, such as Go. For game playing, each simulated episode is one complete play of the game in which both players select actions by the tree and rollout policies. Section 16.6 describes an extension of MCTS used in the AlphaGo program that combines the Monte Carlo evaluations of MCTS with action values learned by a deep artificial neural network via self-play reinforcement learning.

Relating MCTS to the reinforcement learning principles we describe in this book provides some insight into how it achieves such impressive results. At its base, MCTS is a decision-time planning algorithm based on Monte Carlo control applied to simulations

that start from the root state; that is, it is a kind of rollout algorithm as described in the previous section. It therefore benefits from online, incremental, sample-based value estimation and policy improvement. Beyond this, it saves action-value estimates attached to the tree edges and updates them using reinforcement learning’s sample updates. This has the effect of focusing the Monte Carlo trials on trajectories whose initial segments are common to high-return trajectories previously simulated. Further, by incrementally expanding the tree, MCTS effectively grows a lookup table to store a partial action-value function, with memory allocated to the estimated values of state-action pairs visited in the initial segments of high-yielding sample trajectories. MCTS thus avoids the problem of globally approximating an action-value function while it retains the benefit of using past experience to guide exploration.

The striking success of decision-time planning by MCTS has deeply influenced artificial intelligence, and many researchers are studying modifications and extensions of the basic procedure for use in both games and single-agent applications.

8.12 Summary of the Chapter

Planning requires a model of the environment. A *distribution model* consists of the probabilities of next states and rewards for possible actions; a sample model produces single transitions and rewards generated according to these probabilities. Dynamic programming requires a distribution model because it uses *expected updates*, which involve computing expectations over all the possible next states and rewards. A *sample model*, on the other hand, is what is needed to simulate interacting with the environment during which *sample updates*, like those used by many reinforcement learning algorithms, can be used. Sample models are generally much easier to obtain than distribution models.

We have presented a perspective emphasizing the surprisingly close relationships between planning optimal behavior and learning optimal behavior. Both involve estimating the same value functions, and in both cases it is natural to update the estimates incrementally, in a long series of small backing-up operations. This makes it straightforward to integrate learning and planning processes simply by allowing both to update the same estimated value function. In addition, any of the learning methods can be converted into planning methods simply by applying them to simulated (model-generated) experience rather than to real experience. In this case learning and planning become even more similar; they are possibly identical algorithms operating on two different sources of experience.

It is straightforward to integrate incremental planning methods with acting and model-learning. Planning, acting, and model-learning interact in a circular fashion (as in the diagram on page 162), each producing what the other needs to improve; no other interaction among them is either required or prohibited. The most natural approach is for all processes to proceed asynchronously and in parallel. If the processes must share computational resources, then the division can be handled almost arbitrarily—by whatever organization is most convenient and efficient for the task at hand.

In this chapter we have touched upon a number of dimensions of variation among state-space planning methods. One dimension is the variation in the size of updates. The

smaller the updates, the more incremental the planning methods can be. Among the smallest updates are one-step sample updates, as in Dyna. Another important dimension is the distribution of updates, that is, of the focus of search. Prioritized sweeping focuses backward on the predecessors of states whose values have recently changed. On-policy trajectory sampling focuses on states or state-action pairs that the agent is likely to encounter when controlling its environment. This can allow computation to skip over parts of the state space that are irrelevant to the prediction or control problem. Real-time dynamic programming, an on-policy trajectory sampling version of value iteration, illustrates some of the advantages this strategy has over conventional sweep-based policy iteration.

Planning can also focus forward from pertinent states, such as states actually encountered during an agent-environment interaction. The most important form of this is when planning is done at decision time, that is, as part of the action-selection process. Classical heuristic search as studied in artificial intelligence is an example of this. Other examples are rollout algorithms and Monte Carlo Tree Search that benefit from online, incremental, sample-based value estimation and policy improvement.

8.13 Summary of Part I: Dimensions

This chapter concludes Part I of this book. In it we have tried to present reinforcement learning not as a collection of individual methods, but as a coherent set of ideas cutting across methods. Each idea can be viewed as a dimension along which methods vary. The set of such dimensions spans a large space of possible methods. By exploring this space at the level of dimensions we hope to obtain the broadest and most lasting understanding. In this section we use the concept of dimensions in method space to recapitulate the view of reinforcement learning developed so far in this book.

All of the methods we have explored so far in this book have three key ideas in common: first, they all seek to estimate value functions; second, they all operate by backing up values along actual or possible state trajectories; and third, they all follow the general strategy of generalized policy iteration (GPI), meaning that they maintain an approximate value function and an approximate policy, and they continually try to improve each on the basis of the other. These three ideas are central to the subjects covered in this book. We suggest that value functions, backing up value updates, and GPI are powerful organizing principles potentially relevant to any model of intelligence, whether artificial or natural.

Two of the most important dimensions along which the methods vary are shown in Figure 8.11. These dimensions have to do with the kind of update used to improve the value function. The horizontal dimension is whether they are sample updates (based on a sample trajectory) or expected updates (based on a distribution of possible trajectories). Expected updates require a distribution model, whereas sample updates need only a sample model, or can be done from actual experience with no model at all (another dimension of variation). The vertical dimension of Figure 8.11 corresponds to the depth of updates, that is, to the degree of bootstrapping. At three of the four corners of the space are the three primary methods for estimating values: dynamic programming, TD, and Monte Carlo. Along the left edge of the space are the sample-update methods,

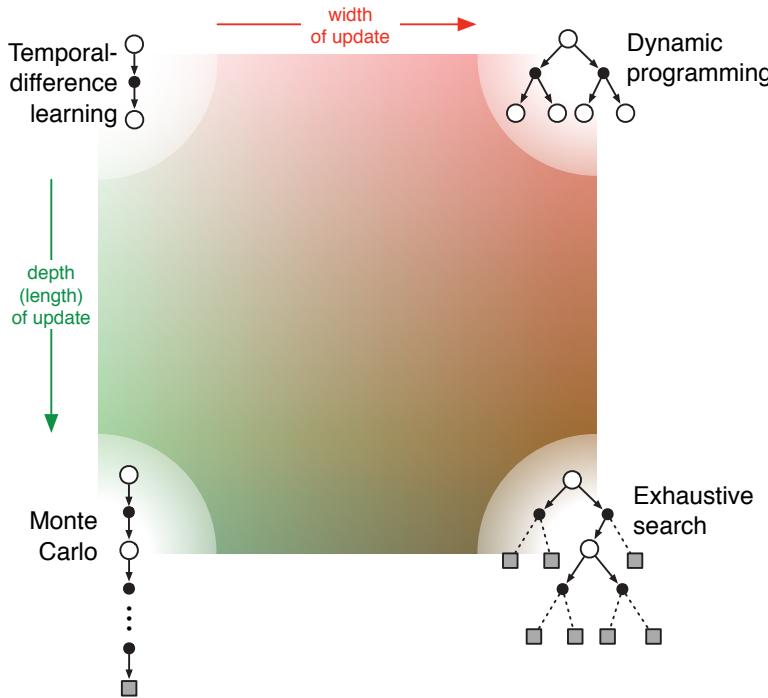


Figure 8.11: A slice through the space of reinforcement learning methods, highlighting the two of the most important dimensions explored in Part I of this book: the depth and width of the updates.

ranging from one-step TD updates to full-return Monte Carlo updates. Between these is a spectrum including methods based on n -step updates (and in Chapter 12 we will extend this to mixtures of n -step updates such as the λ -updates implemented by eligibility traces).

Dynamic programming methods are shown in the extreme upper-right corner of the space because they involve one-step expected updates. The lower-right corner is the extreme case of expected updates so deep that they run all the way to terminal states (or, in a continuing task, until discounting has reduced the contribution of any further rewards to a negligible level). This is the case of exhaustive search. Intermediate methods along this dimension include heuristic search and related methods that search and update up to a limited depth, perhaps selectively. There are also methods that are intermediate along the horizontal dimension. These include methods that mix expected and sample updates, as well as the possibility of methods that mix samples and distributions within a single update. The interior of the square is filled in to represent the space of all such intermediate methods.

A third dimension that we have emphasized in this book is the binary distinction between on-policy and off-policy methods. In the former case, the agent learns the value function for the policy it is currently following, whereas in the latter case it learns the

value function for the policy for a different policy, often the one that the agent currently thinks is best. The policy generating behavior is typically different from what is currently thought best because of the need to explore. This third dimension might be visualized as perpendicular to the plane of the page in Figure 8.11.

In addition to the three dimensions just discussed, we have identified a number of others throughout the book:

Definition of return Is the task episodic or continuing, discounted or undiscounted?

Action values vs. state values vs. afterstate values What kind of values should be estimated? If only state values are estimated, then either a model or a separate policy (as in actor-critic methods) is required for action selection.

Action selection/exploration How are actions selected to ensure a suitable trade-off between exploration and exploitation? We have considered only the simplest ways to do this: ϵ -greedy, optimistic initialization of values, soft-max, and upper confidence bound.

Synchronous vs. asynchronous Are the updates for all states performed simultaneously or one by one in some order?

Real vs. simulated Should one update based on real experience or simulated experience? If both, how much of each?

Location of updates What states or state-action pairs should be updated? Model-free methods can choose only among the states and state-action pairs actually encountered, but model-based methods can choose arbitrarily. There are many possibilities here.

Timing of updates Should updates be done as part of selecting actions, or only afterward?

Memory for updates How long should updated values be retained? Should they be retained permanently, or only while computing an action selection, as in heuristic search?

Of course, these dimensions are neither exhaustive nor mutually exclusive. Individual algorithms differ in many other ways as well, and many algorithms lie in several places along several dimensions. For example, Dyna methods use both real and simulated experience to affect the same value function. It is also perfectly sensible to maintain multiple value functions computed in different ways or over different state and action representations. These dimensions do, however, constitute a coherent set of ideas for describing and exploring a wide space of possible methods.

The most important dimension not mentioned here, and not covered in Part I of this book, is that of function approximation. Function approximation can be viewed as an orthogonal spectrum of possibilities ranging from tabular methods at one extreme through state aggregation, a variety of linear methods, and then a diverse set of nonlinear methods. This dimension is explored in Part II.

Bibliographical and Historical Remarks

- 8.1** The overall view of planning and learning presented here has developed gradually over a number of years, in part by the authors (Sutton, 1990, 1991a, 1991b; Barto, Bradtke, and Singh, 1991, 1995; Sutton and Pinette, 1985; Sutton and Barto, 1981b); it has been strongly influenced by Agre and Chapman (1990; Agre 1988), Bertsekas and Tsitsiklis (1989), Singh (1993), and others. The authors were also strongly influenced by psychological studies of latent learning (Tolman, 1932) and by psychological views of the nature of thought (e.g., Galanter and Gerstenhaber, 1956; Craik, 1943; Campbell, 1960; Dennett, 1978). In Part III of the book, Section 14.6 relates model-based and model-free methods to psychological theories of learning and behavior, and Section 15.11 discusses ideas about how the brain might implement these types of methods.
- 8.2** The terms *direct* and *indirect*, which we use to describe different kinds of reinforcement learning, are from the adaptive control literature (e.g., Goodwin and Sin, 1984), where they are used to make the same kind of distinction. The term *system identification* is used in adaptive control for what we call *model-learning* (e.g., Goodwin and Sin, 1984; Ljung and Söderstrom, 1983; Young, 1984). The Dyna architecture is due to Sutton (1990), and the results in this and the next section are based on results reported there. Barto and Singh (1990) consider some of the issues in comparing direct and indirect reinforcement learning methods. Early work extending Dyna to linear function approximation was done by Sutton, Szepesvári, Geramifard, and Bowling (2008) and by Parr, Li, Taylor, Painter-Wakefield, and Littman (2008).
- 8.3** There have been several works with model-based reinforcement learning that take the idea of exploration bonuses and optimistic initialization to its logical extreme, in which all incompletely explored choices are assumed maximally rewarding and optimal paths are computed to test them. The E³ algorithm of Kearns and Singh (2002) and the R-max algorithm of Brafman and Tennenholtz (2003) are guaranteed to find a near-optimal solution in time polynomial in the number of states and actions. This is usually too slow for practical algorithms but is probably the best that can be done in the worst case.
- 8.4** Prioritized sweeping was developed simultaneously and independently by Moore and Atkeson (1993) and Peng and Williams (1993). The results in the box on page 170 are due to Peng and Williams (1993). The results in the box on page 171 are due to Moore and Atkeson. Key subsequent work in this area includes that by McMahan and Gordon (2005) and by van Seijen and Sutton (2013).
- 8.5** This section was strongly influenced by the experiments of Singh (1993).
- 8.6–7** Trajectory sampling has implicitly been a part of reinforcement learning from the outset, but it was most explicitly emphasized by Barto, Bradtke, and Singh (1995) in their introduction of RTDP. They recognized that Korf's (1990) *learning*

*real-time A** (LRTA*) algorithm is an asynchronous DP algorithm that applies to stochastic problems as well as the deterministic problems on which Korf focused. Beyond LRTA*, RTDP includes the option of updating the values of many states in the time intervals between the execution of actions. Barto et al. (1995) proved the convergence result described here by combining Korf’s (1990) convergence proof for LRTA* with the result of Bertsekas (1982) (also Bertsekas and Tsitsiklis, 1989) ensuring convergence of asynchronous DP for stochastic shortest path problems in the undiscounted case. Combining model-learning with RTDP is called *Adaptive* RTDP, also presented by Barto et al. (1995) and discussed by Barto (2011).

- 8.9** For further reading on heuristic search, the reader is encouraged to consult texts and surveys such as those by Russell and Norvig (2009) and Korf (1988). Peng and Williams (1993) explored a forward focusing of updates much as is suggested in this section.
- 8.10** Abramson’s (1990) expected-outcome model is a rollout algorithm applied to two-person games in which the play of both simulated players is random. He argued that even with random play, it is a “powerful heuristic” that is “precise, accurate, easily estimable, efficiently calculable, and domain-independent.” Tesauro and Galperin (1997) demonstrated the effectiveness of rollout algorithms for improving the play of backgammon programs, adopting the term “rollout” from its use in evaluating backgammon positions by playing out positions with different randomly generating sequences of dice rolls. Bertsekas, Tsitsiklis, and Wu (1997) examine rollout algorithms applied to combinatorial optimization problems, and Bertsekas (2013) surveys their use in discrete deterministic optimization problems, remarking that they are “often surprisingly effective.”
- 8.11** The central ideas of MCTS were introduced by Coulom (2006) and by Kocsis and Szepesvári (2006). They built upon previous research with Monte Carlo planning algorithms as reviewed by these authors. Browne, Powley, Whitehouse, Lucas, Cowling, Rohlfsen, Tavener, Perez, Samothrakis, and Colton (2012) is an excellent survey of MCTS methods and their applications. David Silver contributed to the ideas and presentation in this section.

Part II: Approximate Solution Methods

In the second part of the book we extend the tabular methods presented in the first part to apply to problems with arbitrarily large state spaces. In many of the tasks to which we would like to apply reinforcement learning the state space is combinatorial and enormous; the number of possible camera images, for example, is much larger than the number of atoms in the universe. In such cases we cannot expect to find an optimal policy or the optimal value function even in the limit of infinite time and data; our goal instead is to find a good approximate solution using limited computational resources. In this part of the book we explore such approximate solution methods.

The problem with large state spaces is not just the memory needed for large tables, but the time and data needed to fill them accurately. In many of our target tasks, almost every state encountered will never have been seen before. To make sensible decisions in such states it is necessary to generalize from previous encounters with different states that are in some sense similar to the current one. In other words, the key issue is that of *generalization*. How can experience with a limited subset of the state space be usefully generalized to produce a good approximation over a much larger subset?

Fortunately, generalization from examples has already been extensively studied, and we do not need to invent totally new methods for use in reinforcement learning. To some extent we need only combine reinforcement learning methods with existing generalization methods. The kind of generalization we require is often called *function approximation* because it takes examples from a desired function (e.g., a value function) and attempts to generalize from them to construct an approximation of the entire function. Function approximation is an instance of *supervised learning*, the primary topic studied in machine learning, artificial neural networks, pattern recognition, and statistical curve fitting. In theory, any of the methods studied in these fields can be used in the role of function approximator within reinforcement learning algorithms, although in practice some fit more easily into this role than others.

Reinforcement learning with function approximation involves a number of new issues that do not normally arise in conventional supervised learning, such as nonstationarity, bootstrapping, and delayed targets. We introduce these and other issues successively over the five chapters of this part. Initially we restrict attention to on-policy training, treating in Chapter 9 the prediction case, in which the policy is given and only its value function is approximated, and then in Chapter 10 the control case, in which an approximation to the optimal policy is found. The challenging problem of off-policy learning with function approximation is treated in Chapter 11. In each of these three chapters we will have

to return to first principles and re-examine the objectives of the learning to take into account function approximation. Chapter 12 introduces and analyzes the algorithmic mechanism of *eligibility traces*, which dramatically improves the computational properties of multi-step reinforcement learning methods in many cases. The final chapter of this part explores a different approach to control, *policy-gradient methods*, which approximate the optimal policy directly and need never form an approximate value function (although they may be much more efficient if they do approximate a value function as well the policy).

Chapter 9

On-policy Prediction with Approximation

In this chapter, we begin our study of function approximation in reinforcement learning by considering its use in estimating the state-value function from on-policy data, that is, in approximating v_π from experience generated using a known policy π . The novelty in this chapter is that the approximate value function is represented not as a table but as a parameterized functional form with weight vector $\mathbf{w} \in \mathbb{R}^d$. We will write $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ for the approximate value of state s given weight vector \mathbf{w} . For example, \hat{v} might be a linear function in features of the state, with \mathbf{w} the vector of feature weights. More generally, \hat{v} might be the function computed by a multi-layer artificial neural network, with \mathbf{w} the vector of connection weights in all the layers. By adjusting the weights, any of a wide range of different functions can be implemented by the network. Or \hat{v} might be the function computed by a decision tree, where \mathbf{w} is all the numbers defining the split points and leaf values of the tree. Typically, the number of weights (the dimensionality of \mathbf{w}) is much less than the number of states ($d \ll |\mathcal{S}|$), and changing one weight changes the estimated value of many states. Consequently, when a single state is updated, the change generalizes from that state to affect the values of many other states. Such *generalization* makes the learning potentially more powerful but also potentially more difficult to manage and understand.

Perhaps surprisingly, extending reinforcement learning to function approximation also makes it applicable to partially observable problems, in which the full state is not available to the agent. If the parameterized function form for \hat{v} does not allow the estimated value to depend on certain aspects of the state, then it is just as if those aspects are unobservable. In fact, all the theoretical results for methods using function approximation presented in this part of the book apply equally well to cases of partial observability. What function approximation can't do, however, is augment the state representation with memories of past observations. Some such possible further extensions are discussed briefly in Section 17.3.

9.1 Value-function Approximation

All of the prediction methods covered in this book have been described as updates to an estimated value function that shift its value at particular states toward a “backed-up value,” or *update target*, for that state. Let us refer to an individual update by the notation $s \mapsto u$, where s is the state updated and u is the update target that s ’s estimated value is shifted toward. For example, the Monte Carlo update for value prediction is $S_t \mapsto G_t$, the TD(0) update is $S_t \mapsto R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t)$, and the n -step TD update is $S_t \mapsto G_{t:t+n}$. In the DP (dynamic programming) policy-evaluation update, $s \mapsto \mathbb{E}_{\pi}[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) \mid S_t = s]$, an arbitrary state s is updated, whereas in the other cases the state encountered in actual experience, S_t , is updated.

It is natural to interpret each update as specifying an example of the desired input–output behavior of the value function. In a sense, the update $s \mapsto u$ means that the estimated value for state s should be more like the update target u . Up to now, the actual update has been trivial: the table entry for s ’s estimated value has simply been shifted a fraction of the way toward u , and the estimated values of all other states were left unchanged. Now we permit arbitrarily complex and sophisticated methods to implement the update, and updating at s generalizes so that the estimated values of many other states are changed as well. Machine learning methods that learn to mimic input–output examples in this way are called *supervised learning* methods, and when the outputs are numbers, like u , the process is often called *function approximation*. Function approximation methods expect to receive examples of the desired input–output behavior of the function they are trying to approximate. We use these methods for value prediction simply by passing to them the $s \mapsto u$ of each update as a training example. We then interpret the approximate function they produce as an estimated value function.

Viewing each update as a conventional training example in this way enables us to use any of a wide range of existing function approximation methods for value prediction. In principle, we can use any method for supervised learning from examples, including artificial neural networks, decision trees, and various kinds of multivariate regression. However, not all function approximation methods are equally well suited for use in reinforcement learning. The most sophisticated artificial neural network and statistical methods all assume a static training set over which multiple passes are made. In reinforcement learning, however, it is important that learning be able to occur online, while the agent interacts with its environment or with a model of its environment. To do this requires methods that are able to learn efficiently from incrementally acquired data. In addition, reinforcement learning generally requires function approximation methods able to handle nonstationary target functions (target functions that change over time). For example, in control methods based on GPI (generalized policy iteration) we often seek to learn q_{π} while π changes. Even if the policy remains the same, the target values of training examples are nonstationary if they are generated by bootstrapping methods (DP and TD learning). Methods that cannot easily handle such nonstationarity are less suitable for reinforcement learning.

9.2 The Prediction Objective (\overline{VE})

Up to now we have not specified an explicit objective for prediction. In the tabular case a continuous measure of prediction quality was not necessary because the learned value function could come to equal the true value function exactly. Moreover, the learned values at each state were decoupled—an update at one state affected no other. But with genuine approximation, an update at one state affects many others, and it is not possible to get the values of all states exactly correct. By assumption we have far more states than weights, so making one state's estimate more accurate invariably means making others' less accurate. We are obligated then to say which states we care most about. We must specify a state distribution $\mu(s) \geq 0$, $\sum_s \mu(s) = 1$, representing how much we care about the error in each state s . By the error in a state s we mean the square of the difference between the approximate value $\hat{v}(s, \mathbf{w})$ and the true value $v_\pi(s)$. Weighting this over the state space by μ , we obtain a natural objective function, the *mean square value error*, denoted \overline{VE} :

$$\overline{VE}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2. \quad (9.1)$$

The square root of this measure, the root \overline{VE} , gives a rough measure of how much the approximate values differ from the true values and is often used in plots. Often $\mu(s)$ is chosen to be the fraction of time spent in s . Under on-policy training this is called the *on-policy distribution*; we focus entirely on this case in this chapter. In continuing tasks, the on-policy distribution is the stationary distribution under π .

The on-policy distribution in episodic tasks

In an episodic task, the on-policy distribution is a little different in that it depends on how the initial states of episodes are chosen. Let $h(s)$ denote the probability that an episode begins in each state s , and let $\eta(s)$ denote the number of time steps spent, on average, in state s in a single episode. Time is spent in a state s if episodes start in s , or if transitions are made into s from a preceding state \bar{s} in which time is spent:

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(\bar{s}) \sum_a \pi(a|\bar{s}) p(s|\bar{s}, a), \quad \text{for all } s \in \mathcal{S}. \quad (9.2)$$

This system of equations can be solved for the expected number of visits $\eta(s)$. The on-policy distribution is then the fraction of time spent in each state normalized to sum to one:

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}, \quad \text{for all } s \in \mathcal{S}. \quad (9.3)$$

This is the natural choice without discounting. If there is discounting ($\gamma < 1$) it should be treated as a form of termination, which can be done simply by including a factor of γ in the second term of (9.2).

The two cases, continuing and episodic, behave similarly, but with approximation they must be treated separately in formal analyses, as we will see repeatedly in this part of the book. This completes the specification of the learning objective.

It is not completely clear that the \overline{VE} is the right performance objective for reinforcement learning. Remember that our ultimate purpose—the reason we are learning a value function—is to find a better *policy*. The best value function for this purpose is not necessarily the best for minimizing \overline{VE} . Nevertheless, it is not yet clear what a more useful alternative goal for value prediction might be. For now, we will focus on \overline{VE} .

An ideal goal in terms of \overline{VE} would be to find a *global optimum*, a weight vector \mathbf{w}^* for which $\overline{VE}(\mathbf{w}^*) \leq \overline{VE}(\mathbf{w})$ for all possible \mathbf{w} . Reaching this goal is sometimes possible for simple function approximators such as linear ones, but is rarely possible for complex function approximators such as artificial neural networks and decision trees. Short of this, complex function approximators may seek to converge instead to a *local optimum*, a weight vector \mathbf{w}^* for which $\overline{VE}(\mathbf{w}^*) \leq \overline{VE}(\mathbf{w})$ for all \mathbf{w} in some neighborhood of \mathbf{w}^* . Although this guarantee is only slightly reassuring, it is typically the best that can be said for nonlinear function approximators, and often it is enough. Still, for many cases of interest in reinforcement learning there is no guarantee of convergence to an optimum, or even to within a bounded distance of an optimum. Some methods may in fact diverge, with their \overline{VE} approaching infinity in the limit.

In the last two sections we outlined a framework for combining a wide range of reinforcement learning methods for value prediction with a wide range of function approximation methods, using the updates of the former to generate training examples for the latter. We also described a \overline{VE} performance measure which these methods may aspire to minimize. The range of possible function approximation methods is far too large to cover all, and anyway too little is known about most of them to make a reliable evaluation or recommendation. Of necessity, we consider only a few possibilities. In the rest of this chapter we focus on function approximation methods based on gradient principles, and on linear gradient-descent methods in particular. We focus on these methods in part because we consider them to be particularly promising and because they reveal key theoretical issues, but also because they are simple and our space is limited.

9.3 Stochastic-gradient and Semi-gradient Methods

We now develop in detail one class of learning methods for function approximation in value prediction, those based on stochastic gradient descent (SGD). SGD methods are among the most widely used of all function approximation methods and are particularly well suited to online reinforcement learning.

In gradient-descent methods, the weight vector is a column vector with a fixed number of real valued components, $\mathbf{w} \doteq (w_1, w_2, \dots, w_d)^\top$,¹ and the approximate value function $\hat{v}(s, \mathbf{w})$ is a differentiable function of \mathbf{w} for all $s \in \mathcal{S}$. We will be updating \mathbf{w} at each of a series of discrete time steps, $t = 0, 1, 2, 3, \dots$, so we will need a notation \mathbf{w}_t for the

¹The $^\top$ denotes transpose, needed here to turn the horizontal row vector in the text into a vertical column vector; in this book vectors are generally taken to be column vectors unless explicitly written out horizontally or transposed.

weight vector at each step. For now, let us assume that, on each step, we observe a new example $S_t \mapsto v_\pi(S_t)$ consisting of a (possibly randomly selected) state S_t and its true value under the policy. These states might be successive states from an interaction with the environment, but for now we do not assume so. Even though we are given the exact, correct values, $v_\pi(S_t)$ for each S_t , there is still a difficult problem because our function approximator has limited resources and thus limited resolution. In particular, there is generally no \mathbf{w} that gets all the states, or even all the examples, exactly correct. In addition, we must generalize to all the other states that have not appeared in examples.

We assume that states appear in examples with the same distribution, μ , over which we are trying to minimize the $\overline{\text{VE}}$ as given by (9.1). A good strategy in this case is to try to minimize error on the observed examples. *Stochastic gradient-descent* (SGD) methods do this by adjusting the weight vector after each example by a small amount in the direction that would most reduce the error on that example:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t - \frac{1}{2}\alpha \nabla \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right]^2 \quad (9.4)$$

$$= \mathbf{w}_t + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t), \quad (9.5)$$

where α is a positive step-size parameter, and $\nabla f(\mathbf{w})$, for any scalar expression $f(\mathbf{w})$ that is a function of a vector (here \mathbf{w}), denotes the column vector of partial derivatives of the expression with respect to the components of the vector:

$$\nabla f(\mathbf{w}) \doteq \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \dots, \frac{\partial f(\mathbf{w})}{\partial w_d} \right)^\top. \quad (9.6)$$

This derivative vector is the *gradient* of f with respect to \mathbf{w} . SGD methods are “gradient descent” methods because the overall step in \mathbf{w}_t is proportional to the negative gradient of the example’s squared error (9.4). This is the direction in which the error falls most rapidly. Gradient descent methods are called “stochastic” when the update is done, as here, on only a single example, which might have been selected stochastically. Over many examples, making small steps, the overall effect is to minimize an average performance measure such as the $\overline{\text{VE}}$.

It may not be immediately apparent why SGD takes only a small step in the direction of the gradient. Could we not move all the way in this direction and completely eliminate the error on the example? In many cases this could be done, but usually it is not desirable. Remember that we do not seek or expect to find a value function that has zero error for all states, but only an approximation that balances the errors in different states. If we completely corrected each example in one step, then we would not find such a balance. In fact, the convergence results for SGD methods assume that α decreases over time. If it decreases in such a way as to satisfy the standard stochastic approximation conditions (2.7), then the SGD method (9.5) is guaranteed to converge to a local optimum.

We turn now to the case in which the target output, here denoted $U_t \in \mathbb{R}$, of the t th training example, $S_t \mapsto U_t$, is not the true value, $v_\pi(S_t)$, but some, possibly random, approximation to it. For example, U_t might be a noise-corrupted version of $v_\pi(S_t)$, or it might be one of the bootstrapping targets using \hat{v} mentioned in the previous section. In

in these cases we cannot perform the exact update (9.5) because $v_\pi(S_t)$ is unknown, but we can approximate it by substituting U_t in place of $v_\pi(S_t)$. This yields the following general SGD method for state-value prediction:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha [U_t - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(S_t, \mathbf{w}_t). \quad (9.7)$$

If U_t is an *unbiased* estimate, that is, if $\mathbb{E}[U_t | S_t = s] = v_\pi(s)$, for each t , then \mathbf{w}_t is guaranteed to converge to a local optimum under the usual stochastic approximation conditions (2.7) for decreasing α .

For example, suppose the states in the examples are the states generated by interaction (or simulated interaction) with the environment using policy π . Because the true value of a state is the expected value of the return following it, the Monte Carlo target $U_t \doteq G_t$ is by definition an unbiased estimate of $v_\pi(S_t)$. With this choice, the general SGD method (9.7) converges to a locally optimal approximation to $v_\pi(S_t)$. Thus, the gradient-descent version of Monte Carlo state-value prediction is guaranteed to find a locally optimal solution. Pseudocode for a complete algorithm is shown in the box below.

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

```

Input: the policy  $\pi$  to be evaluated
Input: a differentiable function  $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$ 
Algorithm parameter: step size  $\alpha > 0$ 
Initialize value-function weights  $\mathbf{w} \in \mathbb{R}^d$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )
Loop forever (for each episode):
  Generate an episode  $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$  using  $\pi$ 
  Loop for each step of episode,  $t = 0, 1, \dots, T - 1$ :
     $\mathbf{w} \leftarrow \mathbf{w} + \alpha [G_t - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$ 

```

One does not obtain the same guarantees if a bootstrapping estimate of $v_\pi(S_t)$ is used as the target U_t in (9.7). Bootstrapping targets such as n -step returns $G_{t:t+n}$ or the DP target $\sum_{a,s',r} \pi(a|S_t)p(s',r|S_t,a)[r + \gamma \hat{v}(s',\mathbf{w}_t)]$ all depend on the current value of the weight vector \mathbf{w}_t , which implies that they will be biased and that they will not produce a true gradient-descent method. One way to look at this is that the key step from (9.4) to (9.5) relies on the target being independent of \mathbf{w}_t . This step would not be valid if a bootstrapping estimate were used in place of $v_\pi(S_t)$. Bootstrapping methods are not in fact instances of true gradient descent (Barnard, 1993). They take into account the effect of changing the weight vector \mathbf{w}_t on the estimate, but ignore its effect on the target. They include only a part of the gradient and, accordingly, we call them *semi-gradient methods*.

Although semi-gradient (bootstrapping) methods do not converge as robustly as gradient methods, they do converge reliably in important cases such as the linear case discussed in the next section. Moreover, they offer important advantages that make them often clearly preferred. One reason for this is that they typically enable significantly faster learning, as we have seen in Chapters 6 and 7. Another is that they enable learning to

be continual and online, without waiting for the end of an episode. This enables them to be used on continuing problems and provides computational advantages. A prototypical semi-gradient method is semi-gradient TD(0), which uses $U_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ as its target. Complete pseudocode for this method is given in the box below.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

```

Input: the policy  $\pi$  to be evaluated
Input: a differentiable function  $\hat{v} : S^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$  such that  $\hat{v}(\text{terminal}, \cdot) = 0$ 
Algorithm parameter: step size  $\alpha > 0$ 
Initialize value-function weights  $\mathbf{w} \in \mathbb{R}^d$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )
Loop for each episode:
    Initialize  $S$ 
    Loop for each step of episode:
        Choose  $A \sim \pi(\cdot | S)$ 
        Take action  $A$ , observe  $R, S'$ 
         $\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})] \nabla \hat{v}(S, \mathbf{w})$ 
         $S \leftarrow S'$ 
    until  $S$  is terminal

```

State aggregation is a simple form of generalizing function approximation in which states are grouped together, with one estimated value (one component of the weight vector \mathbf{w}) for each group. The value of a state is estimated as its group's component, and when the state is updated, that component alone is updated. State aggregation is a special case of SGD (9.7) in which the gradient, $\nabla \hat{v}(S_t, \mathbf{w}_t)$, is 1 for S_t 's group's component and 0 for the other components.

Example 9.1: State Aggregation on the 1000-state Random Walk Consider a 1000-state version of the random walk task (Examples 6.2 and 7.1 on pages 125 and 144). The states are numbered from 1 to 1000, left to right, and all episodes begin near the center, in state 500. State transitions are from the current state to one of the 100 neighboring states to its left, or to one of the 100 neighboring states to its right, all with equal probability. Of course, if the current state is near an edge, then there may be fewer than 100 neighbors on that side of it. In this case, all the probability that would have gone into those missing neighbors goes into the probability of terminating on that side (thus, state 1 has a 0.5 chance of terminating on the left, and state 950 has a 0.25 chance of terminating on the right). As usual, termination on the left produces a reward of -1 , and termination on the right produces a reward of $+1$. All other transitions have a reward of zero. We use this task as a running example throughout this section.

Figure 9.1 shows the true value function v_π for this task. It is nearly a straight line, curving very slightly toward the horizontal for the last 100 states at each end. Also shown is the final approximate value function learned by the gradient Monte-Carlo algorithm with state aggregation after 100,000 episodes with a step size of $\alpha = 2 \times 10^{-5}$. For the state aggregation, the 1000 states were partitioned into 10 groups of 100 states each (i.e., states 1–100 were one group, states 101–200 were another, and so on). The staircase effect

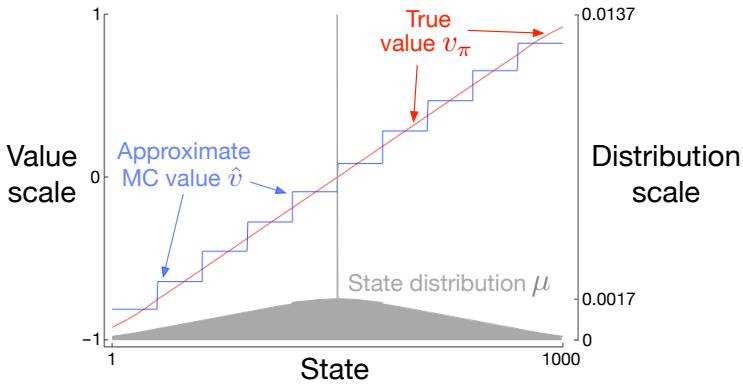


Figure 9.1: Function approximation by state aggregation on the 1000-state random walk task, using the gradient Monte Carlo algorithm (page 202).

shown in the figure is typical of state aggregation; within each group, the approximate value is constant, and it changes abruptly from one group to the next. These approximate values are close to the global minimum of the \overline{VE} (9.1).

Some of the details of the approximate values are best appreciated by reference to the state distribution μ for this task, shown in the lower portion of the figure with a right-side scale. State 500, in the center, is the first state of every episode, but is rarely visited again. On average, about 1.37% of the time steps are spent in the start state. The states reachable in one step from the start state are the second most visited, with about 0.17% of the time steps being spent in each of them. From there μ falls off almost linearly, reaching about 0.0147% at the extreme states 1 and 1000. The most visible effect of the distribution is on the leftmost groups, whose values are clearly shifted higher than the unweighted average of the true values of states within the group, and on the rightmost groups, whose values are clearly shifted lower. This is due to the states in these areas having the greatest asymmetry in their weightings by μ . For example, in the leftmost group, state 100 is weighted more than 3 times more strongly than state 1. Thus the estimate for the group is biased toward the true value of state 100, which is higher than the true value of state 1. ■

9.4 Linear Methods

One of the most important special cases of function approximation is that in which the approximate function, $\hat{v}(\cdot, \mathbf{w})$, is a linear function of the weight vector, \mathbf{w} . Corresponding to every state s , there is a real-valued vector $\mathbf{x}(s) \doteq (x_1(s), x_2(s), \dots, x_d(s))^\top$, with the same number of components as \mathbf{w} . Linear methods approximate the state-value function

by the inner product between \mathbf{w} and $\mathbf{x}(s)$:

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) \doteq \sum_{i=1}^d w_i x_i(s). \quad (9.8)$$

In this case the approximate value function is said to be *linear in the weights*, or simply *linear*.

The vector $\mathbf{x}(s)$ is called a *feature vector* representing state s . Each component $x_i(s)$ of $\mathbf{x}(s)$ is the value of a function $x_i : \mathcal{S} \rightarrow \mathbb{R}$. We think of a *feature* as the entirety of one of these functions, and we call its value for a state s a *feature of s* . For linear methods, features are *basis functions* because they form a linear basis for the set of approximate functions. Constructing d -dimensional feature vectors to represent states is the same as selecting a set of d basis functions. Features may be defined in many different ways; we cover a few possibilities in the next sections.

It is natural to use SGD updates with linear function approximation. The gradient of the approximate value function with respect to \mathbf{w} in this case is

$$\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s).$$

Thus, in the linear case the general SGD update (9.7) reduces to a particularly simple form:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha [U_t - \hat{v}(S_t, \mathbf{w}_t)] \mathbf{x}(S_t).$$

Because it is so simple, the linear SGD case is one of the most favorable for mathematical analysis. Almost all useful convergence results for learning systems of all kinds are for linear (or simpler) function approximation methods.

In particular, in the linear case there is only one optimum (or, in degenerate cases, one set of equally good optima), and thus any method that is guaranteed to converge to or near a local optimum is automatically guaranteed to converge to or near the global optimum. For example, the gradient Monte Carlo algorithm presented in the previous section converges to the global optimum of the $\overline{\text{VE}}$ under linear function approximation if α is reduced over time according to the usual conditions.

The semi-gradient TD(0) algorithm presented in the previous section also converges under linear function approximation, but this does not follow from general results on SGD; a separate theorem is necessary. The weight vector converged to is also not the global optimum, but rather a point near the local optimum. It is useful to consider this important case in more detail, specifically for the continuing case. The update at each time t is

$$\begin{aligned} \mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha \left(R_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t \right) \mathbf{x}_t \\ &= \mathbf{w}_t + \alpha \left(R_{t+1} \mathbf{x}_t - \mathbf{x}_t (\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top \mathbf{w}_t \right), \end{aligned} \quad (9.9)$$

where here we have used the notational shorthand $\mathbf{x}_t = \mathbf{x}(S_t)$. Once the system has reached steady state, for any given \mathbf{w}_t , the expected next weight vector can be written

$$\mathbb{E}[\mathbf{w}_{t+1} | \mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t), \quad (9.10)$$

where

$$\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d \quad \text{and} \quad \mathbf{A} \doteq \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top] \in \mathbb{R}^{d \times d} \quad (9.11)$$

From (9.10) it is clear that, if the system converges, it must converge to the weight vector \mathbf{w}_{TD} at which

$$\begin{aligned} \mathbf{b} - \mathbf{A}\mathbf{w}_{\text{TD}} &= \mathbf{0} \\ \Rightarrow \mathbf{b} &= \mathbf{A}\mathbf{w}_{\text{TD}} \\ \Rightarrow \mathbf{w}_{\text{TD}} &\doteq \mathbf{A}^{-1}\mathbf{b}. \end{aligned} \quad (9.12)$$

This quantity is called the *TD fixed point*. In fact linear semi-gradient TD(0) converges to this point. Some of the theory proving its convergence, and the existence of the inverse above, is given in the box.

Proof of Convergence of Linear TD(0)

What properties assure convergence of the linear TD(0) algorithm (9.9)? Some insight can be gained by rewriting (9.10) as

$$\mathbb{E}[\mathbf{w}_{t+1} | \mathbf{w}_t] = (\mathbf{I} - \alpha\mathbf{A})\mathbf{w}_t + \alpha\mathbf{b}. \quad (9.13)$$

Note that the matrix \mathbf{A} multiplies the weight vector \mathbf{w}_t and not \mathbf{b} ; only \mathbf{A} is important to convergence. To develop intuition, consider the special case in which \mathbf{A} is a diagonal matrix. If any of the diagonal elements are negative, then the corresponding diagonal element of $\mathbf{I} - \alpha\mathbf{A}$ will be greater than one, and the corresponding component of \mathbf{w}_t will be amplified, which will lead to divergence if continued. On the other hand, if the diagonal elements of \mathbf{A} are all positive, then α can be chosen smaller than one over the largest of them, such that $\mathbf{I} - \alpha\mathbf{A}$ is diagonal with all diagonal elements between 0 and 1. In this case the first term of the update tends to shrink \mathbf{w}_t , and stability is assured. In general, \mathbf{w}_t will be reduced toward zero whenever \mathbf{A} is *positive definite*, meaning $y^\top \mathbf{A} y > 0$ for any real vector $y \neq 0$. Positive definiteness also ensures that the inverse \mathbf{A}^{-1} exists.

For linear TD(0), in the continuing case with $\gamma < 1$, the \mathbf{A} matrix (9.11) can be written

$$\begin{aligned} \mathbf{A} &= \sum_s \mu(s) \sum_a \pi(a|s) \sum_{r,s'} p(r,s'|s,a) \mathbf{x}(s) (\mathbf{x}(s) - \gamma \mathbf{x}(s'))^\top \\ &= \sum_s \mu(s) \sum_{s'} p(s'|s) \mathbf{x}(s) (\mathbf{x}(s) - \gamma \mathbf{x}(s'))^\top \\ &= \sum_s \mu(s) \mathbf{x}(s) \left(\mathbf{x}(s) - \gamma \sum_{s'} p(s'|s) \mathbf{x}(s') \right)^\top \\ &= \mathbf{X}^\top \mathbf{D} (\mathbf{I} - \gamma \mathbf{P}) \mathbf{X}, \end{aligned}$$

where $\mu(s)$ is the stationary distribution under π , $p(s'|s)$ is the probability of transition from s to s' under policy π , \mathbf{P} is the $|\mathcal{S}| \times |\mathcal{S}|$ matrix of these probabilities,

\mathbf{D} is the $|S| \times |S|$ diagonal matrix with the $\mu(s)$ on its diagonal, and \mathbf{X} is the $|S| \times d$ matrix with $\mathbf{x}(s)$ as its rows. From here it is clear that the inner matrix $\mathbf{D}(\mathbf{I} - \gamma\mathbf{P})$ is key to determining the positive definiteness of \mathbf{A} .

For a key matrix of this form, positive definiteness is assured if all of its columns sum to a nonnegative number. This was shown by Sutton (1988, p. 27) based on two previously established theorems. One theorem says that any matrix \mathbf{M} is positive definite if and only if the symmetric matrix $\mathbf{S} = \mathbf{M} + \mathbf{M}^\top$ is positive definite (Sutton 1988, appendix). The second theorem says that any symmetric real matrix \mathbf{S} is positive definite if all of its diagonal entries are positive and greater than the sum of the absolute values of the corresponding off-diagonal entries (Varga 1962, p. 23). For our key matrix, $\mathbf{D}(\mathbf{I} - \gamma\mathbf{P})$, the diagonal entries are positive and the off-diagonal entries are negative, so all we have to show is that each row sum plus the corresponding column sum is positive. The row sums are all positive because \mathbf{P} is a stochastic matrix and $\gamma < 1$. Thus it only remains to show that the column sums are nonnegative. Note that the row vector of the column sums of any matrix \mathbf{M} can be written as $\mathbf{1}^\top \mathbf{M}$, where $\mathbf{1}$ is the column vector with all components equal to 1. Let $\boldsymbol{\mu}$ denote the $|S|$ -vector of the $\mu(s)$, where $\boldsymbol{\mu} = \mathbf{P}^\top \boldsymbol{\mu}$ by virtue of $\boldsymbol{\mu}$ being the stationary distribution. The column sums of our key matrix, then, are:

$$\begin{aligned}\mathbf{1}^\top \mathbf{D}(\mathbf{I} - \gamma\mathbf{P}) &= \boldsymbol{\mu}^\top (\mathbf{I} - \gamma\mathbf{P}) \\ &= \boldsymbol{\mu}^\top - \gamma\boldsymbol{\mu}^\top \mathbf{P} \\ &= \boldsymbol{\mu}^\top - \gamma\boldsymbol{\mu}^\top \quad (\text{because } \boldsymbol{\mu} \text{ is the stationary distribution}) \\ &= (1 - \gamma)\boldsymbol{\mu}^\top,\end{aligned}$$

all components of which are positive. Thus, the key matrix and its \mathbf{A} matrix are positive definite, and on-policy TD(0) is stable. (Additional conditions and a schedule for reducing α over time are needed to prove convergence with probability one.)

At the TD fixed point, it has also been proven (in the continuing case) that the $\overline{\text{VE}}$ is within a bounded expansion of the lowest possible error:

$$\overline{\text{VE}}(\mathbf{w}_{\text{TD}}) \leq \frac{1}{1 - \gamma} \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w}). \quad (9.14)$$

That is, the asymptotic error of the TD method is no more than $\frac{1}{1-\gamma}$ times the smallest possible error, that attained in the limit by the Monte Carlo method. Because γ is often near one, this expansion factor can be quite large, so there is substantial potential loss in asymptotic performance with the TD method. On the other hand, recall that the TD methods are often of vastly reduced variance compared to Monte Carlo methods, and thus faster, as we saw in Chapters 6 and 7. Which method will be best depends on the nature of the approximation and problem, and on how long learning continues.

A bound analogous to (9.14) applies to other on-policy bootstrapping methods as well. For example, linear semi-gradient DP (Eq. 9.7 with $U_t \doteq \sum_a \pi(a|S_t) \sum_{s',r} p(s',r|S_t,a)[r + \gamma \hat{v}(s',\mathbf{w}_t)]$) with updates according to the on-policy distribution will also converge to the TD fixed point. One-step semi-gradient *action-value* methods, such as semi-gradient Sarsa(0) covered in the next chapter converge to an analogous fixed point and an analogous bound. For episodic tasks, there is a slightly different but related bound (see Bertsekas and Tsitsiklis, 1996). There are also a few technical conditions on the rewards, features, and decrease in the step-size parameter, which we have omitted here. The full details can be found in the original paper (Tsitsiklis and Van Roy, 1997).

Critical to these convergence results is that states are updated according to the on-policy distribution. For other update distributions, bootstrapping methods using function approximation may actually diverge to infinity. Examples of this and a discussion of possible solution methods are given in Chapter 11.

Example 9.2: Bootstrapping on the 1000-state Random Walk State aggregation is a special case of linear function approximation, so let's return to the 1000-state random walk to illustrate some of the observations made in this chapter. The left panel of Figure 9.2 shows the final value function learned by the semi-gradient TD(0) algorithm (page 203) using the same state aggregation as in Example 9.1. We see that the near-asymptotic TD approximation is indeed farther from the true values than the Monte Carlo approximation shown in Figure 9.1.

Nevertheless, TD methods retain large potential advantages in learning rate, and generalize Monte Carlo methods, as we investigated fully with n -step TD methods in Chapter 7. The right panel of Figure 9.2 shows results with an n -step semi-gradient TD method using state aggregation on the 1000-state random walk that are strikingly similar to those we obtained earlier with tabular methods and the 19-state random walk (Figure 7.2). To obtain such quantitatively similar results we switched the state aggregation to 20 groups of 50 states each. The 20 groups were then quantitatively close

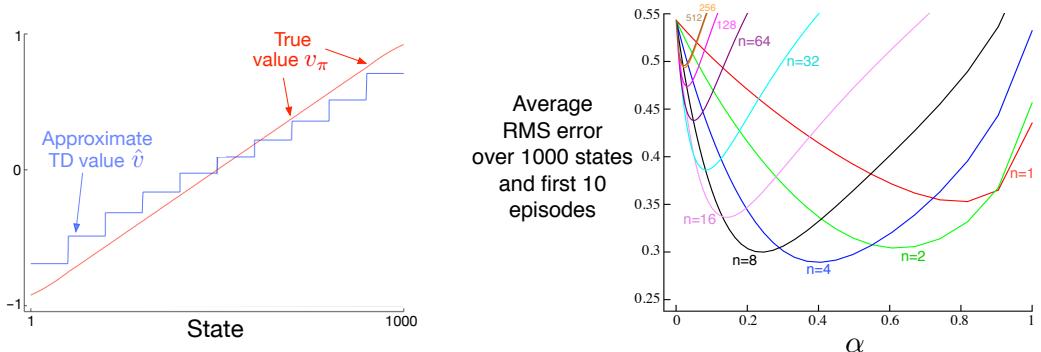


Figure 9.2: Bootstrapping with state aggregation on the 1000-state random walk task. *Left:* Asymptotic values of semi-gradient TD are worse than the asymptotic Monte Carlo values in Figure 9.1. *Right:* Performance of n -step methods with state-aggregation are strikingly similar to those with tabular representations (cf. Figure 7.2). These data are averages over 100 runs.

to the 19 states of the tabular problem. In particular, recall that state transitions were up to 100 states to the left or right. A typical transition would then be of 50 states to the right or left, which is quantitatively analogous to the single-state state transitions of the 19-state tabular system. To complete the match, we use here the same performance measure—an unweighted average of the RMS error over all states and over the first 10 episodes—rather than a \overline{VE} objective as is otherwise more appropriate when using function approximation. ■

The semi-gradient n -step TD algorithm used in the example above is the natural extension of the tabular n -step TD algorithm presented in Chapter 7 to semi-gradient function approximation. Pseudocode is given in the box below.

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated

Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$ such that $\hat{v}(\text{terminal}, \cdot) = 0$

Algorithm parameters: step size $\alpha > 0$, a positive integer n

Initialize value-function weights w arbitrarily (e.g., $w = 0$)

All store and access operations (S_t and R_t) can take their index mod $n + 1$

Loop for each episode:

Initialize and store $S_0 \neq \text{terminal}$

$$T \leftarrow \infty$$

Loop for $t = 0, 1, 2, \dots$:

If $t < T$, then:

Take an action according to $\pi(\cdot|S_t)$

Observe and store the next reward as R_{t+1} and the next state as S_{t+1}

If S_{t+1} is terminal, then $T \leftarrow t + 1$

$\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)

If $\tau \geq 0$:

$$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R$$

If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$

$$(G_{\tau:\tau+n})$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [G - \hat{v}(S_{\tau}, \mathbf{w})] \nabla \hat{v}(S_{\tau}, \mathbf{w})$$

Until $\tau = T - 1$

The key equation of this algorithm, analogous to (7.2), is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha [G_{t:t+n} - \hat{v}(S_t, \mathbf{w}_{t+n-1})] \nabla \hat{v}(S_t, \mathbf{w}_{t+n-1}), \quad 0 \leq t < T, \quad (9.15)$$

where the n -step return is generalized from (7.1) to

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}, \mathbf{w}_{t+n-1}), \quad 0 < t < T-n. \quad (9.16)$$

Exercise 9.1 Show that tabular methods such as presented in Part I of this book are a special case of linear function approximation. What would the feature vectors be? \square

9.5 Feature Construction for Linear Methods

Linear methods are interesting because of their convergence guarantees, but also because in practice they can be very efficient in terms of both data and computation. Whether or not this is so depends critically on how the states are represented in terms of features, which we investigate in this large section. Choosing features appropriate to the task is an important way of adding prior domain knowledge to reinforcement learning systems. Intuitively, the features should correspond to the aspects of the state space along which generalization may be appropriate. If we are valuing geometric objects, for example, we might want to have features for each possible shape, color, size, or function. If we are valuing states of a mobile robot, then we might want to have features for locations, degrees of remaining battery power, recent sonar readings, and so on.

A limitation of the linear form is that it cannot take into account any interactions between features, such as the presence of feature i being good only in the absence of feature j . For example, in the pole-balancing task (Example 3.4), high angular velocity can be either good or bad depending on the angle. If the angle is high, then high angular velocity means an imminent danger of falling—a bad state—whereas if the angle is low, then high angular velocity means the pole is righting itself—a good state. A linear value function could not represent this if its features coded separately for the angle and the angular velocity. It needs instead, or in addition, features for combinations of these two underlying state dimensions. In the following subsections we consider a variety of general ways of doing this.

9.5.1 Polynomials

The states of many problems are initially expressed as numbers, such as positions and velocities in the pole-balancing task (Example 3.4), the number of cars in each lot in the Jack’s car rental problem (Example 4.2), or the gambler’s capital in the gambler problem (Example 4.3). In these types of problems, function approximation for reinforcement learning has much in common with the familiar tasks of interpolation and regression. Various families of features commonly used for interpolation and regression can also be used in reinforcement learning. Polynomials make up one of the simplest families of features used for interpolation and regression. While the basic polynomial features we discuss here do not work as well as other types of features in reinforcement learning, they serve as a good introduction because they are simple and familiar.

As an example, suppose a reinforcement learning problem has states with two numerical dimensions. For a single representative state s , let its two numbers be $s_1 \in \mathbb{R}$ and $s_2 \in \mathbb{R}$. You might choose to represent s simply by its two state dimensions, so that $\mathbf{x}(s) = (s_1, s_2)^\top$, but then you would not be able to take into account any interactions between these dimensions. In addition, if both s_1 and s_2 were zero, then the approximate value would have to also be zero. Both limitations can be overcome by instead representing s by the four-dimensional feature vector $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2)^\top$. The initial 1 feature allows the representation of affine functions in the original state numbers, and the final product feature, $s_1 s_2$, enables interactions to be taken into account. Or you might choose to use higher-dimensional feature vectors like $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1 s_2^2, s_1^2 s_2, s_1^2 s_2^2)^\top$ to

take more complex interactions into account. Such feature vectors enable approximations as arbitrary quadratic functions of the state numbers—even though the approximation is still linear in the weights that have to be learned. Generalizing this example from two to k numbers, we can represent highly-complex interactions among a problem’s state dimensions:

Suppose each state s corresponds to k numbers, s_1, s_2, \dots, s_k , with each $s_i \in \mathbb{R}$. For this k -dimensional state space, each order- n polynomial-basis feature x_i can be written as

$$x_i(s) = \prod_{j=1}^k s_j^{c_{i,j}}, \quad (9.17)$$

where each $c_{i,j}$ is an integer in the set $\{0, 1, \dots, n\}$ for an integer $n \geq 0$. These features make up the order- n polynomial basis for dimension k , which contains $(n+1)^k$ different features.

Higher-order polynomial bases allow for more accurate approximations of more complicated functions. But because the number of features in an order- n polynomial basis grows exponentially with the dimension k of the natural state space (if $n > 0$), it is generally necessary to select a subset of them for function approximation. This can be done using prior beliefs about the nature of the function to be approximated, and some automated selection methods developed for polynomial regression can be adapted to deal with the incremental and nonstationary nature of reinforcement learning.

Exercise 9.2 Why does (9.17) define $(n+1)^k$ distinct features for dimension k ? □

Exercise 9.3 What n and $c_{i,j}$ produce the feature vectors $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1 s_2^2, s_1^2 s_2, s_1^2 s_2^2)^\top$? □

9.5.2 Fourier Basis

Another linear function approximation method is based on the time-honored Fourier series, which expresses periodic functions as weighted sums of sine and cosine basis functions (features) of different frequencies. (A function f is periodic if $f(x) = f(x + \tau)$ for all x and some period τ .) The Fourier series and the more general Fourier transform are widely used in applied sciences in part because if a function to be approximated is known, then the basis function weights are given by simple formulae and, further, with enough basis functions essentially any function can be approximated as accurately as desired. In reinforcement learning, where the functions to be approximated are unknown, Fourier basis functions are of interest because they are easy to use and can perform well in a range of reinforcement learning problems.

First consider the one-dimensional case. The usual Fourier series representation of a function of one dimension having period τ represents the function as a linear combination of sine and cosine functions that are each periodic with periods that evenly divide τ (in other words, whose frequencies are integer multiples of a fundamental frequency $1/\tau$). But if you are interested in approximating an aperiodic function defined over a bounded

interval, then you can use these Fourier basis features with τ set to the length of the interval. The function of interest is then just one period of the periodic linear combination of the sine and cosine features.

Furthermore, if you set τ to twice the length of the interval of interest and restrict attention to the approximation over the half interval $[0, \tau/2]$, then you can use just the cosine features. This is possible because you can represent any *even* function, that is, any function that is symmetric about the origin, with just the cosine basis. So any function over the half-period $[0, \tau/2]$ can be approximated as closely as desired with enough cosine features. (Saying “any function” is not exactly correct because the function has to be mathematically well-behaved, but we skip this technicality here.) Alternatively, it is possible to use just sine features, linear combinations of which are always *odd* functions, that is functions that are anti-symmetric about the origin. But it is generally better to keep just the cosine features because “half-even” functions tend to be easier to approximate than “half-odd” functions because the latter are often discontinuous at the origin. Of course, this does not rule out using both sine and cosine features to approximate over the interval $[0, \tau/2]$, which might be advantageous in some circumstances.

Following this logic and letting $\tau = 2$ so that the features are defined over the half- τ interval $[0, 1]$, the one-dimensional order- n Fourier cosine basis consists of the $n + 1$ features

$$x_i(s) = \cos(i\pi s), \quad s \in [0, 1],$$

for $i = 0, \dots, n$. Figure 9.3 shows one-dimensional Fourier cosine features x_i , for $i = 1, 2, 3, 4$; x_0 is a constant function.

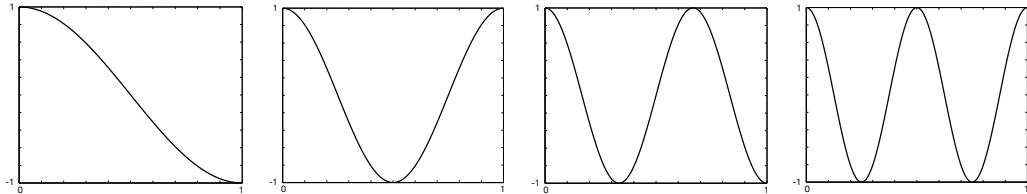


Figure 9.3: One-dimensional Fourier cosine-basis features x_i , $i = 1, 2, 3, 4$, for approximating functions over the interval $[0, 1]$. After Konidaris et al. (2011).

This same reasoning applies to the Fourier cosine series approximation in the multi-dimensional case as described in the box below.

Suppose each state s corresponds to a vector of k numbers, $\mathbf{s} = (s_1, s_2, \dots, s_k)^\top$, with each $s_i \in [0, 1]$. The i th feature in the order- n Fourier cosine basis can then be written

$$x_i(s) = \cos(\pi \mathbf{s}^\top \mathbf{c}^i), \quad (9.18)$$

where $\mathbf{c}^i = (c_1^i, \dots, c_k^i)^\top$, with $c_j^i \in \{0, \dots, n\}$ for $j = 1, \dots, k$ and $i = 1, \dots, (n+1)^k$. This defines a feature for each of the $(n+1)^k$ possible integer vectors \mathbf{c}^i . The inner

product $\mathbf{s}^\top \mathbf{c}^i$ has the effect of assigning an integer in $\{0, \dots, n\}$ to each dimension of \mathbf{s} . As in the one-dimensional case, this integer determines the feature's frequency along that dimension. The features can of course be shifted and scaled to suit the bounded state space of a particular application.

As an example, consider the $k = 2$ case in which $\mathbf{s} = (s_1, s_2)^\top$, where each $\mathbf{c}^i = (c_1^i, c_2^i)^\top$. Figure 9.4 shows a selection of six Fourier cosine features, each labeled by the vector \mathbf{c}^i that defines it (s_1 is the horizontal axis and \mathbf{c}^i is shown as a row vector with the index i omitted). Any zero in \mathbf{c} means the feature is constant along that state dimension. So if $\mathbf{c} = (0, 0)^\top$, the feature is constant over both dimensions; if $\mathbf{c} = (c_1, 0)^\top$ the feature is constant over the second dimension and varies over the first with frequency depending on c_1 ; and similarly, for $\mathbf{c} = (0, c_2)^\top$. When $\mathbf{c} = (c_1, c_2)^\top$ with neither $c_j = 0$, the feature varies along both dimensions and represents an interaction between the two state variables. The values of c_1 and c_2 determine the frequency along each dimension, and their ratio gives the direction of the interaction.

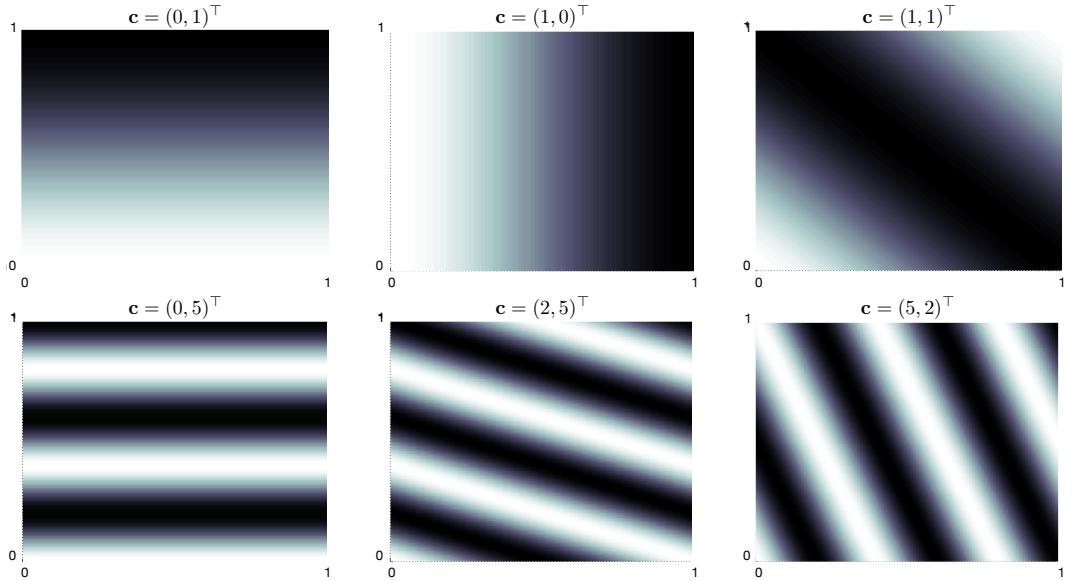


Figure 9.4: A selection of six two-dimensional Fourier cosine features, each labeled by the vector \mathbf{c}^i that defines it (s_1 is the horizontal axis, and \mathbf{c}^i is shown with the index i omitted). After Konidaris et al. (2011).

When using Fourier cosine features with a learning algorithm such as (9.7), semi-gradient TD(0), or semi-gradient Sarsa, it may be helpful to use a different step-size parameter for each feature. If α is the basic step-size parameter, then Konidaris, Osentoski, and Thomas (2011) suggest setting the step-size parameter for feature x_i to $\alpha_i = \alpha / \sqrt{(c_1^i)^2 + \dots + (c_k^i)^2}$ (except when each $c_j^i = 0$, in which case $\alpha_i = \alpha$).

Fourier cosine features with Sarsa can produce good performance compared to several other collections of basis functions, including polynomial and radial basis functions. Not surprisingly, however, Fourier features have trouble with discontinuities because it is difficult to avoid “ringing” around points of discontinuity unless very high frequency basis functions are included.

The number of features in the order- n Fourier basis grows exponentially with the dimension of the state space, but if that dimension is small enough (e.g., $k \leq 5$), then one can select n so that all of the order- n Fourier features can be used. This makes the selection of features more-or-less automatic. For high dimension state spaces, however, it is necessary to select a subset of these features. This can be done using prior beliefs about the nature of the function to be approximated, and some automated selection methods can be adapted to deal with the incremental and nonstationary nature of reinforcement learning. An advantage of Fourier basis features in this regard is that it is easy to select features by setting the \mathbf{c}^i vectors to account for suspected interactions among the state variables and by limiting the values in the \mathbf{c}^j vectors so that the approximation can filter out high frequency components considered to be noise. On the other hand, because Fourier features are non-zero over the entire state space (with the few zeros excepted), they represent global properties of states, which can make it difficult to find good ways to represent local properties.

Figure 9.5 shows learning curves comparing the Fourier and polynomial bases on the 1000-state random walk example. In general, we do not recommend using polynomials for online learning.²

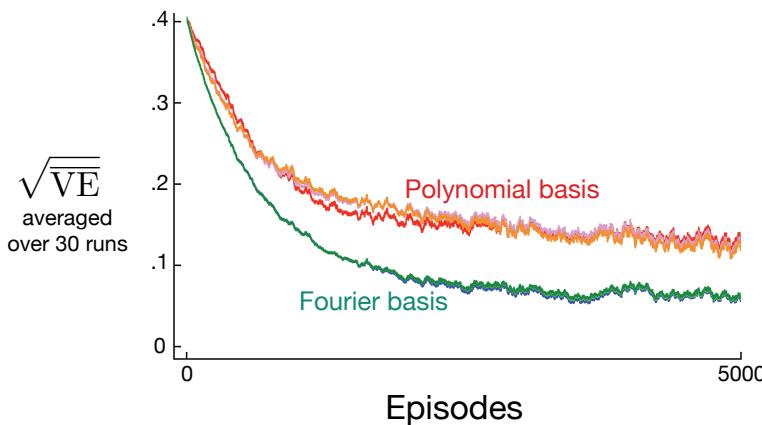


Figure 9.5: Fourier basis vs polynomials on the 1000-state random walk. Shown are learning curves for the gradient Monte Carlo method with Fourier and polynomial bases of order 5, 10, and 20. The step-size parameters were roughly optimized for each case: $\alpha = 0.0001$ for the polynomial basis and $\alpha = 0.00005$ for the Fourier basis. The performance measure (y-axis) is the root mean square value error (9.1).

²There are families of polynomials more complicated than those we have discussed, for example, different families of orthogonal polynomials, and these might work better, but at present there is little experience with them in reinforcement learning.

9.5.3 Coarse Coding

Consider a task in which the natural representation of the state set is a continuous two-dimensional space. One kind of representation for this case is made up of features corresponding to *circles* in state space, as shown to the right. If the state is inside a circle, then the corresponding feature has the value 1 and is said to be *present*; otherwise the feature is 0 and is said to be *absent*. This kind of 1–0-valued feature is called a *binary feature*. Given a state, which binary features are present indicate within which circles the state lies, and thus coarsely code for its location. Representing a state with features that overlap in this way (although they need not be circles or binary) is known as *coarse coding*.

Assuming linear gradient-descent function approximation, consider the effect of the size and density of the circles. Corresponding to each circle is a single weight (a component of \mathbf{w}) that is affected by learning. If we train at one state, a point in the space, then the weights of all circles intersecting that state will be affected. Thus, by (9.8), the approximate value function will be affected at all states within the union of the circles, with a greater effect the more circles a point has “in common” with the state, as shown in Figure 9.6. If the circles are small, then the generalization will be over a short distance, as in Figure 9.7 (left), whereas if they are large, it will be over a large distance, as in Figure 9.7 (middle). Moreover,

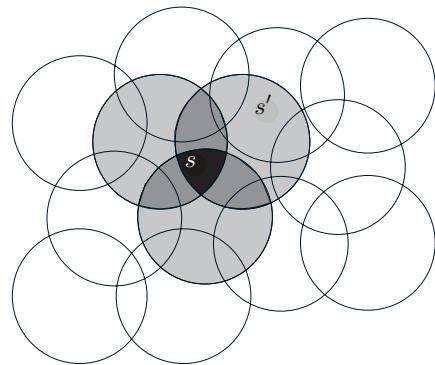


Figure 9.6: Coarse coding. Generalization from state s to state s' depends on the number of their features whose receptive fields (in this case, circles) overlap. These states have one feature in common, so there will be slight generalization between them.

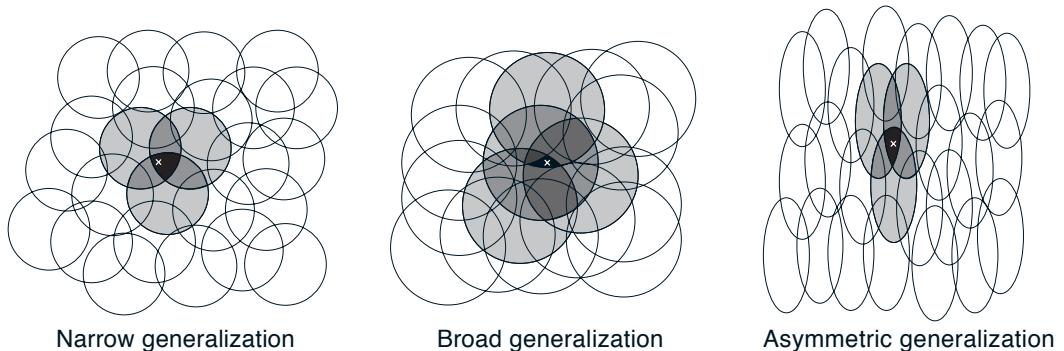


Figure 9.7: Generalization in linear function approximation methods is determined by the sizes and shapes of the features’ receptive fields. All three of these cases have roughly the same number and density of features.

the shape of the features will determine the nature of the generalization. For example, if they are not strictly circular, but are elongated in one direction, then generalization will be similarly affected, as in Figure 9.7 (right).

Features with large receptive fields give broad generalization, but might also seem to limit the learned function to a coarse approximation, unable to make discriminations much finer than the width of the receptive fields. Happily, this is not the case. Initial generalization from one point to another is indeed controlled by the size and shape of the receptive fields, but acuity, the finest discrimination ultimately possible, is controlled more by the total number of features.

Example 9.3: Coarseness of Coarse Coding This example illustrates the effect on learning of the size of the receptive fields in coarse coding. Linear function approximation based on coarse coding and (9.7) was used to learn a one-dimensional square-wave function (shown at the top of Figure 9.8). The values of this function were used as the targets, U_t . With just one dimension, the receptive fields were intervals rather than circles. Learning was repeated with three different sizes of the intervals: narrow, medium, and broad, as shown at the bottom of the figure. All three cases had the same density of features, about 50 over the extent of the function being learned. Training examples were generated uniformly at random over this extent. The step-size parameter was $\alpha = \frac{0.2}{n}$, where n is the number of features that were present at one time. Figure 9.8 shows the functions learned in all three cases over the course of learning. Note that the width of the features had a strong effect early in learning. With broad features, the generalization tended to be broad; with narrow features, only the close neighbors of each trained point were changed, causing the function learned to be more bumpy. However, the final function learned was affected only slightly by the width of the features. Receptive field shape tends to have a strong effect on generalization but little effect on asymptotic solution quality.

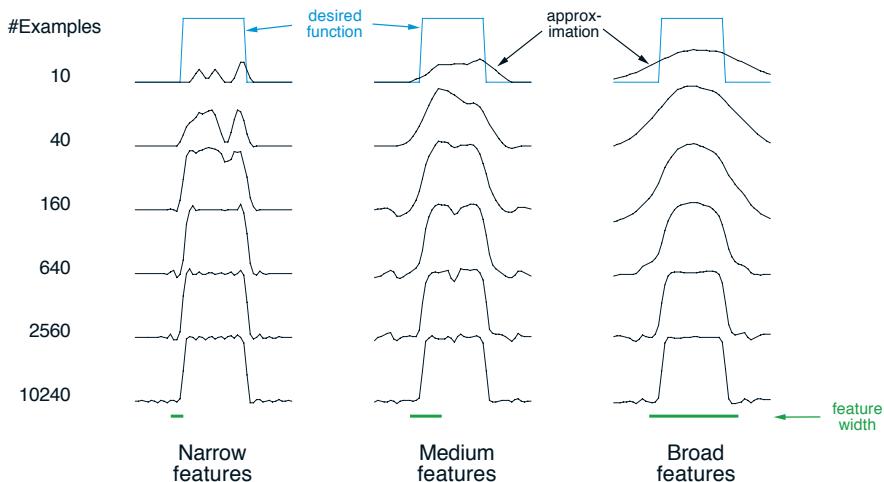


Figure 9.8: Example of feature width's strong effect on initial generalization (first row) and weak effect on asymptotic accuracy (last row). ■

9.5.4 Tile Coding

Tile coding is a form of coarse coding for multi-dimensional continuous spaces that is flexible and computationally efficient. It may be the most practical feature representation for modern sequential digital computers.

In tile coding the receptive fields of the features are grouped into partitions of the state space. Each such partition is called a *tiling*, and each element of the partition is called a *tile*. For example, the simplest tiling of a two-dimensional state space is a uniform grid such as that shown on the left side of Figure 9.9. The tiles or receptive field here are squares rather than the circles in Figure 9.6. If just this single tiling were used, then the state indicated by the white spot would be represented by the single feature whose tile it falls within; generalization would be complete to all states within the same tile and nonexistent to states outside it. With just one tiling, we would not have coarse coding but just a case of state aggregation.

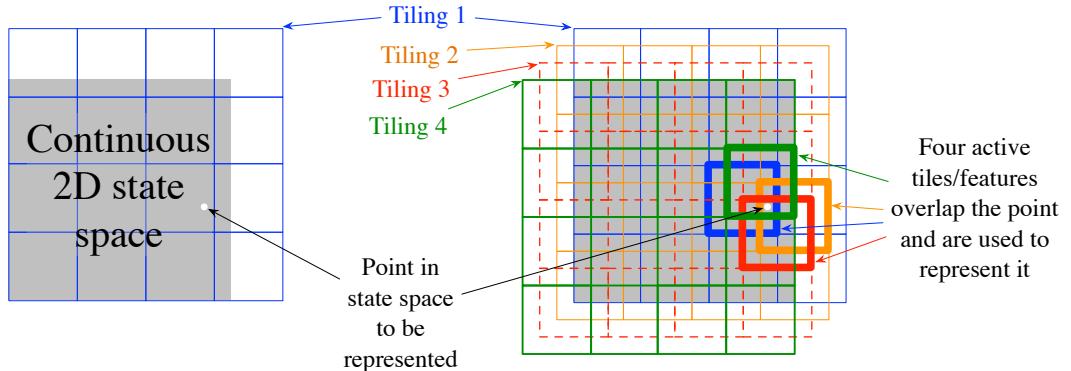


Figure 9.9: Multiple, overlapping grid-tilings on a limited two-dimensional space. These tilings are offset from one another by a uniform amount in each dimension.

To get the strengths of coarse coding requires overlapping receptive fields, and by definition the tiles of a partition do not overlap. To get true coarse coding with tile coding, multiple tilings are used, each offset by a fraction of a tile width. A simple case with four tilings is shown on the right side of Figure 9.9. Every state, such as that indicated by the white spot, falls in exactly one tile in each of the four tilings. These four tiles correspond to four features that become active when the state occurs. Specifically, the feature vector $\mathbf{x}(s)$ has one component for each tile in each tiling. In this example there are $4 \times 4 \times 4 = 64$ components, all of which will be 0 except for the four corresponding to the tiles that s falls within. Figure 9.10 shows the advantage of multiple offset tilings (coarse coding) over a single tiling on the 1000-state random walk example.

An immediate practical advantage of tile coding is that, because it works with partitions, the overall number of features that are active at one time is the same for any state. Exactly one feature is present in each tiling, so the total number of features present is always the same as the number of tilings. This allows the step-size parameter, α , to

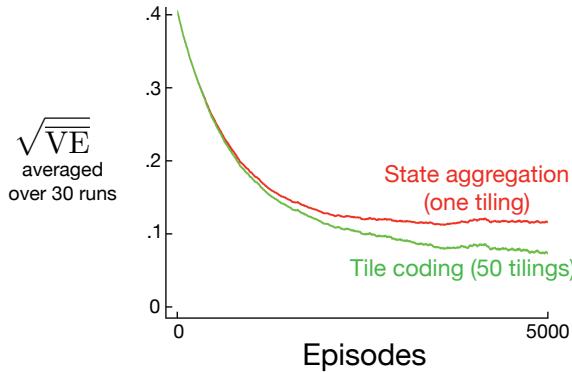


Figure 9.10: Why we use coarse coding. Shown are learning curves on the 1000-state random walk example for the gradient Monte Carlo algorithm with a single tiling and with multiple tilings. The space of 1000 states was treated as a single continuous dimension, covered with tiles each 200 states wide. The multiple tilings were offset from each other by 4 states. The step-size parameter was set so that the initial learning rate in the two cases was the same, $\alpha = 0.0001$ for the single tiling and $\alpha = 0.0001/50$ for the 50 tilings.

be set in an easy, intuitive way. For example, choosing $\alpha = \frac{1}{n}$, where n is the number of tilings, results in exact one-trial learning. If the example $s \mapsto v$ is trained on, then whatever the prior estimate, $\hat{v}(s, \mathbf{w}_t)$, the new estimate will be $\hat{v}(s, \mathbf{w}_{t+1}) = v$. Usually one wishes to change more slowly than this, to allow for generalization and stochastic variation in target outputs. For example, one might choose $\alpha = \frac{1}{10n}$, in which case the estimate for the trained state would move one-tenth of the way to the target in one update, and neighboring states will be moved less, proportional to the number of tiles they have in common.

Tile coding also gains computational advantages from its use of binary feature vectors. Because each component is either 0 or 1, the weighted sum making up the approximate value function (9.8) is almost trivial to compute. Rather than performing d multiplications and additions, one simply computes the indices of the $n \ll d$ active features and then adds up the n corresponding components of the weight vector.

Generalization occurs to states other than the one trained if those states fall within any of the same tiles, proportional to the number of tiles in common. Even the choice of how to offset the tilings from each other affects generalization. If they are offset uniformly in each dimension, as they were in Figure 9.9, then different states can generalize in qualitatively different ways, as shown in the upper half of Figure 9.11. Each of the eight subfigures show the pattern of generalization from a trained state to nearby points. In this example there are eight tilings, thus 64 subregions within a tile that generalize distinctly, but all according to one of these eight patterns. Note how uniform offsets result in a strong effect along the diagonal in many patterns. These artifacts can be avoided if the tilings are offset asymmetrically, as shown in the lower half of the figure. These lower generalization patterns are better because they are all well centered on the trained state with no obvious asymmetries.

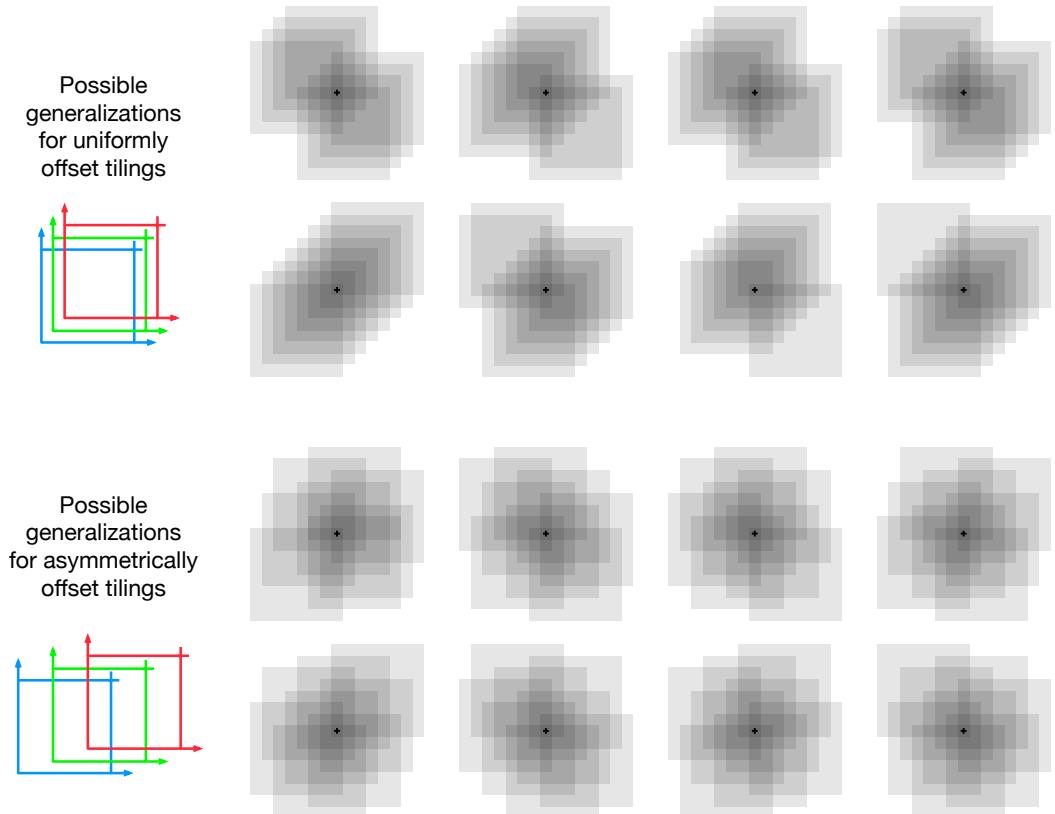


Figure 9.11: Why tile asymmetrical offsets are preferred in tile coding. Shown is the strength of generalization from a trained state, indicated by the small black plus, to nearby states, for the case of eight tilings. If the tilings are uniformly offset (above), then there are diagonal artifacts and substantial variations in the generalization, whereas with asymmetrically offset tilings the generalization is more spherical and homogeneous.

Tilings in all cases are offset from each other by a fraction of a tile width in each dimension. If w denotes the tile width and n the number of tilings, then $\frac{w}{n}$ is a fundamental unit. Within small squares $\frac{w}{n}$ on a side, all states activate the same tiles, have the same feature representation, and the same approximated value. If a state is moved by $\frac{w}{n}$ in any cartesian direction, the feature representation changes by one component/tile. Uniformly offset tilings are offset from each other by exactly this unit distance. For a two-dimensional space, we say that each tiling is offset by the displacement vector $(1, 1)$, meaning that it is offset from the previous tiling by $\frac{w}{n}$ times this vector. In these terms, the asymmetrically offset tilings shown in the lower part of Figure 9.11 are offset by a displacement vector of $(1, 3)$.

Extensive studies have been made of the effect of different displacement vectors on the generalization of tile coding (Parks and Militzer, 1991; An, 1991; An, Miller and Parks,

1991; Miller, An, Glanz and Carter, 1990), assessing their homogeneity and tendency toward diagonal artifacts like those seen for the $(1, 1)$ displacement vectors. Based on this work, Miller and Glanz (1996) recommend using displacement vectors consisting of the first odd integers. In particular, for a continuous space of dimension k , a good choice is to use the first odd integers $(1, 3, 5, 7, \dots, 2k - 1)$, with n (the number of tilings) set to an integer power of 2 greater than or equal to $4k$. This is what we have done to produce the tilings in the lower half of Figure 9.11, in which $k = 2$, $n = 2^3 \geq 4k$, and the displacement vector is $(1, 3)$. In a three-dimensional case, the first four tilings would be offset in total from a base position by $(0, 0, 0)$, $(1, 3, 5)$, $(2, 6, 10)$, and $(3, 9, 15)$. Open-source software that can efficiently make tilings like this for any k is readily available.

In choosing a tiling strategy, one has to pick the number of the tilings and the shape of the tiles. The number of tilings, along with the size of the tiles, determines the resolution or fineness of the asymptotic approximation, as in general coarse coding and illustrated in Figure 9.8. The shape of the tiles will determine the nature of generalization as in Figure 9.7. Square tiles will generalize roughly equally in each dimension as indicated in Figure 9.11 (lower). Tiles that are elongated along one dimension, such as the stripe tilings in Figure 9.12 (middle), will promote generalization along that dimension. The tilings in Figure 9.12 (middle) are also denser and thinner on the left, promoting discrimination along the horizontal dimension at lower values along that dimension. The diagonal stripe tiling in Figure 9.12 (right) will promote generalization along one diagonal. In higher dimensions, axis-aligned stripes correspond to ignoring some of the dimensions in some of the tilings, that is, to hyperplanar slices. Irregular tilings such as shown in Figure 9.12 (left) are also possible, though rare in practice and beyond the standard software.

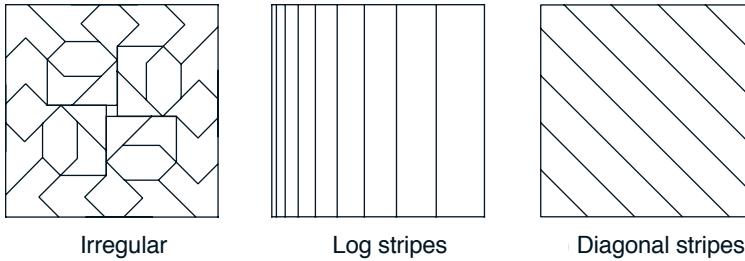
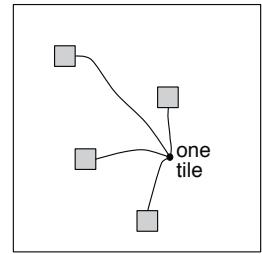


Figure 9.12: Tilings need not be grids. They can be arbitrarily shaped and non-uniform, while still in many cases being computationally efficient to compute.

In practice, it is often desirable to use different shaped tiles in different tilings. For example, one might use some vertical stripe tilings and some horizontal stripe tilings. This would encourage generalization along either dimension. However, with stripe tilings alone it is not possible to learn that a particular conjunction of horizontal and vertical coordinates has a distinctive value (whatever is learned for it will bleed into states with the same horizontal and vertical coordinates). For this one needs the conjunctive rectangular tiles such as originally shown in Figure 9.9. With multiple tilings—some horizontal, some vertical, and some conjunctive—one can get everything: a preference for generalizing along each dimension, yet the ability to learn specific values for conjunctions (see Sutton,

1996 for examples). The choice of tilings determines generalization, and until this choice can be effectively automated, it is important that tile coding enables the choice to be made flexibly and in a way that makes sense to people.

Another useful trick for reducing memory requirements is *hashing*—a consistent pseudo-random collapsing of a large tiling into a much smaller set of tiles. Hashing produces tiles consisting of noncontiguous, disjoint regions randomly spread throughout the state space, but that still form an exhaustive partition. For example, one tile might consist of the four subtiles shown to the right. Through hashing, memory requirements are often reduced by large factors with little loss of performance. This is possible because high resolution is needed in only a small fraction of the state space. Hashing frees us from the curse of dimensionality in the sense that memory requirements need not be exponential in the number of dimensions, but need merely match the real demands of the task. Open-source implementations of tile coding commonly include efficient hashing.



Exercise 9.4 Suppose we believe that one of two state dimensions is more likely to have an effect on the value function than is the other, that generalization should be primarily across this dimension rather than along it. What kind of tilings could be used to take advantage of this prior knowledge? □

9.5.5 Radial Basis Functions

Radial basis functions (RBFs) are the natural generalization of coarse coding to continuous-valued features. Rather than each feature being either 0 or 1, it can be anything in the interval $[0, 1]$, reflecting various *degrees* to which the feature is present. A typical RBF feature, x_i , has a Gaussian (bell-shaped) response $x_i(s)$ dependent only on the distance between the state, s , and the feature's prototypical or center state, c_i , and relative to the feature's width, σ_i :

$$x_i(s) \doteq \exp\left(-\frac{\|s - c_i\|^2}{2\sigma_i^2}\right).$$

The norm or distance metric of course can be chosen in whatever way seems most appropriate to the states and task at hand. The figure below shows a one-dimensional example with a Euclidean distance metric.

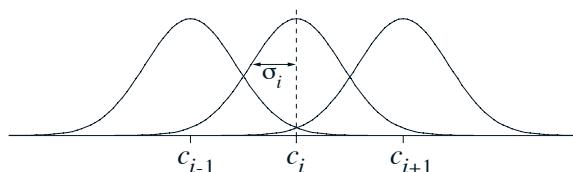


Figure 9.13: One-dimensional radial basis functions.

The primary advantage of RBFs over binary features is that they produce approximate functions that vary smoothly and are differentiable. Although this is appealing, in most cases it has no practical significance. Nevertheless, extensive studies have been made of graded response functions such as RBFs in the context of tile coding (An, 1991; Miller et al., 1991; An et al., 1991; Lane, Handelman and Gelfand, 1992). All of these methods require substantial additional computational complexity (over tile coding) and often reduce performance when there are more than two state dimensions. In high dimensions the edges of tiles are much more important, and it has proven difficult to obtain well controlled graded tile activations near the edges.

An *RBF network* is a linear function approximator using RBFs for its features. Learning is defined by equations (9.7) and (9.8), exactly as in other linear function approximators. In addition, some learning methods for RBF networks change the centers and widths of the features as well, bringing them into the realm of nonlinear function approximators. Nonlinear methods may be able to fit target functions much more precisely. The downside to RBF networks, and to nonlinear RBF networks especially, is greater computational complexity and, often, more manual tuning before learning is robust and efficient.

9.6 Selecting Step-Size Parameters Manually

Most SGD methods require the designer to select an appropriate step-size parameter α . Ideally this selection would be automated, and in some cases it has been, but for most cases it is still common practice to set it manually. To do this, and to better understand the algorithms, it is useful to develop some intuitive sense of the role of the step-size parameter. Can we say in general how it should be set?

Theoretical considerations are unfortunately of little help. The theory of stochastic approximation gives us conditions (2.7) on a slowly decreasing step-size sequence that are sufficient to guarantee convergence, but these tend to result in learning that is too slow. The classical choice $\alpha_t = 1/t$, which produces sample averages in tabular MC methods, is not appropriate for TD methods, for nonstationary problems, or for any method using function approximation. For linear methods, there are recursive least-squares methods that set an optimal *matrix* step size, and these methods can be extended to temporal-difference learning as in the LSTD method described in Section 9.8, but these require $O(d^2)$ step-size parameters, or d times more parameters than we are learning. For this reason we rule them out for use on large problems where function approximation is most needed.

To get some intuitive feel for how to set the step-size parameter manually, it is best to go back momentarily to the tabular case. There we can understand that a step size of $\alpha = 1$ will result in a complete elimination of the sample error after one target (see (2.4) with a step size of one). As discussed on page 201, we usually want to learn slower than this. In the tabular case, a step size of $\alpha = \frac{1}{10}$ would take about 10 experiences to converge approximately to their mean target, and if we wanted to learn in 100 experiences we would use $\alpha = \frac{1}{100}$. In general, if $\alpha = \frac{1}{\tau}$, then the tabular estimate for a state will approach the mean of its targets, with the most recent targets having the greatest effect, after about τ experiences with the state.

With general function approximation there is not such a clear notion of *number of experiences* with a state, as each state may be similar to and dissimilar from all the others to various degrees. However, there is a similar rule that gives similar behavior in the case of linear function approximation. Suppose you wanted to learn in about τ experiences with substantially the same feature vector. A good rule of thumb for setting the step-size parameter of linear SGD methods is then

$$\alpha \doteq (\tau \mathbb{E}[\mathbf{x}^\top \mathbf{x}])^{-1}, \quad (9.19)$$

where \mathbf{x} is a random feature vector chosen from the same distribution as input vectors will be in the SGD. This method works best if the feature vectors do not vary greatly in length; ideally $\mathbf{x}^\top \mathbf{x}$ is a constant.

Exercise 9.5 Suppose you are using tile coding to transform a seven-dimensional continuous state space into binary feature vectors to estimate a state value function $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$. You believe that the dimensions do not interact strongly, so you decide to use eight tilings of each dimension separately (stripe tilings), for $7 \times 8 = 56$ tilings. In addition, in case there are some pairwise interactions between the dimensions, you also take all $\binom{7}{2} = 21$ pairs of dimensions and tile each pair conjunctively with rectangular tiles. You make two tilings for each pair of dimensions, making a grand total of $21 \times 2 + 56 = 98$ tilings. Given these feature vectors, you suspect that you still have to average out some noise, so you decide that you want learning to be gradual, taking about 10 presentations with the same feature vector before learning nears its asymptote. What step-size parameter α should you use? Why? \square

Exercise 9.6 If $\tau = 1$ and $\mathbf{x}(S_t)^\top \mathbf{x}(S_t) = \mathbb{E}[\mathbf{x}^\top \mathbf{x}]$, prove that (9.19) together with (9.7) and linear function approximation results in the error being reduced to zero in one update.

9.7 Nonlinear Function Approximation: Artificial Neural Networks

Artificial neural networks (ANNs) are widely used for nonlinear function approximation. An ANN is a network of interconnected units that have some of the properties of neurons, the main components of nervous systems. ANNs have a long history, with the latest advances in training deeply-layered ANNs (deep learning) being responsible for some of the most impressive abilities of machine learning systems, including reinforcement learning systems. In Chapter 16 we describe several impressive examples of reinforcement learning systems that use ANN function approximation.

Figure 9.14 shows a generic feedforward ANN, meaning that there are no loops in the network, that is, there are no paths within the network by which a unit's output can influence its input. The network in the figure has an output layer consisting of two output units, an input layer with four input units, and two “hidden layers”: layers that are neither input nor output layers. A real-valued weight is associated with each link. A weight roughly corresponds to the efficacy of a synaptic connection in a real neural network (see Section 15.1). If an ANN has at least one loop in its connections, it is a recurrent rather

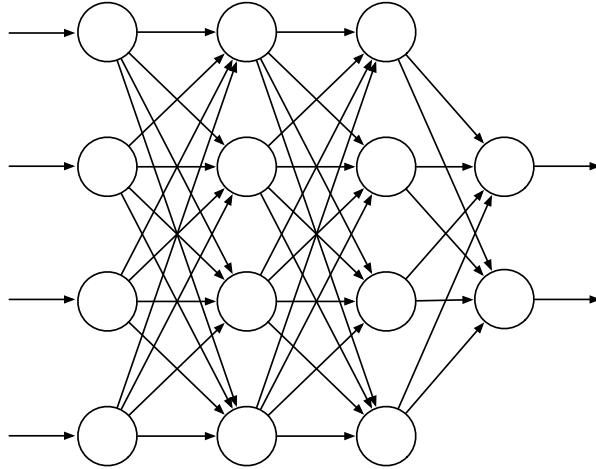


Figure 9.14: A generic feedforward ANN with four input units, two output units, and two hidden layers.

than a feedforward ANN. Although both feedforward and recurrent ANNs have been used in reinforcement learning, here we look only at the simpler feedforward case.

The units (the circles in Figure 9.14) are typically semi-linear units, meaning that they compute a weighted sum of their input signals and then apply to the result a nonlinear function, called the *activation function*, to produce the unit's output, or activation. Different activation functions are used, but they are typically S-shaped, or sigmoid, functions such as the logistic function $f(x) = 1/(1 + e^{-x})$, though sometimes the rectifier nonlinearity $f(x) = \max(0, x)$ is used. A step function like $f(x) = 1$ if $x \geq \theta$, and 0 otherwise, results in a binary unit with threshold θ . The units in a network's input layer are somewhat different in having their activations set to externally-supplied values that are the inputs to the function the network is approximating.

The activation of each output unit of a feedforward ANN is a nonlinear function of the activation patterns over the network's input units. The functions are parameterized by the network's connection weights. An ANN with no hidden layers can represent only a very small fraction of the possible input-output functions. However an ANN with a single hidden layer containing a large enough finite number of sigmoid units can approximate any continuous function on a compact region of the network's input space to any degree of accuracy (Cybenko, 1989). This is also true for other nonlinear activation functions that satisfy mild conditions, but nonlinearity is essential: if all the units in a multi-layer feedforward ANN have linear activation functions, the entire network is equivalent to a network with no hidden layers (because linear functions of linear functions are themselves linear).

Despite this “universal approximation” property of one-hidden-layer ANNs, both experience and theory show that approximating the complex functions needed for many artificial intelligence tasks is made easier—indeed may require—abstractions that are hierarchical compositions of many layers of lower-level abstractions, that is, abstractions

produced by deep architectures such as ANNs with many hidden layers. (See Bengio, 2009, for a thorough review.) The successive layers of a deep ANN compute increasingly abstract representations of the network’s “raw” input, with each unit providing a feature contributing to a hierarchical representation of the overall input-output function of the network.

Training the hidden layers of an ANN is therefore a way to automatically create features appropriate for a given problem so that hierarchical representations can be produced without relying exclusively on hand-crafted features. This has been an enduring challenge for artificial intelligence and explains why learning algorithms for ANNs with hidden layers have received so much attention over the years. ANNs typically learn by a stochastic gradient method (Section 9.3). Each weight is adjusted in a direction aimed at improving the network’s overall performance as measured by an objective function to be either minimized or maximized. In the most common supervised learning case, the objective function is the expected error, or loss, over a set of labeled training examples. In reinforcement learning, ANNs can use TD errors to learn value functions, or they can aim to maximize expected reward as in a gradient bandit (Section 2.8) or a policy-gradient algorithm (Chapter 13). In all of these cases it is necessary to estimate how a change in each connection weight would influence the network’s overall performance, in other words, to estimate the partial derivative of an objective function with respect to each weight, given the current values of all the network’s weights. The gradient is the vector of these partial derivatives.

The most successful way to do this for ANNs with hidden layers (provided the units have differentiable activation functions) is the backpropagation algorithm, which consists of alternating forward and backward passes through the network. Each forward pass computes the activation of each unit given the current activations of the network’s input units. After each forward pass, a backward pass efficiently computes a partial derivative for each weight. (As in other stochastic gradient learning algorithms, the vector of these partial derivatives is an estimate of the true gradient.) In Section 15.10 we discuss methods for training ANNs with hidden layers that use reinforcement learning principles instead of backpropagation. These methods are less efficient than the backpropagation algorithm, but they may be closer to how real neural networks learn.

The backpropagation algorithm can produce good results for shallow networks having 1 or 2 hidden layers, but it may not work well for deeper ANNs. In fact, training a network with $k + 1$ hidden layers can actually result in poorer performance than training a network with k hidden layers, even though the deeper network can represent all the functions that the shallower network can (Bengio, 2009). Explaining results like these is not easy, but several factors are important. First, the large number of weights in a typical deep ANN makes it difficult to avoid the problem of overfitting, that is, the problem of failing to generalize correctly to cases on which the network has not been trained. Second, backpropagation does not work well for deep ANNs because the partial derivatives computed by its backward passes either decay rapidly toward the input side of the network, making learning by deep layers extremely slow, or the partial derivatives grow rapidly toward the input side of the network, making learning unstable. Methods for dealing with these problems are largely responsible for many impressive recent results

achieved by systems that use deep ANNs.

Overfitting is a problem for any function approximation method that adjusts functions with many degrees of freedom on the basis of limited training data. It is less of a problem for online reinforcement learning that does not rely on limited training sets, but generalizing effectively is still an important issue. Overfitting is a problem for ANNs in general, but especially so for deep ANNs because they tend to have very large numbers of weights. Many methods have been developed for reducing overfitting. These include stopping training when performance begins to decrease on validation data different from the training data (cross validation), modifying the objective function to discourage complexity of the approximation (regularization), and introducing dependencies among the weights to reduce the number of degrees of freedom (e.g., weight sharing).

A particularly effective method for reducing overfitting by deep ANNs is the dropout method introduced by Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014). During training, units are randomly removed from the network (dropped out) along with their connections. This can be thought of as training a large number of “thinned” networks. Combining the results of these thinned networks at test time is a way to improve generalization performance. The dropout method efficiently approximates this combination by multiplying each outgoing weight of a unit by the probability that that unit was retained during training. Srivastava et al. found that this method significantly improves generalization performance. It encourages individual hidden units to learn features that work well with random collections of other features. This increases the versatility of the features formed by the hidden units so that the network does not overly specialize to rarely-occurring cases.

Hinton, Osindero, and Teh (2006) took a major step toward solving the problem of training the deep layers of a deep ANN in their work with deep belief networks, layered networks closely related to the deep ANNs discussed here. In their method, the deepest layers are trained one at a time using an unsupervised learning algorithm. Without relying on the overall objective function, unsupervised learning can extract features that capture statistical regularities of the input stream. The deepest layer is trained first, then with input provided by this trained layer, the next deepest layer is trained, and so on, until the weights in all, or many, of the network’s layers are set to values that now act as initial values for supervised learning. The network is then fine-tuned by backpropagation with respect to the overall objective function. Studies show that this approach generally works much better than backpropagation with weights initialized with random values. The better performance of networks trained with weights initialized this way could be due to many factors, but one idea is that this method places the network in a region of weight space from which a gradient-based algorithm can make good progress.

Batch normalization (Ioffe and Szegedy, 2015) is another technique that makes it easier to train deep ANNs. It has long been known that ANN learning is easier if the network input is normalized, for example, by adjusting each input variable to have zero mean and unit variance. Batch normalization for training deep ANNs normalizes the output of deep layers before they feed into the following layer. Ioffe and Szegedy (2015) used statistics from subsets, or “mini-batches,” of training examples to normalize these between-layer signals to improve the learning rate of deep ANNs.

Another technique useful for training deep ANNs is *deep residual learning* (He, Zhang, Ren, and Sun, 2016). Sometimes it is easier to learn how a function differs from the identity function than to learn the function itself. Then adding this difference, or residual function, to the input produces the desired function. In deep ANNs, a block of layers can be made to learn a residual function simply by adding shortcut, or skip, connections around the block. These connections add the input to the block to its output, and no additional weights are needed. He et al. (2016) evaluated this method using deep convolutional networks with skip connections around every pair of adjacent layers, finding substantial improvement over networks without the skip connections on benchmark image classification tasks. Both batch normalization and deep residual learning were used in the reinforcement learning application to the game of Go that we describe in Chapter 16.

A type of deep ANN that has proven to be very successful in applications, including impressive reinforcement learning applications (Chapter 16), is the *deep convolutional network*. This type of network is specialized for processing high-dimensional data arranged in spatial arrays, such as images. It was inspired by how early visual processing works in the brain (LeCun, Bottou, Bengio and Haffner, 1998). Because of its special architecture, a deep convolutional network can be trained by backpropagation without resorting to methods like those described above to train the deep layers.

Figure 9.15 illustrates the architecture of a deep convolutional network. This instance, from LeCun et al. (1998), was designed to recognize hand-written characters. It consists of alternating convolutional and subsampling layers, followed by several fully connected final layers. Each convolutional layer produces a number of feature maps. A feature map is a pattern of activity over an array of units, where each unit performs the same operation on data in its receptive field, which is the part of the data it “sees” from the preceding layer (or from the external input in the case of the first convolutional layer). The units of a feature map are identical to one another except that their receptive fields, which are all the same size and shape, are shifted to different locations on the arrays of incoming data. Units in the same feature map share the same weights. This means that a feature map detects the same feature no matter where it is located in the input

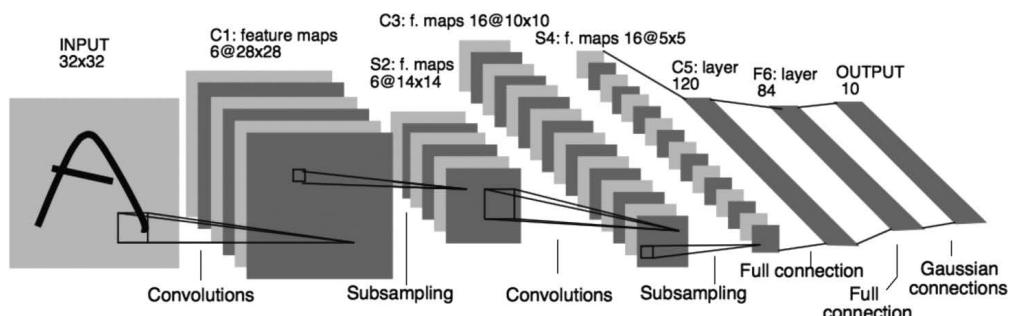


Figure 9.15: Deep Convolutional Network. Republished with permission of Proceedings of the IEEE, from Gradient-based learning applied to document recognition, LeCun, Bottou, Bengio, and Haffner, volume 86, 1998; permission conveyed through Copyright Clearance Center, Inc.

array. In the network in Figure 9.15, for example, the first convolutional layer produces 6 feature maps, each consisting of 28×28 units. Each unit in each feature map has a 5×5 receptive field, and these receptive fields overlap (in this case by four columns and four rows). Consequently, each of the 6 feature maps is specified by just 25 adjustable weights.

The subsampling layers of a deep convolutional network reduce the spatial resolution of the feature maps. Each feature map in a subsampling layer consists of units that average over a receptive field of units in the feature maps of the preceding convolutional layer. For example, each unit in each of the 6 feature maps in the first subsampling layer of the network of Figure 9.15 averages over a 2×2 non-overlapping receptive field over one of the feature maps produced by the first convolutional layer, resulting in six 14×14 feature maps. Subsampling layers reduce the network’s sensitivity to the spatial locations of the features detected, that is, they help make the network’s responses spatially invariant. This is useful because a feature detected at one place in an image is likely to be useful at other places as well.

Advances in the design and training of ANNs—of which we have only mentioned a few—all contribute to reinforcement learning. Although current reinforcement learning theory is mostly limited to methods using tabular or linear function approximation methods, the impressive performances of notable reinforcement learning applications owe much of their success to nonlinear function approximation by multi-layer ANNs. We discuss several of these applications in Chapter 16.

9.8 Least-Squares TD

All the methods we have discussed so far in this chapter have required computation per time step proportional to the number of parameters. With more computation, however, one can do better. In this section we present a method for linear function approximation that is arguably the best that can be done for this case.

As we established in Section 9.4 TD(0) with linear function approximation converges asymptotically (for appropriately decreasing step sizes) to the TD fixed point:

$$\mathbf{w}_{\text{TD}} = \mathbf{A}^{-1}\mathbf{b},$$

where

$$\mathbf{A} \doteq \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top] \quad \text{and} \quad \mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t].$$

Why, one might ask, must we compute this solution iteratively? This is wasteful of data! Could one not do better by computing estimates of \mathbf{A} and \mathbf{b} , and then directly computing the TD fixed point? The *Least-Squares TD* algorithm, commonly known as *LSTD*, does exactly this. It forms the natural estimates

$$\hat{\mathbf{A}}_t \doteq \sum_{k=0}^{t-1} \mathbf{x}_k(\mathbf{x}_k - \gamma\mathbf{x}_{k+1})^\top + \varepsilon\mathbf{I} \quad \text{and} \quad \hat{\mathbf{b}}_t \doteq \sum_{k=0}^{t-1} R_{k+1}\mathbf{x}_k, \quad (9.20)$$

where \mathbf{I} is the identity matrix, and $\varepsilon\mathbf{I}$, for some small $\varepsilon > 0$, ensures that $\widehat{\mathbf{A}}_t$ is always invertible. It might seem that these estimates should both be divided by t , and indeed they should; as defined here, these are really estimates of t times \mathbf{A} and t times \mathbf{b} . However, the extra t factors cancel out when LSTD uses these estimates to estimate the TD fixed point as

$$\mathbf{w}_t \doteq \widehat{\mathbf{A}}_t^{-1} \widehat{\mathbf{b}}_t. \quad (9.21)$$

This algorithm is the most data efficient form of linear TD(0), but it is also more expensive computationally. Recall that semi-gradient TD(0) requires memory and per-step computation that is only $O(d)$.

How complex is LSTD? As it is written above the complexity seems to increase with t , but the two approximations in (9.20) could be implemented incrementally using the techniques we have covered earlier (e.g., in Chapter 2) so that they can be done in constant time per step. Even so, the update for $\widehat{\mathbf{A}}_t$ would involve an outer product (a column vector times a row vector) and thus would be a matrix update; its computational complexity would be $O(d^2)$, and of course the memory required to hold the $\widehat{\mathbf{A}}_t$ matrix would be $O(d^2)$.

A potentially greater problem is that our final computation (9.21) uses the inverse of $\widehat{\mathbf{A}}_t$, and the computational complexity of a general inverse computation is $O(d^3)$. Fortunately, an inverse of a matrix of our special form—a sum of outer products—can also be updated incrementally with only $O(d^2)$ computations, as

$$\begin{aligned} \widehat{\mathbf{A}}_t^{-1} &= \left(\widehat{\mathbf{A}}_{t-1} + \mathbf{x}_{t-1}(\mathbf{x}_{t-1} - \gamma \mathbf{x}_t)^\top \right)^{-1} && \text{(from (9.20))} \\ &= \widehat{\mathbf{A}}_{t-1}^{-1} - \frac{\widehat{\mathbf{A}}_{t-1}^{-1} \mathbf{x}_{t-1} (\mathbf{x}_{t-1} - \gamma \mathbf{x}_t)^\top \widehat{\mathbf{A}}_{t-1}^{-1}}{1 + (\mathbf{x}_{t-1} - \gamma \mathbf{x}_t)^\top \widehat{\mathbf{A}}_{t-1}^{-1} \mathbf{x}_{t-1}}, \end{aligned} \quad (9.22)$$

for $t > 0$, with $\widehat{\mathbf{A}}_0 \doteq \varepsilon\mathbf{I}$. Although the identity (9.22), known as *the Sherman-Morrison formula*, is superficially complicated, it involves only vector-matrix and vector-vector multiplications and thus is only $O(d^2)$. Thus we can store the inverse matrix $\widehat{\mathbf{A}}_t^{-1}$, maintain it with (9.22), and then use it in (9.21), all with only $O(d^2)$ memory and per-step computation. The complete algorithm is given in the box on the next page.

Of course, $O(d^2)$ is still significantly more expensive than the $O(d)$ of semi-gradient TD. Whether the greater data efficiency of LSTD is worth this computational expense depends on how large d is, how important it is to learn quickly, and the expense of other parts of the system. The fact that LSTD requires no step-size parameter is sometimes also touted, but the advantage of this is probably overstated. LSTD does not require a step size, but it does require ε ; if ε is chosen too small the sequence of inverses can vary wildly, and if ε is chosen too large then learning is slowed. In addition, LSTD's lack of a step-size parameter means that it never forgets. This is sometimes desirable, but it is problematic if the target policy π changes as it does in reinforcement learning and GPI. In control applications, LSTD typically has to be combined with some other mechanism to induce forgetting, mooting any initial advantage of not requiring a step-size parameter.

LSTD for estimating $\hat{v} = \mathbf{w}^\top \mathbf{x}(\cdot) \approx v_\pi$ ($O(d^2)$ version)

Input: feature representation $\mathbf{x} : \mathcal{S}^+ \rightarrow \mathbb{R}^d$ such that $\mathbf{x}(\text{terminal}) = \mathbf{0}$

Algorithm parameter: small $\varepsilon > 0$

$$\widehat{\mathbf{A}}^{-1} \leftarrow \varepsilon^{-1} \mathbf{I}$$

A $d \times d$ matrix

$$\widehat{\mathbf{b}} \leftarrow \mathbf{0}$$

A d -dimensional vector

Loop for each episode:

 Initialize S ; $\mathbf{x} \leftarrow \mathbf{x}(S)$

 Loop for each step of episode:

 Choose and take action $A \sim \pi(\cdot | S)$, observe R, S' ; $\mathbf{x}' \leftarrow \mathbf{x}(S')$

$$\mathbf{v} \leftarrow \widehat{\mathbf{A}}^{-1}^\top (\mathbf{x} - \gamma \mathbf{x}')$$

$$\widehat{\mathbf{A}}^{-1} \leftarrow \widehat{\mathbf{A}}^{-1} - (\widehat{\mathbf{A}}^{-1} \mathbf{x}) \mathbf{v}^\top / (1 + \mathbf{v}^\top \mathbf{x})$$

$$\widehat{\mathbf{b}} \leftarrow \widehat{\mathbf{b}} + R \mathbf{x}$$

$$\mathbf{w} \leftarrow \widehat{\mathbf{A}}^{-1} \widehat{\mathbf{b}}$$

$$S \leftarrow S'; \mathbf{x} \leftarrow \mathbf{x}'$$

 until S' is terminal

9.9 Memory-based Function Approximation

So far we have discussed the *parametric* approach to approximating value functions. In this approach, a learning algorithm adjusts the parameters of a functional form intended to approximate the value function over a problem's entire state space. Each update, $s \mapsto g$, is a training example used by the learning algorithm to change the parameters with the aim of reducing the approximation error. After the update, the training example can be discarded (although it might be saved to be used again). When an approximate value of a state (which we will call the *query state*) is needed, the function is simply evaluated at that state using the latest parameters produced by the learning algorithm.

Memory-based function approximation methods are very different. They simply save training examples in memory as they arrive (or at least save a subset of the examples) without updating any parameters. Then, whenever a query state's value estimate is needed, a set of examples is retrieved from memory and used to compute a value estimate for the query state. This approach is sometimes called *lazy learning* because processing training examples is postponed until the system is queried to provide an output.

Memory-based function approximation methods are prime examples of *nonparametric* methods. Unlike parametric methods, the approximating function's form is not limited to a fixed parameterized class of functions, such as linear functions or polynomials, but is instead determined by the training examples themselves, together with some means for combining them to output estimated values for query states. As more training examples accumulate in memory, one expects nonparametric methods to produce increasingly accurate approximations of any target function.

There are many different memory-based methods depending on how the stored training examples are selected and how they are used to respond to a query. Here, we focus on *local-learning* methods that approximate a value function only locally in the neighborhood of the current query state. These methods retrieve a set of training examples from memory whose states are judged to be the most relevant to the query state, where relevance usually depends on the distance between states: the closer a training example's state is to the query state, the more relevant it is considered to be, where distance can be defined in many different ways. After the query state is given a value, the local approximation is discarded.

The simplest example of the memory-based approach is the *nearest neighbor* method, which simply finds the example in memory whose state is closest to the query state and returns that example's value as the approximate value of the query state. In other words, if the query state is s , and $s' \mapsto g$ is the example in memory in which s' is the closest state to s , then g is returned as the approximate value of s . Slightly more complicated are *weighted average* methods that retrieve a set of nearest neighbor examples and return a weighted average of their target values, where the weights generally decrease with increasing distance between their states and the query state. *Locally weighted regression* is similar, but it fits a surface to the values of a set of nearest states by means of a parametric approximation method that minimizes a weighted error measure like (9.1), where the weights depend on distances from the query state. The value returned is the evaluation of the locally-fitted surface at the query state, after which the local approximation surface is discarded.

Being nonparametric, memory-based methods have the advantage over parametric methods of not limiting approximations to pre-specified functional forms. This allows accuracy to improve as more data accumulates. Memory-based *local* approximation methods have other properties that make them well suited for reinforcement learning. Because trajectory sampling is of such importance in reinforcement learning, as discussed in Section 8.6, memory-based local methods can focus function approximation on local neighborhoods of states (or state-action pairs) visited in real or simulated trajectories. There may be no need for global approximation because many areas of the state space will never (or almost never) be reached. In addition, memory-based methods allow an agent's experience to have a relatively immediate affect on value estimates in the neighborhood of the current state, in contrast with a parametric method's need to incrementally adjust parameters of a global approximation.

Avoiding global approximation is also a way to address the curse of dimensionality. For example, for a state space with k dimensions, a tabular method storing a global approximation requires memory exponential in k . On the other hand, in storing examples for a memory-based method, each example requires memory proportional to k , and the memory required to store, say, n examples is linear in n . Nothing is exponential in k or n . Of course, the critical remaining issue is whether a memory-based method can answer queries quickly enough to be useful to an agent. A related concern is how speed degrades as the size of the memory grows. Finding nearest neighbors in a large database can take too long to be practical in many applications.

Proponents of memory-based methods have developed ways to accelerate the nearest neighbor search. Using parallel computers or special purpose hardware is one approach; another is the use of special multi-dimensional data structures to store the training data. One data structure studied for this application is the *k-d tree* (short for *k*-dimensional tree), which recursively splits a *k*-dimensional space into regions arranged as nodes of a binary tree. Depending on the amount of data and how it is distributed over the state space, nearest-neighbor search using *k-d* trees can quickly eliminate large regions of the space in the search for neighbors, making the searches feasible in some problems where naive searches would take too long.

Locally weighted regression additionally requires fast ways to do the local regression computations which have to be repeated to answer each query. Researchers have developed many ways to address these problems, including methods for forgetting entries in order to keep the size of the database within bounds. The Bibliographic and Historical Comments section at the end of this chapter points to some of the relevant literature, including a selection of papers describing applications of memory-based learning to reinforcement learning.

9.10 Kernel-based Function Approximation

Memory-based methods such as the weighted average and locally weighted regression methods described above depend on assigning weights to examples $s' \mapsto g$ in the database depending on the distance between s' and a query states s . The function that assigns these weights is called a *kernel function*, or simply a *kernel*. In the weighted average and locally weighted regressions methods, for example, a kernel function $k : \mathbb{R} \rightarrow \mathbb{R}$ assigns weights to distances between states. More generally, weights do not have to depend on distances; they can depend on some other measure of similarity between states. In this case, $k : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}$, so that $k(s, s')$ is the weight given to data about s' in its influence on answering queries about s .

Viewed slightly differently, $k(s, s')$ is a measure of the strength of generalization from s' to s . Kernel functions numerically express how *relevant* knowledge about any state is to any other state. As an example, the strengths of generalization for tile coding shown in Figure 9.11 correspond to different kernel functions resulting from uniform and asymmetrical tile offsets. Although tile coding does not explicitly use a kernel function in its operation, it generalizes according to one. In fact, as we discuss more below, the strength of generalization resulting from linear parametric function approximation can always be described by a kernel function.

Kernel regression is the memory-based method that computes a kernel weighted average of the targets of *all* examples stored in memory, assigning the result to the query state. If \mathcal{D} is the set of stored examples, and $g(s')$ denotes the target for state s' in a stored example, then kernel regression approximates the target function, in this case a value function depending on \mathcal{D} , as

$$\hat{v}(s, \mathcal{D}) = \sum_{s' \in \mathcal{D}} k(s, s')g(s'). \quad (9.23)$$

The weighted average method described above is a special case in which $k(s, s')$ is non-zero only when s and s' are close to one another so that the sum need not be computed over all of \mathcal{D} .

A common kernel is the Gaussian radial basis function (RBF) used in RBF function approximation as described in Section 9.5.5. In the method described there, RBFs are features whose centers and widths are either fixed from the start, with centers presumably concentrated in areas where many examples are expected to fall, or are adjusted in some way during learning. Barring methods that adjust centers and widths, this is a linear parametric method whose parameters are the weights of each RBF, which are typically learned by stochastic gradient, or semi-gradient, descent. The form of the approximation is a linear combination of the pre-determined RBFs. Kernel regression with an RBF kernel differs from this in two ways. First, it is memory-based: the RBFs are centered on the states of the stored examples. Second, it is nonparametric: there are no parameters to learn; the response to a query is given by (9.23).

Of course, many issues have to be addressed for practical implementation of kernel regression, issues that are beyond the scope of our brief discussion. However, it turns out that any linear parametric regression method like those we described in Section 9.4, with states represented by feature vectors $\mathbf{x}(s) = (x_1(s), x_2(s), \dots, x_d(s))^\top$, can be recast as kernel regression where $k(s, s')$ is the inner product of the feature vector representations of s and s' ; that is

$$k(s, s') = \mathbf{x}(s)^\top \mathbf{x}(s'). \quad (9.24)$$

Kernel regression with this kernel function produces the same approximation that a linear parametric method would if it used these feature vectors and learned with the same training data.

We skip the mathematical justification for this, which can be found in any modern machine learning text, such as Bishop (2006), and simply point out an important implication. Instead of constructing features for linear parametric function approximators, one can instead construct kernel functions directly without referring at all to feature vectors. Not all kernel functions can be expressed as inner products of feature vectors as in (9.24), but a kernel function that can be expressed like this can offer significant advantages over the equivalent parametric method. For many sets of feature vectors, (9.24) has a compact functional form that can be evaluated without any computation taking place in the d -dimensional feature space. In these cases, kernel regression is much less complex than directly using a linear parametric method with states represented by these feature vectors. This is the so-called “kernel trick” that allows effectively working in the high-dimension of an expansive feature space while actually working only with the set of stored training examples. The kernel trick is the basis of many machine learning methods, and researchers have shown how it can sometimes benefit reinforcement learning.

9.11 Looking Deeper at On-policy Learning: Interest and Emphasis

The algorithms we have considered so far in this chapter have treated all the states encountered equally, as if they were all equally important. In some cases, however, we are more interested in some states than others. In discounted episodic problems, for example, we may be more interested in accurately valuing early states in the episode than in later states where discounting may have made the rewards much less important to the value of the start state. Or, if an action-value function is being learned, it may be less important to accurately value poor actions whose value is much less than the greedy action. Function approximation resources are always limited, and if they were used in a more targeted way, then performance could be improved.

One reason we have treated all states encountered equally is that then we are updating according to the on-policy distribution, for which stronger theoretical results are available for semi-gradient methods. Recall that the on-policy distribution was defined as the distribution of states encountered in an MDP while following the target policy. Now we will generalize this concept significantly. Rather than having one on-policy distribution for the MDP, we will have many. All of them will have in common that they are a distribution of states encountered in trajectories while following the target policy, but they will vary in how the trajectories are, in a sense, initiated.

We now introduce some new concepts. First we introduce a non-negative scalar measure, a random variable I_t called *interest*, indicating the degree to which we are interested in accurately valuing the state (or state-action pair) at time t . If we don't care at all about the state, then the interest should be zero; if we fully care, it might be one, though it is formally allowed to take any non-negative value. The interest can be set in any causal way; for example, it may depend on the trajectory up to time t or the learned parameters at time t . The distribution μ in the $\overline{\text{VE}}$ (9.1) is then defined as the distribution of states encountered while following the target policy, weighted by the interest. Second, we introduce another non-negative scalar random variable, the *emphasis* M_t . This scalar multiplies the learning update and thus emphasizes or de-emphasizes the learning done at time t . The general n -step learning rule, replacing (9.15), is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha M_t [G_{t:t+n} - \hat{v}(S_t, \mathbf{w}_{t+n-1})] \nabla \hat{v}(S_t, \mathbf{w}_{t+n-1}), \quad 0 \leq t < T, \quad (9.25)$$

with the n -step return given by (9.16) and the emphasis determined recursively from the interest by:

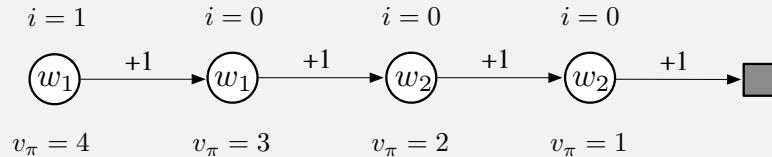
$$M_t = I_t + \gamma^n M_{t-n}, \quad 0 \leq t < T, \quad (9.26)$$

with $M_t \doteq 0$, for all $t < 0$. These equations are taken to include the Monte Carlo case, for which $G_{t:t+n} = G_t$, all the updates are made at end of the episode, $n = T - t$, and $M_t = I_t$.

Example 9.4 illustrates how interest and emphasis can result in more accurate value estimates.

Example 9.4: Interest and Emphasis

To see the potential benefits of using interest and emphasis, consider the four-state Markov reward process shown below:



Episodes start in the leftmost state, then transition one state to the right, with a reward of $+1$, on each step until the terminal state is reached. The true value of the first state is thus 4, of the second state 3, and so on as shown below each state. These are the true values; the estimated values can only approximate these because they are constrained by the parameterization. There are two components to the parameter vector $\mathbf{w} = (w_1, w_2)^\top$, and the parameterization is as written inside each state. The estimated values of the first two states are given by w_1 alone and thus must be the same even though their true values are different. Similarly, the estimated values of the third and fourth states are given by w_2 alone and must be the same even though their true values are different. Suppose that we are interested in accurately valuing only the leftmost state; we assign it an interest of 1 while all the other states are assigned an interest of 0, as indicated above the states.

First consider applying gradient Monte Carlo algorithms to this problem. The algorithms presented earlier in this chapter that do not take into account interest and emphasis (in (9.7) and the box on page 202) will converge (for decreasing step sizes) to the parameter vector $\mathbf{w}_\infty = (3.5, 1.5)$, which gives the first state—the only one we are interested in—a value of 3.5 (i.e., intermediate between the true values of the first and second states). The methods presented in this section that do use interest and emphasis, on the other hand, will learn the value of the first state exactly correctly; w_1 will converge to 4 while w_2 will never be updated because the emphasis is zero in all states save the leftmost.

Now consider applying two-step semi-gradient TD methods. The methods from earlier in this chapter without interest and emphasis (in (9.15) and (9.16) and the box on page 209) will again converge to $\mathbf{w}_\infty = (3.5, 1.5)$, while the methods with interest and emphasis converge to $\mathbf{w}_\infty = (4, 2)$. The latter produces the exactly correct values for the first state and for the third state (which the first state bootstraps from) while never making any updates corresponding to the second or fourth states.

9.12 Summary

Reinforcement learning systems must be capable of *generalization* if they are to be applicable to artificial intelligence or to large engineering applications. To achieve this, any of a broad range of existing methods for *supervised-learning function approximation* can be used simply by treating each update as a training example.

Perhaps the most suitable supervised learning methods are those using *parameterized function approximation*, in which the policy is parameterized by a weight vector \mathbf{w} . Although the weight vector has many components, the state space is much larger still, and we must settle for an approximate solution. We defined the *mean square value error*, $\overline{\text{VE}}(\mathbf{w})$, as a measure of the error in the values $v_{\pi_{\mathbf{w}}}(s)$ for a weight vector \mathbf{w} under the *on-policy distribution*, μ . The $\overline{\text{VE}}$ gives us a clear way to rank different value-function approximations in the on-policy case.

To find a good weight vector, the most popular methods are variations of *stochastic gradient descent* (SGD). In this chapter we have focused on the *on-policy* case with a *fixed policy*, also known as policy evaluation or prediction; a natural learning algorithm for this case is *n-step semi-gradient TD*, which includes gradient Monte Carlo and semi-gradient TD(0) algorithms as the special cases when $n = \infty$ and $n = 1$ respectively. Semi-gradient TD methods are not true gradient methods. In such bootstrapping methods (including DP), the weight vector appears in the update target, yet this is not taken into account in computing the gradient—thus they are *semi-gradient* methods. As such, they cannot rely on classical SGD results.

Nevertheless, good results can be obtained for semi-gradient methods in the special case of *linear* function approximation, in which the value estimates are sums of features times corresponding weights. The linear case is the most well understood theoretically and works well in practice when provided with appropriate features. Choosing the features is one of the most important ways of adding prior domain knowledge to reinforcement learning systems. They can be chosen as polynomials, but this case generalizes poorly in the online learning setting typically considered in reinforcement learning. Better is to choose features according the Fourier basis, or according to some form of coarse coding with sparse overlapping receptive fields. Tile coding is a form of coarse coding that is particularly computationally efficient and flexible. Radial basis functions are useful for one- or two-dimensional tasks in which a smoothly varying response is important. LSTD is the most data-efficient linear TD prediction method, but requires computation proportional to the square of the number of weights, whereas all the other methods are of complexity linear in the number of weights. Nonlinear methods include artificial neural networks trained by backpropagation and variations of SGD; these methods have become very popular in recent years under the name *deep reinforcement learning*.

Linear semi-gradient *n*-step TD is guaranteed to converge under standard conditions, for all n , to a $\overline{\text{VE}}$ that is within a bound of the optimal error (achieved asymptotically by Monte Carlo methods). This bound is always tighter for higher n and approaches zero as $n \rightarrow \infty$. However, in practice very high n results in very slow learning, and some degree of bootstrapping ($n < \infty$) is usually preferable, just as we saw in comparisons of tabular *n*-step methods in Chapter 7 and in comparisons of tabular TD and Monte Carlo methods in Chapter 6.

Exercise 9.7 One of the simplest artificial neural networks consists of a single semi-linear unit with a logistic nonlinearity. The need to handle approximate value functions of this form is common in games that end with either a win or a loss, in which case the value of a state can be interpreted as the probability of winning. Derive the learning algorithm for this case, from (9.7), such that no gradient notation appears.

**Exercise 9.8* Arguably, the squared error used to derive (9.7) is inappropriate for the case treated in the preceding exercise, and the right error measure is the *cross-entropy loss* (which you can find on Wikipedia). Repeat the derivation in Section 9.3, using the cross-entropy loss instead of the squared error in (9.4), all the way to an explicit form with no gradient or logarithm notation in it. Is your final form more complex, or simpler, than that you obtained in the preceding exercise?

Bibliographical and Historical Remarks

Generalization and function approximation have always been an integral part of reinforcement learning. Bertsekas and Tsitsiklis (1996), Bertsekas (2012), and Sugiyama et al. (2013) present the state of the art in function approximation in reinforcement learning. Some of the early work with function approximation in reinforcement learning is discussed at the end of this section.

9.3 Gradient-descent methods for minimizing mean square error in supervised learning are well known. Widrow and Hoff (1960) introduced the least-mean-square (LMS) algorithm, which is the prototypical incremental gradient-descent algorithm. Details of this and related algorithms are provided in many texts (e.g., Widrow and Stearns, 1985; Bishop, 1995; Duda and Hart, 1973).

Semi-gradient TD(0) was first explored by Sutton (1984, 1988), as part of the linear TD(λ) algorithm that we will treat in Chapter 12. The term “semi-gradient” to describe these bootstrapping methods is new to the second edition of this book.

The earliest use of state aggregation in reinforcement learning may have been Michie and Chambers’s BOXES system (1968). The theory of state aggregation in reinforcement learning has been developed by Singh, Jaakkola, and Jordan (1995) and Tsitsiklis and Van Roy (1996). State aggregation has been used in dynamic programming from its earliest days (e.g., Bellman, 1957a).

9.4 Sutton (1988) proved convergence of linear TD(0) in the mean to the minimal \overline{V} solution for the case in which the feature vectors, $\{\mathbf{x}(s) : s \in \mathcal{S}\}$, are linearly independent. Convergence with probability 1 was proved by several researchers at about the same time (Peng, 1993; Dayan and Sejnowski, 1994; Tsitsiklis, 1994; Gurvits, Lin, and Hanson, 1994). In addition, Jaakkola, Jordan, and Singh (1994) proved convergence under online updating. All of these results assumed linearly independent feature vectors, which implies at least as many components to \mathbf{w}_t as there are states. Convergence for the more important case of general (dependent) feature vectors was first shown by Dayan (1992). A significant

generalization and strengthening of Dayan’s result was proved by Tsitsiklis and Van Roy (1997). They proved the main result presented in this section, the bound on the asymptotic error of linear bootstrapping methods.

- 9.5** Our presentation of the range of possibilities for linear function approximation is based on that by Barto (1990).
- 9.5.2** Konidaris, Osentoski, and Thomas (2011) introduced the Fourier basis in a simple form suitable for reinforcement learning problems with multi-dimensional continuous state spaces and functions that do not have to be periodic.
- 9.5.3** The term *coarse coding* is due to Hinton (1984), and our Figure 9.6 is based on one of his figures. Waltz and Fu (1965) provide an early example of this type of function approximation in a reinforcement learning system.
- 9.5.4** Tile coding, including hashing, was introduced by Albus (1971, 1981). He described it in terms of his “cerebellar model articulator controller,” or CMAC, as tile coding is sometimes known in the literature. The term “tile coding” was new to the first edition of this book, though the idea of describing CMAC in these terms is taken from Watkins (1989). Tile coding has been used in many reinforcement learning systems (e.g., Shewchuk and Dean, 1990; Lin and Kim, 1991; Miller, Scalera, and Kim, 1994; Sofge and White, 1992; Tham, 1994; Sutton, 1996; Watkins, 1989) as well as in other types of learning control systems (e.g., Kraft and Campagna, 1990; Kraft, Miller, and Dietz, 1992). This section draws heavily on the work of Miller and Glanz (1996). General software for tile coding is available in several languages (e.g., see <http://incompleteideas.net/tiles/tiles3.html>).
- 9.5.5** Function approximation using radial basis functions has received wide attention ever since being related to ANNs by Broomhead and Lowe (1988). Powell (1987) reviewed earlier uses of RBFs, and Poggio and Girosi (1989, 1990) extensively developed and applied this approach.
- 9.6** Automatic methods for adapting the step-size parameter include RMSprop (Tieleman and Hinton, 2012), Adam (Kingma and Ba, 2015), stochastic meta-descent methods such as Delta-Bar-Delta (Jacobs, 1988), its incremental generalization (Sutton, 1992b, c; Mahmood et al., 2012), and nonlinear generalizations (Schraudolph, 1999, 2002). Methods explicitly designed for reinforcement learning include AlphaBound (Dabney and Barto, 2012), SID and NOSID (Dabney, 2014), TIDBD (Kearney et al., in preparation) and the application of stochastic meta-descent to policy gradient learning (Schraudolph, Yu, and Aberdeen, 2006).
- 9.7** The introduction of the threshold logic unit as an abstract model neuron by McCulloch and Pitts (1943) was the beginning of ANNs. The history of ANNs as learning methods for classification or regression has passed through several stages: roughly, the Perceptron (Rosenblatt, 1962) and ADALINE (ADaptive LINEar Element) (Widrow and Hoff, 1960) stage of learning by single-layer ANNs, the

error-backpropagation stage (LeCun, 1985; Rumelhart, Hinton, and Williams, 1986) of learning by multi-layer ANNs, and the current deep-learning stage with its emphasis on representation learning (e.g., Bengio, Courville, and Vincent, 2012; Goodfellow, Bengio, and Courville, 2016). Examples of the many books on ANNs are Haykin (1994), Bishop (1995), and Ripley (2007).

ANNs as function approximation for reinforcement learning goes back to the early work of Farley and Clark (1954), who used reinforcement-like learning to modify the weights of linear threshold functions representing policies. Widrow, Gupta, and Maitra (1973) presented a neuron-like linear threshold unit implementing a learning process they called *learning with a critic* or *selective bootstrap adaptation*, a reinforcement-learning variant of the ADALINE algorithm. Werbos (1987, 1994) developed an approach to prediction and control that uses ANNs trained by error backpropagation to learn policies and value functions using TD-like algorithms. Barto, Sutton, and Brouwer (1981) and Barto and Sutton (1981b) extended the idea of an associative memory network (e.g., Kohonen, 1977; Anderson, Silverstein, Ritz, and Jones, 1977) to reinforcement learning. Barto, Anderson, and Sutton (1982) used a two-layer ANN to learn a nonlinear control policy, and emphasized the first layer's role of learning a suitable representation. Hampson (1983, 1989) was an early proponent of multilayer ANNs for learning value functions. Barto, Sutton, and Anderson (1983) presented an actor-critic algorithm in the form of an ANN learning to balance a simulated pole (see Sections 15.7 and 15.8). Barto and Anandan (1985) introduced a stochastic version of Widrow et al.'s (1973) selective bootstrap algorithm called the *associative reward-penalty (A_{R-P}) algorithm*. Barto (1985, 1986) and Barto and Jordan (1987) described multi-layer ANNs consisting of A_{R-P} units trained with a globally-broadcast reinforcement signal to learn classification rules that are not linearly separable. Barto (1985) discussed this approach to ANNs and how this type of learning rule is related to others in the literature at that time. (See Section 15.10 for additional discussion of this approach to training multi-layer ANNs.) Anderson (1986, 1987, 1989) evaluated numerous methods for training multilayer ANNs and showed that an actor-critic algorithm in which both the actor and critic were implemented by two-layer ANNs trained by error backpropagation outperformed single-layer ANNs in the pole-balancing and tower of Hanoi tasks. Williams (1988) described several ways that backpropagation and reinforcement learning can be combined for training ANNs. Gullapalli (1990) and Williams (1992) devised reinforcement learning algorithms for neuron-like units having continuous, rather than binary, outputs. Barto, Sutton, and Watkins (1990) argued that ANNs can play significant roles for approximating functions required for solving sequential decision problems. Williams (1992) related REINFORCE learning rules (Section 13.3) to the error backpropagation method for training multi-layer ANNs. Tesauro's TD-Gammon (Tesauro 1992, 1994; Section 16.1) influentially demonstrated the learning abilities of $TD(\lambda)$ algorithm with function approximation by multi-layer ANNs in learning to play backgammon. The *AlphaGo*, *AlphaGo Zero*, and *AlphaZero* programs of Silver et al. (2016, 2017a, b; Section 16.6) used reinforcement learning with

deep convolutional ANNs in achieving impressive results with the game of Go. Schmidhuber (2015) reviews applications of ANNs in reinforcement learning, including applications of recurrent ANNs.

9.8 LSTD is due to Bradtke and Barto (see Bradtke, 1993, 1994; Bradtke and Barto, 1996; Bradtke, Ydstie, and Barto, 1994), and was further developed by Boyan (1999, 2002), Nedić and Bertsekas (2003), and Yu (2010). The incremental update of the inverse matrix has been known at least since 1949 (Sherman and Morrison, 1949). An extension of least-squares methods to control was introduced by Lagoudakis and Parr (2003; Buşoniu, Lazaric, Ghavamzadeh, Munos, Babuška, and De Schutter, 2012).

9.9 Our discussion of memory-based function approximation is largely based on the review of locally weighted learning by Atkeson, Moore, and Schaal (1997). Atkeson (1992) discussed the use of locally weighted regression in memory-based robot learning and supplied an extensive bibliography covering the history of the idea. Stanfill and Waltz (1986) influentially argued for the importance of memory based methods in artificial intelligence, especially in light of parallel architectures then becoming available, such as the Connection Machine. Baird and Klopf (1993) introduced a novel memory-based approach and used it as the function approximation method for Q-learning applied to the pole-balancing task. Schaal and Atkeson (1994) applied locally weighted regression to a robot juggling control problem, where it was used to learn a system model. Peng (1995) used the pole-balancing task to experiment with several nearest-neighbor methods for approximating value functions, policies, and environment models. Tadepalli and Ok (1996) obtained promising results with locally-weighted linear regression to learn a value function for a simulated automatic guided vehicle task. Bottou and Vapnik (1992) demonstrated surprising efficiency of several local learning algorithms compared to non-local algorithms in some pattern recognition tasks, discussing the impact of local learning on generalization.

Bentley (1975) introduced k -d trees and reported observing average running time of $O(\log n)$ for nearest neighbor search over n records. Friedman, Bentley, and Finkel (1977) clarified the algorithm for nearest neighbor search with k -d trees. Omohundro (1987) discussed efficiency gains possible with hierarchical data structures such as k -d-trees. Moore, Schneider, and Deng (1997) introduced the use of k -d trees for efficient locally weighted regression.

9.10 The origin of kernel regression is the *method of potential functions* of Aizerman, Braverman, and Rozonoer (1964). They likened the data to point electric charges of various signs and magnitudes distributed over space. The resulting electric potential over space produced by summing the potentials of the point charges corresponded to the interpolated surface. In this analogy, the kernel function is the potential of a point charge, which falls off as the reciprocal of the distance from the charge. Connell and Utgoff (1987) applied an actor–critic method to the pole-balancing task in which the critic approximated the value function

using kernel regression with an inverse-distance weighting. Predating widespread interest in kernel regression in machine learning, these authors did not use the term kernel, but referred to “Shepard’s method” (Shepard, 1968). Other kernel-based approaches to reinforcement learning include those of Ormoneit and Sen (2002), Dietterich and Wang (2002), Xu, Xie, Hu, and Lu (2005), Taylor and Parr (2009), Barreto, Precup, and Pineau (2011), and Bhat, Farias, and Moallemi (2012).

9.11 For Emphatic-TD methods, see the bibliographical notes to Section 11.8.

The earliest example we know of in which function approximation methods were used for learning value functions was Samuel’s checkers player (1959, 1967). Samuel followed Shannon’s (1950) suggestion that a value function did not have to be exact to be a useful guide to selecting moves in a game and that it might be approximated by a linear combination of features. In addition to linear function approximation, Samuel experimented with lookup tables and hierarchical lookup tables called signature tables (Griffith, 1966, 1974; Page, 1977; Biermann, Fairfield, and Beres, 1982).

At about the same time as Samuel’s work, Bellman and Dreyfus (1959) proposed using function approximation methods with DP. (It is tempting to think that Bellman and Samuel had some influence on one another, but we know of no reference to the other in the work of either.) There is now a fairly extensive literature on function approximation methods and DP, such as multigrid methods and methods using splines and orthogonal polynomials (e.g., Bellman and Dreyfus, 1959; Bellman, Kalaba, and Kotkin, 1963; Daniel, 1976; Whitt, 1978; Reetz, 1977; Schweitzer and Seidmann, 1985; Chow and Tsitsiklis, 1991; Kushner and Dupuis, 1992; Rust, 1996).

Holland’s (1986) classifier system used a selective feature-match technique to generalize evaluation information across state–action pairs. Each classifier matched a subset of states having specified values for a subset of features, with the remaining features having arbitrary values (“wild cards”). These subsets were then used in a conventional state-aggregation approach to function approximation. Holland’s idea was to use a genetic algorithm to evolve a set of classifiers that collectively would implement a useful action-value function. Holland’s ideas influenced the early research of the authors on reinforcement learning, but we focused on different approaches to function approximation. As function approximators, classifiers are limited in several ways. First, they are state-aggregation methods, with concomitant limitations in scaling and in representing smooth functions efficiently. In addition, the matching rules of classifiers can implement only aggregation boundaries that are parallel to the feature axes. Perhaps the most important limitation of conventional classifier systems is that the classifiers are learned via the genetic algorithm, an evolutionary method. As we discussed in Chapter 1, there is available during learning much more detailed information about how to learn than can be used by evolutionary methods. This perspective led us to instead adapt supervised learning methods for use in reinforcement learning, specifically gradient-descent and ANN methods. These differences between Holland’s approach and ours are not surprising because Holland’s ideas were developed during a period when ANNs were generally regarded as being too weak in computational power to be useful, whereas our work was at the beginning of

the period that saw widespread questioning of that conventional wisdom. There remain many opportunities for combining aspects of these different approaches.

Christensen and Korf (1986) experimented with regression methods for modifying coefficients of linear value function approximations in the game of chess. Chapman and Kaelbling (1991) and Tan (1991) adapted decision-tree methods for learning value functions. Explanation-based learning methods have also been adapted for learning value functions, yielding compact representations (Yee, Saxena, Utgoff, and Barto, 1990; Dietterich and Flann, 1995).