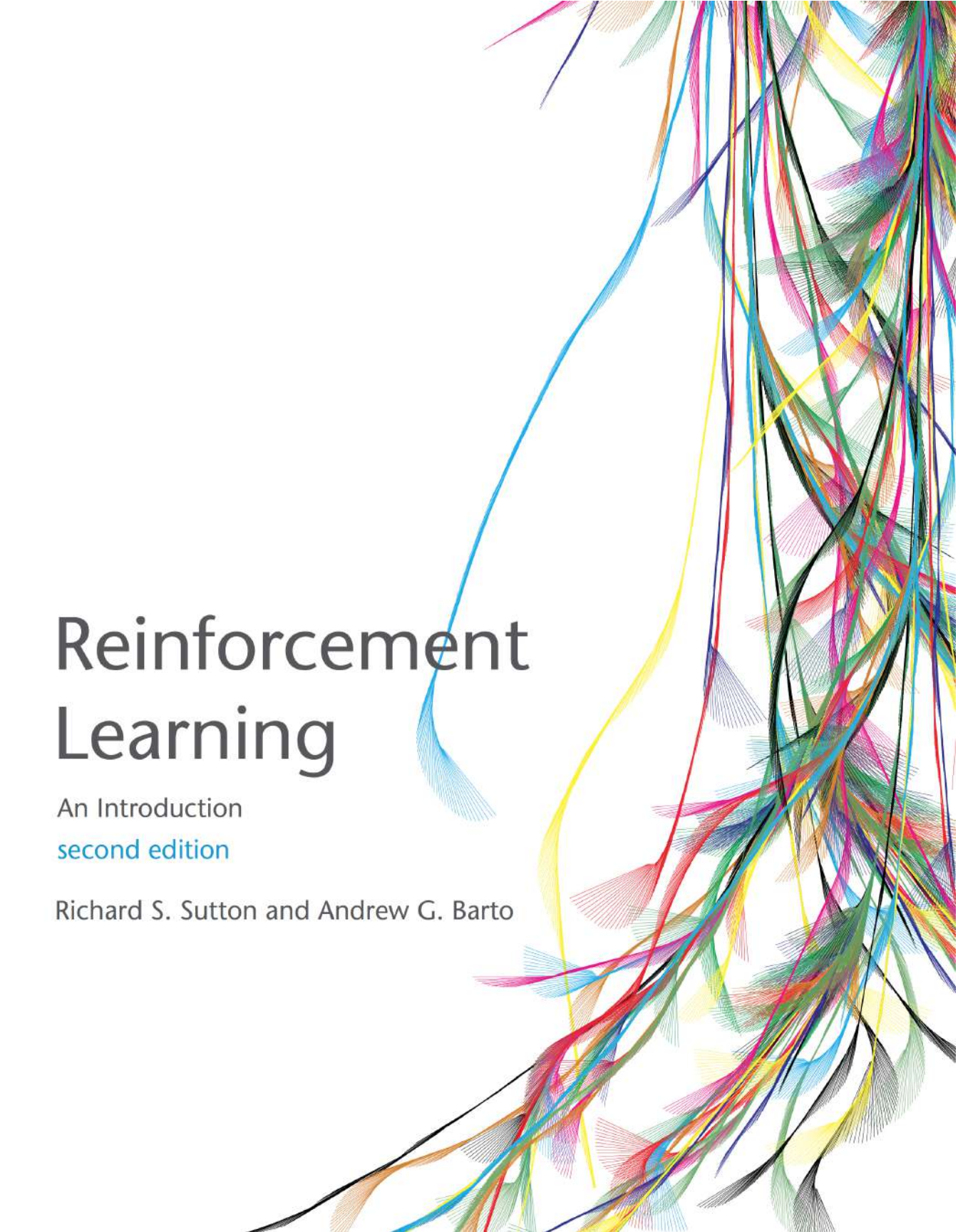


Reinforcement Learning

An Introduction

second edition

Richard S. Sutton and Andrew G. Barto

Adaptive Computation and Machine Learning

Francis Bach

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

The cover design is based on the trajectories of a simulated bicycle controlled by a reinforcement learning system developed by Jette Randløv.

Reinforcement Learning:

An Introduction

second edition

Richard S. Sutton and Andrew G. Barto

The MIT Press

Cambridge, Massachusetts

London, England

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

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 $\boldsymbol{\phi}_t$), 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, $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$, 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 $p(s', r \mid s, a)$ 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 Szepesvári (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 Klopf. 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 Ph.D. 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 Ph.D. 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, 15.4, 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, Pawel 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
$f : \mathcal{X} \to \mathcal{Y}$	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
$\mathbb{1}_{predicate}$	indicator function ($\mathbb{1}_{predicate} \doteq 1$ if the *predicate* 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., $s \in \mathcal{S}$, $r \in \mathcal{R}$
$\lvert\mathcal{S}\rvert$	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 *deterministic* policy π
$\pi(a\lvert s)$	probability of taking action a in state s under *stochastic* 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)
$\bar{G}_{t:h}$	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)
$p(s', r \,\vert\, s, a)$	probability of transition to state s' with reward r, from state s and action a
$p(s' \,\vert\, s, a)$	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, e.g., $\bar{V}_t(s) \doteq \sum_a \pi(a\lvert s) Q_t(s, a)$
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
$w_i, w_{t,i}$	ith 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
$x_i(s), x_i(s,a)$	ith 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, $\mathbf{w}^\top\mathbf{x} \doteq \sum_i w_i x_i$; e.g., $\hat{v}(s,\mathbf{w}) \doteq \mathbf{w}^\top\mathbf{x}(s)$
$\mathbf{v}, \mathbf{v}_t$	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)

$\mathbf{A}$	$d \times d$ matrix $\mathbf{A} \doteq \mathbb{E}\Big[\mathbf{x}_t\big(\mathbf{x}_t - \gamma\mathbf{x}_{t+1}\big)^\top\Big]$
$\mathbf{b}$	d-dimensional vector $\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t]$
$\mathbf{w}_{\text{TD}}$	TD fixed point $\mathbf{w}_{\text{TD}} \doteq \mathbf{A}^{-1}\mathbf{b}$ (a d-vector, Section 9.4)
$\mathbf{I}$	identity matrix
$\mathbf{P}$	$\lvert\mathcal{S}\rvert \times \lvert\mathcal{S}\rvert$ matrix of state-transition probabilities under π
$\mathbf{D}$	$\lvert\mathcal{S}\rvert \times \lvert\mathcal{S}\rvert$ diagonal matrix with $\boldsymbol{\mu}$ on its diagonal
$\mathbf{X}$	$\lvert\mathcal{S}\rvert \times d$ 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 $\overline{\text{VE}}(\mathbf{w}) \doteq \lVert v_{\mathbf{w}} - v_\pi\rVert^2_\mu$ (Section 9.2)
$\overline{\text{BE}}(\mathbf{w})$	mean square Bellman error $\overline{\text{BE}}(\mathbf{w}) \doteq \big\lVert\bar{\delta}_{\mathbf{w}}\big\rVert^2_\mu$
$\overline{\text{PBE}}(\mathbf{w})$	mean square projected Bellman error $\overline{\text{PBE}}(\mathbf{w}) \doteq \big\lVert\Pi\bar{\delta}_{\mathbf{w}}\big\rVert^2_\mu$
$\overline{\text{TDE}}(\mathbf{w})$	mean square temporal-difference error $\overline{\text{TDE}}(\mathbf{w}) \doteq \mathbb{E}_b\big[\rho_t\delta_t^2\big]$ (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.[1] That is, we adopt the perspective 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

[1] The relationships to psychology and neuroscience are summarized in Chapters 14 and 15.

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 knowledgable external supervisor. Each example is a description of a situation together with a specification—the label—of the correct action the system should take to 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 much of machine learning research is concerned with supervised learning without explicitly specifying how such an ability would finally 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. One must 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 1960's, 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 still common today, but not dominant. From our point of view, it was simply 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 this 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 the move, then the update to the estimated value of S_t, denoted $V(S_t)$, can be written as

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

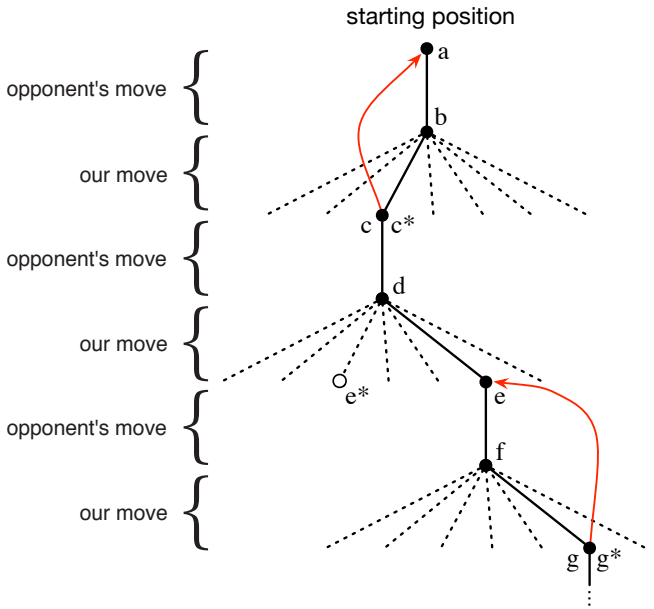


Figure 1.1: A sequence of tic-tac-toe moves. The solid black lines represent the moves taken during a game; the dashed lines represent moves that we (our reinforcement learning player) considered but did not make. Our second move was an exploratory move, meaning that it was taken even though another 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.

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 (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 its 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 with reinforcement learning. Artificial neural networks and deep learning (Section 9.6) 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? □

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? □

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? □

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; and 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 offline 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 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 Ph.D. 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 artificial neural network textbooks have used the term "trial-and-error" to describe 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 McClaren, 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, 1969a,b). 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 consistently emphasized the role of trial and error and learning as essential aspects of artificial intelligence (Michie, 1974).

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 is 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 *Al*gorithm *o*f *p*attern *ex*traction) 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 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, and Koza, 1992).

The individual most responsible for reviving the trial-and-error thread to reinforcement learning within artificial intelligence was Harry Klopf (1972, 1975, 1982). Klopf 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 Klopf, 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. Klopf'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) Ph.D. dissertation and extended to use backpropagation neural networks in Anderson's (1986) Ph.D. 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 TD(λ) 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 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 Szepesvári (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 actions are taken in more than one 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:

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

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 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:

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

where $\mathbb{1}_{predicate}$ 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

$$A_t \doteq \operatorname*{arg\,max}_a Q_t(a), \tag{2.2}$$

where $\operatorname{arg\,max}_a$ denotes the action a for which the expression that follows is maximized (again, 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 $q_*(a)$. 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? □

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, $q_*(a)$, $a = 1, \ldots, 10$,

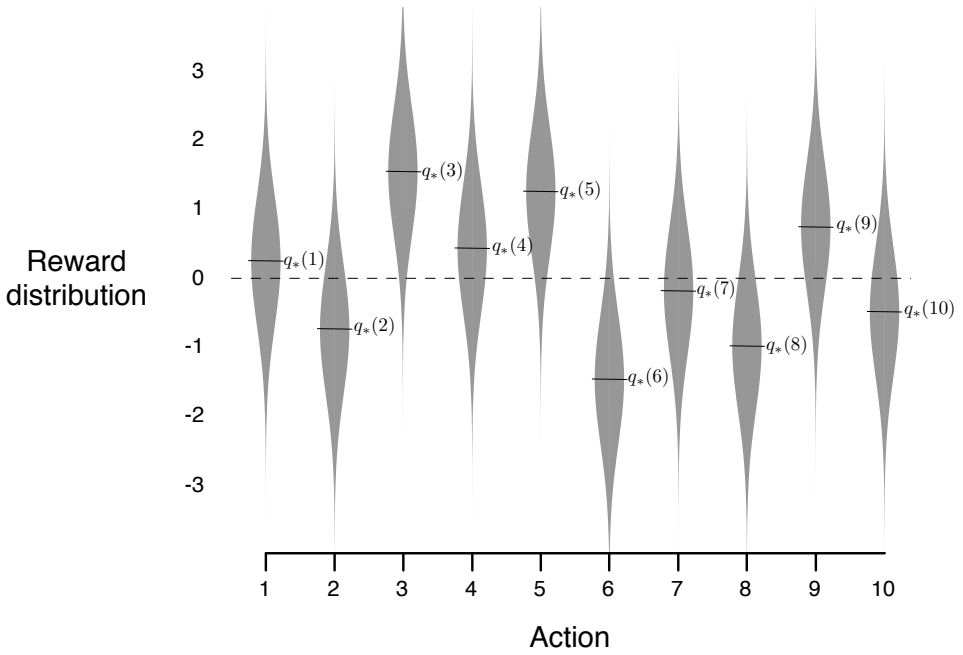


Figure 2.1: An example bandit problem from the 10-armed testbed. The true value $q_*(a)$ of each of the ten actions was selected according to a normal distribution with mean zero and unit variance, and then the actual rewards were selected according to a mean $q_*(a)$ unit variance normal distribution,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

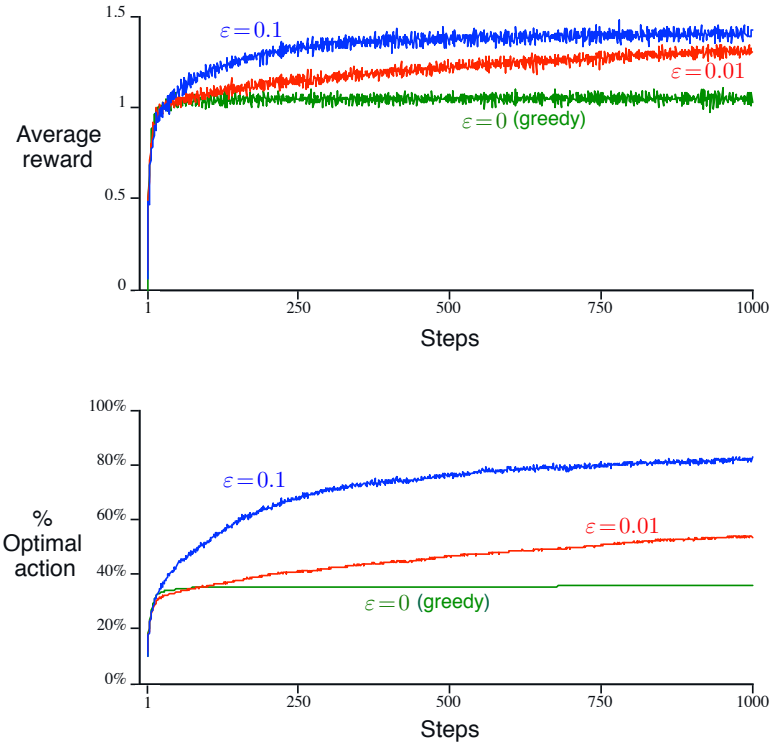


Figure 2.2: Average performance of ε-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems. All methods used sample averages as their action-value estimates.

often got stuck performing suboptimal actions. The lower graph 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? □

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. □

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 ith 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

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

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 nth reward, R_n, the new average of all n rewards can be computed by

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

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

$$\textit{NewEstimate} \leftarrow \textit{OldEstimate} \;+\; \textit{StepSize}\,\Big[\textit{Target} - \textit{OldEstimate}\Big]. \qquad (2.4)$$

The expression $\big[\textit{Target} - \textit{OldEstimate}\big]$ 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 nth 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 nth 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:
$Q(a) \leftarrow 0$
$N(a) \leftarrow 0$

Loop forever:
$A \leftarrow \begin{cases} \operatorname{argmax}_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$
$R \leftarrow bandit(A)$
$N(A) \leftarrow N(A) + 1$
$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\big[R - Q(A)\big]$

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

$$Q_{n+1} \doteq Q_n + \alpha\Big[R_n - Q_n\Big], \tag{2.5}$$

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:

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

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 nth 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:

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

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? □

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 zero 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. □

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

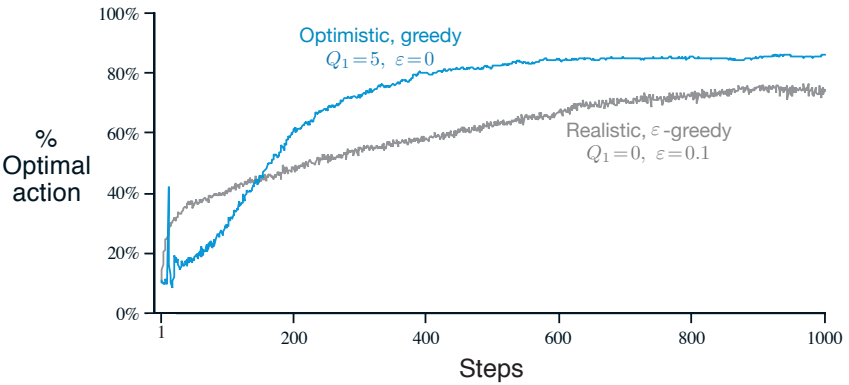


Figure 2.3: The effect of optimistic initial action-value estimates on the 10-armed testbed. Both methods used a constant step-size parameter, $\alpha = 0.1$.

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? □

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

$$\beta_n \doteq \alpha/\bar{o}_n, \tag{2.8}$$

to process the nth 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:

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

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

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

$$A_t \doteq \operatorname*{arg\,max}_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right], \tag{2.10}$$

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.

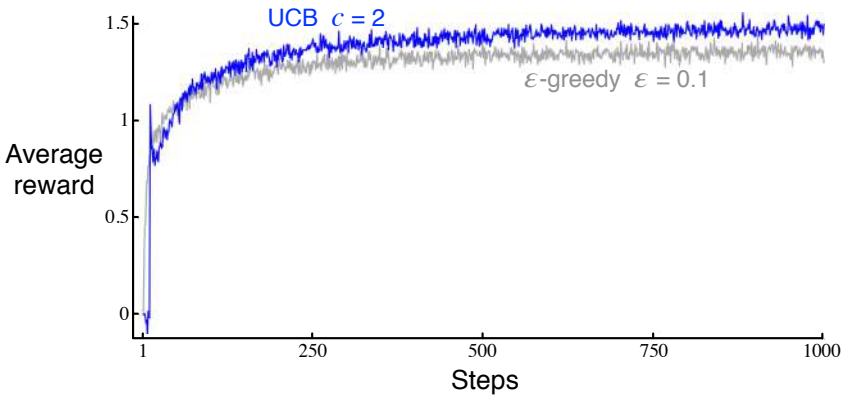


Figure 2.4: Average performance of UCB action selection on the 10-armed testbed. As shown, UCB generally performs better than ε-greedy action selection, except in the first k steps, when it selects randomly among the as-yet-untried actions.

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. □

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 $H_t(a)$. 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:

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

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. □

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:

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

where $\alpha > 0$ is a step-size parameter, and $\bar{R}_t \in \mathbb{R}$ is the average of all the rewards up to but not including time t, which can be computed incrementally as described in Section 2.4 (or Section 2.5 if the problem is nonstationary).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.

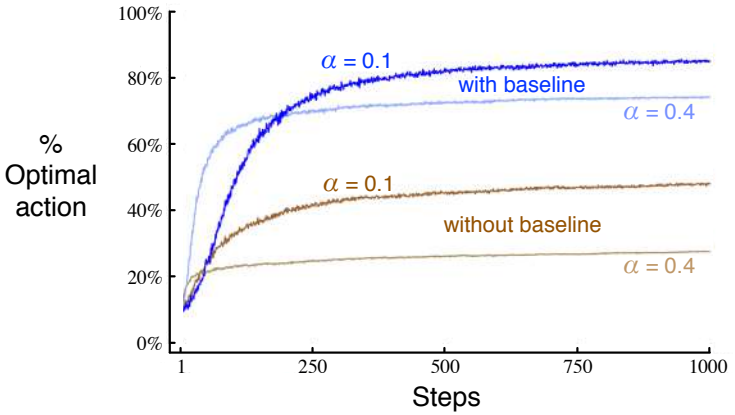


Figure 2.5: Average performance of the gradient bandit algorithm with and without a reward baseline on the 10-armed testbed when the $q_*(a)$ are chosen to be near $+4$ rather than near zero.

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 proportional to the increment's effect on performance:

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

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:

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

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 $\pi_t(x)/\pi_t(x)$:

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

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:

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

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 $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)\big(\mathbb{1}_{a=x} - \pi_t(a)\big)$, where $\mathbb{1}_{a=x}$ is defined to be 1 if $a = x$, else 0. Assuming that for now, we have

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

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 proportional to the sample. Substituting a sample of the expectation above for the performance gradient in (2.13) yields:

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

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

Thus it remains only to show that $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)\big(\mathbb{1}_{a=x} - \pi_t(a)\big)$, 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

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

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, e.g., 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. This would appear to you as a single, nonstationary k-armed bandit task whose true action values change randomly from step to step. You could try using one of the methods described in this chapter that can handle nonstationarity, but unless the true action values change slowly, these methods will not work very well. 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 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 0.1 and 0.2 with probability 0.5 (case A), and 0.9 and 0.8 with probability 0.5 (case B). If you are not able to tell which case you face at any step, what is the best expectation of success 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 expectation of success 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

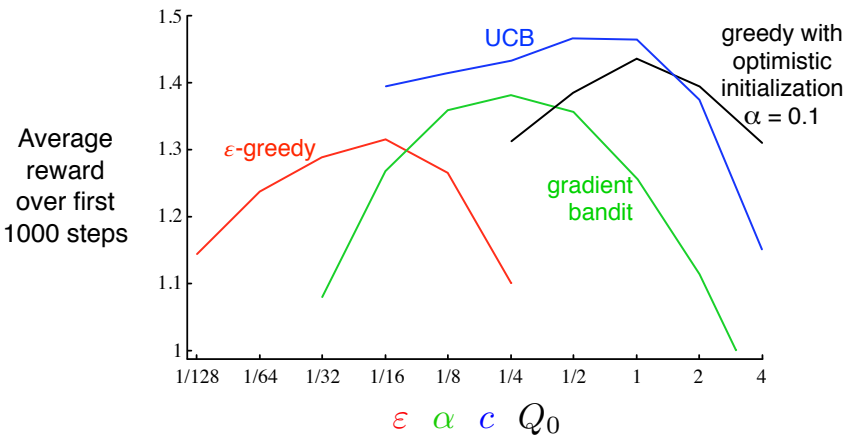


Figure 2.6: A parameter study of the various bandit algorithms presented in this chapter. Each point is the average reward obtained over 1000 steps with a particular algorithm at a particular setting of its parameter.

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, and 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. □

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), and 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 there and by Greensmith, Bartlett, and Baxter (2002, 2004) and 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, e.g., 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, Van Roy, Kazerouni, Osband, and Wen (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 tradeoff 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.

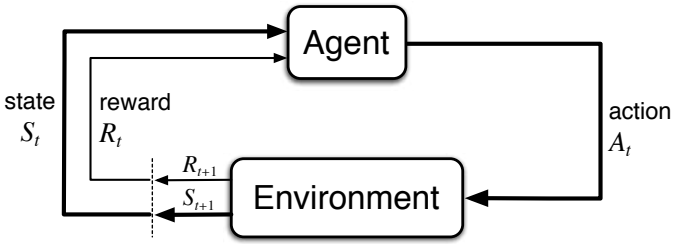


Figure 3.1: The agent–environment interaction in a Markov decision process.

More specifically, the agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots$.[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:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{3.1}$$

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:

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

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 $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is an ordinary deterministic function of four arguments. The '|' in the middle of it comes from the notation for conditional probability,

[1] We 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.

[2] We 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).

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

[4] We 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

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

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 only 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 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 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 $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$),

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

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

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

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

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

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 the 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 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 can 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? □

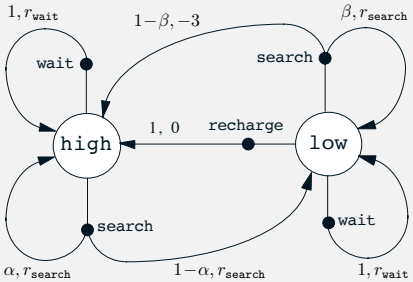
Exercise 3.3 Consider the problem of driving. You could define the actions in terms of the accelerator, steering wheel, and brake, that is, where your body meets the machine. Or you could define them farther out—say, where the rubber meets the road, considering your actions to be tire torques. Or you 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 $\mathcal{S} = \{\texttt{high}, \texttt{low}\}$. 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 $\mathcal{A}(\texttt{high}) = \{\texttt{search}, \texttt{wait}\}$ and $\mathcal{A}(\texttt{low}) = \{\texttt{search}, \texttt{wait}, \texttt{recharge}\}$.

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 $r_{\texttt{search}}$ and $r_{\texttt{wait}}$, with $r_{\texttt{search}} > r_{\texttt{wait}}$, respectively denote the expected number of cans the robot will collect (and hence the expected reward) while searching and while waiting. 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'	$p(s' \mid s, a)$	$r(s, a, s')$
high	search	high	α	$r_{\texttt{search}}$
high	search	low	$1 - \alpha$	$r_{\texttt{search}}$
low	search	high	$1 - \beta$	-3
low	search	low	β	$r_{\texttt{search}}$
high	wait	high	1	$r_{\texttt{wait}}$
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	$r_{\texttt{wait}}$
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 $p(s', r|s, a) > 0$. □

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 robot *what* you want it to achieve, not *how* you want it achieved.[6]

3.3 Returns and Episodes

So far we have discussed the objective of learning informally. 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 $R_{t+1}, R_{t+2}, R_{t+3}, \ldots$, 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:

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

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

[5] Better places for imparting this kind of prior knowledge are the initial policy or initial value function, or in influences on these.

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

[7] Episodes are sometimes called "trials" in the literature.

be $T = \infty$, and the return, which is what we are trying to maximize, could itself easily be infinite. (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*:

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

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:

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

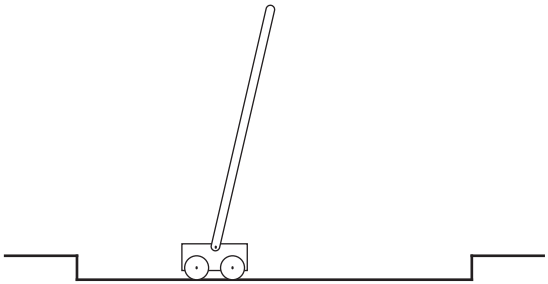
Note that this works for all time steps $t < T$, even if termination occurs at $t + 1$, if we define $G_T = 0$. This often makes it easy to compute returns from reward sequences.

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

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

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). □

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 $-\gamma^K$, where K is the number of time steps before failure. 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? □

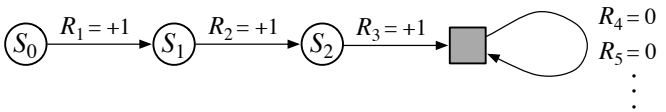
Exercise 3.10 Prove the second equality in (3.10). □

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 single 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 $+1, +1, +1, 0, 0, 0, \ldots$. 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

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

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 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 stochastic policy π, then what is the expectation of R_{t+1} in terms of π and the four-argument function p (3.2)? □

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

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

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:

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

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. □

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

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.

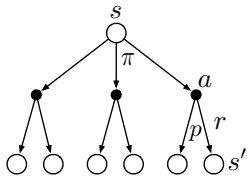
These kinds of methods are presented in Chapter 5. Of course, if there are very many states, then it may not be practical to keep separate averages for each state individually. Instead, the agent would have to maintain v_π and q_π as parameterized functions (with fewer parameters than states) and adjust 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:

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

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'.

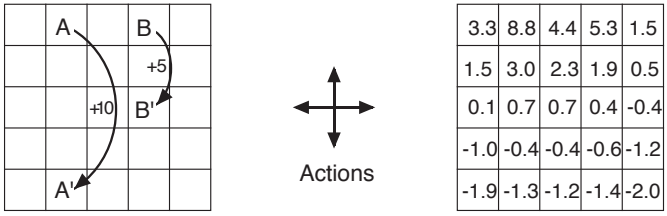


Figure 3.2: Gridworld example: exceptional reward dynamics (left) and state-value function for the equiprobable random policy (right).

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, but its expected return is less than 10, its immediate reward, because from A the agent is taken to A', from which it is likely to run into the edge of the grid. State B, on the other hand, is valued more than 5, its immediate reward, 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? □

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. □

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, $v_{\texttt{putt}}(s)$, 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 greater. 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.

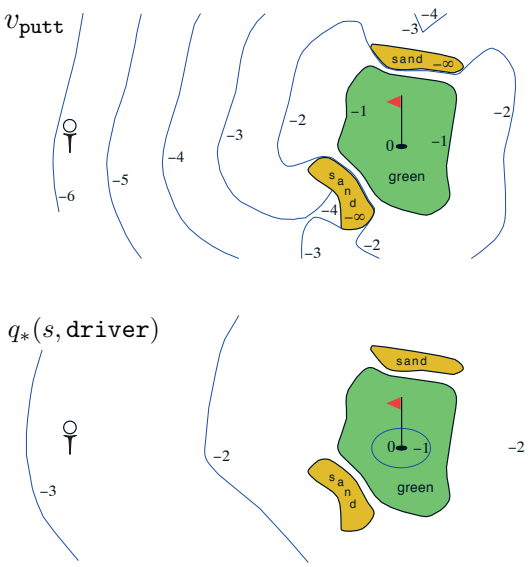
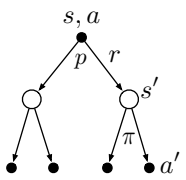


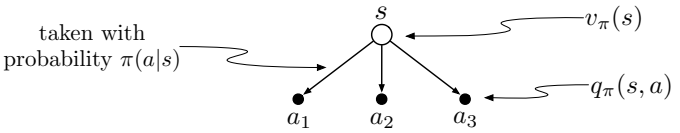
Figure 3.3: A golf example: the state-value function for putting (upper) and the optimal action-value function for using the driver (lower). ■

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. □



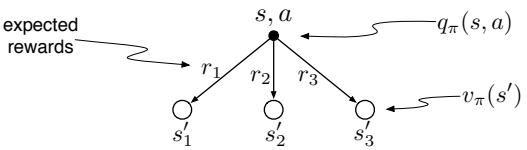
q_π backup diagram

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. □

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. □

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

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

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:

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

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 $q_*(s, \texttt{driver})$. 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 $q_*(s, \texttt{driver})$ 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:

$$\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] && \text{(by (3.9))} \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] && (3.18) \\
&= \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma v_*(s')\big]. && (3.19)
\end{aligned}$$

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

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

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).

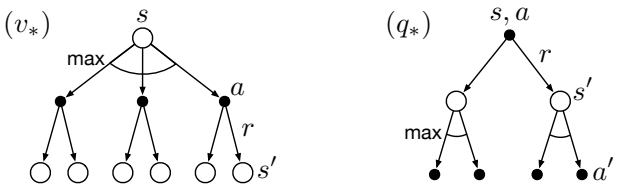


Figure 3.4: Backup diagrams for v_* and q_*

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.

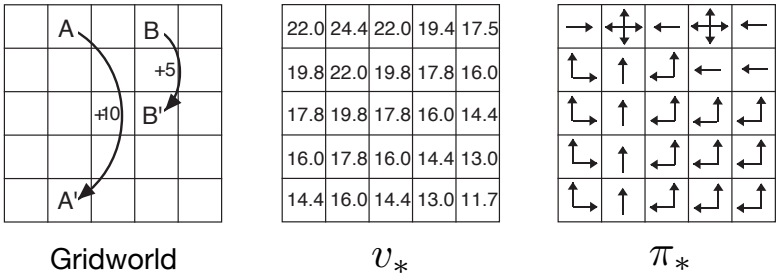


Figure 3.5: Optimal solutions to the gridworld example. ■

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 $v_*(\texttt{h})$ can be written as follows:

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

Following the same procedure for $v_*(\texttt{l})$ yields the equation

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

For any choice of $r_\texttt{s}$, $r_\texttt{w}$, α, β, and γ, with $0 \le \gamma < 1$, $0 \le \alpha, \beta \le 1$, there is exactly one pair of numbers, $v_*(\texttt{h})$ and $v_*(\texttt{l})$, that simultaneously satisfy these two nonlinear equations. ■

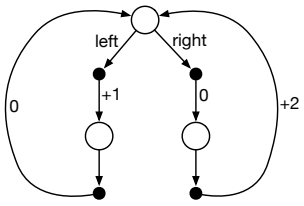
Explicitly solving the Bellman optimality equation provides one route to finding an optimal policy, and thus to solving the reinforcement learning problem. However, this solution is rarely directly useful. It is akin to an exhaustive search, looking ahead at all possibilities, computing their probabilities 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) we accurately know the dynamics of the environment; (2) we have enough computational resources to complete the computation of the solution; and (3) 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, $q_*(s, \texttt{putter})$, for the golf example. □

Exercise 3.22 Consider the continuing MDP shown on 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 to varying degrees. 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 completely known and controllable by the agent; everything outside is incompletely controllable but 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* 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* 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 has a complete and accurate environment model, the agent is typically unable to perform enough computation per time step to fully use it. The memory available is also an important constraint. 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 of which we are aware in which reinforcement learning was discussed using the MDP formalism is Andreae's (1969b) 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 $p(s', r \mid s, a)$ is slightly unusual. It is more common in the MDP literature to describe the dynamics in terms of the state transition probabilities $p(s' \mid s, a)$ 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 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, e.g., 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.

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:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \,|\, s, a)\Big[r + \gamma v_*(s')\Big], \text{ or} \qquad (4.1) \\ q_*(s,a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a\Big] \\ &= \sum_{s',r} p(s', r \,|\, s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big], \qquad (4.2) \end{aligned}$$

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 $s \in \mathcal{S}$,

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] && \text{(from (3.9))} \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] && (4.3) \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a) \Big[r + \gamma v_\pi(s') \Big], && (4.4) \end{aligned}$$

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 $v_0, v_1, v_2, \ldots$, 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:

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

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 $k \to \infty$ 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. Of course it is easier to 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.

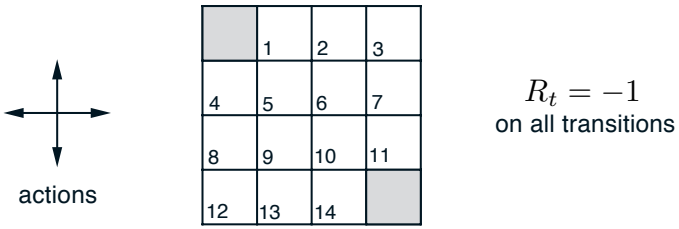
A complete in-place version of iterative policy evaluation is shown in pseudocode in the box below. Note how it handles termination. Formally, iterative policy evaluation converges only in the limit, but in practice it must be halted short of this. The pseudocode tests the quantity $\max_{s\in\mathcal{S}} |v_{k+1}(s)-v_k(s)|$ 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 $V(terminal) = 0$

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

Example 4.1 Consider the 4×4 gridworld shown below.



The nonterminal states are $\mathcal{S} = \{1, 2, \ldots, 14\}$. There are four actions possible in each state, $\mathcal{A} = \{\texttt{up}, \texttt{down}, \texttt{right}, \texttt{left}\}$, 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, $p(6, -1 \,|\, 5, \texttt{right}) = 1$, $p(7, -1 \,|\, 7, \texttt{right}) = 1$, and $p(10, r \,|\, 5, \texttt{right}) = 0$ 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 $q_\pi(11, \texttt{down})$? What is $q_\pi(7, \texttt{down})$? □

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) for the action-value function q_π and its successive approximation by a sequence of functions $q_0, q_1, q_2, \ldots$? □

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

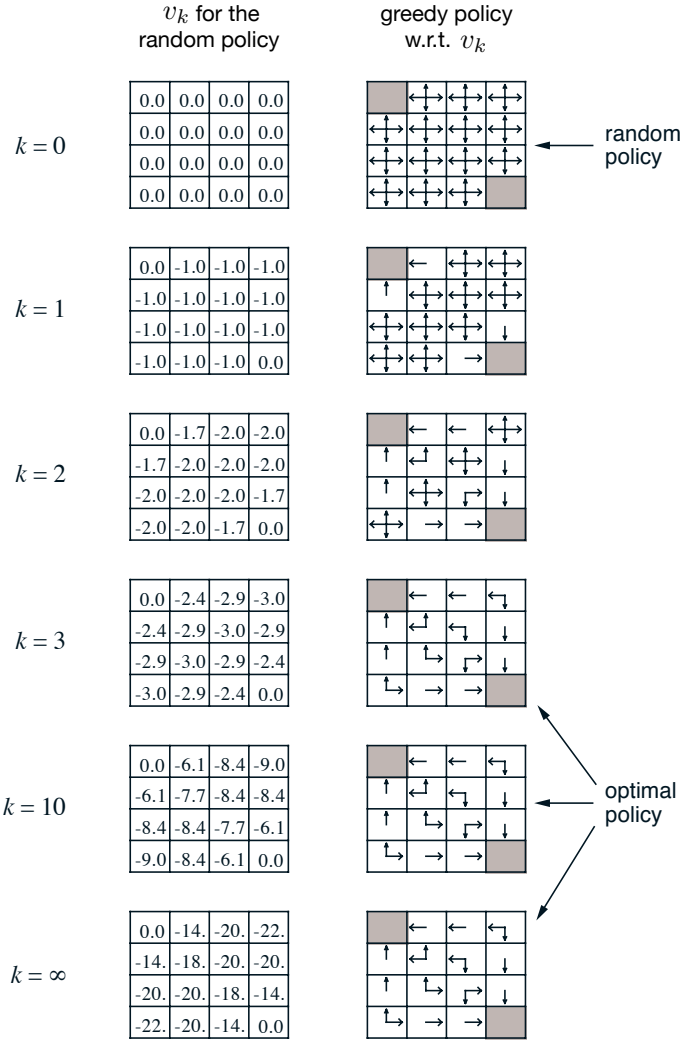


Figure 4.1: Convergence of iterative policy evaluation on a small gridworld. The left column is the sequence of approximations of the state-value function for the random policy (all actions equally likely). The right column is the sequence of greedy policies corresponding to the value function estimates (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

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

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 $v_{\pi'}(s)$:

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

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 to a particular action. It is a natural extension

to consider changes at *all* states and to *all* possible actions, 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

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

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 $s \in \mathcal{S}$:

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

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. The value function of any such policy, $v_{\pi'}(s)$, can be seen by inspection to be either -1, -2, or -3 at 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 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 $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r \,|\, s, \pi(s))\big[r + \gamma V(s')\big]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   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 $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V(s')\big]$
   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.

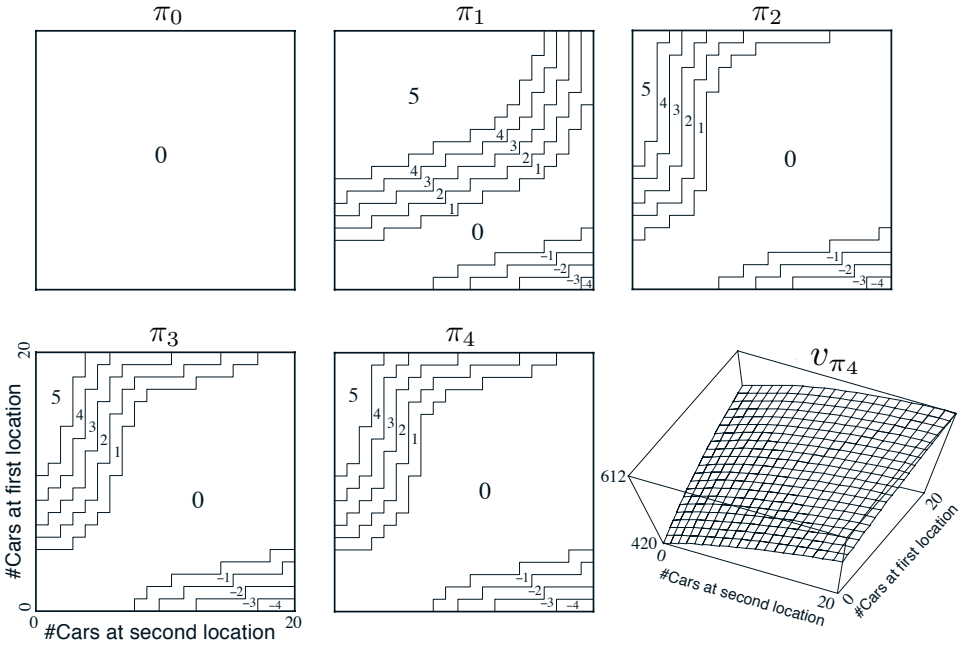


Figure 4.2: The sequence of policies found by policy iteration on Jack's car rental problem, and the final state-value function. The first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second (negative 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 the example of Jack's car rental illustrates, and as is also illustrated by 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 ok for pedagogy, but not for actual use. Modify the pseudocode so that convergence is guaranteed. □

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. □

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. □

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. □

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:

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

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 $V(\mathit{terminal}) = 0$

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

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

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, $s \in \{1, 2, \ldots, 99\}$ and the actions are stakes, $a \in \{0, 1, \ldots, \min(s, 100 - s)\}$. 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?

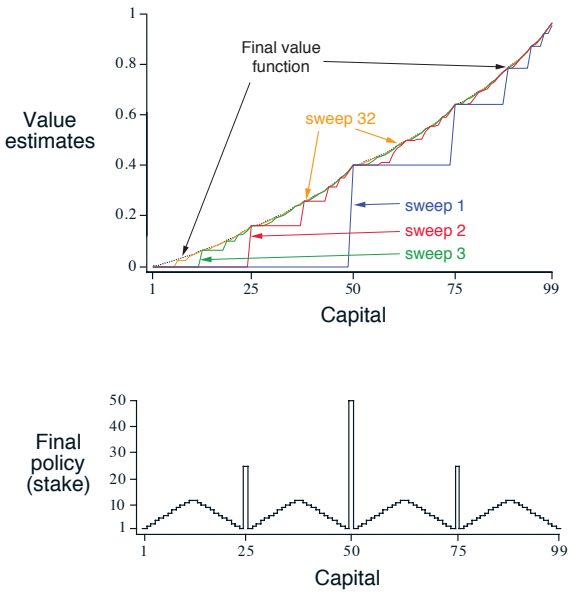


Figure 4.3: The solution to the gambler's problem for $p_h = 0.4$. The upper graph shows the value function found by successive sweeps of value iteration. The lower graph shows the final policy.

■

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? □

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 $\theta \to 0$? □

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

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 stochastic). (In 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.) 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.

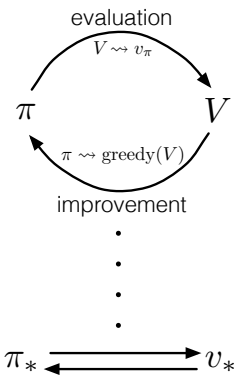
Asynchronous algorithms also make it easier to intermix computation with real-time interaction. To solve a given MDP, we can run an iterative DP algorithm *at the same time that an agent is actually experiencing the MDP*. The agent's experience can be used

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

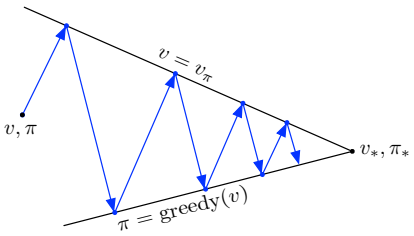
4.6 Generalized Policy Iteration

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 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 the (worst case) time 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 functions 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 (1969b) mentioned DP in the context of reinforcement learning, specifically policy iteration, although he did not make specific connections between DP and learning algorithms. 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).