Reinforcement Learning: An Introduction

Second edition, in progress

****Draft****

Richard S. Sutton and Andrew G. Barto

A Bradford Book

The MIT Press
Cambridge, Massachusetts
London, England

In memory of A. Harry Klopf

Contents

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.

[Three paragraphs elided in favor of updated content in the second edition.]

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

We also wish to thank the many people who have read drafts of this book and provided valuable comments, including Tom Kalt, John Tsitsiklis, 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.

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 2013. 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, on-line 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 with outstanding contributions by many active researchers. 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, since the first edition, our deeper understanding of some topics 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. Another weakness is the excess of subscripts and superscripts. In this edition we use the explicit notation of $p(s', r \,|\, s, a)$ for the joint probability for the next state and reward given the current state and action. All the changes in notation are summarized in a table on page xv.

The second edition is significantly expanded, and its top-level organization has been revamped. 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 whole 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, and AlphaGo (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 60% longer than 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), Weiring 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 2.2, 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 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 thank many readers on the internet for finding errors and potential points of confusion in the second edition, and specifically 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 owe Phil Thomas thanks for helping us make these chapters accessible to non-psychologists and non-neuroscientists. We thank Jim Houk for introducing us to the subject of information processing in the basal ganglia. 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 commented 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, Stefan Dernbach, Clemens Rosenbaum, and Patrick Taylor 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 12, to Shantong Zhang for replicating and thus verifying almost all the experimental results, to Kris De Asis for improving the new technical content of Chapter 12, 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 would also like to gratefully acknowledge 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.

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
$\Pr\{X=x\}$	probability that a random variable X takes on the value x
$\mathbb{E}[X]$	expectation of a random variable X
$\arg\max_a f(a)$	a value of a at which $f(a)$ takes its maximal value
$\ln x$	natural logarithm of x
$\exp(x)$	e^x, where e is the base of the natural logarithm
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
$\mathbb{1}_{predicate}$	An indicator function (1 is 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
$\pi_t(a)$	probability of selecting action a on time t
$\bar{R}_t$	estimate at time t of the expected reward given π

In a Markov Decision Process:

s, s'	states
a	an action
r	a reward
$\mathbb{R}$	set of real numbers
$\mathcal{S}$	set of all nonterminal states
$\mathcal{S}^+$	set of all states, including the terminal state
$\mathcal{A}$	set of all actions
$\mathcal{R}$	set of all possible rewards, a finite subset of $\mathbb{R}$
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
$\pi(a\|s)$	probability of taking action a in state s under *stochastic* policy π
G_t	return (cumulative discounted reward) following time t (Section 3.3)
$\bar{G}_{t:h}$	flat return (uncorrected, undiscounted) from $t+1$ to h (Section 5.8)
$G_t^{\lambda s}$	λ-return, corrected by estimated state values (Section 12.1)
$G_t^{\lambda a}$	λ-return, corrected by estimated action values (Section 12.1)
$G_{t:h}^{\lambda s}$	truncated, corrected λ-return, with state values (Section 12.3)
$G_{t:h}^{\lambda a}$	truncated, corrected λ-return, with action values (Section 12.3)
$p(s', r \mid s, a)$	probability of transition to state s' with reward r, from state s and action a
$p(s' \mid s, a)$	probability of transition to state s', from state s taking 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_*
δ_t	temporal-difference error at t (a random variable) (Section 6.1)
$\mathbf{w}, \mathbf{w}_t$	d-vector of weights underlying an approximate value function
$w_i, w_{t,i}$	ith component of learnable weight vector
d	dimensionality—the number of components of the main weight vector
m	number of 1s in a sparse binary feature vector, or dimensionality of a secondary 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 $\mathbf{w}$
$\mathbf{x}(s)$	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 feature vector
$\mathbf{x}_t$	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 $\mathbf{w}$
$\mathbf{e}_t$	d-vector of eligibility traces at time t
$h(s, a)$	a preference for selecting action a in state s
$\boldsymbol{\theta}, \boldsymbol{\theta}_t$	parameter vector of target policy (Chapter 12)
$\pi_{\boldsymbol{\theta}}$	policy corresponding to parameter $\boldsymbol{\theta}$
$J(\pi), J(\boldsymbol{\theta})$	performance measure for policy π or $\pi_{\boldsymbol{\theta}}$
b	*behavior policy* selecting actions while learning about *target policy* π, or a *baseline function* $b : \mathcal{S} \mapsto \mathbb{R}$ for policy-gradient methods
$\rho_{t:h}$	importance sampling ratio for time t to time h (Section 5.5)
ρ_t	importance sampling ratio for time t alone, $\rho_t = \rho_{t:t}$
$r(\pi)$	average reward (reward rate) for policy π (Section 10.3)
$\bar{R}_t$	estimate of $r(\pi)$ at time t

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 explore idealized learning situations and evaluate the effectiveness of various learning methods. 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, 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 class of problems, and the field that studies these problems and their 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 a many confusions.

Reinforcement learning problems involve learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. In an essential way, these are *closed-loop* problems because the learning system's actions influence its later inputs. Moreover, the learner is not told which actions to take, as in many forms of machine learning, but instead must discover which actions yield the most reward by trying them out. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These three characteristics—being closed-loop in an essential way, not having direct instructions as to what actions to take, and that the consequences of actions, including reward signals, play out over extended time periods—are the three most important distinguishing features of the reinforcement learning problem.

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 as well.

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 their purest forms.

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, as well as trying to incorporate vast amounts of domain knowledge. 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 jug. 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 options 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 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 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.

A *reward signal* defines the goal in 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. Expressed this way, we hope it is clear that value functions formalize a basic and familiar idea.

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 we have learned about reinforcement learning over the last few 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

From the preceding discussion, it should be clear that 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. 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 main concern is not with designing the state signal, but with deciding what action to take as a function of whatever state signal is available. (We do touch briefly on state design and construction in the last chapter in Section 17.3.)

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, methods such as genetic algorithms, genetic programming, simulated annealing, and other optimization methods have been used to approach reinforcement learning problems without ever appealing to value functions. These methods evaluate the "lifetime" behavior of many non-learning agents, each using a different policy for interacting with its environment, and select those that are able to obtain the most reward. We call these *evolutionary* methods because their operation is analogous to the way biological evolution produces organisms with skilled behavior even when 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.

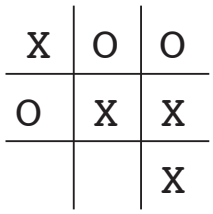
Our focus is on reinforcement learning methods that learn while interacting with the environment, which evolutionary methods do not do. Methods able to take advantage of the details of individual behavioral interactions can be much more efficient than evolutionary methods in many cases. Evolutionary 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.

However, we do include some methods that, like evolutionary methods, do not appeal to value functions. These methods search in spaces of policies defined by a collection of numerical parameters. They estimate the directions the parameters should be adjusted in order to most rapidly improve a policy's performance. Unlike evolutionary methods, however, they produce these estimates while the agent is interacting with its environment and so can take advantage of the details of individual behavioral interactions. Methods like this have proven useful in many problems, and some of the simplest reinforcement learning methods fall into this category (see Chapter 13). In the end, however, the best methods of this type tend to include value functions in some form.

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, 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?



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 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 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.

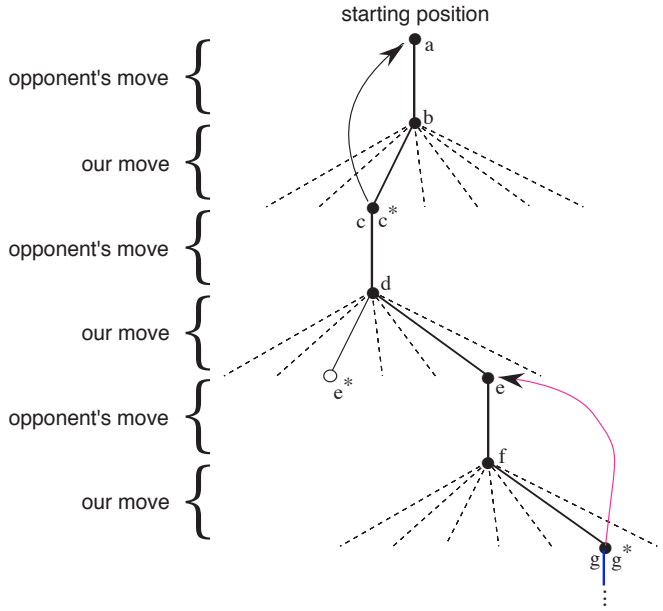


Figure 1.1: A sequence of tic-tac-toe moves. The solid lines represent the moves taken during a game; the dashed lines represent moves that we (our reinforcement learning player) considered but did not make. Our second move was an exploratory move, meaning that it was taken even though another sibling 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 curved arrow in which estimated values are moved up the tree from later nodes to earlier as detailed in the text.

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 denote the state before the greedy move, and s' the state after the move, then the update to the estimated value of s, denoted $V(s)$, can be written as

$$V(s) \leftarrow V(s) + \alpha\Big[V(s') - V(s)\Big],$$

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, $V(s') - V(s)$, between estimates at two different 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 the 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 now plays at the level of the world's best human players (see Chapter 16). The 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. 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. We also had 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 "think about" 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. No model is 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 set of probabilities. What 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 relying on 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 functions are the key features of 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 scalar 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 that started 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 other 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. Although the two threads have been largely independent, the exceptions revolve around a third, less distinct thread concerning temporal-difference methods such as the one 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 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 Markovian decision processes (MDPs), and 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 off-line computation depending essentially on accurate system models and analytic solutions to the Bellman equation. Further, the simplest form of dynamic programming is a computation that proceeds backwards in time, making it difficult to see how it could be involved in a learning process that must proceed in a forward direction. Some of the earliest work in dynamic programming, such as that by Bellman and Dreyfus (1959) might now be classified as following a learning approach. Witten's (1977) work (discussed below) certainly qualifies as a combination of learning and dynamic-programming ideas. Werbos (1987) argued explicitly greater interrelation of dynamic programming and learning methods and its relevance to understanding neural and cognitive mechanisms. For us the full integration of dynamic programming methods with on-line 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 (Watkins, 1989). 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 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 would 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, that 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 Chapter 14. According to American psychologist R. S. Woodworth 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 (Woodworth, 1938). 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 accumulating 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 and experimental methods of B. F. Skinner (e.g., Hull, 1943; Skinner, 1938).

The term "reinforcement" in the context of animal learning came into use well after Thorndike's expression of the Law of Effect, to the best of our knowledge first appearing in this context in the 1927 English translation of Pavlov's monograph on conditioned reflexes. Reinforcement is the strengthening of a pattern of behavior as a result of an animal receiving a stimulus—a reinforcer—in an appropriate temporal relationship with another stimulus or with a response. Some psychologists extended its meaning to include the process of weakening in addition to strengthening, as well applying when the omission or termination of an event changes behavior. Reinforcement produces changes in behavior that persist after the reinforcer is withdrawn, so that a stimulus that attracts an animal's attention or that energizes its behavior without producing lasting changes is not considered to be 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, already known for his "mechanical tortoise" (Walter, 1950), built a version capable of a simple form of learning (Walter, 1951). 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 (Shannon, 1951, 1952). 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 fascinating 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, 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 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 of these was the work by a New Zealand researcher named John Andreae. Andreae (1963) 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, 1969a). 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.

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 reinforce 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. Stochastic learning automata were foreshadowed by earlier work in psychology, beginning with William Estes' 1950 effort toward a statistical theory of learning (Estes, 1950) and further developed by others, most famously by psychologist Robert Bush and statistician Frederick Mosteller (Bush and Mosteller, 1955).

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. Although reinforcement learning in economics developed largely independently of the early work in artificial intelligence, reinforcement learning and game theory is a topic of current interest in both fields, but one that is beyond the scope of this book. Camerer (2003) discusses the reinforcement learning tradition in economics, and Nowé et al. (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 that 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. 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 neural network learning, in particular, how it could produce learning algorithms for multilayer networks (Barto, Anderson, and Sutton, 1982;

Barto and Anderson, 1985; Barto and Anandan, 1985; Barto, 1985, 1986; Barto and Jordan, 1987). We say more about reinforcement learning and neural networks in Chapter 15.

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.

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 on-line. (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, 1978b, 1978c) 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 appear to have been 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 in 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) 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. 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 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 et al. (2013). Books that take a control or operations research perspective include those of Si et al. (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 (1992), 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 See Chapter 6 for references to the kind of temporal-difference method used in the tic-tac-toe example.

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 the use of eligibility traces. 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:

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

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 some simple methods for estimating the values of actions and for using the estimates to make action selection decisions. 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

$$A_t \doteq \arg\max_a Q_t(a), \tag{2.2}$$

where $\arg\max_a$ 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?

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? □

2.3 The 10-armed Testbed

To roughly assess the relative effectiveness of the greedy and ε-greedy 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, $q_*(a)$, $a = 1, \ldots, 10$, 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

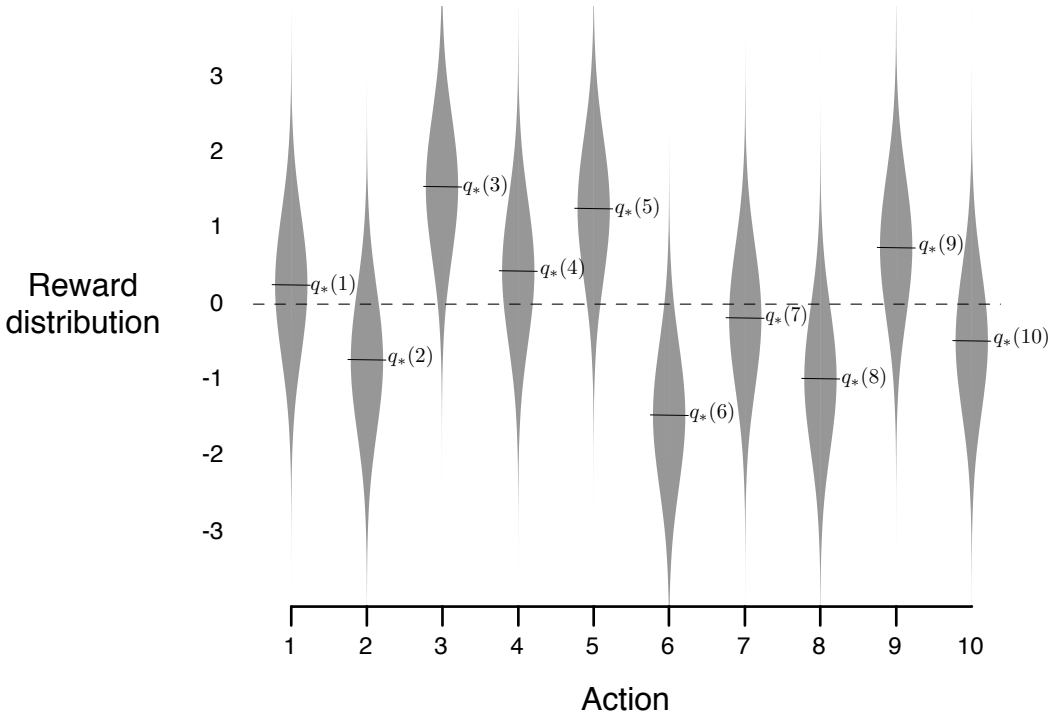


Figure 2.1: An example bandit problem from the 10-armed testbed. The true value $q_*(a)$ of each of the ten actions was selected according to a normal distribution with mean zero and unit variance, and then the actual rewards were selected according to a mean $q_*(a)$ unit variance normal distribution,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

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 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

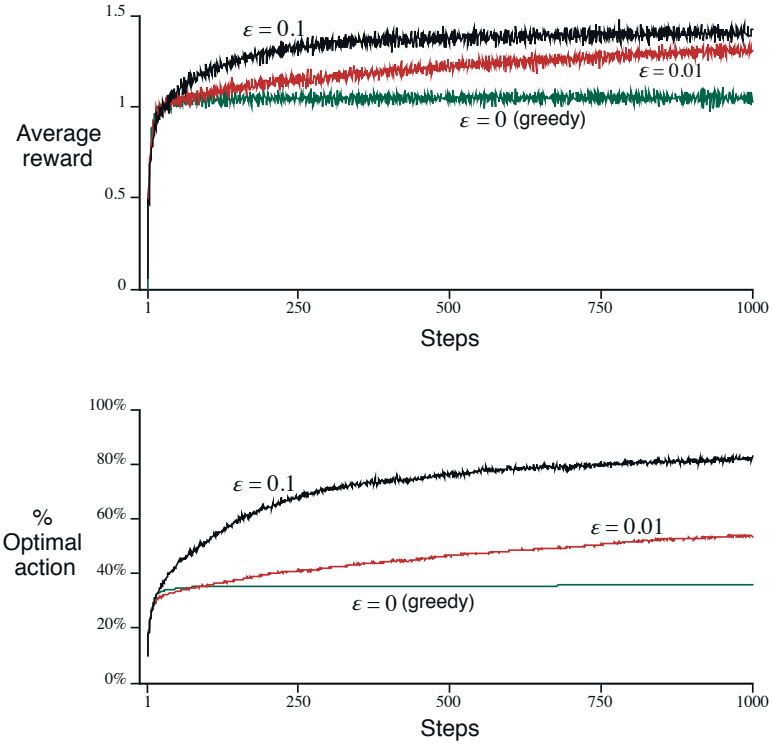


Figure 2.2: Average performance of ε-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems. All methods used sample averages as their action-value estimates.

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 policy changes. Reinforcement learning requires a balance between exploration and exploitation.

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. Pseudocode for a complete bandit algorithm using incrementally computed sample averages and ε-greedy action selection is shown in the box on the next page. 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:
$Q(a) \leftarrow 0$
$N(a) \leftarrow 0$

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

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

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

The expression $\big[\mathit{Target} - \mathit{OldEstimate}\big]$ 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 ($\mathit{StepSize}$) used in the incremental method described above 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 $\alpha_t(a)$. We sometimes use the informal shorthand $\alpha = \frac{1}{n}$ when $\alpha_t(a) = \frac{1}{n}$, leaving the dependence of n on the action implicit, just as we have in this section.

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.[1] 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

[1] The notation $(a, b]$ as a set denotes the real interval between a and b including b but not including a. Thus, here we are saying that $0 < \alpha \leq 1$.

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

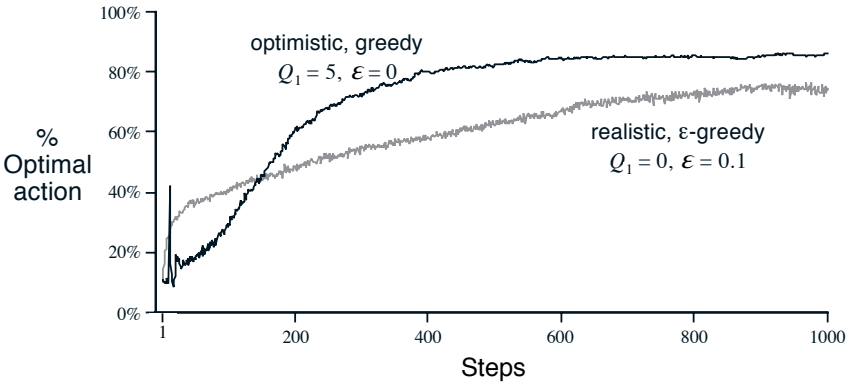


Figure 2.3: The effect of optimistic initial action-value estimates on the 10-armed testbed. Both methods used a constant step-size parameter, $\alpha = 0.1$.

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 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? □

2.7 Upper-Confidence-Bound Action Selection

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

$$A_t \doteq \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right], \tag{2.8}$$

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

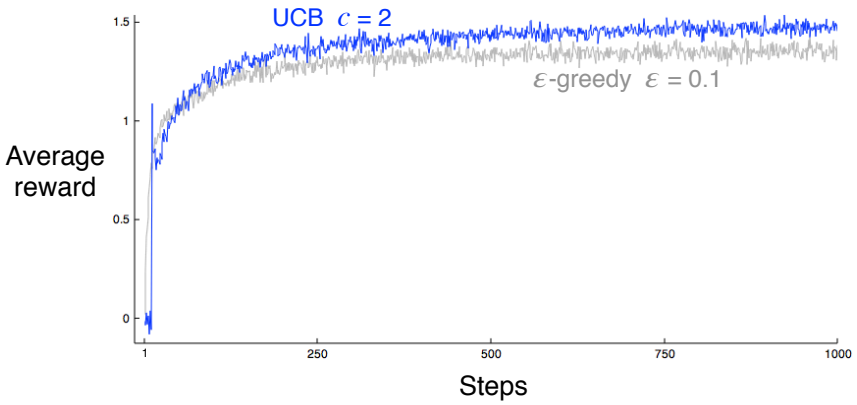


Figure 2.4: Average performance of UCB action selection on the 10-armed testbed. As shown, UCB generally performs better than ε-greedy action selection, except in the first k steps, when it selects randomly among the as-yet-untried actions.

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.

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 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:

$$\Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \doteq \pi_t(a), \tag{2.9}$$

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 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 2.7 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 this setting based on the idea of stochastic gradient ascent. On each step, after selecting action A_t and receiving the reward R_t, preferences are updated by:

$$\begin{aligned} H_{t+1}(A_t) &\doteq H_t(A_t) + \alpha\big(R_t - \bar{R}_t\big)\big(1 - \pi_t(A_t)\big), \qquad \text{and} \\ H_{t+1}(a) &\doteq H_t(a) - \alpha\big(R_t - \bar{R}_t\big)\pi_t(a), \qquad\qquad \text{for all } a \neq A_t, \end{aligned} \tag{2.10}$$

where $\alpha > 0$ is a step-size parameter, and $\bar{R}_t \in \mathbb{R}$ is the average of all the rewards up through and including time t, which can be computed incrementally as described in Section 2.3 (or Section 2.4 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 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 (2.10)), then performance would be significantly degraded, as shown in the figure.

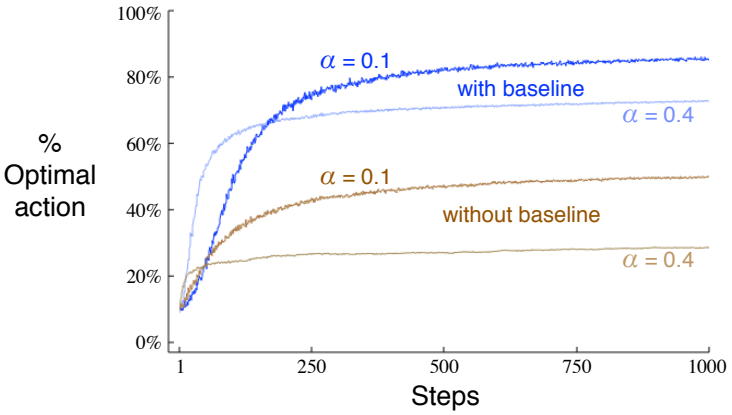


Figure 2.5: Average performance of the gradient bandit algorithm with and without a reward baseline on the 10-armed testbed when the $q_*(a)$ are chosen to be near +4 rather than near zero.

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 preference $H_t(a)$ would be incremented proportional to the increment's effect on performance:

$$H_{t+1}(a) \doteq H_t(a) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}, \tag{2.11}$$

where the measure of performance here is the expected reward:

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

and the measure of the increment's effect is the *partial derivative* of this performance measure with respect to the preference. Of course, it is not possible to implement gradient ascent exactly in our case because by assumption we do not know the $q_*(b)$, but in fact the updates of our algorithm (2.10) are equal to (2.11) 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:

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

where X_t can be any scalar that does not depend on b. We can include it here because the gradient sums to zero over all the actions, $\sum_b \frac{\partial \pi_t(b)}{\partial H_t(a)} = 0$. As $H_t(a)$ is changed, some actions' probabilities go up and some down, but the sum of the changes must be zero because the sum of the probabilities must remain one.

$$\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} = \sum_b \pi_t(b) \big(q_*(b) - X_t\big) \frac{\partial \pi_t(b)}{\partial H_t(a)} / \pi_t(b)$$

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

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

where here we have chosen $X_t = \bar{R}_t$ and substituted R_t for $q_*(A_t)$, which is permitted because $\mathbb{E}[R_t | A_t] = q_*(A_t)$ and because the R_t (given A_t) is uncorrelated with anything else. Shortly we will establish that $\frac{\partial \pi_t(b)}{\partial H_t(a)} = \pi_t(b)\big(\mathbb{1}_{a=b} - \pi_t(a)\big)$, where $\mathbb{1}_{a=b}$ is defined to be 1 if $a = b$, 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 (2.11) 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 (2.10).

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

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

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 2.8 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

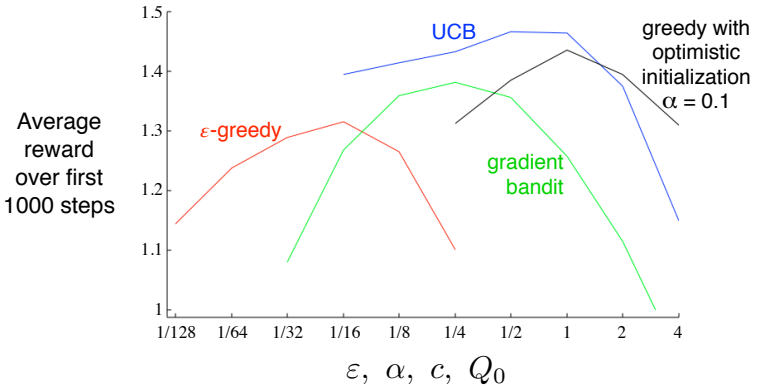


Figure 2.6: A parameter study of the various bandit algorithms presented in this chapter. Each point is the average reward obtained over 1000 steps with a particular algorithm at a particular setting of its parameter.

their parameter, neither too large nor too small. In assessing 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 special functions called *Gittins indices*. These provide an optimal solution to a certain kind of bandit problem more general than that considered here, but this approach assumes that the prior distribution of possible problems is known. Unfortunately, neither the theory nor the computational tractability of this method appear to generalize to the full reinforcement learning problem that we consider in the rest of the book.

Bayesian methods 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 2.9 (programming) Make a figure analogous to Figure 2.6 for the non-stationary 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, 1976). 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 (2001, 2004) and Dick (2015).

The term *soft-max* for the action selection rule (2.9) 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 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.

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. See Figure 3.1.

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

[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

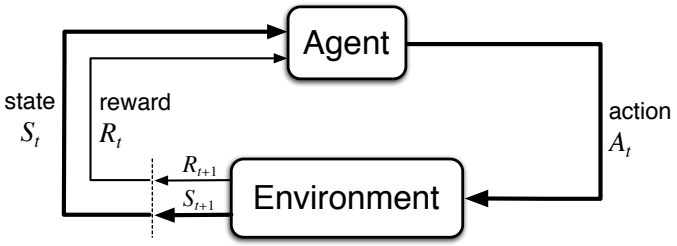


Figure 3.1: The agent–environment interaction in a Markov decision process.

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:

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

for all $s', s \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$. The dot over the equals sign in this 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 function p here is an ordinary deterministic function of four arguments. The '|' in the middle of it does not have a formal meaning. It just reminds us that p specifies a probability distribution for each choice of s and a, that is, that

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

The probabilities given by the four-argument function p completely characterize the dynamics of a finite MDP. From it, 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),

$$p(s' \,|\, s, a) \doteq \Pr\{S_t{=}s' \mid S_{t-1}{=}s, A_{t-1}{=}a\} = \sum_{r \in \mathcal{R}} p(s', r \,|\, s, a), \tag{3.3}$$

the expected rewards for state–action pairs,

$$r(s, a) \doteq \mathbb{E}[R_t \mid S_{t-1}{=}s, A_{t-1}{=}a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \,|\, s, a), \tag{3.4}$$

and the expected rewards for state–action–next-state triples,

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

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.

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

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. This agent has to decide whether the robot should (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. This decision has to be made either periodically or whenever certain events occur, such as finding an empty can. The agent therefore has three actions, and the state is primarily determined by the state of the battery. The rewards might be zero most of the time, but then become positive when the robot secures an empty can, or large and negative if the battery runs all the way down. In this example, the reinforcement learning agent is not the entire robot. The states it monitors describe conditions within the robot itself, not conditions of the robot's external environment. The agent's environment therefore includes the rest of the robot, which might contain other complex decision-making systems, as well as the robot's external environment. ■

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? □

Exercise 3.4 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 the four-argument function p (3.2)? □

Example 3.4: Recycling Robot MDP The recycling robot (Example 3.3) can be turned into a simple example of an MDP by simplifying it and providing some more details. (Our aim is to produce a simple example, not a particularly realistic one.) Recall that the agent makes a decision at times determined by external events (or by other parts of the robot's control system). At each such time the robot decides whether it should (1) actively search for a can, (2) remain stationary and wait for someone to bring it a can, or (3) go back to home base to recharge its battery. Suppose the environment works as follows. 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).

The agent makes its decisions solely as a function of the energy level of the battery. It can distinguish two levels, `high` and `low`, so that the state set is $\mathcal{S} = \{\texttt{high}, \texttt{low}\}$. Let us call the possible decisions—the agent's actions—`wait`, `search`, and `recharge`. 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 agent's action sets are

$$\begin{aligned} \mathcal{A}(\texttt{high}) &\doteq \{\texttt{search}, \texttt{wait}\} \\ \mathcal{A}(\texttt{low}) &\doteq \{\texttt{search}, \texttt{wait}, \texttt{recharge}\}. \end{aligned}$$

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, to keep things simple, 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, as in Table 3.1.

A *transition graph* is a useful way to summarize the dynamics of a finite MDP. Figure 3.2 shows the transition graph for the recycling robot example. 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 3.5 Give a table analogous to to Table 3.1, 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 $p(s', r|s, a) > 0$. □

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	$r_{\texttt{wait}}$
low	wait	high	0	$r_{\texttt{wait}}$
low	wait	low	1	$r_{\texttt{wait}}$
low	recharge	high	1	0
low	recharge	low	0	0.

Table 3.1: Transition probabilities and expected rewards for the finite MDP of the recycling robot example. There is a row for each possible combination of current state, s, next state, s', and action possible in the current state, $a \in \mathcal{A}(s)$.

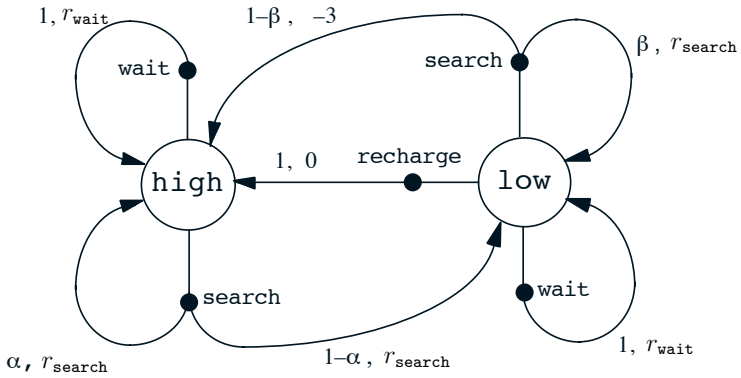


Figure 3.2: Transition graph for the recycling robot example.

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.

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:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T, \tag{3.6}$$

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*,[6] 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 (3.6) is problematic for continuing tasks because the final time step would be $T = \infty$, and the return, which is what we are trying to maximize, could 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.

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

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

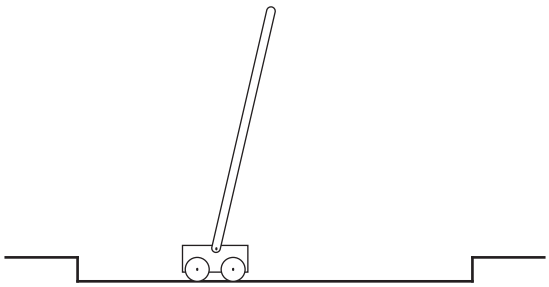
In particular, it chooses A_t to maximize the expected *discounted return*:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{3.7}$$

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 (3.7) 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 (3.7) 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.

Example 3.5: 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 (3.6). 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? □

Returns at successive time steps are related to each other in a way that is important for the theory

and algorithms of reinforcement learning:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma\big(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots\big) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{3.8}$$

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.

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. □

Note that although the return (3.7) 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

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}. \tag{3.9}$$

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? □

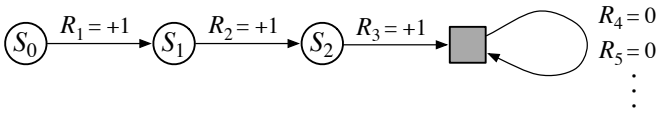
Exercise 3.10 Prove (3.9). □

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 (3.6) and as a sum over an infinite number of terms in the other (3.7). These 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 (3.7), 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 also write the return as

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{3.10}$$

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.

The *value* 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

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s\right], \text{ for all } s \in \mathcal{S}, \tag{3.11}$$

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:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a\right]. \tag{3.12}$$

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

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

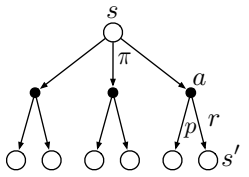
of this kind *Monte Carlo methods* because they involve averaging over many random samples of actual returns. These kinds of methods are presented in Chapter 5. Of course, if there are very many states, then it may not be practical to keep separate averages for each state individually. Instead, the agent 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 (3.8). 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:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] && \text{(by (3.8))} \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \,|\, s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \Big] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r \,|\, s, a) \Big[r + \gamma v_\pi(s') \Big], \quad \text{for all } s \in \mathcal{S}, && (3.13) \end{aligned}$$

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 (3.13) 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 (3.13) 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.



Update 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 *update diagrams* because they diagram relationships that form the basis of the update 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 update diagrams throughout the book to provide graphical summaries of the algorithms we discuss. (Note that, unlike transition graphs, the state nodes of update diagrams do not necessarily represent distinct states; for example, a state might be its own successor.)

Example 3.6: Gridworld Figure 3.3 (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

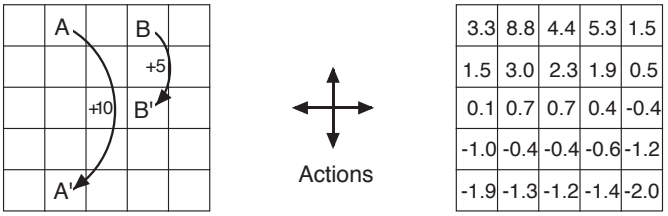


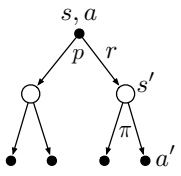
Figure 3.3: Gridworld example: exceptional reward dynamics (left) and state-value function for the equiprobable random policy (right).

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′.

Suppose the agent selects all four actions with equal probability in all states. Figure 3.3 (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 (3.13). 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 3.11 The Bellman equation (3.13) must hold for each state for the value function v_π shown in Figure 3.3 (right). As an example, 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 3.12 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 update diagram to the right corresponds to this equation. Show the sequence of equations analogous to (3.13), but for action values. □



q_π update diagram

Example 3.7: 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 3.4 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

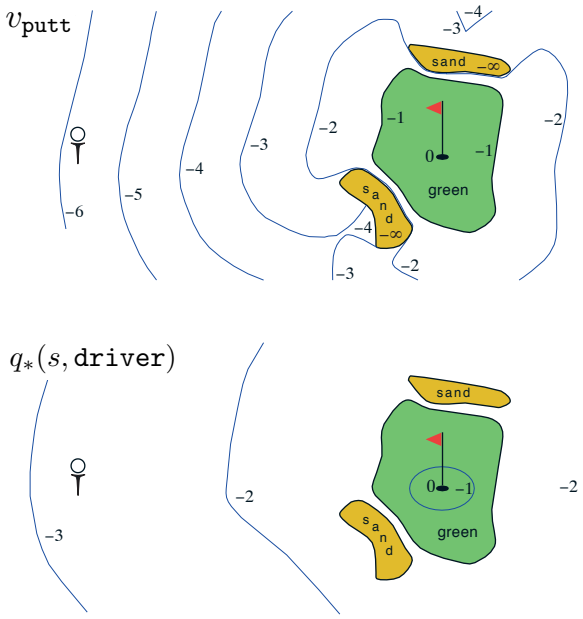


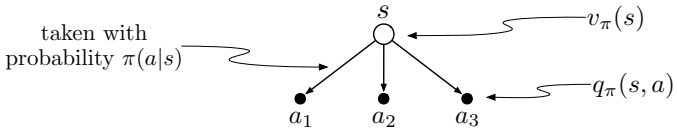
Figure 3.4: A golf example: the state-value function for putting (above) and the optimal action-value function for using the driver (below).

of -3, and so on to get all the contour lines shown in the figure. Putting doesn't get us out of sand traps, so they have a value of $-\infty$. Overall, it takes us six strokes to get from the tee to the hole by putting. ■

Exercise 3.13 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 (3.7), 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 3.14 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. □

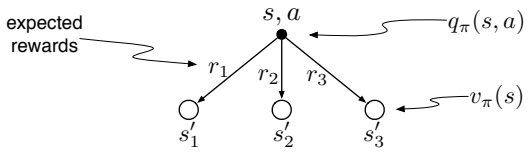
Exercise 3.15 The value of a state depends on the values of the actions possible in that state and on how likely each action is to be taken under the current policy. We can think of this in terms of a small update diagram rooted at the state and considering each possible action:



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 3.16 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 update 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 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

$$v_*(s) \doteq \max_\pi v_\pi(s), \tag{3.14}$$

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

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

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

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:

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]\,. \tag{3.16}$$

Example 3.8: Optimal Value Functions for Golf The lower part of Figure 3.4 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 (3.13). 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:

$$\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] && \text{(by (3.8))} \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] && (3.17) \\
&= \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma v_*(s')\big]. && (3.18)
\end{aligned}$$

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

$$\begin{aligned}
q_*(s,a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a\Big] \\
&= \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big]. && (3.19)
\end{aligned}$$

The update diagrams in Figure 3.5 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 update 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 update diagram on the left graphically represents the Bellman optimality equation (3.18) and the diagram on the right graphically represents (3.19).

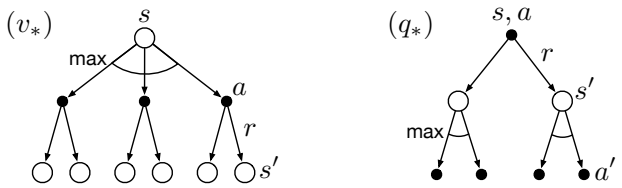


Figure 3.5: Update diagrams for v_* and q_*

For finite MDPs, the Bellman optimality equation for v_π (3.18) has a unique solution independent of the policy. 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 3.9: Bellman Optimality Equations for the Recycling Robot Using (3.18), 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`. Since 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. ■

Example 3.10: Solving the Gridworld Suppose we solve the Bellman equation for v_* for the simple grid task introduced in Example 3.6 and shown again in Figure 3.6 (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 3.6 (middle) shows the optimal value function, and Figure 3.6 (right) shows the corresponding optimal policies. Where there are multiple arrows in a cell, all of the corresponding actions are optimal.

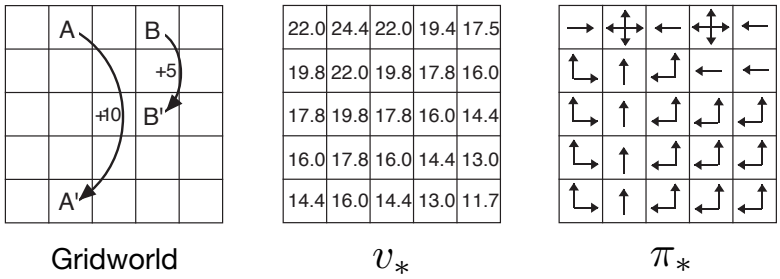


Figure 3.6: Optimal solutions to the gridworld example.

■

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. Since 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 (3.18) 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 3.17 Draw or describe the optimal state-value function for the golf example. □

Exercise 3.18 Draw or describe the contours of the optimal action-value function for putting, $q_*(s, \texttt{putter})$, for the golf example. □

Exercise 3.19 Give the Bellman equation for q_* for the recycling robot. □

Exercise 3.20 Figure 3.6 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 (3.7) to express this value symbolically, and then to compute it to three decimal places. □

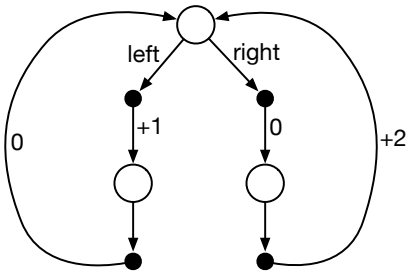
Exercise 3.21 Consider the continuing MDP shown on to the right. The only decision to be made is that in the top state, where two actions are available, left and right. The numbers show the rewards that are received deterministically after each action. There are exactly two deterministic policies, π_{left} and π_{right}. What policy is optimal if $\gamma = 0$? If $\gamma = 0.9$? If $\gamma = 0.5$? □

Exercise 3.22 Give an equation for v_* in terms of q_*. □

Exercise 3.23 Give an equation for q_* in terms of v_* and the world's dynamics, $p(s', r \mid s, a)$. □

Exercise 3.24 Give an equation for π_* in terms of q_*. □

Exercise 3.25 Give an equation for π_* in terms of v_* and the world's dynamics, $p(s', r \mid s, a)$. □



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

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

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

The bioreactor example is based on the work of Ungar (1990) and Miller and Williams (1992). The recycling robot example was inspired by the can-collecting robot built by Jonathan Connell (1989).

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 DP in terms of contraction mappings.

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

The golf example was suggested by Chris Watkins.

Chapter 4

Dynamic Programming

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

Starting with this chapter, we usually assume that the environment is a finite MDP. That is, we assume that its state, action, and reward sets, $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$, are finite, and that its dynamics are given by a set of probabilities $p(s', r \,|\, s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}$, $r \in \mathcal{R}$, 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 Chapter 9 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:

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

or

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

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

4.1 Policy Evaluation (Prediction)

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

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] && \text{(from (3.8))} \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] && (4.3) \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a)\Big[r + \gamma v_\pi(s')\Big], && (4.4) \end{aligned}$$

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 a update can be expressed in an equation, as above, or in an update diagram like those introduced in Chapter 3. For example, the update diagram corresponding to the expected update used in iterative policy evaluation is shown on page 47.

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

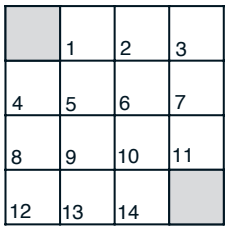
in-place algorithm also converges to v_π; in fact, it usually converges faster than the two-array version, as you might expect, since it uses new data as soon as they are available. We think of the updates as being done in a *sweep* through the state space. For the in-place algorithm, the order in which 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 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 boxed algorithm 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

Input π, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  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 $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$

Example 4.1 Consider the 4×4 gridworld shown below.



$R_t = -1$
on all transitions

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

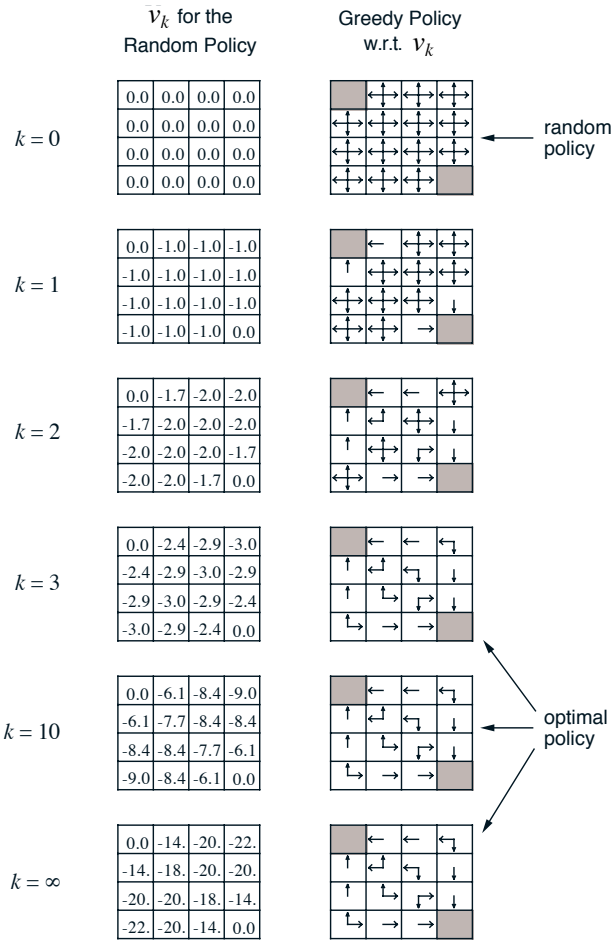


Figure 4.1: Convergence of iterative policy evaluation on a small gridworld. The left column is the sequence of approximations of the state-value function for the random policy (all actions equal). The right column is the sequence of greedy policies corresponding to the value function estimates (arrows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

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

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

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 at least one state. This result applies in particular to the two policies that we considered in the previous paragraph, an original deterministic policy, π, and a changed policy, π', that is identical to π except that $\pi'(s) = a \neq \pi(s)$. Obviously, (4.7) holds at all states other than s. 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 $v_{\pi'}(s)$:

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

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

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

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 on the next page. Note that each policy evaluation, itself an iterative computation, is started

Policy iteration (using iterative policy evaluation)

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      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)

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

This algorithm has a subtle bug, in that it may never terminate if the policy continually switches between two or more policies that are equally good. The bug can be fixed by adding additional flags, but it makes the pseudocode so ugly that it is not worth it.

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 often converges in surprisingly few iterations. This is illustrated by the example in Figure 4.1. The bottom-left diagram 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.

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

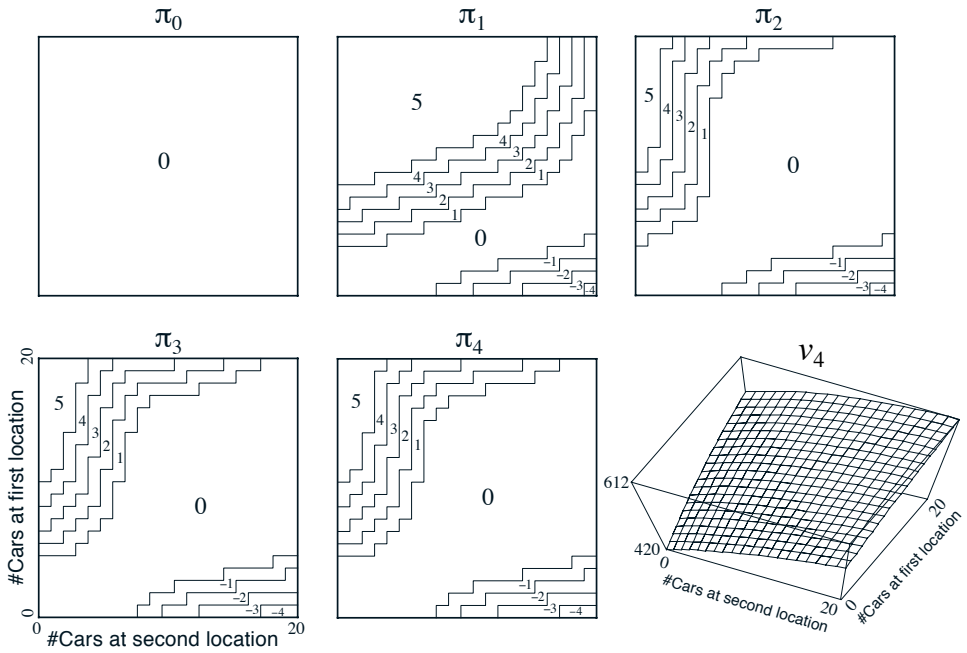


Figure 4.2: The sequence of policies found by policy iteration on Jack's car rental problem, and the final state-value function. The first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second (negative 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.

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. ■

Exercise 4.4 (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. If your computer is too slow for the full problem, cut all the numbers of cars in half. □

Exercise 4.5 How would policy iteration be defined for action values? Give a complete algorithm for computing q_*, analogous to that on page 65 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 $\epsilon/|\mathcal{A}(s)|$. 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_* (page 65). □

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 update diagrams for these algorithms on page 47 (policy evaluation) and on the left of Figure 3.5 (value iteration). These two are the natural update 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

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

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

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. Since 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?

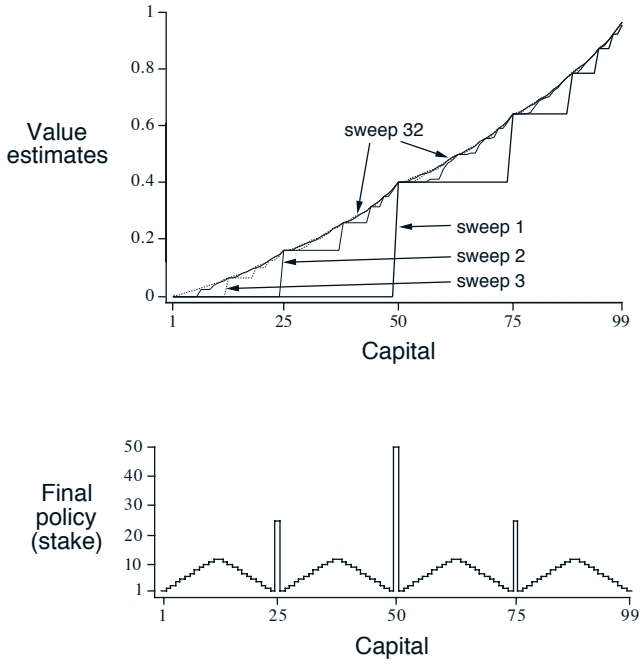


Figure 4.3: The solution to the gambler's problem for $p_h = 0.4$. The upper graph shows the value function found by successive sweeps of value iteration. The lower graph shows the final policy. ■

Exercise 4.7 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 4.8 (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 4.9 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.

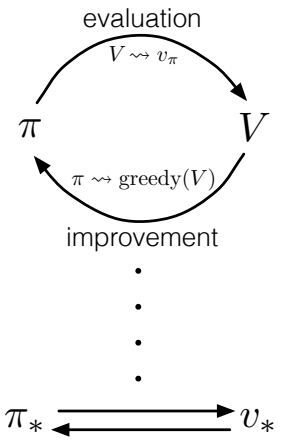
Of course, avoiding sweeps does not necessarily mean that we can get away with less computation. It just means that an algorithm does not need to get locked into any hopelessly long sweep before it can make progress improving a policy. We can try to take advantage of this flexibility by selecting the 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

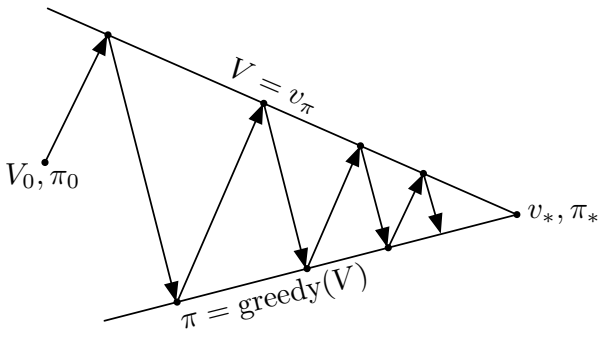
Policy iteration consists of two simultaneous, interacting processes, one making the value function consistent with the current policy (policy evaluation), and the other making the policy greedy with respect to the current value function (policy improvement). In policy iteration, these two processes 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. It is easy to see that 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 *update 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 it can be performed in parallel, with separate processors simultaneously updating the values of individual states using input from other processors. 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 (1957).

Chapter 5

Monte Carlo Methods

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

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

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

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

5.1 Monte Carlo Prediction

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

In particular, suppose we wish to estimate $v_\pi(s)$, the value of a state s under policy π, given a set of episodes obtained by following π and passing through s. Each occurrence of state s in an episode is called a *visit* to s. Of course, s may be visited multiple times in the same episode; let us call the first time it is visited in an episode the *first visit* to s. The *first-visit MC method* estimates $v_\pi(s)$ as the average of the returns following first visits to s, whereas the *every-visit MC method* averages the returns following all visits to s. These two Monte Carlo (MC) methods are very similar but have slightly different theoretical properties. First-visit MC has been most widely studied, dating back to the 1940s, and is the one we focus on in this chapter. Every-visit MC extends more naturally to function approximation and eligibility traces, as discussed in Chapters 9 and 12. First-visit MC is shown in procedural form in the box below.

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

Initialize:
    $\pi \leftarrow$ policy to be evaluated
    $V \leftarrow$ an arbitrary state-value function
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
    Generate an episode using π
    For each state s appearing in the episode:
        $G \leftarrow$ return following the first occurrence of s
        Append G to $Returns(s)$
        $V(s) \leftarrow$ average($Returns(s)$)

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

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

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

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

Consider the policy that sticks if the player's sum is 20 or 21, and otherwise hits. To find the state-value function for this policy by a Monte Carlo approach, one simulates many blackjack games using the policy and averages the returns following each state. Note that in this task the same state never recurs within one episode, so there is no difference between first-visit and every-visit MC methods. In this way, we obtained the estimates of the state-value function shown in Figure 5.1. The estimates for states with a usable ace are less certain and less regular because these states are less common. In any event, after 500,000 games the value function is very well approximated.

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

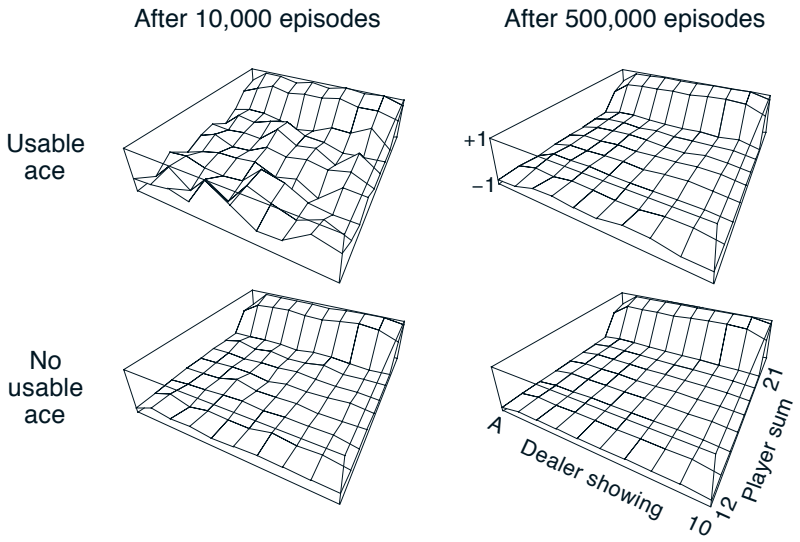


Figure 5.1: Approximate state-value functions for the blackjack policy that sticks only on 20 or 21, computed by Monte Carlo policy evaluation. ■

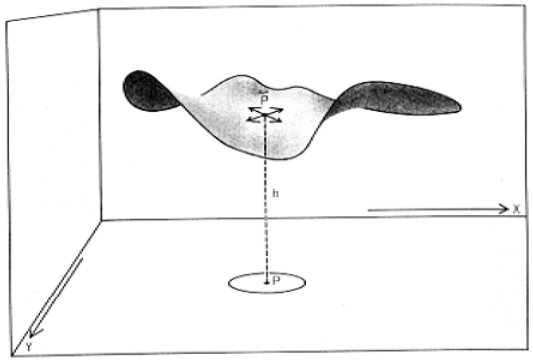
Can we generalize the idea of update diagrams to Monte Carlo algorithms? The general idea of an update diagram is to show at the top the root node to be updated and to show below all the transitions and leaf nodes whose rewards and estimated values contribute to the update. For Monte Carlo estimation of v_π, the root is a state node, and below it is the entire trajectory of transitions along a particular single episode, ending at the terminal state, as shown to the right. Whereas the DP diagram (page 47) shows all possible transitions, the Monte Carlo diagram shows only those sampled on the one episode. Whereas the DP diagram includes only one-step transitions, the Monte Carlo diagram goes all the way to the end of the episode. These differences in the diagrams accurately reflect the fundamental differences between the algorithms.

Monte Carlo update diagram for v_π

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

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

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



A bubble on a wire loop

This is similar to the kind of problem for which Monte Carlo methods were originally designed. Instead of the iterative computation described above, imagine standing on the surface and taking a random walk, stepping randomly from grid point to neighboring grid point, with equal probability, until you reach the boundary. It turns out that the expected value of the height at the boundary is a close approximation to the height of the desired surface at the starting point (in fact, it is exactly the value computed by the iterative method described above). Thus, one can closely approximate the height of the surface at a point by simply averaging the boundary heights of many walks started at the point. If one is interested in only the value at one point, or any fixed small set of points, then this Monte Carlo method can be far more efficient than the iterative method based on local consistency.

■

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

5.2 Monte Carlo Estimation of Action Values

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

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

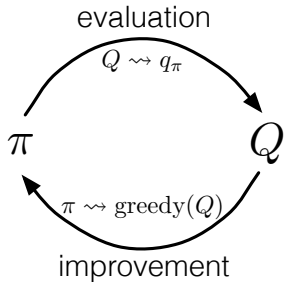
The only complication is that many state–action pairs may never be visited. If π is a deterministic policy, then in following π one will observe returns only for one of the actions from each state. With no returns to average, the Monte Carlo estimates of the other actions will not improve with experience.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

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

The assumption of exploring starts is sometimes useful, but of course it cannot be relied upon in general, particularly when learning directly from actual interaction with an environment. In that case the starting conditions are unlikely to be so helpful. The most common alternative approach to assuring that all state–action pairs are encountered is to consider only policies that are stochastic with a nonzero probability of selecting all actions in each state. We discuss two important variants of this approach in later sections. For now, we retain the assumption of exploring starts and complete the presentation of a full Monte Carlo control method.

Exercise 5.2 What is the update diagram for Monte Carlo estimation of q_π? □

5.3 Monte Carlo Control



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

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

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

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

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

$$\pi(s) \doteq \arg\max_a q(s,a). \tag{5.1}$$

Policy improvement then can be done by constructing each π_{k+1} as the greedy policy with respect to q_{π_k}. The policy improvement theorem (Section 4.2) then applies to π_k and π_{k+1} because, for all $s \in \mathcal{S}$,

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

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

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

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

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

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

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

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s,a) \leftarrow$ arbitrary
  $\pi(s) \leftarrow$ arbitrary
  $Returns(s,a) \leftarrow$ empty list

Repeat forever:
  Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ s.t. all pairs have probability > 0
  Generate an episode starting from S_0, A_0, following π
  For each pair s, a appearing in the episode:
    $G \leftarrow$ return following the first occurrence of s, a
    Append G to $Returns(s,a)$
    $Q(s,a) \leftarrow$ average$(Returns(s,a))$
  For each s in the episode:
    $\pi(s) \leftarrow \arg\max_a Q(s,a)$

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

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

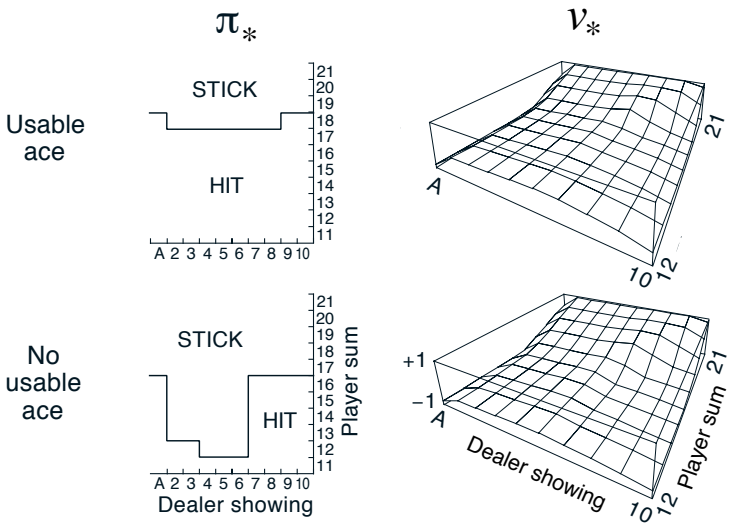


Figure 5.2: The optimal policy and state-value function for blackjack, found by Monte Carlo ES (Figure 5.4). The state-value function shown was computed from the action-value function found by Monte Carlo ES. ■

5.4 Monte Carlo Control without Exploring Starts

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

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

probability, $1-\varepsilon+\frac{\epsilon}{|\mathcal{A}(s)|}$, is given to the greedy action. The ε-greedy policies are examples of *ε-soft* policies, defined as policies for which $\pi(a|s) \geq \frac{\epsilon}{|\mathcal{A}(s)|}$ for all states and actions, for some $\varepsilon > 0$. Among ε-soft policies, ε-greedy policies are in some sense those that are closest to greedy.

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

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

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s,a) \leftarrow$ arbitrary
  $Returns(s,a) \leftarrow$ empty list
  $\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
  (a) Generate an episode using π
  (b) For each pair s, a appearing in the episode:
    $G \leftarrow$ return following the first occurrence of s, a
    Append G to $Returns(s,a)$
    $Q(s,a) \leftarrow$ average$(Returns(s,a))$
  (c) For each s in the episode:
    $A^* \leftarrow \arg\max_a Q(s,a)$ (with ties broken arbitrarily)
    For all $a \in \mathcal{A}(s)$:

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

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

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

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

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

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

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

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

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

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

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

In essence, we have shown in the last few pages that policy iteration works for ε-soft policies. Using the natural notion of greedy policy for ε-soft policies, one is assured of improvement on every step, except when the best policy has been found among the ε-soft policies. This analysis is independent of how the action-value functions are determined at each stage, but it does assume that they are computed exactly. This brings us to roughly the same point as in the previous section. Now we only achieve the best policy among the ε-soft policies, but on the other hand, we have eliminated the assumption of exploring starts.

5.5 Off-policy Prediction via Importance Sampling

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

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

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

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

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

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

where p here is the state-transition probability function defined by (3.3). Thus, the relative probability of the trajectory under the target and behavior policies (the importance-sampling ratio) is

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

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

Now we are ready to give a Monte Carlo algorithm that uses a batch of observed episodes following policy b to estimate $v_\pi(s)$. It is convenient here to number time steps in a way that increases across episode boundaries. That is, if the first episode of the batch ends in a terminal state at time 100, then the next episode begins at time $t = 101$. This enables us to use time-step numbers to refer to particular steps in particular episodes. In particular, we can define the set of all time steps in which state s

is visited, denoted $\mathcal{T}(s)$. This is for an every-visit method; for a first-visit method, $\mathcal{T}(s)$ would only include time steps that were first visits to s within their episodes. Also, let $T(t)$ denote the first time of termination following time t, and G_t denote the return after t up through $T(t)$. Then $\{G_t\}_{t\in\mathcal{T}(s)}$ are the returns that pertain to state s, and $\left\{\rho_{t:T(t)-1}\right\}_{t\in\mathcal{T}(s)}$ are the corresponding importance-sampling ratios. To estimate $v_\pi(s)$, we simply scale the returns by the ratios and average the results:

$$V(s) \doteq \frac{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|}. \tag{5.4}$$

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

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

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

or zero if the denominator is zero. To understand these two varieties of importance sampling, consider their estimates after observing a single return. In the weighted-average estimate, the ratio $\rho_{t:T(t)-1}$ for the single return cancels in the numerator and denominator, so that the estimate is equal to the observed return independent of the ratio (assuming the ratio is nonzero). Given that this return was the only one observed, this is a reasonable estimate, but its expectation is $v_b(s)$ rather than $v_\pi(s)$, and in this statistical sense it is biased. In contrast, the simple average (5.4) is always $v_\pi(s)$ in expectation (it is unbiased), but it can be extreme. Suppose the ratio were ten, indicating that the trajectory observed is ten times as likely under the target policy as under the behavior policy. In this case the ordinary importance-sampling estimate would be *ten times* the observed return. That is, it would be quite far from the observed return even though the episode's trajectory is considered very representative of the target policy.

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

A complete every-visit MC algorithm for off-policy policy evaluation using weighted importance sampling is given in the next section on page 90.

Example 5.4: Off-policy Estimation of a Blackjack State Value
We applied both ordinary and weighted importance-sampling methods to estimate the value of a single blackjack state from off-policy data. Recall that one of the advantages of Monte Carlo methods is that they can be used to evaluate a single state without forming estimates for any other states. In this example, we evaluated the state in which the dealer is showing a deuce, the sum of the player's cards is 13, and the player has a usable ace (that is, the player holds an ace and a deuce, or equivalently three aces). The data was generated by starting in this state then choosing to hit or stick at random with equal probability (the behavior policy). The target policy was to stick only on a sum of 20 or 21, as in Example 5.1. The value of this state under the target policy is approximately -0.27726 (this was determined by separately generating one-hundred million episodes using the target policy and averaging

their returns). Both off-policy methods closely approximated this value after 1000 off-policy episodes using the random policy. To make sure they did this reliably, we performed 100 independent runs, each starting from estimates of zero and learning for 10,000 episodes. Figure 5.3 shows the resultant learning curves—the squared error of the estimates of each method as a function of number of episodes, averaged over the 100 runs. The error approaches zero for both algorithms, but the weighted importance-sampling method has much lower error at the beginning, as is typical in practice.

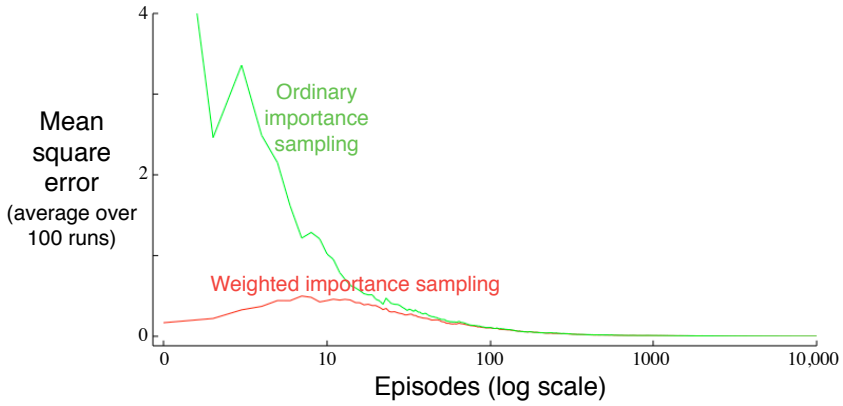


Figure 5.3: Weighted importance sampling produces lower error estimates of the value of a single blackjack state from off-policy episodes (see Example 5.4). ■

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

The lower part of Figure 5.4 shows ten independent runs of the first-visit MC algorithm using ordinary importance sampling. Even after millions of episodes, the estimates fail to converge to the correct value of 1. In contrast, the weighted importance-sampling algorithm would give an estimate of exactly 1 everafter the first episode that ended with the back action. All returns not equal to 1 (that is, ending with the end action) would be inconsistent with the target policy and thus would have a $\rho_{t:T(t)-1}$ of zero and contribute neither to the numerator nor denominator of (5.5). The weighted importance-sampling algorithm produces a weighted average of only the returns consistent with the target policy, and all of these would be exactly 1.

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

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

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

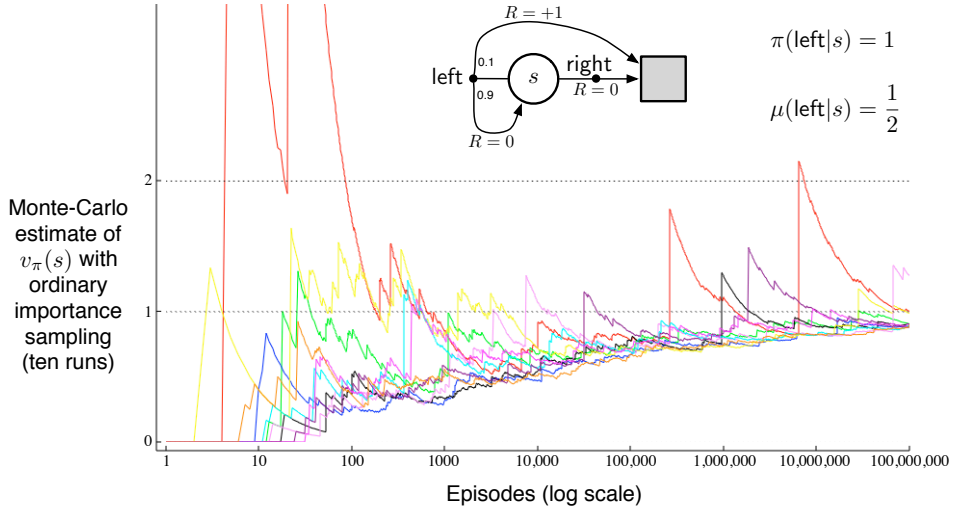


Figure 5.4: Ordinary importance sampling produces surprisingly unstable estimates on the one-state MDP shown inset (Example 5.5). The correct estimate here is 1 ($\gamma = 1$), and, even though this is the expected value of a sample return (after importance sampling), the variance of the samples is infinite,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

importance-sampling-scaled return is infinite:

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

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

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

Exercise 5.3 What is the equation analogous to (5.5) for *action* values $Q(s,a)$ instead of state values $V(s)$, again given returns generated using b? □

Exercise 5.4 In learning curves such as those shown in Figure 5.3 error generally decreases with training, as indeed happened for the ordinary importance-sampling method. But for the weighted importance-sampling method error first increased and then decreased. Why do you think this happened? □

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

5.6 Incremental Implementation

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

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

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

$$V_n \doteq \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}, \qquad n \geq 2, \tag{5.6}$$

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

$$V_{n+1} \doteq V_n + \frac{W_n}{C_n}\Big[G_n - V_n\Big], \qquad n \geq 1, \tag{5.7}$$

and

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

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

Exercise 5.6 Modify the algorithm for first-visit MC policy evaluation (Section 5.1) to use the incremental implementation for sample averages described in Section 2.4. □

Exercise 5.7 Derive the weighted-average update rule (5.7) from (5.6). Follow the pattern of the derivation of the unweighted rule (2.3). □

Off-policy MC prediction, for estimating $Q \approx q_\pi$

Input: an arbitrary target policy π

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$

Repeat forever:
$b \leftarrow$ any policy with coverage of π
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ downto 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
$W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
If $W = 0$ then ExitForLoop

5.7 Off-policy Monte Carlo Control

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

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

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

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

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

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

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ downto 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then ExitForLoop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

Exercise 5.8: Racetrack (programming) Consider driving a race car around a turn like those shown in Figure 5.5. You want to go as fast as possible, but not so fast as to run off the track. In our simplified racetrack, the car is at one of a discrete set of grid positions, the cells in the diagram. The velocity is also discrete, a number of grid cells moved horizontally and vertically per time step. The actions are increments to the velocity components. Each may be changed by $+1$, -1, or 0 in one step, for a total of nine actions. Both velocity components are restricted to be nonnegative and less than 5, and they cannot both be zero except at the starting line. Each episode begins in one of the randomly selected start states with both velocity components zero and ends when the car crosses the finish line. The rewards are -1 for each step until the car crosses the finish line. If the car hits the track boundary, it is moved back to a random position on the starting line, both velocity components are reduced to zero, and the episode continues. Before updating the car's location at each time step, check to see if the projected path of the car intersects the track boundary. If it intersects the finish line, the episode

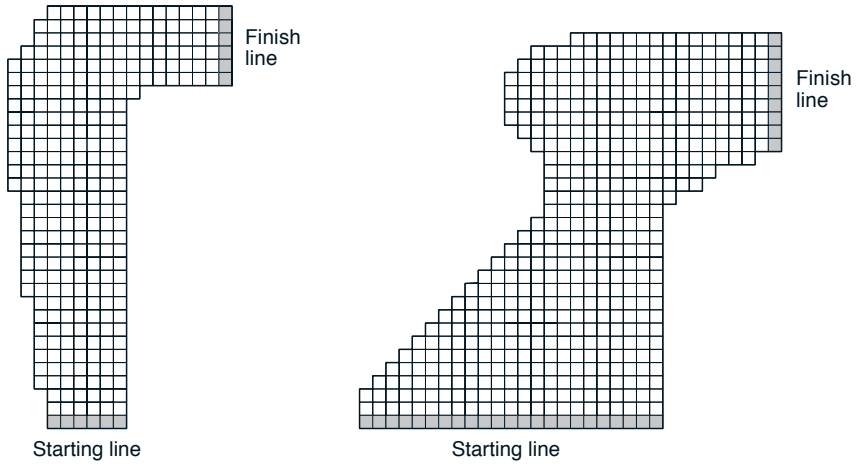


Figure 5.5: A couple of right turns for the racetrack task.

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

5.8 *Discounting-aware Importance Sampling

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

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

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

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

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

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

Now we need to scale the flat partial returns by an importance sampling ratio that is similarly truncated. As $\bar{G}_{t:h}$ only involves rewards up to a horizon h, we only need the ratio of the probabilities up to h. We define an ordinary importance-sampling estimator, analogous to (5.4), as

$$V(s) \doteq \frac{\sum_{t\in\mathcal{T}(s)} \Big((1-\gamma)\sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1}\rho_{t:h-1}\bar{G}_{t:h} \quad + \quad \gamma^{T(t)-t-1}\rho_{t:T(t)-1}\bar{G}_{t:T(t)}\Big)}{|\mathcal{T}(s)|}, \tag{5.8}$$

and a weighted importance-sampling estimator, analogous to (5.5), as

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

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

5.9 *Per-reward Importance Sampling

There is one more way in which the structure of the return as a sum of rewards can be taken into account in off-policy importance sampling, a way that may be able to reduce variance even in the absence of discounting (that is, even if $\gamma = 1$). In the off-policy estimators (5.4) and (5.5), each term of the sum in the numerator is itself a sum:

$$\begin{aligned}\rho_{t:T-1}G_t &= \rho_{t:T-1}\left(R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1}R_T\right)\\ &= \rho_{t:T-1}R_{t+1} + \gamma\rho_{t:T-1}R_{t+2} + \cdots + \gamma^{T-t-1}\rho_{t:T-1}R_T.\end{aligned} \tag{5.10}$$

The off-policy estimators rely on the expected values of these terms; let us see if we can write them in a simpler way. Note that each sub-term of (5.10) is a product of a random reward and a random importance-sampling ratio. For example, the first sub-term can be written, using (5.3), as

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

Now notice that, of all these factors, only the first and the last (the reward) are correlated; all the other ratios are independent random variables whose expected value is one:

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

Thus, because the expectation of the product of independent random variables is the product of their expectations, all the ratios except the first drop out in expectation, leaving just

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

If we repeat this analysis for the kth term of (5.10), we get

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

It follows then that the expectation of our original term (5.10) can be written

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

where

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

We call this idea *per-reward* importance sampling. It follows immediately that there is an alternate importance-sampling estimator, with the same unbiased expectation as the ordinary-importance-sampling estimator (5.4), using $\tilde{G}_t$:

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \tilde{G}_t}{|\mathcal{T}(s)|}, \tag{5.11}$$

which we might expect to sometimes be of lower variance.

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

***Exercise 5.9** Modify the algorithm for off-policy Monte Carlo control (page 91) to use the idea of the truncated weighted-average estimator (5.9). Note that you will first need to convert this equation to action values. □

5.10 Summary

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

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

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

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

exploring and tries to find the best policy that still explores. In *off-policy* methods, the agent also explores, but learns a deterministic optimal policy that may be unrelated to the policy followed.

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

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

Bibliographical and Historical Remarks

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

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

Barto and Duff (1994) discussed policy evaluation in the context of classical Monte Carlo algorithms for solving systems of linear equations. They used the analysis of Curtiss (1954) to point out the computational advantages of Monte Carlo policy evaluation for large problems.

5.3–4 Monte Carlo ES was introduced in the 1998 edition of this book. That may have been the first explicit connection between Monte Carlo estimation and control methods based on policy iteration. An early use of Monte Carlo methods to estimate action values in a reinforcement learning context was by Michie and Chambers (1968). In pole balancing (page 44), they used averages of episode durations to assess the worth (expected balancing "life") of each possible action in each state, and then used these assessments to control action selections. Their method is similar in spirit to Monte Carlo ES with every-visit MC estimates. Narendra and Wheeler (1986) studied a Monte Carlo method for ergodic finite Markov chains that used the return accumulated between successive visits to the same state as a reward for adjusting a learning automaton's action probabilities.

5.5 Efficient off-policy learning has become recognized as an important challenge that arises in several fields. For example, it is closely related to the idea of "interventions" and "counterfactuals" in probabalistic graphical (Bayesian) models (e.g., Pearl, 1995; Balke and Pearl, 1994). Off-policy methods using importance sampling have a long history and yet still are not well

understood. Weighted importance sampling, which is also sometimes called normalized importance sampling (e.g., Koller and Friedman, 2009), is discussed by Rubinstein (1981), Hesterberg (1988), Shelton (2001), and Liu (2001) among others.

The target policy in off-policy learning is sometimes referred to in the literature as the "estimation" policy, as it was in the first edition of this book.

5.7 The racetrack exercise is adapted from Barto, Bradtke, and Singh (1995), and from Gardner (1973).

5.8 Our treatment of the idea of discounting-aware importance sampling is based on the analysis of Sutton, Mahmood, Precup, and van Hasselt (2014). It has been worked out most fully to date by Mahmood (in preparation; Mahmood, van Hasselt, and Sutton, 2014).

5.9 Per-reward importance sampling was introduced by Precup, Sutton, and Singh (2000), who called it "per-decision" importance sampling. These works also combine off-policy learning with temporal-difference learning, eligibility traces, and approximation methods, introducing subtle issues that we consider in later chapters.

Chapter 6

Temporal-Difference Learning

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

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

6.1 TD Prediction

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

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big], \tag{6.1}$$

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

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

immediately on transition to S_{t+1} and receiving R_{t+1}. In effect, the target for the Monte Carlo update is G_t, whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. This TD method is called *TD(0)*, or

one-step TD, because it is a special case of the TD(λ) and n-step TD methods developed in Chapter 12 and Chapter 7. The box below specifies TD(0) completely in procedural form.

Tabular TD(0) for estimating v_π

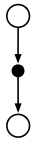
Input: the policy π to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0$, for all $s \in \mathcal{S}^+$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

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

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] && (6.3)\\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] && \text{(from (3.8))}\\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]\,. && (6.4) \end{aligned}$$

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

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


TD(0)

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

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

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

as a sum of TD errors:

$$\begin{aligned} G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) && \text{(from (3.8))} \\ &= \delta_t + \gamma\big(G_{t+1} - V(S_{t+1})\big) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\big(G_{t+2} - V(S_{t+2})\big) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}\big(G_T - V(S_T)\big) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}\big(0 - 0\big) \\ &= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k. && (6.6) \end{aligned}$$

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

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

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

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

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

A simple way to view the operation of Monte Carlo methods is to plot the predicted total time (the last column) over the sequence, as in Figure 6.1 (left). The arrows show the changes in predictions recommended by the constant-α MC method (6.1), for $\alpha = 1$. These are exactly the errors between the estimated value (predicted time to go) in each state and the actual return (actual time to go). For

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

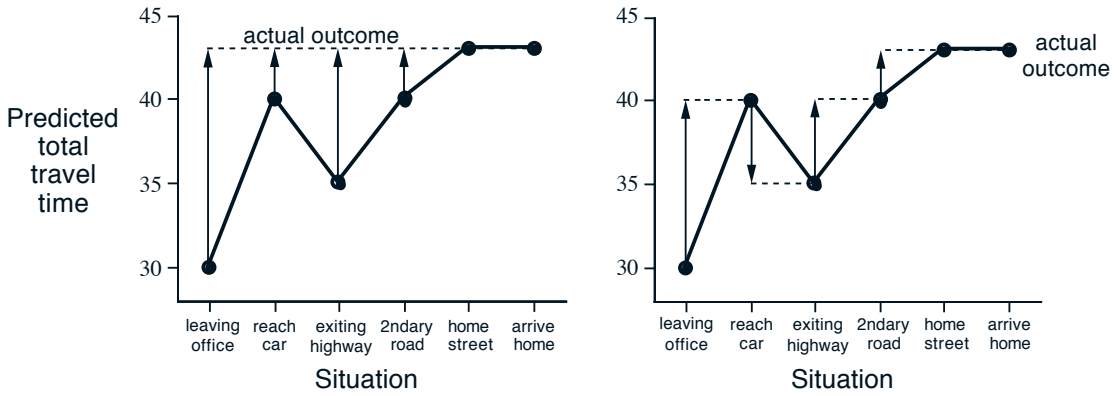


Figure 6.1: Changes recommended in the driving home example by Monte Carlo methods (left) and TD methods (right).

example, when you exited the highway you thought it would take only 15 minutes more to get home, but in fact it took 23 minutes. Equation 6.1 applies at this point and determines an increment in the estimate of time to go after exiting the highway. The error, $G_t - V(S_t)$, at this time is eight minutes. Suppose the step-size parameter, α, is 1/2. Then the predicted time to go after exiting the highway would be revised upward by four minutes as a result of this experience. This is probably too large a change in this case; the truck was probably just an unlucky break. In any event, the change can only be made off-line, that is, after you have reached home. Only at this point do you know any of the actual returns.

Is it necessary to wait until the final outcome is known before learning can begin? Suppose on another day you again estimate when leaving your office that it will take 30 minutes to drive home, but then you become stuck in a massive traffic jam. Twenty-five minutes after leaving the office you are still bumper-to-bumper on the highway. You now estimate that it will take another 25 minutes to get home, for a total of 50 minutes. As you wait in traffic, you already know that your initial estimate of 30 minutes was too optimistic. Must you wait until you get home before increasing your estimate for the initial state? According to the Monte Carlo approach you must, because you don't yet know the true return.

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

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

6.2 Advantages of TD Prediction Methods

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

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

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

But are TD methods sound? Certainly it is convenient to learn one guess from the next, without waiting for an actual outcome, but can we still guarantee convergence to the correct answer? Happily, the answer is yes. For any fixed policy π, TD(0) has been proved to converge to v_π, in the mean for a constant step-size parameter if it is sufficiently small, and with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions (2.7). Most convergence proofs apply only to the table-based case of the algorithm presented above (6.2), but some also apply to the case of general linear function approximation. These results are discussed in a more general setting in Chapter 9.

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

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

Exercise 6.3 From Figure 6.2 (left) it appears that the first episode results in a change in only $V(\mathsf{A})$. What does this tell you about what happened on the first episode? Why was only the estimate for this one state changed? By exactly how much was it changed? □

Exercise 6.4 The specific results shown in Figure 6.2 (right) are dependent on the value of the step-size parameter, α. Do you think the conclusions about which algorithm is better would be affected if a wider range of α values were used? Is there a different, fixed value of α at which either algorithm would have performed significantly better than shown? Why or why not? □

***Exercise 6.5** In Figure 6.2 (right) the RMS error of the TD method seems to go down and then up again, particularly at high α's. What could have caused this? Do you think this always occurs, or might it be a function of how the approximate value function was initialized? □

Exercise 6.6 Above we stated that the true values for the random walk task are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}$, and $\frac{5}{6}$, for states A through E. Describe at least two different ways that these could have been computed. Which would you guess we actually used? Why? □

Example 6.2: Random Walk In this example we empirically compare the prediction abilities of TD(0) and constant-α MC applied to the small Markov reward process shown in the upper part of the figure below. All episodes start in the center state, C, and proceed either left or right by one state on each step, with equal probability. This behavior can be thought of as due to the combined effect of a fixed policy and an environment's state-transition probabilities, but we do not care which; we are concerned only with predicting returns however they are generated. Episodes terminate either on the extreme left or the extreme right. When an episode terminates on the right, a reward of +1 occurs; all other rewards are zero. For example, a typical epsiode might consist of the following state-and-reward sequence: $\mathsf{C}, 0, \mathsf{B}, 0, \mathsf{C}, 0, \mathsf{D}, 0, \mathsf{E}, 1$. Because this task is undiscounted, the true value of each state is the probability of terminating on the right if starting from that state. Thus, the true value of the center state is $v_\pi(\mathsf{C}) = 0.5$. The true values of all the states, A through E, are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}$, and $\frac{5}{6}$. The left part of Figure 6.2 shows the values learned by TD(0) approaching the true values as more episodes are experienced. Averaging over many episode sequences, the right part of the figure shows the average error in the predictions found by TD(0) and constant-α MC, for a variety of values of α, as a function of number of episodes. In all cases the approximate value function was initialized to the intermediate value $V(s) = 0.5$, for all s. The TD method was consistently better than the MC method on this task.

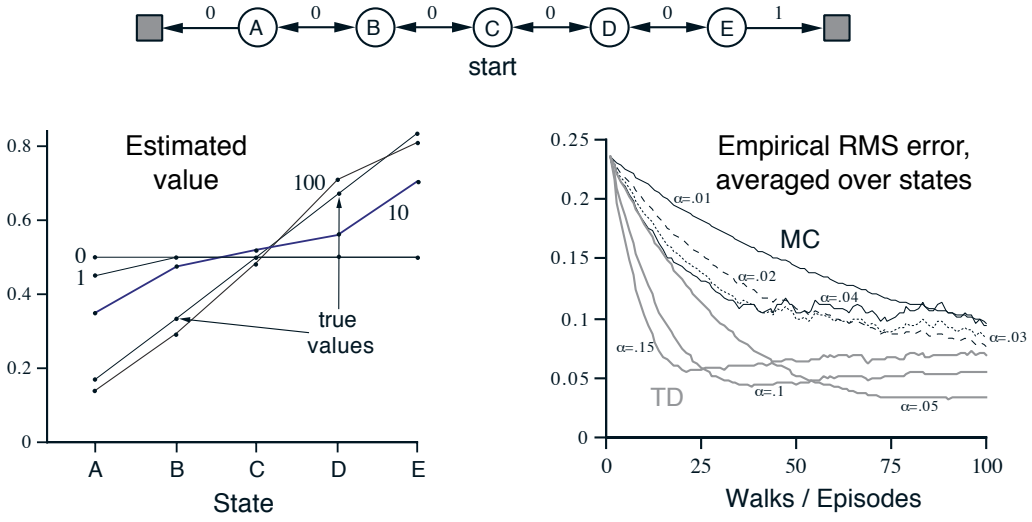


Figure 6.2: Results with the 5-state random walk. *Top:* The small Markov reward process generating the episodes. *Left:* Results from a single run after various numbers of episodes. The estimate after 100 episodes is about as close as they ever get to the true values; with a constant step-size parameter ($\alpha = 0.1$ in this example), the values fluctuate indefinitely in response to the outcomes of the most recent episodes. *Right:* Learning curves for TD(0) and constant-α MC methods, for various values of α. The performance measure shown is the root mean-squared (RMS) error between the value function learned and the true value function, averaged over the five states. These data are averages over 100 different sequences of episodes.

■

6.3 Optimality of TD(0)

Suppose there is available only a finite amount of experience, say 10 episodes or 100 time steps. In this case, a common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer. Given an approximate value function, V, the increments specified by (6.1) or (6.2) are computed for every time step t at which a nonterminal state is visited, but the value function is changed only once, by the sum of all the increments. Then all the available experience is processed again with the new value function to produce a new overall increment, and so on, until the value function converges. We call this *batch updating* because updates are made only after processing each complete *batch* of training data.

Under batch updating, TD(0) converges deterministically to a single answer independent of the step-size parameter, α, as long as α is chosen to be sufficiently small. The constant-α MC method also converges deterministically under the same conditions, but to a different answer. Understanding these two answers will help us understand the difference between the two methods. Under normal updating the methods do not move all the way to their respective batch answers, but in some sense they take steps in these directions. Before trying to understand the two answers in general, for all possible tasks, we first look at a few examples.

Example 6.3: Random walk under batch updating Batch-updating versions of TD(0) and constant-α MC were applied as follows to the random walk prediction example (Example 6.2). After each new episode, all episodes seen so far were treated as a batch. They were repeatedly presented to the algorithm, either TD(0) or constant-α MC, with α sufficiently small that the value function converged. The resulting value function was then compared with v_π, and the average root mean-squared error across the five states (and across 100 independent repetitions of the whole experiment) was plotted to obtain the learning curves shown in Figure 6.3. Note that the batch TD method was consistently better than the batch Monte Carlo method. ■

Under batch training, constant-α MC converges to values, $V(s)$, that are sample averages of the actual returns experienced after visiting each state s. These are optimal estimates in the sense that they minimize the mean-squared error from the actual returns in the training set. In this sense it is surprising that the batch TD method was able to perform better according to the root mean-squared error measure shown in Figure 6.3. How is it that batch TD was able to perform better than this optimal method? The answer is that the Monte Carlo method is optimal only in a limited way, and that TD is optimal in a way that is more relevant to predicting returns. But first let's develop our

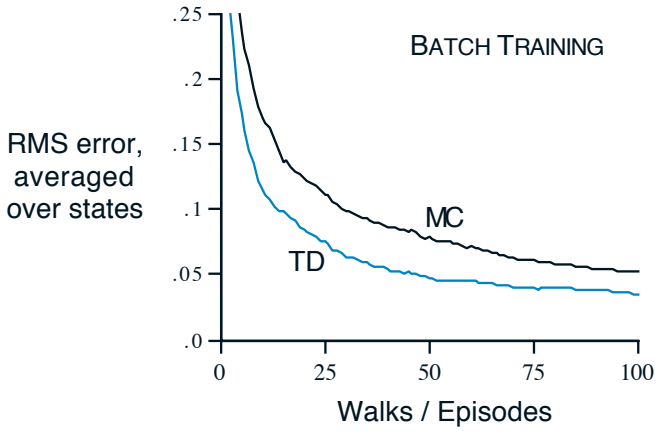


Figure 6.3: Performance of TD(0) and constant-α MC under batch training on the random walk task.

intuitions about different kinds of optimality through another example. Consider Example 6.4, on the next page.

Example 6.4 illustrates a general difference between the estimates found by batch TD(0) and batch Monte Carlo methods. Batch Monte Carlo methods always find the estimates that minimize mean-squared error on the training set, whereas batch TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process. In general, the *maximum-likelihood estimate* of a parameter is the parameter value whose probability of generating the data is greatest. In this case, the maximum-likelihood estimate is the model of the Markov process formed in the obvious way from the observed episodes: the estimated transition probability from i to j is the fraction of observed transitions from i that went to j, and the associated expected reward is the average of the rewards observed on those transitions. Given this model, we can compute the estimate of the value function that would be exactly correct if the model were exactly correct. This is called the *certainty-equivalence estimate* because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated. In general, batch TD(0) converges to the certainty-equivalence estimate.

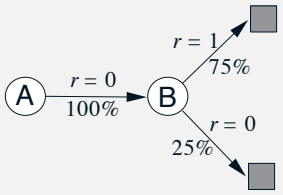
Example 6.4 You are the Predictor

Place yourself now in the role of the predictor of returns for an unknown Markov reward process. Suppose you observe the following eight episodes:

A, 0, B, 0	B, 1
B, 1	B, 1
B, 1	B, 1
B, 1	B, 0

This means that the first episode started in state A, transitioned to B with a reward of 0, and then terminated from B with a reward of 0. The other seven episodes were even shorter, starting from B and terminating immediately. Given this batch of data, what would you say are the optimal predictions, the best values for the estimates $V(\mathsf{A})$ and $V(\mathsf{B})$? Everyone would probably agree that the optimal value for $V(\mathsf{B})$ is $\frac{3}{4}$, because six out of the eight times in state B the process terminated immediately with a return of 1, and the other two times in B the process terminated immediately with a return of 0.

But what is the optimal value for the estimate $V(\mathsf{A})$ given this data? Here there are two reasonable answers. One is to observe that 100% of the times the process was in state A it traversed immediately to B (with a reward of 0); and since we have already decided that B has value $\frac{3}{4}$, therefore A must have value $\frac{3}{4}$ as well. One way of viewing this answer is that it is based on first modeling the Markov process, in this case as shown to the right, and then computing the correct estimates given the model, which indeed in this case gives $V(\mathsf{A}) = \frac{3}{4}$. This is also the answer that batch TD(0) gives.



The other reasonable answer is simply to observe that we have seen A once and the return that followed it was 0; we therefore estimate $V(\mathsf{A})$ as 0. This is the answer that batch Monte Carlo methods give. Notice that it is also the answer that gives minimum squared error on the training data. In fact, it gives zero error on the data. But still we expect the first answer to be better. If the process is Markov, we expect that the first answer will produce lower error on *future* data, even though the Monte Carlo answer is better on the existing data.

■

This helps explain why TD methods converge more quickly than Monte Carlo methods. In batch form, TD(0) is faster than Monte Carlo methods because it computes the true certainty-equivalence estimate. This explains the advantage of TD(0) shown in the batch results on the random walk task (Figure 6.3). The relationship to the certainty-equivalence estimate may also explain in part the speed advantage of nonbatch TD(0) (e.g., Figure 6.2, right). Although the nonbatch methods do not achieve either the certainty-equivalence or the minimum squared-error estimates, they can be understood as moving roughly in these directions. Nonbatch TD(0) may be faster than constant-α MC because it is moving toward a better estimate, even though it is not getting all the way there. At the current time nothing more definite can be said about the relative efficiency of on-line TD and Monte Carlo methods.

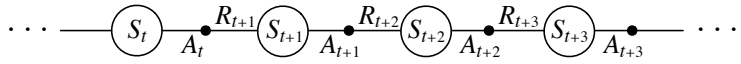
Finally, it is worth noting that although the certainty-equivalence estimate is in some sense an optimal solution, it is almost never feasible to compute it directly. If N is the number of states, then just forming the maximum-likelihood estimate of the process may require N^2 memory, and computing the corresponding value function requires on the order of N^3 computational steps if done conventionally. In these terms it is indeed striking that TD methods can approximate the same solution using memory no more than N and repeated computations over the training set. On tasks with large state spaces, TD methods may be the only feasible way of approximating the certainty-equivalence solution.

*__Exercise 6.7__ Design an off-policy version of the TD(0) update that can be used with arbitrary target policy π and covering behavior policy b, using at each step t the importance sampling ratio $\rho_{t:t}$ (5.3). □

6.4 Sarsa: On-policy TD Control

We turn now to the use of TD prediction methods for the control problem. As usual, we follow the pattern of generalized policy iteration (GPI), only this time using TD methods for the evaluation or prediction part. As with Monte Carlo methods, we face the need to trade off exploration and exploitation, and again approaches fall into two main classes: on-policy and off-policy. In this section we present an on-policy TD control method.

The first step is to learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $q_\pi(s,a)$ for the current behavior policy π and for all states s and actions a. This can be done using essentially the same TD method described above for learning v_π. Recall that an episode consists of an alternating sequence of states and state–action pairs:



In the previous section we considered transitions from state to state and learned the values of states. Now we consider transitions from state–action pair to state–action pair, and learn the values of state–action pairs. Formally these cases are identical: they are both Markov chains with a reward process. The theorems assuring the convergence of state values under TD(0) also apply to the corresponding algorithm for action values:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1},A_{t+1}) - Q(S_t,A_t)\Big]. \tag{6.7}$$

This update is done after every transition from a nonterminal state S_t. If S_{t+1} is terminal, then $Q(S_{t+1},A_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events, $(S_t,A_t,R_{t+1},S_{t+1},A_{t+1})$, that make up a transition from one state–action pair to the next. This quintuple gives rise to the name *Sarsa* for the algorithm. The update diagram for Sarsa is as shown to the right.


Sarsa

Exercise 6.8 Show that an action-value version of (6.6) holds for the action-value form of the TD error $\delta_t = R_{t+1} + \gamma Q(S_{t+1},A_{t+1} - Q(S_t,A_t)$, again assuming that the values don't

change from step to step. □

It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method. As in all on-policy methods, we continually estimate q_π for the behavior policy π, and at the same time change π toward greediness with respect to q_π. The general form of the Sarsa control algorithm is given in the box below.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q. For example, one could use ε-greedy or ε-soft policies. According to Satinder Singh (personal communication), Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state–action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be arranged, for example, with ε-greedy policies by setting $\varepsilon = 1/t$), but this result has not yet been published in the literature.

Example 6.5: Windy Gridworld Shown inset in Figure 6.4 is a standard gridworld, with start and goal states, but with one difference: there is a crosswind upward through the middle of the grid. The actions are the standard four—`up`, `down`, `right`, and `left`—but in the middle region the resultant next states are shifted upward by a "wind," the strength of which varies from column to column. The strength of the wind is given below each column, in number of cells shifted upward. For example, if you are one cell to the right of the goal, then the action `left` takes you to the cell just above the goal. Let us treat this as an undiscounted episodic task, with constant rewards of -1 until the goal state is reached.

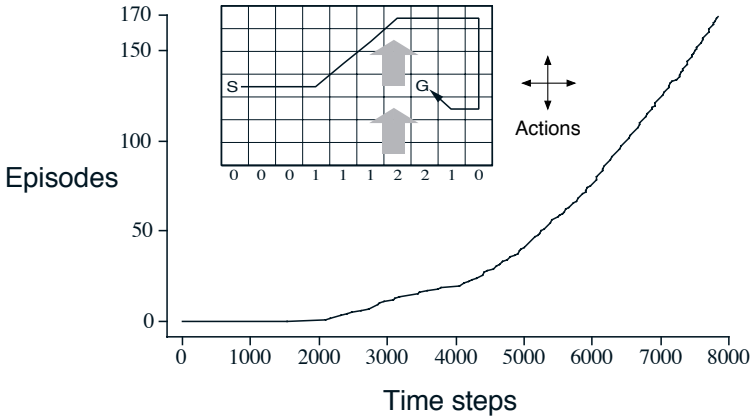


Figure 6.4: Results of Sarsa applied to a gridworld (shown inset) in which movement is altered by a location-dependent, upward "wind." A trajectory under the optimal policy is also shown.

The graph in Figure 6.4 shows the results of applying ε-greedy Sarsa to this task, with $\varepsilon = 0.1$, $\alpha = 0.5$, and the initial values $Q(s,a) = 0$ for all s, a. The increasing slope of the graph shows that the goal is reached more and more quickly over time. By 8000 time steps, the greedy policy was long since optimal (a trajectory from it is shown inset); continued ε-greedy exploration kept the average episode length at about 17 steps, two more than the minimum of 15. Note that Monte Carlo methods cannot easily be used on this task because termination is not guaranteed for all policies. If a policy was ever found that caused the agent to stay in the same state, then the next episode would never end. Step-by-step learning methods such as Sarsa do not have this problem because they quickly learn *during the episode* that such policies are poor, and switch to something else. ■

Exercise 6.9: *Windy Gridworld with King's Moves* Re-solve the windy gridworld task assuming eight possible actions, including the diagonal moves, rather than the usual four. How much better can you do with the extra actions? Can you do even better by including a ninth action that causes no movement at all other than that caused by the wind? □

Exercise 6.10: *Stochastic Wind* Re-solve the windy gridworld task with King's moves, assuming that the effect of the wind, if there is any, is stochastic, sometimes varying by 1 from the mean values given for each column. That is, a third of the time you move exactly according to these values, as in the previous exercise, but also a third of the time you move one cell above that, and another third of the time you move one cell below that. For example, if you are one cell to the right of the goal and you move `left`, then one-third of the time you move one cell above the goal, one-third of the time you move two cells above the goal, and one-third of the time you move to the goal. □

6.5 Q-learning: Off-policy TD Control

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as *Q-learning* (Watkins, 1989), defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]. \tag{6.8}$$

In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state–action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. As we observed in Chapter 5, this is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to q_*.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

What is the update diagram for Q-learning? The rule (6.8) updates a state–action pair, so the top node, the root of the update, must be a small, filled action node. The update is also *from* action nodes, maximizing over all those actions possible in the next state. Thus the bottom nodes of the update diagram should be all these action nodes. Finally, remember that we indicate taking the maximum of these "next action" nodes with an arc across them (Figure 3.5-right). Can you guess now what the diagram is? If so, please do make a guess before turning to the answer in Figure 6.6 on page 109.

Example 6.6: Cliff Walking This gridworld example compares Sarsa and Q-learning, highlighting the difference between on-policy (Sarsa) and off-policy (Q-learning) methods. Consider the gridworld shown in the upper part of Figure 6.5. This is a standard undiscounted, episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Reward is -1 on all transitions except those into the region marked "The Cliff." Stepping into this region incurs a reward of -100 and sends the agent instantly back to the start.

The lower part of Figure 6.5 shows the performance of the Sarsa and Q-learning methods with ε-greedy action selection, $\varepsilon = 0.1$. After an initial transient, Q-learning learns values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the ε-greedy action selection. Sarsa, on the other hand, takes the action selection into account and learns the longer but safer path through the upper part of the grid. Although Q-learning actually learns the values of the optimal policy, its on-line performance is worse than that of Sarsa, which learns the roundabout policy. Of course, if ε were gradually reduced, then both methods would asymptotically converge to the optimal policy.

Figure 6.5: The cliff-walking task. The results are from a single run, but smoothed by averaging the reward sums from 10 successive episodes. ■

Exercise 6.11 Why is Q-learning considered an *off-policy* control method? □

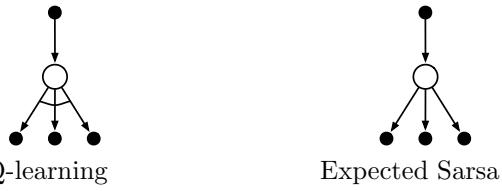


Figure 6.6: The update diagrams for Q-learning and expected Sarsa.

6.6 Expected Sarsa

Consider the learning algorithm that is just like Q-learning except that instead of the maximum over next state–action pairs it uses the expected value, taking into account how likely each action is under the current policy. That is, consider the algorithm with the update rule

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \mathbb{E}[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\Big] \\ &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big], \end{aligned} \tag{6.9}$$

but that otherwise follows the schema of Q-learning. Given the next state, S_{t+1}, this algorithm moves *deterministically* in the same direction as Sarsa moves *in expectation*, and accordingly it is called *expected Sarsa*. Its update diagram is shown on the right in Figure 6.6.

Expected Sarsa is more complex computationally than Sarsa but, in return, it eliminates the variance due to the random selection of A_{t+1}. Given the same amount of experience we might expect it to perform slightly better than Sarsa, and indeed it generally does. Figure 6.7 shows summary results on the cliff-walking task with Expected Sarsa compared to Sarsa and Q-learning. Expected Sarsa retains the significant advantage of Sarsa over Q-learning on this problem. In addition, Expected Sarsa

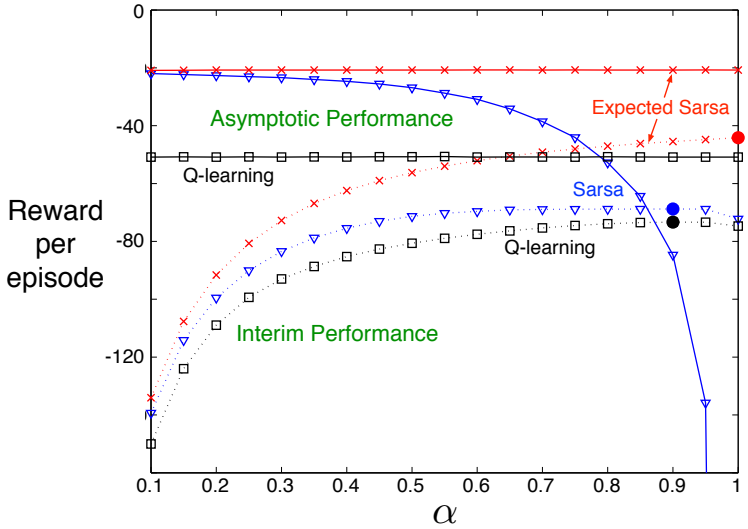


Figure 6.7: Interim and asymptotic performance of TD control methods on the cliff-walking task as a function of α. All algorithms used an ε-greedy policy with $\varepsilon = 0.1$. Asymptotic performance is an average over 100,000 episodes whereas interim performance is an average over the first 100 episodes. These data are averages of over 50,000 and 10 runs for the interim and asymptotic cases respectively. The solid circles mark the best interim performance of each method. Adapted from van Seijen et al. (2009).

shows a significant improvement over Sarsa over a wide range of values for the step-size parameter α. In cliff walking the state transitions are all deterministic and all randomness comes from the policy. In such cases, Expected Sarsa can safely set $\alpha = 1$ without suffering any degradation of asymptotic performance, whereas Sarsa can only perform well in the long run at a small value of α, at which short-term performance is poor. In this and other examples there is a consistent empirical advantage of Expected Sarsa over Sarsa.

In these cliff walking results Expected Sarsa was used on-policy, but in general it might use a policy different from the target policy π to generate behavior, in which case it becomes an off-policy algorithm. For example, suppose π is the greedy policy while behavior is more exploratory; then Expected Sarsa is exactly Q-learning. In this sense Expected Sarsa subsumes and generalizes Q-learning while reliably improving over Sarsa. Except for the small additional computational cost, Expected Sarsa may completely dominate both of the other more-well-known TD control algorithms.

6.7 Maximization Bias and Double Learning

All the control algorithms that we have discussed so far involve maximization in the construction of their target policies. For example, in Q-learning the target policy is the greedy policy given the current action values, which is defined with a max, and in Sarsa the policy is often ε-greedy, which also involves a maximization operation. In these algorithms, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias. To see why, consider a single state s where there are many actions a whose true values, $q(s,a)$, are all zero but whose estimated values, $Q(s,a)$, are uncertain and thus distributed some above and some below zero. The maximum of the true values is zero, but the maximum of the estimates is positive, a positive bias. We call this *maximization bias*.

Example 6.7: Maximization Bias Example The small MDP shown inset in Figure 6.8 provides a simple example of how maximization bias can harm the performance of TD control algorithms. The MDP has two non-terminal states A and B. Episodes always start in A with a choice between two actions, left and right. The right action transitions immediately to the terminal state with a reward and return of zero. The left action transitions to B, also with a reward of zero, from which there are many possible actions all of which cause immediate termination with a reward drawn from a normal

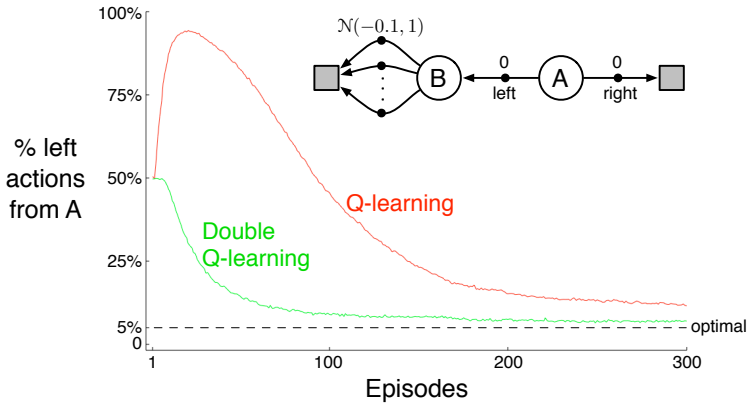


Figure 6.8: Comparison of Q-learning and Double Q-learning on a simple episodic MDP (shown inset). Q-learning initially learns to take the left action much more often than the right action, and always takes it significantly more often than the 5% minimum probability enforced by ε-greedy action selection with $\varepsilon = 0.1$. In contrast, Double Q-learning is almost unaffected by maximization bias. These data are averaged over 10,000 runs. The initial action-value estimates were zero. Any ties in ε-greedy action selection were broken randomly.

distribution with mean -0.1 and variance 1.0. Thus, the expected return for any trajectory starting with left is -0.1, and thus taking left in state A is always a mistake. Nevertheless, our control methods may favor left because of maximization bias making B appear to have a positive value. Figure 6.8 shows that Q-learning with ε-greedy action selection initially learns to strongly favor the left action on this example. Even at asymptote, Q-learning takes the left action about 5% more often than is optimal at our parameter settings ($\varepsilon = 0.1$, $\alpha = 0.1$, and $\gamma = 1$). ■

Are there algorithms that avoid maximization bias? To start, consider a bandit case in which we have noisy estimates of the value of each of many actions, obtained as sample averages of the rewards received on all the plays with each action. As we discussed above, there will be a positive maximization bias if we use the maximum of the estimates as an estimate of the maximum of the true values. One way to view the problem is that it is due to using the same samples (plays) both to determine the maximizing action and to estimate its value. Suppose we divided the plays in two sets and used them to learn two independent estimates, call them $Q_1(a)$ and $Q_2(a)$, each an estimate of the true value $q(a)$, for all $a \in \mathcal{A}$. We could then use one estimate, say Q_1, to determine the maximizing action $A^* = \operatorname{arg\,max}_a Q_1(a)$, and the other, Q_2, to provide the estimate of its value, $Q_2(A^*) = Q_2(\operatorname{arg\,max}_a Q_1(a))$. This estimate will then be unbiased in the sense that $\mathbb{E}[Q_2(A^*)] = q(A^*)$. We can also repeat the process with the role of the two estimates reversed to yield a second unbiased estimate $Q_1(\operatorname{arg\,max}_a Q_2(a))$. This is the idea of *double learning*. Note that although we learn two estimates, only one estimate is updated on each play; double learning doubles the memory requirements, but does not increase the amount of computation per step.

The idea of double learning extends naturally to algorithms for full MDPs. For example, the double learning algorithm analogous to Q-learning, called Double Q-learning, divides the time steps in two, perhaps by flipping a coin on each step. If the coin comes up heads, the update is

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q_2\big(S_{t+1}, \operatorname*{arg\,max}_a Q_1(S_{t+1}, a)\big) - Q_1(S_t, A_t)\Big]. \tag{6.10}$$

If the coin comes up tails, then the same update is done with Q_1 and Q_2 switched, so that Q_2 is updated. The two approximate value functions are treated completely symmetrically. The behavior policy can use both action-value estimates. For example, an ε-greedy policy for Double Q-learning could be based on the average (or sum) of the two action-value estimates. A complete algorithm for Double Q-learning is given below. This is the algorithm used to produce the results in Figure 6.8. In that example, double learning seems to eliminate the harm caused by maximization bias. Of course there are also double versions of Sarsa and Expected Sarsa.

Double Q-learning

Initialize $Q_1(s,a)$ and $Q_2(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily
Initialize $Q_1(\textit{terminal-state}, \cdot) = Q_2(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q_1 and Q_2 (e.g., ε-greedy in $Q_1 + Q_2$)
    Take action A, observe R, S'
    With 0.5 probabilility:
      $Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{arg\,max}_a Q_1(S',a)\big) - Q_1(S,A)\Big)$
    else:
      $Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{arg\,max}_a Q_2(S',a)\big) - Q_2(S,A)\Big)$
    $S \leftarrow S'$
  until S is terminal

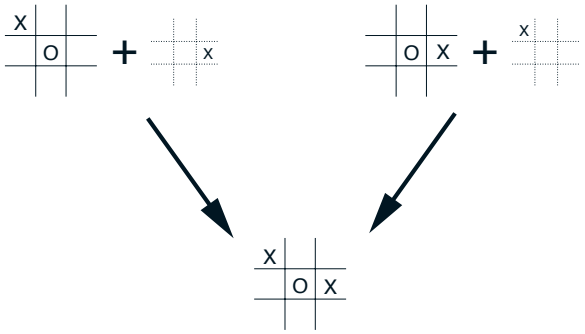
***Exercise 6.12** What are the update equations for Double Expected Sarsa with an ε-greedy target

policy? □

6.8 Games, Afterstates, and Other Special Cases

In this book we try to present a uniform approach to a wide class of tasks, but of course there are always exceptional tasks that are better treated in a specialized way. For example, our general approach involves learning an *action*-value function, but in Chapter 1 we presented a TD method for learning to play tic-tac-toe that learned something much more like a *state*-value function. If we look closely at that example, it becomes apparent that the function learned there is neither an action-value function nor a state-value function in the usual sense. A conventional state-value function evaluates states in which the agent has the option of selecting an action, but the state-value function used in tic-tac-toe evaluates board positions *after* the agent has made its move. Let us call these *afterstates*, and value functions over these, *afterstate value functions*. Afterstates are useful when we have knowledge of an initial part of the environment's dynamics but not necessarily of the full dynamics. For example, in games we typically know the immediate effects of our moves. We know for each possible chess move what the resulting position will be, but not how our opponent will reply. Afterstate value functions are a natural way to take advantage of this kind of knowledge and thereby produce a more efficient learning method.

The reason it is more efficient to design algorithms in terms of afterstates is apparent from the tic-tac-toe example. A conventional action-value function would map from positions *and* moves to an estimate of the value. But many position–move pairs produce the same resulting position, as in this example:



In such cases the position–move pairs are different but produce the same "afterposition," and thus must have the same value. A conventional action-value function would have to separately assess both pairs, whereas an afterstate value function would immediately assess both equally. Any learning about the position–move pair on the left would immediately transfer to the pair on the right.

Afterstates arise in many tasks, not just games. For example, in queuing tasks there are actions such as assigning customers to servers, rejecting customers, or discarding information. In such cases the actions are in fact defined in terms of their immediate effects, which are completely known.

It is impossible to describe all the possible kinds of specialized problems and corresponding specialized learning algorithms. However, the principles developed in this book should apply widely. For example, afterstate methods are still aptly described in terms of generalized policy iteration, with a policy and (afterstate) value function interacting in essentially the same way. In many cases one will still face the choice between on-policy and off-policy methods for managing the need for persistent exploration.

Exercise 6.13 Describe how the task of Jack's Car Rental (Example 4.2) could be reformulated in terms of afterstates. Why, in terms of this specific task, would such a reformulation be likely to speed convergence? □

6.9 Summary

In this chapter we introduced a new kind of learning method, temporal-difference (TD) learning, and showed how it can be applied to the reinforcement learning problem. As usual, we divided the overall problem into a prediction problem and a control problem. TD methods are alternatives to Monte Carlo methods for solving the prediction problem. In both cases, the extension to the control problem is via the idea of generalized policy iteration (GPI) that we abstracted from dynamic programming. This is the idea that approximate policy and value functions should interact in such a way that they both move toward their optimal values.

One of the two processes making up GPI drives the value function to accurately predict returns for the current policy; this is the prediction problem. The other process drives the policy to improve locally (e.g., to be ε-greedy) with respect to the current value function. When the first process is based on experience, a complication arises concerning maintaining sufficient exploration. We can classify TD control methods according to whether they deal with this complication by using an on-policy or off-policy approach. Sarsa is an on-policy method, and Q-learning is an off-policy method. Expected Sarsa is also an off-policy method as we present it here. There is a third way in which TD methods can be extended to control which we did not include in this chapter, called actor–critic methods. These methods are covered in full in Chapter 13.

The methods presented in this chapter are today the most widely used reinforcement learning methods. This is probably due to their great simplicity: they can be applied on-line, with a minimal amount of computation, to experience generated from interaction with an environment; they can be expressed nearly completely by single equations that can be implemented with small computer programs. In the next few chapters we extend these algorithms, making them slightly more complicated and significantly more powerful. All the new algorithms will retain the essence of those introduced here: they will be able to process experience on-line, with relatively little computation, and they will be driven by TD errors. The special cases of TD methods introduced in the present chapter should rightly be called *one-step, tabular, model-free* TD methods. In the next two chapters we extend them to multistep forms (a link to Monte Carlo methods) and forms that include a model of the environment (a link to planning and dynamic programming). Then, in the second part of the book we extend them to various forms of function approximation rather than tables (a link to deep learning and artificial neural networks).

Finally, in this chapter we have discussed TD methods entirely within the context of reinforcement learning problems, but TD methods are actually more general than this. They are general methods for learning to make long-term predictions about dynamical systems. For example, TD methods may be relevant to predicting financial data, life spans, election outcomes, weather patterns, animal behavior, demands on power stations, or customer purchases. It was only when TD methods were analyzed as pure prediction methods, independent of their use in reinforcement learning, that their theoretical properties first came to be well understood. Even so, these other potential applications of TD learning methods have not yet been extensively explored.

Bibliographical and Historical Remarks

As we outlined in Chapter 1, the idea of TD learning has its early roots in animal learning psychology and artificial intelligence, most notably the work of Samuel (1959) and Klopf (1972). Samuel's work is described as a case study in Section 16.2. Also related to TD learning are Holland's (1975, 1976) early ideas about consistency among value predictions. These influenced one of the authors (Barto), who was a graduate student from 1970 to 1975 at the University of Michigan, where Holland was teaching. Holland's ideas led to a number of TD-related systems, including the work of Booker (1982) and the bucket brigade of Holland (1986), which is related to Sarsa as discussed below.

6.1–2 Most of the specific material from these sections is from Sutton (1988), including the TD(0) algorithm, the random walk example, and the term "temporal-difference learning." The characterization of the relationship to dynamic programming and Monte Carlo methods was influenced by Watkins (1989), Werbos (1987), and others. The use of update diagrams was new to the first edition of this book, wherein they were called "backup diagrams."

Tabular TD(0) was proved to converge in the mean by Sutton (1988) and with probability 1 by Dayan (1992), based on the work of Watkins and Dayan (1992). These results were extended and strengthened by Jaakkola, Jordan, and Singh (1994) and Tsitsiklis (1994) by using extensions of the powerful existing theory of stochastic approximation. Other extensions and generalizations are covered in later chapters.

6.3 The optimality of the TD algorithm under batch training was established by Sutton (1988). Illuminating this result is Barnard's (1993) derivation of the TD algorithm as a combination of one step of an incremental method for learning a model of the Markov chain and one step of a method for computing predictions from the model. The term *certainty equivalence* is from the adaptive control literature (e.g., Goodwin and Sin, 1984).

6.4 The Sarsa algorithm was introduced by Rummery and Niranjan (1994). They explored it in conjunction with neural networks and called it "Modified Connectionist Q-learning". The name "Sarsa" was introduced by Sutton (1996). The convergence of one-step tabular Sarsa (the form treated in this chapter) has been proved by Satinder Singh (personal communication). The "windy gridworld" example was suggested by Tom Kalt.

Holland's (1986) bucket brigade idea evolved into an algorithm closely related to Sarsa. The original idea of the bucket brigade involved chains of rules triggering each other; it focused on passing credit back from the current rule to the rules that triggered it. Over time, the bucket brigade came to be more like TD learning in passing credit back to any temporally preceding rule, not just to the ones that triggered the current rule. The modern form of the bucket brigade, when simplified in various natural ways, is nearly identical to one-step Sarsa, as detailed by Wilson (1994).

6.5 Q-learning was introduced by Watkins (1989), whose outline of a convergence proof was made rigorous by Watkins and Dayan (1992). More general convergence results were proved by Jaakkola, Jordan, and Singh (1994) and Tsitsiklis (1994).

6.6 Expected Sarsa was first described in an exercise in the first edition of this book, then fully investigated by van Seijen, van Hasselt, Whiteson, and Weiring (2009). They established its convergence properties and conditions under which it will outperform regular Sarsa and Q-learning. Our Figure 6.7 is adapted from their results. Our presentation differs slightly from theirs in that they define "Expected Sarsa" to be an on-policy method exclusively, whereas we use this name for the general algorithm in which the target and behavior policies are allowed to differ. The general off-policy view of Expected Sarsa was first noted by van Hasselt (2011), who called it "General Q-learning".

6.7 Maximization bias and double learning were introduced and extensively investigated by Hado van Hasselt (2010, 2011). The example MDP in Figure 6.8 was adapted from that in his Figure 4.1 (van Hasselt, 2011).

6.8 The notion of an afterstate is the same as that of a "post-decision state" (Van Roy, Bertsekas, Lee, and Tsitsiklis, 1997; Powell, 2010).

Chapter 7

n-step Bootstrapping

In this chapter we unify the Monte Carlo (MC) methods and the one-step temporal-difference (TD) methods presented in the previous two chapters. Neither MC methods nor one-step TD methods are always the best. In this chapter we present *n-step TD methods* that generalize both methods so that one can shift from one to the other smoothly as needed to meet the demands of a particular task. n-step methods span a spectrum with MC methods at one end and one-step TD methods at the other. The best methods are often intermediate between the two extremes.

Another way of looking at the benefits of n-step methods is that they free you from the tyranny of the time step. With one-step TD methods the same time step determines how often the action can be changed and the time interval over which bootstrapping is done. In many applications one wants to be able to update the action very fast to take into account anything that has changed, but bootstrapping works best if it is over a length of time in which a significant and recognizable state change has occurred. With one-step TD methods, these time intervals are the same, and so a compromise must be made. n-step methods enable bootstrapping to occur over multiple steps, freeing us from the tyranny of the single time step.

The idea of n-step methods is usually used as an introduction to the algorithmic idea of *eligibility traces* (Chapter 12), which enable bootstrapping over multiple time intervals simultaneously. Here we instead consider the n-step bootstrapping idea on its own, postponing the treatment of eligibility-trace mechanisms until later. This allows us to separate the issues better, dealing with as many of them as possible in the simpler n-step setting.

As usual, we first consider the prediction problem and then the control problem. That is, we first consider how n-step methods can help in predicting returns as a function of state for a fixed policy (i.e., in estimating v_π). Then we extend the ideas to action values and control methods.

7.1 n-step TD Prediction

What is the space of methods lying between Monte Carlo and TD methods? Consider estimating v_π from sample episodes generated using π. Monte Carlo methods perform an update for each state based on the entire sequence of observed rewards from that state until the end of the episode. The update of one-step TD methods, on the other hand, is based on just the one next reward, bootstrapping from the value of the state one step later as a proxy for the remaining rewards. One kind of intermediate method, then, would perform an update based on an intermediate number of rewards: more than one, but less than all of them until termination. For example, a two-step update would be based on the first two rewards and the estimated value of the state two steps later. Similarly, we could have three-step

updates, four-step updates, and so on. Figure 7.1 shows the update diagrams of the spectrum of *n-step updates* for v_π, with the one-step TD update on the left and the up-until-termination Monte Carlo update on the right.

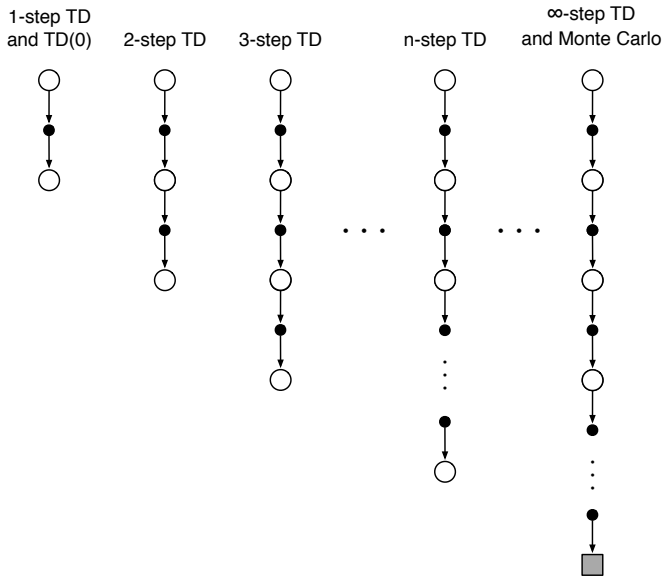


Figure 7.1: The update diagrams of n-step methods. These methods form a spectrum ranging from one-step TD methods to Monte Carlo methods.

The methods that use n-step updates are still TD methods because they still change an earlier estimate based on how it differs from a later estimate. Now the later estimate is not one step later, but n steps later. Methods in which the temporal difference extends over n steps are called *n-step TD methods*. The TD methods introduced in the previous chapter all used one-step updates, which is why we called them one-step TD methods.

More formally, consider the update of the estimated value of state S_t as a result of the state–reward sequence, $S_t, R_{t+1}, S_{t+1}, R_{t+2}, \ldots, R_T, S_T$ (omitting the actions). We know that in Monte Carlo updates the estimate of $v_\pi(S_t)$ is updated in the direction of the complete return:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T,$$

where T is the last time step of the episode. Let us call this quantity the *target* of the update. Whereas in Monte Carlo updates the target is the return, in one-step updates the target is the first reward plus the discounted estimated value of the next state, which we call the *one-step return*:

$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1}),$$

where $V_t : \mathcal{S} \to \mathbb{R}$ here is the estimate at time t of v_π. The subscripts on $G_{t:t+1}$ indicate that it is a truncated return for time t using rewards up until time $t+1$, with the discounted estimate $\gamma V_t(S_{t+1})$ taking the place of the other terms $\gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$ of the full return, as discussed in the previous chapter. Our point now is that this idea makes just as much sense after two steps as it does after one. The target for a two-step update is the *two-step return*:

$$G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$$

where now $\gamma^2 V_{t+1}(S_{t+2})$ corrects for the absence of the terms $\gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots + \gamma^{T-t-1} R_T$. Similarly, the target for an arbitrary n-step update is the *n-step return*:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}), \tag{7.1}$$

for all n, t such that $n \geq 1$ and $0 \leq t < T-n$. All n-step returns can be considered approximations to the full return, truncated after n steps and then corrected for the remaining missing terms by $V_{t+n-1}(S_{t+n})$. If $t+n \geq T$ (if the n-step return extends to or beyond termination), then all the missing terms are taken as zero, and the n-step return defined to be equal to the ordinary full return ($G_{t:t+n} \doteq G_t$ if $t+n \geq T$).

Note that n-step returns for $n > 1$ involve future rewards and states that are not available at the time of transition from t to $t+1$. No real algorithm can use the n-step return until after it has seen R_{t+n} and computed V_{t+n-1}. The first time these are available is $t+n$. The natural state-value learning algorithm for using n-step returns is thus

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha\big[G_{t:t+n} - V_{t+n-1}(S_t)\big], \qquad 0 \leq t < T, \tag{7.2}$$

while the values of all other states remain unchanged: $V_{t+n}(s) = V_{t+n-1}(s)$, for all $s \neq S_t$. We call this algorithm *n-step TD*. Note that no changes at all are made during the first $n-1$ steps of each episode. To make up for that, an equal number of additional updates are made at the end of the episode, after termination and before starting the next episode.

n-step TD for estimating $V \approx v_\pi$

Initialize $V(s)$ arbitrarily, $s \in \mathcal{S}$
Parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations (for S_t and R_t) can take their index mod n

Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | | Take an action according to $\pi(\cdot|S_t)$
  | | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | | If S_{t+1} is terminal, then $T \leftarrow t+1$
  | $\tau \leftarrow t-n+1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | | If $\tau+n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_{\tau:\tau+n})$
  | | $V(S_\tau) \leftarrow V(S_\tau) + \alpha\,[G - V(S_\tau)]$
  Until $\tau = T-1$

Exercise 7.1 In Chapter 6 we noted that the Monte Carlo error can be written as the sum of TD errors (6.6) if the value estimates don't change from step to step. Show that the n-step error used in (7.2) can also be written as a sum TD errors (again if the value estimates don't change) generalizing the earlier result. □

Exercise 7.2 (programming) With an n-step method, the value estimates *do* change from step to step, so an algorithm that used the sum of TD errors (see previous exercise) in place of the error in (7.2) would actually be a slightly different algorithm. Would it be a better algorithm or a worse one? Devise and program a small experiment to answer this question empirically. □

The n-step return uses the value function V_{t+n-1} to correct for the missing rewards beyond R_{t+n}. An important property of n-step returns is that their expectation is guaranteed to be a better estimate of v_π than V_{t+n-1} is, in a worst-state sense. That is, the worst error of the expected n-step return is

guaranteed to be less than or equal to γ^n times the worst error under V_{t+n-1}:

$$\max_s \left| \mathbb{E}_\pi[G_{t:t+n} | S_t = s] - v_\pi(s) \right| \;\leq\; \gamma^n \max_s \left| V_{t+n-1}(s) - v_\pi(s) \right|, \tag{7.3}$$

for all $n \geq 1$. This is called the *error reduction property* of n-step returns. Because of the error reduction property, one can show formally that all n-step TD methods converge to the correct predictions under appropriate technical conditions. The n-step TD methods thus form a family of sound methods, with one-step TD methods and Monte Carlo methods as extreme members.

Example 7.1: n-step TD Methods on the Random Walk Consider using n-step TD methods on the random walk task described in Example 6.2 and shown in Figure 6.2 (top). Suppose the first episode progressed directly from the center state, C, to the right, through D and E, and then terminated on the right with a return of 1. Recall that the estimated values of all the states started at an intermediate value, $V(s) = 0.5$. As a result of this experience, a one-step method would change only the estimate for the last state, $V(\mathsf{E})$, which would be incremented toward 1, the observed return. A two-step method, on the other hand, would increment the values of the two states preceding termination: $V(\mathsf{D})$ and $V(\mathsf{E})$ both would be incremented toward 1. A three-step method, or any n-step method for $n > 2$, would increment the values of all three of the visited states toward 1, all by the same amount.

Which value of n is better? Figure 7.2 shows the results of a simple empirical test for a larger random walk process, with 19 states instead of 5 (and with a -1 outcome on the left, all values initialized to 0), which we use as a running example in this chapter. Results are shown for n-step TD methods with a range of values for n and α. The performance measure for each parameter setting, shown on the vertical axis, is the square-root of the average squared error between the predictions at the end of the episode for the 19 states and their true values, then averaged over the first 10 episodes and 100 repetitions of the whole experiment (the same sets of walks were used for all parameter settings). Note that methods with an intermediate value of n worked best. This illustrates how the generalization of TD and Monte Carlo methods to n-step methods can potentially perform better than either of the two extreme methods.

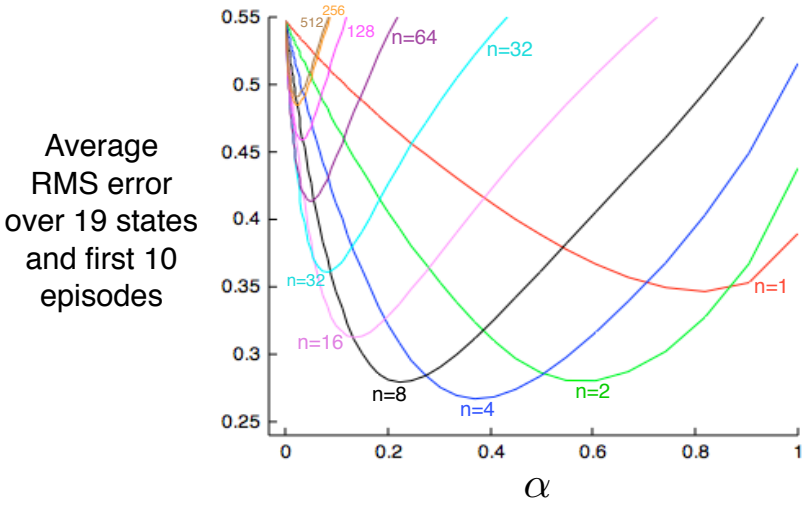


Figure 7.2: Performance of n-step TD methods as a function of α, for various values of n, on a 19-state random walk task (Example 7.1). ■

Exercise 7.3 Why do you think a larger random walk task (19 states instead of 5) was used in the examples of this chapter? Would a smaller walk have shifted the advantage to a different value of n? How about the change in left-side outcome from 0 to -1 made in the larger walk? Do you think that made any difference in the best value of n? □

7.2 n-step Sarsa

How can n-step methods be used not just for prediction, but for control? In this section we show how n-step methods can be combined with Sarsa in a straightforward way to produce an on-policy TD control method. The n-step version of Sarsa we call n-step Sarsa, and the original version presented in the previous chapter we henceforth call *one-step Sarsa*, or *Sarsa(0)*.

The main idea is to simply switch states for actions (state–action pairs) and then use an ε-greedy policy. The update diagrams for n-step Sarsa, shown in Figure 7.3 are like those of n-step TD (Figure 7.1), strings of alternating states and actions, except that the Sarsa ones all start and end with an action rather a state. We redefine n-step returns (update targets) in terms of estimated action values:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}), \quad n \geq 1, 0 \leq t < T - n, \quad (7.4)$$

with $G_{t:t+n} \doteq G_t$ if $t + n \geq T$. The natural algorithm is then

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right], \qquad 0 \leq t < T, \quad (7.5)$$

while the values of all other states remain unchanged: $Q_{t+n}(s, a) = Q_{t+n-1}(s, a)$, for all s, a such that $s \neq S_t$ or $a \neq A_t$. This is the algorithm we call *n-step Sarsa*. Pseudocode is shown in the box on the next page, and an example of why it can speed up learning compared to one-step methods is given in Figure 7.4.

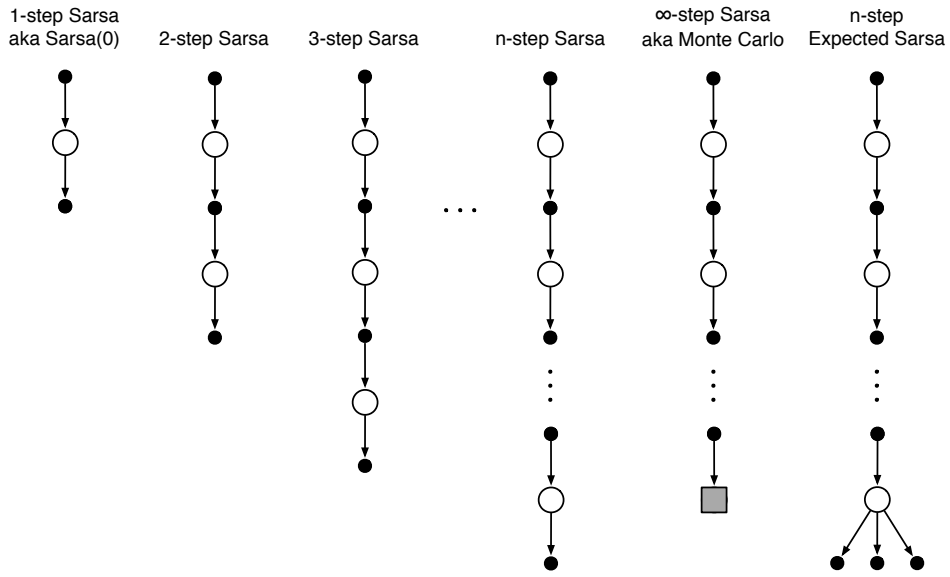


Figure 7.3: The update diagrams for the spectrum of n-step methods for state–action values. They range from the one-step update of Sarsa(0) to the up-until-termination update of the Monte Carlo method. In between are the n-step updates, based on n steps of real rewards and the estimated value of the nth next state–action pair, all appropriately discounted. On the far right is the update diagram for n-step Expected Sarsa.

n-step Sarsa for estimating $Q \approx q_*$, or $Q \approx q_\pi$ for a given π

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ε-greedy with respect to Q, or to a fixed given policy
Parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations (for S_t, A_t, and R_t) can take their index mod n

Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take action A_t
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then:
  | $T \leftarrow t+1$
  | else:
  | Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\,[G - Q(S_\tau, A_\tau)]$
  | If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε-greedy wrt Q
  Until $\tau = T - 1$

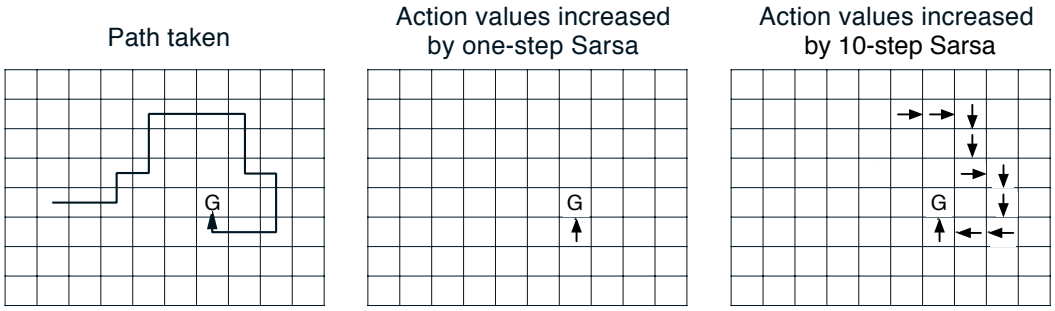


Figure 7.4: Gridworld example of the speedup of policy learning due to the use of n-step methods. The first panel shows the path taken by an agent in a single episode, ending at a location of high reward, marked by the G. In this example the values were all initially 0, and all rewards were zero except for a positive reward at G. The arrows in the other two panels show which action values were strengthened as a result of this path by one-step and n-step Sarsa methods. The one-step method strengthens only the last action of the sequence of actions that led to the high reward, whereas the n-step method strengthens the last n actions of the sequence, so that much more is learned from the one episode.

What about Expected Sarsa? The update diagram for the n-step version of Expected Sarsa is shown on the far right in Figure 7.3. It consists of a linear string of sample actions and states, just as in n-step Sarsa, except that its last element is a branch over all action possibilities weighted, as always, by their probability under π. This algorithm can be described by the same equation as n-step Sarsa (above)

except with the n-step return redefined as

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \sum_a \pi(a|S_{t+n}) Q_{t+n-1}(S_{t+n}, a), \tag{7.6}$$

for all n and t such that $n \geq 1$ and $0 \leq t \leq T - n$.

7.3 n-step Off-policy Learning by Importance Sampling

Recall that off-policy learning is learning the value function for one policy, π, while following another policy, b. Often, π is the greedy policy for the current action-value-function estimate, and b is a more exploratory policy, perhaps ε-greedy. In order to use the data from b we must take into account the difference between the two policies, using their relative probability of taking the actions that were taken (see Section 5.5). In n-step methods, returns are constructed over n steps, so we are interested in the relative probability of just those n actions. For example, to make a simple off-policy version of n-step TD, the update for time t (actually made at time $t+n$) can simply be weighted by $\rho_{t:t+n-1}$:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha \rho_{t:t+n-1} \left[G_{t:t+n} - V_{t+n-1}(S_t) \right], \quad 0 \leq t < T, \tag{7.7}$$

where $\rho_{t:t+n-1}$, called the *importance sampling ratio*, is the relative probability under the two policies of taking the n actions from A_t to A_{t+n-1} (cf. Eq. 5.3):

$$\rho_{t:h} \doteq \prod_{k=t}^{\min(h, T-1)} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \tag{7.8}$$

For example, if any one of the actions would never be taken by π (i.e., $\pi(A_k|S_k) = 0$) then the n-step return should be given zero weight and be totally ignored. On the other hand, if by chance an action is taken that π would take with much greater probability than b does, then this will increase the weight that would otherwise be given to the return. This makes sense because that action is characteristic of π (and therefore we want to learn about it) but is selected only rarely by b and thus rarely appears in the data. To make up for this we have to over-weight it when it does occur. Note that if the two policies are actually the same (the on-policy case) then the importance sampling ratio is always 1. Thus our new update (7.7) generalizes and can completely replace our earlier n-step TD update. Similarly, our previous n-step Sarsa update can be completely replaced by a simple off-policy form:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha \rho_{t+1:t+n-1} \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right], \tag{7.9}$$

for $0 \leq t < T$. Note that the importance sampling ratio here starts one step later than for n-step TD (above). This is because here we are updating a state–action pair. We do not have to care how likely we were to select the action; now that we have selected it we want to learn fully from what happens, with importance sampling only for subsequent actions. Pseudocode for the full algorithm is shown in the box on the next page.

The off-policy version of n-step Expected Sarsa would use the same update as above for n-step Sarsa except that the importance sampling ratio would have one less factor in it. That is, the above equation would use $\rho_{t+1:t+n-2}$ instead of $\rho_{t+1:t+n-1}$, and of course it would use the Expected Sarsa version of the n-step return (7.6). This is because in Expected Sarsa all possible actions are taken into account in the last state; the one actually taken has no effect and does not have to be corrected for.

Off-policy n-step Sarsa for estimating $Q \approx q_*$, or $Q \approx q_\pi$ for a given π

Input: an arbitrary behavior policy b such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ε-greedy with respect to Q, or as a fixed given policy
Parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations (for S_t, A_t, and R_t) can take their index mod n

Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim b(\cdot|S_0)$
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take action A_t
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then:
  | $T \leftarrow t+1$
  | else:
  | Select and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
  | If $\tau \geq 0$:
  | $\rho \leftarrow \prod_{i=\tau+1}^{\min(\tau+n-1, T-1)} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$ ($\rho_{\tau+1:t+n-1}$)
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ ($G_{\tau:\tau+n}$)
  | $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\rho[G - Q(S_\tau, A_\tau)]$
  | If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε-greedy wrt Q
  Until $\tau = T - 1$

7.4 *Per-reward Off-policy Methods

The multi-step off-policy methods presented in the previous section are very simple and conceptually clear, but are probably not the most efficient. A more sophisticated approach would use per-reward importance sampling ideas such as were introduced in Section 5.9. To understand this approach, first note that the ordinary n-step return (7.1), like all returns, can be written recursively:

$$G_{t:h} = R_{t+1} + \gamma G_{t+1:h}.$$

Now consider the effect of following a behavior policy $b \neq \pi$ that is not the same as the target policy π. All of the resulting experience, including the first reward R_{t+1} and the next state S_{t+1} must be weighted by the importance sampling ratio for time t, $\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$. One might be tempted to simply weight the righthand side of the above equation, but one can do better. Suppose the action at time t would never be selected by π, so that ρ_t is zero. Then a simple weighting would result in the n-step return being zero, which could result in high variance when it was used as a target. Instead, in this more sophisticated approach, one uses an alternate, *off-policy* definition of the n-step return, as

$$G_{t:h} \doteq \rho_t \left(R_{t+1} + \gamma G_{t+1:h}\right) + (1 - \rho_t) V_{h-1}(S_t), \qquad t < h \leq T, \tag{7.10}$$

where $G_{t:t} \doteq V_{t-1}(S_t)$. Now, if ρ_t is zero, instead of the target being zero and causing the estimate to shrink, the target is the same as the estimate and causes no change. The importance sampling ratio

being zero means we should ignore the sample, so leaving the estimate unchanged seems an appropriate outcome. Notice that the second, additional term does not change the expected update; the importance sampling ratio has expected value one (Section 5.9) and is uncorrelated with the estimate, so the expected value of the second term is zero. Also note that the off-policy definition (7.10) is a strict generalization of the earlier on-policy definition of the n-step return (7.1), as the two are identical in the on-policy case, in which ρ_t is always 1.

For a conventional n-step method, the learning rule to use in conjunction with (7.10) is the n-step TD update (7.2), which has no explicit importance sampling ratios other than those embedded in G.

Exercise 7.4 Write the pseudocode for the off-policy state-value prediction algorithm described above. □

For action values, the off-policy definition of the n-step return is a little different because the first action does not play a role in the importance sampling. We are learning the value of that action and it does not matter if it was unlikely or even impossible under the target policy. It has been taken and now full unit weight must be given to the reward and state that follows it. Importance sampling will apply only to the actions that follow it. The off-policy recursive definition of the n-step return for action values is

$$G_{t:h} \doteq R_{t+1} + \gamma \left(\rho_{t+1} G_{t+1:h} + (1 - \rho_{t+1})\bar{Q}_{t+1}\right), \quad t < h \leq T, \tag{7.11}$$

with $\bar{Q}_t \doteq \sum_a \pi(a|S_t) Q_{t-1}(S_t, a)$. A complete n-step off-policy action-value prediction algorithm would combine (7.11) and (7.5). If the recursion ends with $G_{t:t} \doteq Q(S_t, A_t)$, then the resultant algorithm is analogous to Sarsa, and if $G_{t:t} \doteq \bar{Q}_t$, then the resultant algorithm is analogous to Expected Sarsa.

Exercise 7.5 Write the pseudocode for the off-policy action-value prediction algorithm described immediately above. Specify both Sarsa and Expected Sarsa variations. □

Exercise 7.6 Show that the general (off-policy) version of the n-step return (7.10) can still be written exactly and compactly as the sum of state-based TD errors (6.5) if the approximate state value function does not change. □

Exercise 7.7 Repeat the above exercise for the action version of the off-policy n-step return (7.11) and the Expected Sarsa TD error (the quantity in brackets in Equation 6.9). □

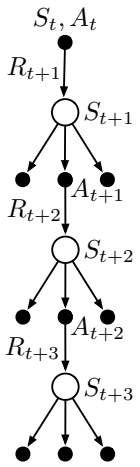
Exercise 7.8 (programming) Devise a small off-policy prediction problem and use it to show that the off-policy learning algorithm using (7.10) and (7.2) is more data efficient than the simpler algorithm using (7.1) and (7.7). □

The importance sampling that we have used in this section, the previous section, and in Chapter 5 enables off-policy learning, but at the cost of increasing the variance of the updates. The high variance forces us to use a small step-size parameter, resulting in slow learning. It is probably inevitable that off-policy training is slower than on-policy training—after all, the data is less relevant to what you are trying to learn. However, it is probably also true that the methods we have presented here can be improved on. One possibility is to rapidly adapt the step sizes to the observed variance, as in the Autostep method (Mahmood et al, 2012). Another promising approach is the invariant updates of Karampatziakis and Langford (2010) as extended to TD by Tian (in preparation). The usage technique of Mahmood (2017; Mahmood and Sutton, 2015) is probably also part of the solution. In the next section we consider an off-policy learning method that does not use importance sampling.

7.5 Off-policy Learning Without Importance Sampling: The n-step Tree Backup Algorithm

Is off-policy learning possible without importance sampling? Q-learning and Expected Sarsa from Chapter 6 do this for the one-step case, but is there a corresponding multi-step algorithm? In this section we present just such an n-step method, called the *tree-backup algorithm.*

The idea of the algorithm is suggested by the 3-step tree-backup update diagram shown to the right. Down the central spine and labeled in the diagram are three sample states and rewards, and two sample actions. These are the random variables representing the events occurring after the initial state–action pair S_t, A_t. Hanging off to the sides of each state are the actions that were *not* selected. (For the last state, all the actions are considered to have not (yet) been selected.) Because we have no sample data for the unselected actions, we bootstrap and use the estimates of their values in forming the target for the update. This slightly extends the idea of an update diagram. So far we have always updated the estimated value of the node at the top of the diagram toward a target combining the rewards along the way (appropriately discounted) and the estimated values of the nodes at the bottom. In the tree-backup update diagram, the target includes all these things *plus* the estimated values of the dangling action nodes hanging off the sides, at all levels. This is why it is called a *tree-backup* update; it is an update from the entire tree of of estimated action values.



the 3-step tree-backup update

More precisely, the update is from the estimated action values of the *leaf nodes* of the tree. The action nodes in the interior, corresponding to the actual actions taken, do not participate. Each leaf node contributes to the target with a weight proportional to its probability of occurring under the target policy π. Thus each first-level action a contributes with a weight of $\pi(a|S_{t+1})$, except that the action actually taken, A_{t+1}, does not contribute at all. Its probability, $\pi(A_{t+1}|S_{t+1})$, is used to weight all the second-level action values. Thus, each non-selected second-level action a' contributes with weight $\pi(A_{t+1}|S_{t+1})\pi(a'|S_{t+2})$. Each third-level action contributes with weight $\pi(A_{t+1}|S_{t+1})\pi(A_{t+2}|S_{t+2})\pi(a''|S_{t+3})$, and so on. It is as if each arrow to an action node in the diagram is weighted by the action's probability of being selected under the target policy and, if there is a tree below the action, then that weight applies to all the leaf nodes in the tree.

We can think of the 3-step tree-backup update as consisting of 6 half-steps, alternating between sample half-steps from an action to a subsequent state, and expected half-steps considering from that state all possible actions with their probabilities of occuring under the policy.

The one-step return (target) of the tree-backup algorithm is the same as that of Expected Sarsa. It can be written

$$\begin{aligned} G_{t:t+1} &\doteq R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q_t(S_{t+1}, a) \\ &= \delta'_t + Q_{t-1}(S_t, A_t), \end{aligned}$$

where δ'_t is a modified form of the TD error from Expected Sarsa:

$$\delta'_t \doteq R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q_t(S_{t+1}, a) - Q_{t-1}(S_t, A_t). \tag{7.12}$$

With these, the general n-step returns of the tree-backup algorithm can be defined recursively, and then as a sum of TD errors:

$$\begin{aligned} G_{t:t+n} &\doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_t(S_{t+1}, a) \;+\; \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n} \\ &= \delta'_t + Q_{t-1}(S_t, A_t) - \gamma \pi(A_{t+1}|S_{t+1}) Q_t(S_{t+1}, A_{t+1}) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n} \end{aligned} \tag{7.13}$$

$$
\begin{aligned}
&= Q_{t-1}(S_t, A_t) + \delta'_t + \gamma\pi(A_{t+1}|S_{t+1})\big(G_{t+1:t+n} - Q_t(S_{t+1}, A_{t+1})\big) \\
&= Q_{t-1}(S_t, A_t) + \delta'_t + \gamma\pi(A_{t+1}|S_{t+1})\delta'_{t+1} + \gamma^2\pi(A_{t+1}|S_{t+1})\pi(A_{t+2}|S_{t+2})\delta'_{t+2} + \cdots \\
&= Q_{t-1}(S_t, A_t) + \sum_{k=t}^{\min(t+n-1,T-1)} \delta'_k \prod_{i=t+1}^{k} \gamma\pi(A_i|S_i),
\end{aligned}
$$

under the usual convention that a degenerate product with no factors is 1. This target is then used with the usual action-value update rule from n-step Sarsa:

$$
Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha\left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)\right], \tag{7.5}
$$

while the values of all other state–action pairs remain unchanged: $Q_{t+n}(s, a) = Q_{t+n-1}(s, a)$, for all s, a such that $s \neq S_t$ or $a \neq A_t$. Pseudocode for this algorithm is shown in the box on the previous page.

n-step Tree Backup for estimating $Q \approx q_*$, or $Q \approx q_\pi$ for a given π

Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ε-greedy with respect to Q, or as a fixed given policy
Parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations can take their index mod n

Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$
  Store $Q(S_0, A_0)$ as Q_0
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
  | If $t < T$:
  | | Take action A_t
  | | Observe the next reward R; observe and store the next state as S_{t+1}
  | | If S_{t+1} is terminal:
  | | | $T \leftarrow t + 1$
  | | | Store $R - Q_t$ as δ_t
  | | else:
  | | | Store $R + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, a) - Q_t$ as δ_t
  | | | Select arbitrarily and store an action as A_{t+1}
  | | | Store $Q(S_{t+1}, A_{t+1})$ as Q_{t+1}
  | | | Store $\pi(A_{t+1}|S_{t+1})$ as π_{t+1}
  | $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
  | If $\tau \geq 0$:
  | | $Z \leftarrow 1$
  | | $G \leftarrow Q_\tau$
  | | For $k = \tau, \ldots, \min(\tau + n - 1, T - 1)$:
  | | | $G \leftarrow G + Z\delta_k$
  | | | $Z \leftarrow \gamma Z\pi_{k+1}$
  | | $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\left[G - Q(S_\tau, A_\tau)\right]$
  | | If π is being learned, then ensure that $\pi(a|S_\tau)$ is ε-greedy wrt $Q(S_\tau, \cdot)$
  Until $\tau = T - 1$

7.6 *A Unifying Algorithm: n-step $Q(\sigma)$

So far in this chapter we have considered three different kinds of action-value algorithms, corresponding to the first three update diagrams shown in Figure 7.5. n-step Sarsa has all sample transitions, the tree-backup algorithm has all state-to-action transitions fully branched without sampling, and n-step Expected Sarsa has all sample transitions except for the last state-to-action one, which is fully branched with an expected value. To what extent can these algorithms be unified?

One idea for unification is suggested by the fourth update diagram in Figure 7.5. This is the idea that one might decide on a step-by-step basis whether one wanted to take the action as a sample, as in Sarsa, or consider the expectation over all actions instead, as in the tree-backup update. Then, if one chose always to sample, one would obtain Sarsa, whereas if one chose never to sample, one would get the tree-backup algorithm. Expected Sarsa would be the case where one chose to sample for all steps except for the last one. And of course there would be many other possibilities, as suggested by the last diagram in the figure. To increase the possibilities even further we can consider a continuous variation between sampling and expectation. Let $\sigma_t \in [0, 1]$ denote the degree of sampling on step t, with $\sigma = 1$ denoting full sampling and $\sigma = 0$ denoting a pure expectation with no sampling. The random variable σ_t might be set as a function of the state, action, or state–action pair at time t. We call this proposed new algorithm n-step $Q(\sigma)$.

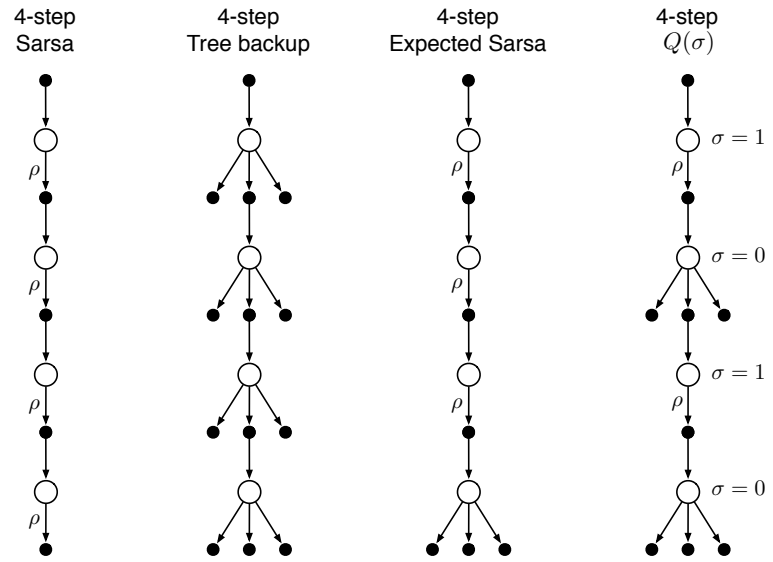


Figure 7.5: The update diagrams of the three kinds of n-step action-value updates considered so far in this chapter (4-step case) plus the update diagram of a fourth kind of update that unifies them all. The 'ρ's indicate half transitions on which importance sampling is required in the off-policy case. The fourth kind of update unifies all the others by choosing on a state-by-state basis whether to sample ($\sigma_t = 1$) or not ($\sigma_t = 0$).

Now let us develop the equations of n-step $Q(\sigma)$. First note that the n-step return of Sarsa (7.4) can be written in terms of its own pure-sample-based TD error:

$$G_{t:t+n} = Q_{t-1}(S_t, A_t) + \sum_{k=t}^{\min(t+n-1,T-1)} \gamma^{k-t} \left[R_{k+1} + \gamma Q_k(S_{k+1}, A_{k+1}) - Q_{k-1}(S_k, A_k)\right]$$

This suggests that we may be able to cover both cases if we generalize the TD error to slide with σ_t from its expectation to its sampling form:

$$\delta_t \doteq R_{t+1} + \gamma\big[\sigma_{t+1} Q_t(S_{t+1}, A_{t+1}) + (1-\sigma_{t+1})\bar{Q}_{t+1}\big] - Q_{t-1}(S_t, A_t), \tag{7.14}$$

with

$$\bar{Q}_t \doteq \sum_a \pi(a|S_t) Q_{t-1}(S_t, a), \tag{7.15}$$

as usual. Using these we can define the n-step returns of $Q(\sigma)$ as:

$$\begin{aligned}
G_{t:t+1} &\doteq R_{t+1} + \gamma\big[\sigma_{t+1} Q_t(S_{t+1}, A_{t+1}) + (1-\sigma_{t+1})\bar{Q}_{t+1}\big] \\
&= \delta_t + Q_{t-1}(S_t, A_t), \\
G_{t:t+2} &\doteq R_{t+1} + \gamma\big[\sigma_{t+1} Q_t(S_{t+1}, A_{t+1}) + (1-\sigma_{t+1})\bar{Q}_{t+1}\big] \\
&\quad - \gamma(1-\sigma_{t+1})\pi(A_{t+1}|S_{t+1}) Q_t(S_{t+1}, A_{t+1}) \\
&\quad + \gamma(1-\sigma_{t+1})\pi(A_{t+1}|S_{t+1})\big[R_{t+2} + \gamma\big[\sigma_{t+2} Q_t(S_{t+2}, A_{t+2}) + (1-\sigma_{t+2})\bar{Q}_{t+2}\big]\big] \\
&\quad - \gamma\sigma_{t+1} Q_t(S_{t+1}, A_{t+1}) \\
&\quad + \gamma\sigma_{t+1}\big[R_{t+2} + \gamma\big[\sigma_{t+2} Q_t(S_{t+2}, A_{t+2}) + (1-\sigma_{t+2})\bar{Q}_{t+2}\big]\big] \\
&= Q_{t-1}(S_t, A_t) + \delta_t \\
&\quad + \gamma(1-\sigma_{t+1})\pi(A_{t+1}|S_{t+1})\delta_{t+1} \\
&\quad + \gamma\sigma_{t+1}\delta_{t+1} \\
&= Q_{t-1}(S_t, A_t) + \delta_t + \gamma\big[(1-\sigma_{t+1})\pi(A_{t+1}|S_{t+1}) + \sigma_{t+1}\big]\delta_{t+1} \\
G_{t:t+n} &\doteq Q_{t-1}(S_t, A_t) + \sum_{k=t}^{\min(t+n-1,T-1)} \delta_k \prod_{i=t+1}^{k} \gamma\big[(1-\sigma_i)\pi(A_i|S_i) + \sigma_i\big].
\end{aligned} \tag{7.16}$$

Under on-policy training, this return is ready to be used in an update such as that for n-step Sarsa (7.5). For the off-policy case, we need to take σ into account in the importance sampling ratio, which we redefine more generally as

$$\rho_{t:h} \doteq \prod_{k=t}^{\min(h,T-1)} \left(\sigma_k \frac{\pi(A_k|S_k)}{b(A_k|S_k)} + 1 - \sigma_k\right). \tag{7.17}$$

After this we can then use the usual general (off-policy) update for n-step Sarsa (7.9). A complete algorithm is given in the box on the next page.

Off-policy n-step $Q(\sigma)$ for estimating $Q \approx q_*$, or $Q \approx q_\pi$ for a given π

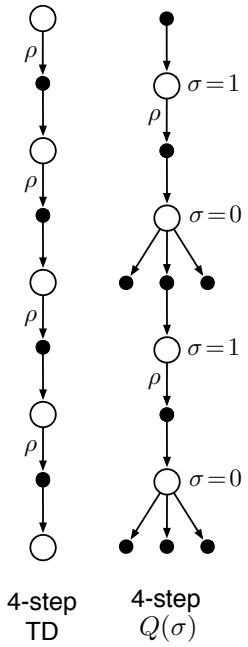
Input: an arbitrary behavior policy b such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ε-greedy with respect to Q, or as a fixed given policy
Parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations can take their index mod n

Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim b(\cdot|S_0)$
  Store $Q(S_0, A_0)$ as Q_0
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
  | If $t < T$:
  | | Take action A_t
  | | Observe the next reward R; observe and store the next state as S_{t+1}
  | | If S_{t+1} is terminal:
  | | | $T \leftarrow t + 1$
  | | | Store $\delta_t \leftarrow R - Q_t$
  | | else:
  | | | Select and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$
  | | | Select and store σ_{t+1}
  | | | Store $Q(S_{t+1}, A_{t+1})$ as Q_{t+1}
  | | | Store $R + \gamma\sigma_{t+1}Q_{t+1} + \gamma(1-\sigma_{t+1})\sum_a \pi(a|S_{t+1})Q(S_{t+1},a) - Q_t$ as δ_t
  | | | Store $\pi(A_{t+1}|S_{t+1})$ as π_{t+1}
  | | | Store $\frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})}$ as ρ_{t+1}
  | $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
  | If $\tau \geq 0$:
  | | $\rho \leftarrow 1$
  | | $Z \leftarrow 1$
  | | $G \leftarrow Q_\tau$
  | | For $k = \tau, \ldots, \min(\tau + n - 1, T - 1)$:
  | | | $G \leftarrow G + Z\delta_k$
  | | | $Z \leftarrow \gamma Z\big[(1-\sigma_{k+1})\pi_{k+1} + \sigma_{k+1}\big]$
  | | | $\rho \leftarrow \rho(1 - \sigma_k + \sigma_k\rho_k)$
  | | $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\rho\,[G - Q(S_\tau, A_\tau)]$
  | | If π is being learned, then ensure that $\pi(a|S_\tau)$ is ε-greedy wrt $Q(S_\tau, \cdot)$
  Until $\tau = T - 1$

7.7 Summary

In this chapter we have developed a range of temporal-difference learning methods that lie in between the one-step TD methods of the previous chapter and the Monte Carlo methods of the chapter before. Methods that involve an intermediate amount of bootstrapping are important because they will typically perform better than either extreme.

Our focus in this chapter has been on n-step methods, which look ahead to the next n rewards, states, and actions. The two 4-step update diagrams to the right together summarize most of the methods introduced. The state-value update shown is for n-step TD with importance sampling, and the action-value update is for n-step $Q(\sigma)$, which generalizes Expected Sarsa and Q-learning. All n-step methods involve a delay of n time steps before updating, as only then are all the required future events known. A further drawback is that they involve more computation per time step than previous methods. Compared to one-step methods, n-step methods also require more memory to record the states, actions, rewards, and sometimes other variables over the last n time steps. Eventually, in Chapter 12, we will see how multi-step TD methods can be implemented with minimal memory and computational complexity using eligibility traces, but there will always be some additional computation beyond one-step methods. Such costs can be well worth paying to escape the tyranny of the single time step.



Although n-step methods are more complex than those using eligibility traces, they have the great benefit of being conceptually clear. We have sought to take advantage of this by developing two approaches to off-policy learning in the n-step case. One, based on importance sampling is conceptually simple but can be of high variance. If the target and behavior policies are very different it probably needs some new algorithmic ideas before it can be efficient and practical. The other, based on tree-backup updates, is the natural extension of Q-learning to the multi-step case with stochastic target policies. It involves no importance sampling but, again if the target and behavior policies are substantially different, the bootstrapping may span only a few steps even if n is large.

Bibliographical and Historical Remarks

7.1–2 The notion of n-step returns is due to Watkins (1989), who also first discussed their error reduction property. n-step algorithms were explored in the first edition of this book, in which they were treated as of conceptual interest, but not feasible in practice. The work of Cichosz (1995) and particularly van Seijen (2016) showed that they are actually completely practical algorithms. Given this, and their conceptual clarity and simplicity, we have chosen to highlight them here in the second edition. In particular, we now postpone all discussion of the backward view and of eligibility traces until Chapter 12.

The results in the random walk examples were made for this text based on work of Sutton (1988) and Singh and Sutton (1996). The use of update diagrams to describe these and other algorithms in this chapter is new.

7.3–5 The developments in these sections are based on the work of Precup, Sutton, and Singh (2000), Precup, Sutton, and Dasgupta (2001), and Sutton, Mahmood, Precup, and van Hasselt (2014).

The tree-backup algorithm is due to Precup, Sutton, and Singh (2000), but the presentation of it here is new.

7.6 The $Q(\sigma)$ algorithm is new to this text, but has been explored further by De Asis, Hernandez-Garcia, Holland, and Sutton (2017).

Chapter 8

Planning and Learning with Tabular Methods

In this chapter we develop a unified view of reinforcement learning methods that require a model of the environment, such as dynamic programming and heuristic search, and methods that can be used without a model, such as Monte Carlo and temporal-difference methods. These are respectively called *model-based* and *model-free* reinforcement learning methods. Model-based methods rely on *planning* as their primary component, while model-free methods primarily rely on *learning*. Although there are real differences between these two kinds of methods, there are also great similarities. In particular, the heart of both kinds of methods is the computation of value functions. Moreover, all the methods are based on looking ahead to future events, computing a backed-up value, and then using it as an update target for an approximate value function. Earlier in this book we presented Monte Carlo and temporal-difference methods as distinct alternatives, then showed how they can be unified by n-step methods. Our goal in this chapter is a similar integration of model-based and model-free methods. Having established these as distinct in earlier chapters, we now explore the extent to which they can be intermixed.

8.1 Models and Planning

By a *model* of the environment we mean anything that an agent can use to predict how the environment will respond to its actions. Given a state and an action, a model produces a prediction of the resultant next state and next reward. If the model is stochastic, then there are several possible next states and next rewards, each with some probability of occurring. Some models produce a description of all possibilities and their probabilities; these we call *distribution models*. Other models produce just one of the possibilities, sampled according to the probabilities; these we call *sample models*. For example, consider modeling the sum of a dozen dice. A distribution model would produce all possible sums and their probabilities of occurring, whereas a sample model would produce an individual sum drawn according to this probability distribution. The kind of model assumed in dynamic programming—estimates of the MDP's dynamics, $p(s', r \mid s, a)$—is a distribution model. The kind of model used in the blackjack example in Chapter 5 is a sample model. Distribution models are stronger than sample models in that they can always be used to produce samples. However, in many applications it is much easier to obtain sample models than distribution models. The dozen dice are a simple example of this. It would be easy to write a computer program to simulate the dice rolls and return the sum, but harder and more error-prone to figure out all the possible sums and their probabilities.

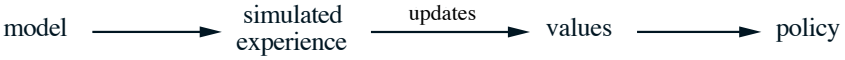
Models can be used to mimic or simulate experience. Given a starting state and action, a sample

model produces a possible transition, and a distribution model generates all possible transitions weighted by their probabilities of occurring. Given a starting state and a policy, a sample model could produce an entire episode, and a distribution model could generate all possible episodes and their probabilities. In either case, we say the model is used to *simulate* the environment and produce *simulated experience*.

The word *planning* is used in several different ways in different fields. We use the term to refer to any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment:

In artificial intelligence, there are two distinct approaches to planning according to our definition. *State-space planning*, which includes the approach we take in this book, is viewed primarily as a search through the state space for an optimal policy or an optimal path to a goal. Actions cause transitions from state to state, and value functions are computed over states. In what we call *plan-space planning*, planning is instead a search through the space of plans. Operators transform one plan into another, and value functions, if any, are defined over the space of plans. Plan-space planning includes evolutionary methods and "partial-order planning," a common kind of planning in artificial intelligence in which the ordering of steps is not completely determined at all stages of planning. Plan-space methods are difficult to apply efficiently to the stochastic sequential decision problems that are the focus in reinforcement learning, and we do not consider them further (but see, e.g., Russell and Norvig, 2010).

The unified view we present in this chapter is that all state-space planning methods share a common structure, a structure that is also present in the learning methods presented in this book. It takes the rest of the chapter to develop this view, but there are two basic ideas: (1) all state-space planning methods involve computing value functions as a key intermediate step toward improving the policy, and (2) they compute value functions by update operations applied to simulated experience. This common structure can be diagrammed as follows:



Dynamic programming methods clearly fit this structure: they make sweeps through the space of states, generating for each state the distribution of possible transitions. Each distribution is then used to compute a backed-up value (update target) and update the state's estimated value. In this chapter we argue that various other state-space planning methods also fit this structure, with individual methods differing only in the kinds of updates they do, the order in which they do them, and in how long the backed-up information is retained.

Viewing planning methods in this way emphasizes their relationship to the learning methods that we have described in this book. The heart of both learning and planning methods is the estimation of value functions by backing-up update operations. The difference is that whereas planning uses simulated experience generated by a model, learning methods use real experience generated by the environment. Of course this difference leads to a number of other differences, for example, in how performance is assessed and in how flexibly experience can be generated. But the common structure means that many ideas and algorithms can be transferred between planning and learning. In particular, in many cases a learning algorithm can be substituted for the key update step of a planning method. Learning methods require only experience as input, and in many cases they can be applied to simulated experience just as well as to real experience. The box below shows a simple example of a planning method based on one-step tabular Q-learning and on random samples from a sample model. This method, which we call *random-sample one-step tabular Q-planning*, converges to the optimal policy for the model under the same conditions that one-step tabular Q-learning converges to the optimal policy for the real environment (each state–action pair must be selected an infinite number of times in Step 1, and α must decrease appropriately over time).

Random-sample one-step tabular Q-planning

Do forever:

1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(s)$, at random
2. Send S, A to a sample model, and obtain
   a sample next reward, R, and a sample next state, S'
3. Apply one-step tabular Q-learning to S, A, R, S':
   $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

In addition to the unified view of planning and learning methods, a second theme in this chapter is the benefits of planning in small, incremental steps. This enables planning to be interrupted or redirected at any time with little wasted computation, which appears to be a key requirement for efficiently intermixing planning with acting and with learning of the model. Planning in very small steps may be the most efficient approach even on pure planning problems if the problem is too large to be solved exactly.

8.2 Dyna: Integrating Planning, Acting, and Learning

When planning is done on-line, while interacting with the environment, a number of interesting issues arise. New information gained from the interaction may change the model and thereby interact with planning. It may be desirable to customize the planning process in some way to the states or decisions currently under consideration, or expected in the near future. If decision making and model learning are both computation-intensive processes, then the available computational resources may need to be divided between them. To begin exploring these issues, in this section we present Dyna-Q, a simple architecture integrating the major functions needed in an on-line planning agent. Each function appears in Dyna-Q in a simple, almost trivial, form. In subsequent sections we elaborate some of the alternate ways of achieving each function and the trade-offs between them. For now, we seek merely to illustrate the ideas and stimulate your intuition.

Within a planning agent, there are at least two roles for real experience: it can be used to improve the model (to make it more accurately match the real environment) and it can be used to directly improve the value function and policy using the kinds of reinforcement learning methods we have discussed in previous chapters. The former we call *model-learning*, and the latter we call *direct reinforcement learning* (direct RL). The possible relationships between experience, model, values, and policy are summarized in Figure 8.1. Each arrow shows a relationship of influence and presumed improvement. Note how experience can improve value functions and policies either directly or indirectly via the model. It is the latter, which is sometimes called *indirect reinforcement learning*, that is involved in planning.

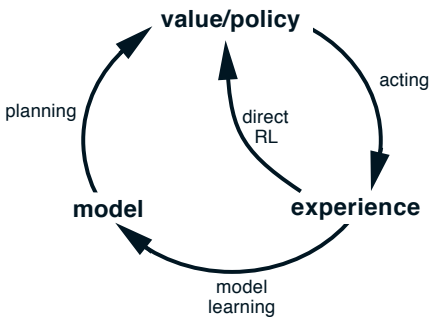


Figure 8.1: Relationships among learning, planning, and acting.

Both direct and indirect methods have advantages and disadvantages. Indirect methods often make fuller use of a limited amount of experience and thus achieve a better policy with fewer environmental interactions. On the other hand, direct methods are much simpler and are not affected by biases in the design of the model. Some have argued that indirect methods are always superior to direct ones, while others have argued that direct methods are responsible for most human and animal learning. Related debates in psychology and artificial intelligence concern the relative importance of cognition as opposed to trial-and-error learning, and of deliberative planning as opposed to reactive decision making (see Chapter 14 for discussion of some of these issues from the perspective of psychology). Our view is that the contrast between the alternatives in all these debates has been exaggerated, that more insight can be gained by recognizing the similarities between these two sides than by opposing them. For example, in this book we have emphasized the deep similarities between dynamic programming and temporal-difference methods, even though one was designed for planning and the other for model-free learning.

Dyna-Q includes all of the processes shown in Figure 8.1—planning, acting, model-learning, and direct RL—all occurring continuously. The planning method is the random-sample one-step tabular Q-planning method given in Figure 8.1. The direct RL method is one-step tabular Q-learning. The model-learning method is also table-based and assumes the environment is deterministic. After each transition $S_t, A_t \rightarrow R_{t+1}, S_{t+1}$, the model records in its table entry for S_t, A_t the prediction that R_{t+1}, S_{t+1} will deterministically follow. Thus, if the model is queried with a state–action pair that has been experienced before, it simply returns the last-observed next state and next reward as its prediction. During planning, the Q-planning algorithm randomly samples only from state–action pairs that have previously been experienced (in Step 1), so the model is never queried with a pair about which it has no information.

The overall architecture of Dyna agents, of which the Dyna-Q algorithm is one example, is shown in Figure 8.2. The central column represents the basic interaction between agent and environment, giving rise to a trajectory of real experience. The arrow on the left of the figure represents direct reinforcement learning operating on real experience to improve the value function and the policy. On the right are model-based processes. The model is learned from real experience and gives rise to simulated experience. We use the term *search control* to refer to the process that selects the starting states and actions for the

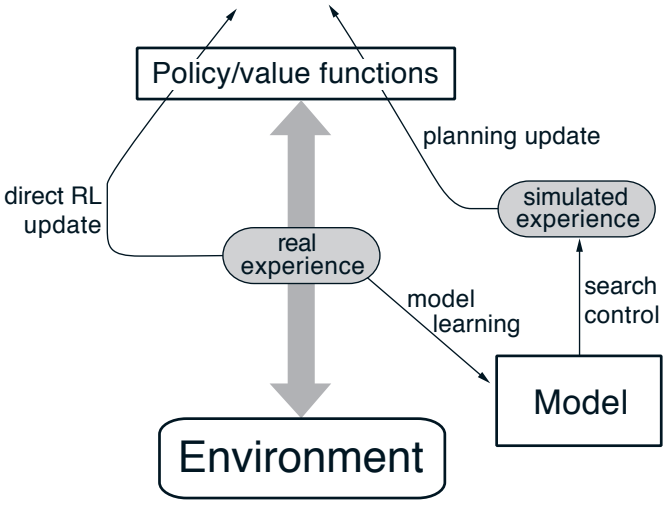


Figure 8.2: The general Dyna Architecture. Real experience, passing back and forth between the environment and the policy, affects policy and value functions in much the same way as does simulated experience generated by the model of the environment.

simulated experiences generated by the model. Finally, planning is achieved by applying reinforcement learning methods to the simulated experiences just as if they had really happened. Typically, as in Dyna-Q, the same reinforcement learning method is used both for learning from real experience and for planning from simulated experience. The reinforcement learning method is thus the "final common path" for both learning and planning. Learning and planning are deeply integrated in the sense that they share almost all the same machinery, differing only in the source of their experience.

Conceptually, planning, acting, model-learning, and direct RL occur simultaneously and in parallel in Dyna agents. For concreteness and implementation on a serial computer, however, we fully specify the order in which they occur within a time step. In Dyna-Q, the acting, model-learning, and direct RL processes require little computation, and we assume they consume just a fraction of the time. The remaining time in each step can be devoted to the planning process, which is inherently computation-intensive. Let us assume that there is time in each step, after acting, model-learning, and direct RL, to complete n iterations (Steps 1–3) of the Q-planning algorithm. In the pseudocode algorithm for Dyna-Q in the box below, $Model(s, a)$ denotes the contents of the (predicted next state and reward) for state–action pair (s, a). Direct reinforcement learning, model-learning, and planning are implemented by steps (d), (e), and (f), respectively. If (e) and (f) were omitted, the remaining algorithm would be one-step tabular Q-learning.

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \epsilon$-greedy(S,Q)
- (c) Execute action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
- (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Repeat n times:
  - $S \leftarrow$ random previously observed state
  - $A \leftarrow$ random action previously taken in S
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Example 8.1: Dyna Maze Consider the simple maze shown inset in Figure 8.3. In each of the 47 states there are four actions, `up`, `down`, `right`, and `left`, which take the agent deterministically to the corresponding neighboring states, except when movement is blocked by an obstacle or the edge of the maze, in which case the agent remains where it is. Reward is zero on all transitions, except those into the goal state, on which it is +1. After reaching the goal state (G), the agent returns to the start state (S) to begin a new episode. This is a discounted, episodic task with $\gamma = 0.95$.

The main part of Figure 8.3 shows average learning curves from an experiment in which Dyna-Q agents were applied to the maze task. The initial action values were zero, the step-size parameter was $\alpha = 0.1$, and the exploration parameter was $\epsilon = 0.1$. When selecting greedily among actions, ties were broken randomly. The agents varied in the number of planning steps, n, they performed per real step. For each n, the curves show the number of steps taken by the agent to reach the goal in each episode, averaged over 30 repetitions of the experiment. In each repetition, the initial seed for the random number generator was held constant across algorithms. Because of this, the first episode was exactly the same (about 1700 steps) for all values of n, and its data are not shown in the figure. After the first episode, performance improved for all values of n, but much more rapidly for larger values. Recall that the $n = 0$ agent is a nonplanning agent, using only direct reinforcement learning (one-step tabular Q-learning). This was by far the slowest agent on this problem, despite the fact that the parameter values (α and ε) were optimized for it. The nonplanning agent took about 25 episodes to reach (ε-)optimal

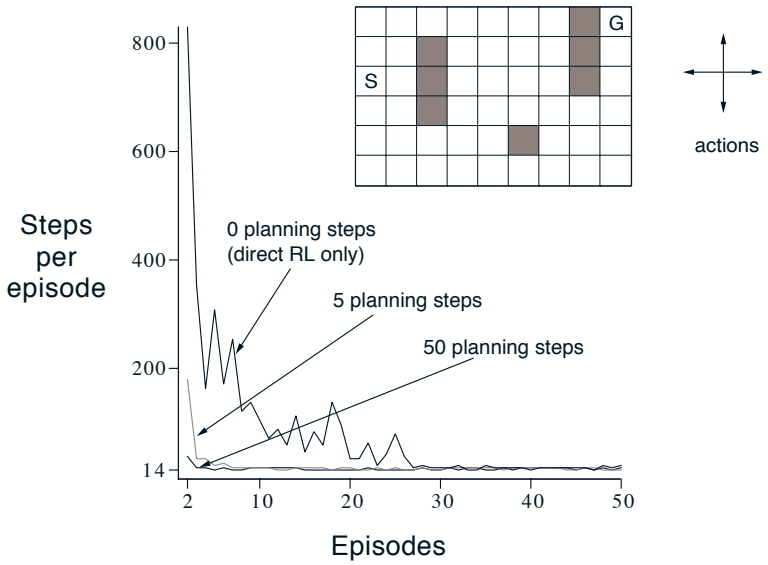


Figure 8.3: A simple maze (inset) and the average learning curves for Dyna-Q agents varying in their number of planning steps (n) per real step. The task is to travel from S to G as quickly as possible.

performance, whereas the $n = 5$ agent took about five episodes, and the $n = 50$ agent took only three episodes.

Figure 8.4 shows why the planning agents found the solution so much faster than the nonplanning agent. Shown are the policies found by the $n = 0$ and $n = 50$ agents halfway through the second episode. Without planning ($n = 0$), each episode adds only one additional step to the policy, and so only one step (the last) has been learned so far. With planning, again only one step is learned during the first episode, but here during the second episode an extensive policy has been developed that by the episode's end will reach almost back to the start state. This policy is built by the planning process while the agent is still wandering near the start state. By the end of the third episode a complete optimal policy will have been found and perfect performance attained.

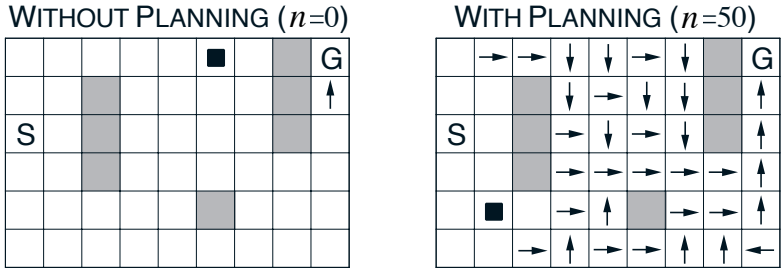


Figure 8.4: Policies found by planning and nonplanning Dyna-Q agents halfway through the second episode. The arrows indicate the greedy action in each state; if no arrow is shown for a state, then all of its action values were equal. The black square indicates the location of the agent. ■

In Dyna-Q, learning and planning are accomplished by exactly the same algorithm, operating on real experience for learning and on simulated experience for planning. Because planning proceeds incrementally, it is trivial to intermix planning and acting. Both proceed as fast as they can. The agent is always reactive and always deliberative, responding instantly to the latest sensory information and

yet always planning in the background. Also ongoing in the background is the model-learning process. As new information is gained, the model is updated to better match reality. As the model changes, the ongoing planning process will gradually compute a different way of behaving to match the new model.

Exercise 8.1 The nonplanning method looks particularly poor in Figure 8.4 because it is a one-step method; a method using multi-step bootstrapping would do better. Do you think one of the multi-step bootstrapping methods from Chapter 7 could do as well as the Dyna method? Explain why or why not.

8.3 When the Model Is Wrong

In the maze example presented in the previous section, the changes in the model were relatively modest. The model started out empty, and was then filled only with exactly correct information. In general, we cannot expect to be so fortunate. Models may be incorrect because the environment is stochastic and only a limited number of samples have been observed, or because the model was learned using function approximation that has generalized imperfectly, or simply because the environment has changed and its new behavior has not yet been observed. When the model is incorrect, the planning process is likely to compute a suboptimal policy.

In some cases, the suboptimal policy computed by planning quickly leads to the discovery and correction of the modeling error. This tends to happen when the model is optimistic in the sense of predicting greater reward or better state transitions than are actually possible. The planned policy attempts to exploit these opportunities and in doing so discovers that they do not exist.

Example 8.2: Blocking Maze A maze example illustrating this relatively minor kind of modeling error and recovery from it is shown in Figure 8.5. Initially, there is a short path from start to goal, to the right of the barrier, as shown in the upper left of the figure. After 1000 time steps, the short path is "blocked," and a longer path is opened up along the left-hand side of the barrier, as shown in upper right of the figure. The graph shows average cumulative reward for a Dyna-Q agent and an enhanced Dyna-Q+ agent to be described shortly. The first part of the graph shows that both Dyna agents found the short path within 1000 steps. When the environment changed, the graphs become flat, indicating a period during which the agents obtained no reward because they were wandering around behind the barrier. After a while, however, they were able to find the new opening and the new optimal behavior.

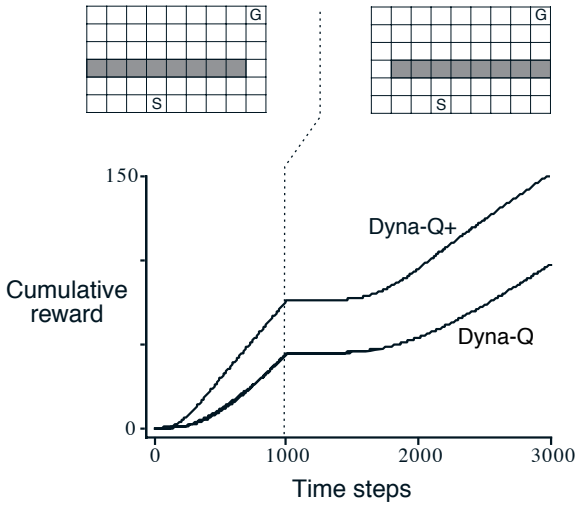


Figure 8.5: Average performance of Dyna agents on a blocking task. The left environment was used for the first 1000 steps, the right environment for the rest. Dyna-Q+ is Dyna-Q with an exploration bonus that encourages exploration. ■

Greater difficulties arise when the environment changes to become *better* than it was before, and yet the formerly correct policy does not reveal the improvement. In these cases the modeling error may not be detected for a long time, if ever, as we see in the next example.

Example 8.3: Shortcut Maze The problem caused by this kind of environmental change is illustrated by the maze example shown in Figure 8.6. Initially, the optimal path is to go around the left side of the barrier (upper left). After 3000 steps, however, a shorter path is opened up along the right side, without disturbing the longer path (upper right). The graph shows that the regular Dyna-Q agent never switched to the shortcut. In fact, it never realized that it existed. Its model said that there was no shortcut, so the more it planned, the less likely it was to step to the right and discover it. Even with an ε-greedy policy, it is very unlikely that an agent will take so many exploratory actions as to discover the shortcut.

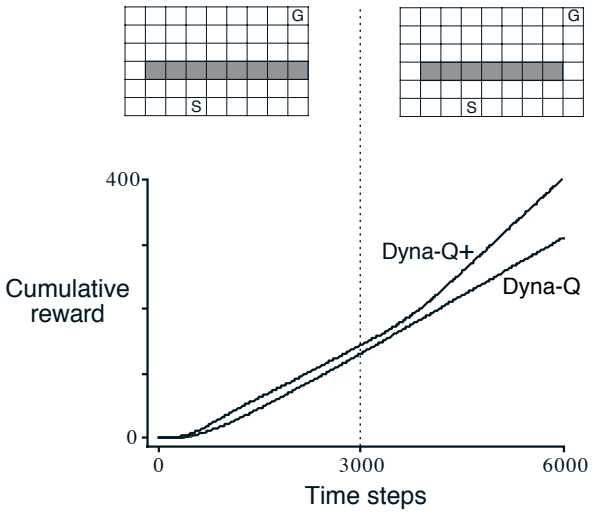


Figure 8.6: Average performance of Dyna agents on a shortcut task. The left environment was used for the first 3000 steps, the right environment for the rest. ■

The general problem here is another version of the conflict between exploration and exploitation. In a planning context, exploration means trying actions that improve the model, whereas exploitation means behaving in the optimal way given the current model. We want the agent to explore to find changes in the environment, but not so much that performance is greatly degraded. As in the earlier exploration/exploitation conflict, there probably is no solution that is both perfect and practical, but simple heuristics are often effective.

The Dyna-Q+ agent that did solve the shortcut maze uses one such heuristic. This agent keeps track for each state–action pair of how many time steps have elapsed since the pair was last tried in a real interaction with the environment. The more time that has elapsed, the greater (we might presume) the chance that the dynamics of this pair has changed and that the model of it is incorrect. To encourage behavior that tests long-untried actions, a special "bonus reward" is given on simulated experiences involving these actions. In particular, if the modeled reward for a transition is r, and the transition has not been tried in τ time steps, then planning updates are done as if that transition produced a reward of $r + \kappa\sqrt{\tau}$, for some small κ. This encourages the agent to keep testing all accessible state transitions and even to find long sequences of actions in order to carry out such tests.[1] Of course all this testing

[1]The Dyna-Q+ agent was changed in two other ways as well. First, actions that had never been tried before from a state were allowed to be considered in the planning step (f) of the Tabular Dyna-Q algorithm in the box above. Second, the initial model for such actions was that they would lead back to the same state with a reward of zero.

has its cost, but in many cases, as in the shortcut maze, this kind of computational curiosity is well worth the extra exploration.

Exercise 8.2 Why did the Dyna agent with exploration bonus, Dyna-Q+, perform better in the first phase as well as in the second phase of the blocking and shortcut experiments?

Exercise 8.3 Careful inspection of Figure 8.6 reveals that the difference between Dyna-Q+ and Dyna-Q narrowed slightly over the first part of the experiment. What is the reason for this?

Exercise 8.4 (programming) The exploration bonus described above actually changes the estimated values of states and actions. Is this necessary? Suppose the bonus $\kappa\sqrt{\tau}$ was used not in updates, but solely in action selection. That is, suppose the action selected was always that for which $Q(S_t, a) + \kappa\sqrt{\tau(S_t, a)}$ was maximal. Carry out a gridworld experiment that tests and illustrates the strengths and weaknesses of this alternate approach.

8.4 Prioritized Sweeping

In the Dyna agents presented in the preceding sections, simulated transitions are started in state–action pairs selected uniformly at random from all previously experienced pairs. But a uniform selection is usually not the best; planning can be much more efficient if simulated transitions and updates are focused on particular state–action pairs. For example, consider what happens during the second episode of the first maze task (Figure 8.4). At the beginning of the second episode, only the state–action pair leading directly into the goal has a positive value; the values of all other pairs are still zero. This means that it is pointless to perform updates along almost all transitions, because they take the agent from one zero-valued state to another, and thus the updates would have no effect. Only an update along a transition into the state just prior to the goal, or from it, will change any values. If simulated transitions are generated uniformly, then many wasteful updates will be made before stumbling onto one of these useful ones. As planning progresses, the region of useful updates grows, but planning is still far less efficient than it would be if focused where it would do the most good. In the much larger problems that are our real objective, the number of states is so large that an unfocused search would be extremely inefficient.

This example suggests that search might be usefully focused by working *backward* from goal states. Of course, we do not really want to use any methods specific to the idea of "goal state." We want methods that work for general reward functions. Goal states are just a special case, convenient for stimulating intuition. In general, we want to work back not just from goal states but from any state whose value has changed. Suppose that the values are initially correct given the model, as they were in the maze example prior to discovering the goal. Suppose now that the agent discovers a change in the environment and changes its estimated value of one state, either up or down. Typically, this will imply that the values of many other states should also be changed, but the only useful one-step updates are those of actions that lead directly into the one state whose value has been changed. If the values of these actions are updated, then the values of the predecessor states may change in turn. If so, then actions leading into them need to be updated, and then *their* predecessor states may have changed. In this way one can work backward from arbitrary states that have changed in value, either performing useful updates or terminating the propagation. This general idea might be termed *backward focusing* of planning computations.

As the frontier of useful updates propagates backward, it often grows rapidly, producing many state–action pairs that could usefully be updated. But not all of these will be equally useful. The values of some states may have changed a lot, whereas others may have changed little. The predecessor pairs of those that have changed a lot are more likely to also change a lot. In a stochastic environment, variations in estimated transition probabilities also contribute to variations in the sizes of changes and in the urgency with which pairs need to be updated. It is natural to prioritize the updates

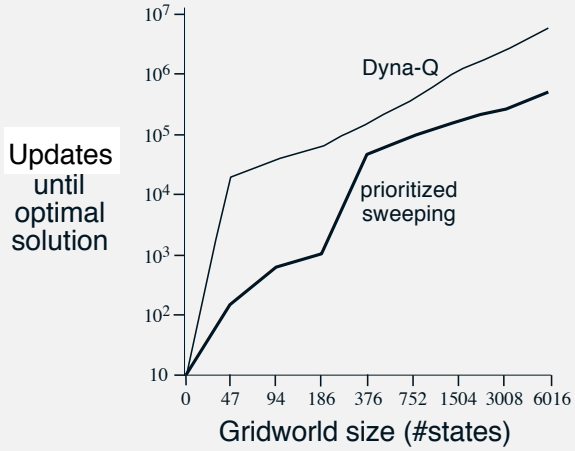
according to a measure of their urgency, and perform them in order of priority. This is the idea behind *prioritized sweeping*. A queue is maintained of every state–action pair whose estimated value would change nontrivially if updated , prioritized by the size of the change. When the top pair in the queue is updated, the effect on each of its predecessor pairs is computed. If the effect is greater than some small threshold, then the pair is inserted in the queue with the new priority (if there is a previous entry of the pair in the queue, then insertion results in only the higher priority entry remaining in the queue). In this way the effects of changes are efficiently propagated backward until quiescence. The full algorithm for the case of deterministic environments is given in the box.

Prioritized sweeping for a deterministic environment

Initialize $Q(s,a)$, $Model(s,a)$, for all s,a, and $PQueue$ to empty
Do forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow policy(S,Q)$
(c) Execute action A; observe resultant reward, R, and state, S'
(d) $Model(S,A) \leftarrow R, S'$
(e) $P \leftarrow |R + \gamma \max_a Q(S',a) - Q(S,A)|$.
(f) if $P > \theta$, then insert S,A into $PQueue$ with priority P
(g) Repeat n times, while $PQueue$ is not empty:
$S,A \leftarrow first(PQueue)$
$R,S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
Repeat, for all $\bar{S},\bar{A}$ predicted to lead to S:
$\bar{R} \leftarrow$ predicted reward for $\bar{S},\bar{A},S$
$P \leftarrow |\bar{R} + \gamma \max_a Q(S,a) - Q(\bar{S},\bar{A})|$.
if $P > \theta$ then insert $\bar{S},\bar{A}$ into $PQueue$ with priority P

Example 8.4 Prioritized Sweeping on Mazes

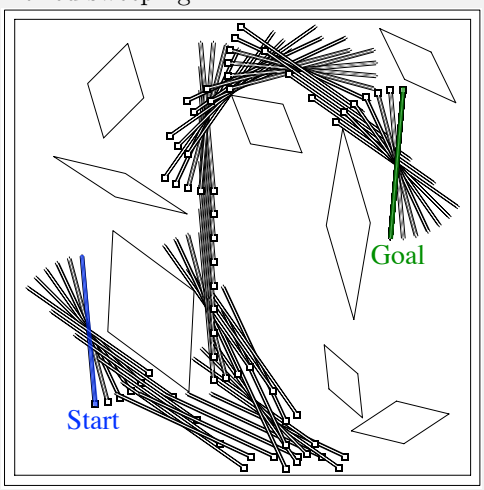
Prioritized sweeping has been found to dramatically increase the speed at which optimal solutions are found in maze tasks, often by a factor of 5 to 10. A typical example is shown below.



These data are for a sequence of maze tasks of exactly the same structure as the one shown in Figure 8.3, except that they vary in the grid resolution. Prioritized sweeping maintained a decisive advantage over unprioritized Dyna-Q. Both systems made at most $n = 5$ updates per environmental interaction. ■

Example 8.5 Rod Maneuvering

The objective in this task is to maneuver a rod around some awkwardly placed obstacles within a limited rectangular work space to a goal position in the fewest number of steps. The rod can be translated along its long axis or perpendicular to that axis, or it can be rotated in either direction around its center. The distance of each movement is approximately 1/20 of the work space, and the rotation increment is 10 degrees. Translations are deterministic and quantized to one of 20×20 positions. The figure below shows the obstacles and the shortest solution from start to goal, found by prioritized sweeping.



This problem is deterministic, but has four actions and 14,400 potential states (some of these are unreachable because of the obstacles). This problem is probably too large to be solved with unprioritized methods. Figure reprinted from Moore and Atkeson (1993). ■

Extensions of prioritized sweeping to stochastic environments are straightforward. The model is maintained by keeping counts of the number of times each state–action pair has been experienced and of what the next states were. It is natural then to update each pair not with a sample update, as we have been using so far, but with an expected update, taking into account all possible next states and their probabilities of occurring.

Prioritized sweeping is just one way of distributing computations to improve planning efficiency, and probably not the best way. One of prioritized sweeping's limitations is that it uses *expected* updates, which in stochastic environments may waste lots of computation on low-probability transitions. As we show in the following section, sample updates can in many cases get closer to the true value function with less computation despite the variance introduced by sampling. Sample updates can win because they break the overall backing-up computation into smaller pieces—those corresponding to individual transitions—which then enables it to be focused more narrowly on the pieces that will have the largest impact. This idea was taken to what may be its logical limit in the "small backups" introduced by van Seijen and Sutton (2013). These are updates along a single transition, like a sample update, but based on the probability of the transition without sampling, as in an expected update. By selecting the order in which small updates are done it is possible to greatly improve planning efficiency beyond that possible with prioritized sweeping.

We have suggested in this chapter that all kinds of state-space planning can be viewed as sequences of value updates, varying only in the type of update, expected or sample, large or small, and in the order in which the updates are done. In this section we have emphasized backward focusing, but this

is just one strategy. For example, another would be to focus on states according to how easily they can be reached from the states that are visited frequently under the current policy, which might be called *forward focusing*. Peng and Williams (1993) and Barto, Bradtke and Singh (1995) have explored versions of forward focusing, and the methods introduced in the next few sections take it to an extreme form.

8.5 Expected vs. Sample Updates

The examples in the previous sections give some idea of the range of possibilities for combining methods of learning and planning. In the rest of this chapter, we analyze some of the component ideas involved, starting with the relative advantages of expected and sample updates.

Much of this book has been about different kinds of value-function updates, and we have considered a great many varieties. Focusing for the moment on one-step updates, they vary primarily along three binary dimensions. The first two dimensions are whether they update state values or action values and whether they estimate the value for the optimal policy or for an arbitrary given policy. These two dimensions give rise to four classes of updates for approximating the four value functions, q_*, v_*, q_π, and

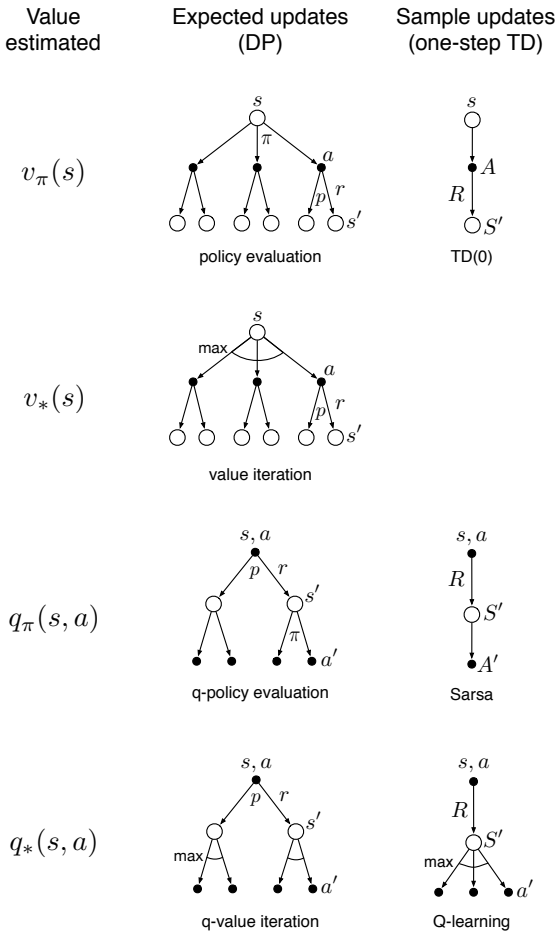


Figure 8.7: Diagrams of all the one-step updates considered in this book.

v_π. The other binary dimension is whether the updates are *expected* updates, considering all possible events that might happen, or *sample* updates, considering a single sample of what might happen. These three binary dimensions give rise to eight cases, seven of which correspond to specific algorithms, as shown in Figure 8.7. (The eighth case does not seem to correspond to any useful update.) Any of these one-step updates can be used in planning methods. The Dyna-Q agents discussed earlier use q_* sample updates, but they could just as well use q_* expected updates, or either expected or sample q_π updates. The Dyna-AC system uses v_π sample updates together with a learning policy structure. For stochastic problems, prioritized sweeping is always done using one of the expected updates.

When we introduced one-step sample updates in Chapter 6, we presented them as substitutes for expected updates. In the absence of a distribution model, expected updates are not possible, but sample updates can be done using sample transitions from the environment or a sample model. Implicit in that point of view is that expected updates, if possible, are preferable to sample updates. But are they? Expected updates certainly yield a better estimate because they are uncorrupted by sampling error, but they also require more computation, and computation is often the limiting resource in planning. To properly assess the relative merits of expected and sample updates for planning we must control for their different computational requirements.

For concreteness, consider the expected and sample updates for approximating q_*, and the special case of discrete states and actions, a table-lookup representation of the approximate value function, Q, and a model in the form of estimated dynamics, $\hat{p}(s', r \,|\, s, a)$. The expected update for a state–action pair, s, a, is:

$$Q(s,a) \leftarrow \sum_{s',r} \hat{p}(s', r \,|\, s, a) \Big[r + \gamma \max_{a'} Q(s', a') \Big]. \tag{8.1}$$

The corresponding sample update for s, a, given a sample next state and reward, S' and R (from the model), is the Q-learning-like update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \Big[R + \gamma \max_{a'} Q(S', a') - Q(s,a) \Big], \tag{8.2}$$

where α is the usual positive step-size parameter.

The difference between these expected and sample updates is significant to the extent that the environment is stochastic, specifically, to the extent that, given a state and action, many possible next states may occur with various probabilities. If only one next state is possible, then the expected and sample updates given above are identical (taking $\alpha = 1$). If there are many possible next states, then there may be significant differences. In favor of the expected update is that it is an exact computation, resulting in a new $Q(s, a)$ whose correctness is limited only by the correctness of the $Q(s', a')$ at successor states. The sample update is in addition affected by sampling error. On the other hand, the sample update is cheaper computationally because it considers only one next state, not all possible next states. In practice, the computation required by update operations is usually dominated by the number of state–action pairs at which Q is evaluated. For a particular starting pair, s, a, let b be the *branching factor* (i.e., the number of possible next states, s', for which $\hat{p}(s' \,|\, s, a) > 0$). Then an expected update of this pair requires roughly b times as much computation as a sample update.

If there is enough time to complete an expected update, then the resulting estimate is generally better than that of b sample updates because of the absence of sampling error. But if there is insufficient time to complete a expected update, then sample updates are always preferable because they at least make some improvement in the value estimate with fewer than b updates. In a large problem with many state–action pairs, we are often in the latter situation. With so many state–action pairs, expected updates of all of them would take a very long time. Before that we may be much better off with a few sample updates at many state–action pairs than with expected updates at a few pairs. Given a unit of computational effort, is it better devoted to a few expected updates or to b times as many sample updates?

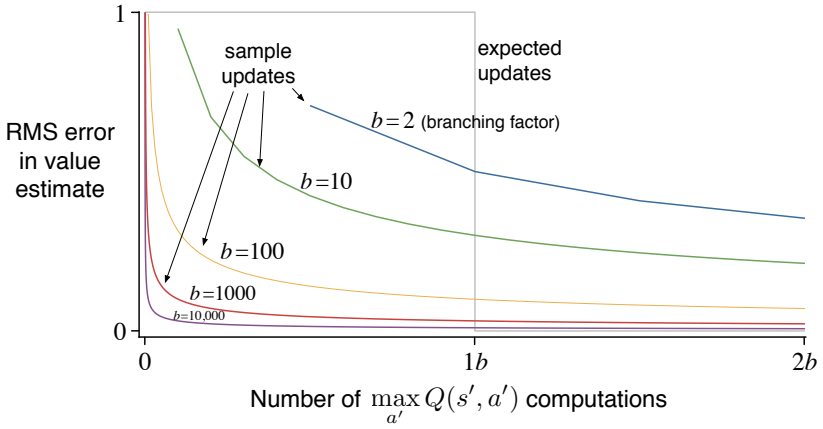


Figure 8.8: Comparison of efficiency of expected and sample updates.

Figure 8.8 shows the results of an analysis that suggests an answer to this question. It shows the estimation error as a function of computation time for expected and sample updates for a variety of branching factors, b. The case considered is that in which all b successor states are equally likely and in which the error in the initial estimate is 1. The values at the next states are assumed correct, so the expected update reduces the error to zero upon its completion. In this case, sample updates reduce the error according to $\sqrt{\frac{b-1}{bt}}$ where t is the number of sample updates that have been performed (assuming sample averages, i.e., $\alpha = 1/t$). The key observation is that for moderately large b the error falls dramatically with a tiny fraction of b updates. For these cases, many state–action pairs could have their values improved dramatically, to within a few percent of the effect of an expected update, in the same time that a single state–action pair could undergo an expected update.

The advantage of sample updates shown in Figure 8.8 is probably an underestimate of the real effect. In a real problem, the values of the successor states would be estimates that are themselves updated. By causing estimates to be more accurate sooner, sample updates will have a second advantage in that the values backed up from the successor states will be more accurate. These results suggest that sample updates are likely to be superior to expected updates on problems with large stochastic branching factors and too many states to be solved exactly.

8.6 Trajectory Sampling

In this section we compare two ways of distributing updates. The classical approach, from dynamic programming, is to perform sweeps through the entire state (or state–action) space, updating each state (or state–action pair) once per sweep. This is problematic on large tasks because there may not be time to complete even one sweep. In many tasks the vast majority of the states are irrelevant because they are visited only under very poor policies or with very low probability. Exhaustive sweeps implicitly devote equal time to all parts of the state space rather than focusing where it is needed. As we discussed in Chapter 4, exhaustive sweeps and the equal treatment of all states that they imply are not necessary properties of dynamic programming. In principle, updates can be distributed any way one likes (to assure convergence, all states or state–action pairs must be visited in the limit an infinite number of times; although an exception to this is discussed in Section 8.7 below), but in practice exhaustive sweeps are often used.

The second approach is to sample from the state or state–action space according to some distribution. One could sample uniformly, as in the Dyna-Q agent, but this would suffer from some of the same problems as exhaustive sweeps. More appealing is to distribute updates according to the on-policy

distribution, that is, according to the distribution observed when following the current policy. One advantage of this distribution is that it is easily generated; one simply interacts with the model, following the current policy. In an episodic task, one starts in a start state (or according to the starting-state distribution) and simulates until the terminal state. In a continuing task, one starts anywhere and just keeps simulating. In either case, sample state transitions and rewards are given by the model, and sample actions are given by the current policy. In other words, one simulates explicit individual trajectories and performs updates at the state or state–action pairs encountered along the way. We call this way of generating experience and updates *trajectory sampling*.

It is hard to imagine any efficient way of distributing updates according to the on-policy distribution other than by trajectory sampling. If one had an explicit representation of the on-policy distribution, then one could sweep through all states, weighting the update of each according to the on-policy distribution, but this leaves us again with all the computational costs of exhaustive sweeps. Possibly one could sample and update individual state–action pairs from the distribution, but even if this could be done efficiently, what benefit would this provide over simulating trajectories? Even knowing the on-policy distribution in an explicit form is unlikely. The distribution changes whenever the policy changes, and computing the distribution requires computation comparable to a complete policy evaluation. Consideration of such other possibilities makes trajectory sampling seem both efficient and elegant.

Is the on-policy distribution of updates a good one? Intuitively it seems like a good choice, at least better than the uniform distribution. For example, if you are learning to play chess, you study positions that might arise in real games, not random positions of chess pieces. The latter may be valid states, but to be able to accurately value them is a different skill from evaluating positions in real games. We will also see in Chapter 9 that the on-policy distribution has significant advantages when function approximation is used. Whether or not function approximation is used, one might expect on-policy focusing to significantly improve the speed of planning.

Focusing on the on-policy distribution could be beneficial because it causes vast, uninteresting parts of the space to be ignored, or it could be detrimental because it causes the same old parts of the space to be updated over and over. We conducted a small experiment to assess the effect empirically. To isolate the effect of the update distribution, we used entirely one-step expected tabular updates, as defined by (8.1). In the *uniform* case, we cycled through all state–action pairs, updating each in place, and in the *on-policy* case we simulated episodes, all starting in the same state, updating each state–action pair that occurred under the current ϵ-greedy policy ($\epsilon = 0.1$). The tasks were undiscounted episodic tasks, generated randomly as follows. From each of the $|\mathcal{S}|$ states, two actions were possible, each of which resulted in one of b next states, all equally likely, with a different random selection of b states for each state–action pair. The branching factor, b, was the same for all state–action pairs. In addition, on all transitions there was a 0.1 probability of transition to the terminal state, ending the episode. We used episodic tasks to get a clear measure of the quality of the current policy. At any point in the planning process one can stop and exhaustively compute $v_{\tilde{\pi}}(s_0)$, the true value of the start state under the greedy policy, $\tilde{\pi}$, given the current action-value function Q, as an indication of how well the agent would do on a new episode on which it acted greedily (all the while assuming the model is correct).

The upper part of Figure 8.9 shows results averaged over 200 sample tasks with 1000 states and branching factors of 1, 3, and 10. The quality of the policies found is plotted as a function of the number of expected updates completed. In all cases, sampling according to the on-policy distribution resulted in faster planning initially and retarded planning in the long run. The effect was stronger, and the initial period of faster planning was longer, at smaller branching factors. In other experiments, we found that these effects also became stronger as the number of states increased. For example, the lower part of Figure 8.9 shows results for a branching factor of 1 for tasks with 10,000 states. In this case the advantage of on-policy focusing is large and long-lasting.

All of these results make sense. In the short term, sampling according to the on-policy distribution

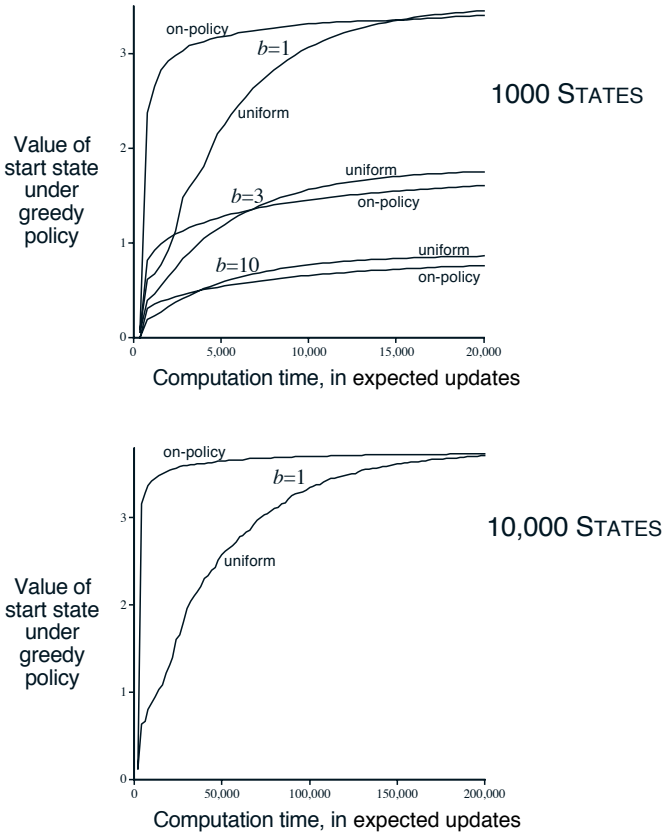


Figure 8.9: Relative efficiency of updates distributed uniformly across the state space versus focused on simulated on-policy trajectories, each starting in the same state. Results are for randomly generated tasks of two sizes and various branching factors, b.

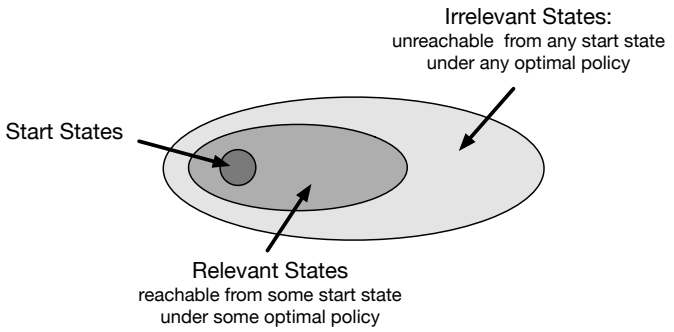
helps by focusing on states that are near descendants of the start state. If there are many states and a small branching factor, this effect will be large and long-lasting. In the long run, focusing on the on-policy distribution may hurt because the commonly occurring states all already have their correct values. Sampling them is useless, whereas sampling other states may actually perform some useful work. This presumably is why the exhaustive, unfocused approach does better in the long run, at least for small problems. These results are not conclusive because they are only for problems generated in a particular, random way, but they do suggest that sampling according to the on-policy distribution can be a great advantage for large problems, in particular for problems in which a small subset of the state–action space is visited under the on-policy distribution.

8.7 Real-time Dynamic Programming

Real-time dynamic programming, or RTDP, is an on-policy trajectory-sampling version of DP's value-iteration algorithm. Because it is closely related to conventional sweep-based policy iteration, RTDP illustrates in a particularly clear way some of the advantages that on-policy trajectory sampling can provide. RTDP updates the values of states visited in actual or simulated trajectories by means of expected tabular value-iteration updates as defined by (4.10). It is basically the algorithm that produced the on-policy results shown in Figure 8.9.

The close connection between RTDP and conventional DP makes it possible to derive some theoretical results by adapting existing theory. RTDP is an example of an *asynchronous* DP algorithm as described in Section 4.5. Asynchronous DP algorithms are not organized in terms of systematic sweeps of the state set; they update state values in any order whatsoever, using whatever values of other states happen to be available. In RTDP, the update order is dictated by the order states are visited in real or simulated trajectories.

If trajectories can start only from a designated set of start states, and if you are interested in the prediction problem for a given policy, then on-policy trajectory sampling allows the algorithm to completely skip states that cannot be reached by the given policy from any of the start states: unreachable states are irrelevant to the prediction problem. For a control problem, where the goal is to find an optimal policy instead of evaluating a given policy, there might well be states that cannot be reached by any optimal policy from any of the start states, and there is no need to specify optimal actions for these irrelevant states. What is needed is an *optimal partial policy*, meaning a policy that is optimal for the relevant states but can specify arbitrary actions, or even be undefined, for the irrelevant states (see the illustration below).



But *finding* such an optimal partial policy with an on-policy trajectory-sampling control method, such as Sarsa (Section 6.4), in general requires visiting all state–action pairs—even those that will turn out to be irrelevant—an infinite number of times. This can be done, for example, by using exploring starts (Section 5.3). This is true for RTDP as well: for episodic tasks with exploring starts, RTDP is an asynchronous value-iteration algorithm that converges to optimal polices for discounted finite MDPs (and for the undiscounted case under certain conditions). Unlike the situation for a prediction problem, it is generally not possible to stop updating any state or state–action pair if convergence to an optimal policy is important.

The most interesting result for RTDP is that for certain types of problems satisfying reasonable conditions, RTDP is guaranteed to find a policy that is optimal on the relevant states without visiting every state infinitely often, or even without visiting some states at all. Indeed, in some problems, only a small fraction of the states need to be visited. This can be a great advantage for problems with very large state sets, where even a single sweep may not be feasible.

The tasks for which this result holds are undiscounted episodic tasks for MDPs with absorbing goal states that generate zero rewards, as described in Section 3.4. At every step of a real or simulated trajectory, RTDP selects a greedy action (breaking ties randomly) and applies the expected value-iteration update operation to the current state. It can also update the values of an arbitrary collection of other states at each step; for example, it can update the values of states visited in a limited-horizon look-ahead search from the current state.

For these problems, with each episode beginning in a state randomly chosen from the set of start states, and ending at a goal state, RTDP converges (with probability one) to a policy that is optimal for all the relevant states[2] provided the following conditions are satisfied: 1) the initial value of every goal state is zero, 2) there exists at least one policy that guarantees that a goal state will be reached with probability one from any start state, 3) all rewards for transitions from non-goal states are strictly negative, and 4) all the initial values are equal to, or greater than, their optimal values (which can be

[2]This policy might be stochastic because RTDP continues to randomly select among all the greedy actions.

satisfied by simply setting the initial values of all states to zero). This result was proved by Barto, Bradtke, and Singh (1995) by combining results for asynchronous DP with results about a heuristic search algorithm known as *learning real-time A** due to Korf (1990).

Tasks having these properties are examples of *stochastic optimal path problems*, which are usually stated in terms of cost minimization instead as reward maximization, as we do here. Maximizing the negative returns in our version is equivalent to minimizing the costs of paths from a start state to a goal state. Examples of this kind of task are minimum-time control tasks, where each time step required to reach a goal produces a reward of -1, or problems like the Golf example in Section 3.5, whose objective is to hit the hole with the fewest strokes.

Example 8.6: RTDP on the Racetrack The racetrack problem of Exercise 5.7 (page 91) is a stochastic optimal path problem. Comparing RTDP and the conventional DP value iteration algorithm on an example racetrack problem illustrates some of the advantages of on-policy trajectory sampling.

Recall from the exercise that an agent has to learn how to drive a car around a turn like those shown in Figure 5.5 and cross the finish line as quickly as possible while staying on the track. Start states are all the zero-speed states on the starting line; the goal states are all the states that can be reached in one time step by crossing the finish line from inside the track. Unlike Exercise 5.7, here there is no limit on the car's speed, so the state set is potentially infinite. However, the set of states that can be reached from the set of start states via any policy is finite and can be considered to be the state set of the problem. Each episode begins in a randomly selected start state and ends when the car crosses the finish line. The rewards are -1 for each step until the car crosses the finish line. If the car hits the track boundary, it is moved back to a random start state, and the episode continues.

A racetrack similar to the small racetrack on the left of Figure 5.5 has 9,115 states reachable from start states by any policy, only 599 of which are relevant, meaning that they are reachable from some start state via some optimal policy. (The number of relevant states was estimated by counting the states visited while executing optimal actions for 10^7 episodes.)

The table below compares solving this task by conventional DP and by RTDP. These results are averages over 25 runs, each begun with a different random number seed. Conventional DP in this case is value iteration using exhaustive sweeps of the state set, with values updated one state at a time in place, meaning that the update for each state uses the most recent values of the other states (This is the Gauss-Seidel version of value iteration, which was found to be approximately twice as fast as the Jacobi version on this problem. See Section 4.8.) No special attention was paid to the ordering of the updates; other orderings could have produced faster convergence. Initial values were all zero for each run of both methods. DP was judged to have converged when the maximum change in a state value over a sweep was less than 10^{-4}, and RTDP was judged to have converged when the average time to cross the finish line over 20 episodes appeared to stabilize at an asymptotic number of steps. This version of RTDP updated only the value of the current state on each step.

	DP	RTDP
Average computation to convergence	28 sweeps	4000 episodes
Average number of updates to convergence	252,784	127,600
Average number of updates per episode	—	31.9
% of states updated $\leq$ 100 times	—	98.45
% of states updated $\leq$ 10 times	—	80.51
% of states updated 0 times	—	3.18

Both methods produced policies averaging between 14 and 15 steps to cross the finish line, but RTDP required only roughly half of the updates that DP did. This is the result of RTDP's on-policy trajectory sampling. Whereas the value of every state was updated in each sweep of DP, RTDP focused updates on fewer states. In an average run, RTDP updated the values of 98.45% of the states no more than 100 times and 80.51% of the states no more than 10 times; the values of about 290 states were not updated at all in an average run. ■

Another advantage of RTDP is that as the value function approaches the optimal value function v_*, the policy used by the agent to generate trajectories approaches an optimal policy because it is always greedy with respect to the current value function. This is in contrast to the situation in conventional value iteration. In practice, value iteration terminates when the value function changes by only a small amount in a sweep, which is how we terminated it to obain the results in the table above. At this point, the value function closely approximates v_*, and a greedy policy is close to an optimal policy. However, it is possible that policies that are greedy with respect to the latest value function were optimal, or nearly so, long before value iteration terminates. (Recall from Chapter 4 that optimal policies can be greedy with respect to many different value functions, not just v_*.) Checking for the emergence of an optimal policy before value iteration converges is not a part of the conventional DP algorithm and requires considerable additional computation.

In the racetrack example, by running many test episodes after each DP sweep, with actions selected greedily according to the result of that sweep, it was possible to estimate the earliest point in the DP computation at which the approximated optimal evaluation function was good enough so that the corresponding greedy policy was nearly optimal. For this racetrack, a close-to-optimal policy emerged after 15 sweeps of value iteration, or after 136,725 value-iteration updates. This is considerably less than the 252,784 updates DP needed to converge to v_*, but sill more than the 127,600 updates RTDP required.

Although these simulations are certainly not definitive comparisons of the RTDP with conventional sweep-based value iteration, they illustrate some of advantages of on-policy trajectory sampling. Whereas conventional value iteration continued to update the value of all the states, RTDP strongly focused on subsets of the states that were relevant to the problem's objective. This focus became increasingly narrow as learning continued. Because the convergence theorem for RTDP applies to the simulations, we know that RTDP eventually would have focused only on relevant states, i.e., on states making up optimal paths. RTDP achieved nearly optimal control with about 50% of the computation required by sweep-based value iteration.

8.8 Planning at Decision Time

Planning can be used in at least two ways. The one we have considered so far in this chapter, typified by dynamic programming and Dyna, is to use planning to gradually improve a policy or value function on the basis of simulated experience obtained from a model (either a sample or a distribution model). Selecting actions is then a matter of comparing the current state's action values obtained from a table in the tabular case we have thus far considered, or by evaluating a mathematical expression in the approximate methods we consider in Part II below. Well before an action is selected for any current state S_t, planning has played a part in improving the table entries, or the mathematical expression, needed to select the action for many states, including S_t. Used this way, planning is not focussed on the current state. We call planning used in this way *background planning*.

The other way to use planning is to begin and complete it *after* encountering each new state S_t, as a computation whose output is the selection of a single action A_t; on the next step planning begins anew with S_{t+1} to produce A_{t+1}, and so on. The simplest, and almost degenerate, example of this use of planning is when only state values are available, and an action is selected by comparing the values of model-predicted next states for each action (or by comparing the values of afterstates as in the tic-tac-toe example in Chapter 1). More generally, planning used in this way can look much deeper than one-step-ahead and evaluate action choices leading to many different predicted state and reward trajectories. Unlike the first use of planning, here planning focusses on a particular state. We call this *decision-time planning*.

These two ways of thinking about planning—using simulated experience to gradually improve a policy or value function, or using simulated experience to select an action for the current state—can blend

together in natural and interesting ways, but they have tended to be studied separately, and that is a good way to first understand them. Let us now take a closer look at decision-time planning.

Even when planning is only done at decision time, we can still view it, as we did in Section 8.1, as proceeding from simulated experience to updates and values, and ultimately to a policy. It is just that now the values and policy are specific to the current state and the action choices available there, so much so that the values and policy created by the planning process are typically discarded after being used to select the current action. In many applications this is not a great loss because there are very many states and we are unlikely to return to the same state for a long time. In general, one may want to do a mix of both: focus planning on the current state *and* store the results of planning so as to be that much farther along should one return to the same state later. Decision-time planning is most useful in applications in which fast responses are not required. In chess playing programs, for example, one may be permitted seconds or minutes of computation for each move, and strong programs may plan dozens of moves ahead within this time. On the other hand, if low latency action selection is the priority, then one is generally better off doing planning in the background to compute a policy that can then be rapidly applied to each newly encountered state.

8.9 Heuristic Search

The classical state-space planning methods in artificial intelligence are decision-time planning methods collectively known as *heuristic search*. In heuristic search, for each state encountered, a large tree of possible continuations is considered. The approximate value function is applied to the leaf nodes and then backed up toward the current state at the root. The backing up within the search tree is just the same as in the expected updates with maxes (those for v_* and q_*) discussed throughout this book. The backing up stops at the state–action nodes for the current state. Once the backed-up values of these nodes are computed, the best of them is chosen as the current action, and then all backed-up values are discarded.

In conventional heuristic search no effort is made to save the backed-up values by changing the approximate value function. In fact, the value function is generally designed by people and never changed as a result of search. However, it is natural to consider allowing the value function to be improved over time, using either the backed-up values computed during heuristic search or any of the other methods presented throughout this book. In a sense we have taken this approach all along. Our greedy, ε-greedy, and UCB (Section 2.7) action-selection methods are not unlike heuristic search, albeit on a smaller scale. For example, to compute the greedy action given a model and a state-value function, we must look ahead from each possible action to each possible next state, take into account the rewards and estimated values, and then pick the best action. Just as in conventional heuristic search, this process computes backed-up values of the possible actions, but does not attempt to save them. Thus, heuristic search can be viewed as an extension of the idea of a greedy policy beyond a single step.

The point of searching deeper than one step is to obtain better action selections. If one has a perfect model and an imperfect action-value function, then in fact deeper search will usually yield better policies.[3] Certainly, if the search is all the way to the end of the episode, then the effect of the imperfect value function is eliminated, and the action determined in this way must be optimal. If the search is of sufficient depth k such that γ^k is very small, then the actions will be correspondingly near optimal. On the other hand, the deeper the search, the more computation is required, usually resulting in a slower response time. A good example is provided by Tesauro's grandmaster-level backgammon player, TD-Gammon (Section 16.1). This system used TD learning to learn an afterstate value function through many games of self-play, using a form of heuristic search to make its moves. As a model, TD-Gammon used a priori knowledge of the probabilities of dice rolls and the assumption that the opponent

[3]There are interesting exceptions to this. See, e.g., Pearl (1984).

always selected the actions that TD-Gammon rated as best for it. Tesauro found that the deeper the heuristic search, the better the moves made by TD-Gammon, but the longer it took to make each move. Backgammon has a large branching factor, yet moves must be made within a few seconds. It was only feasible to search ahead selectively a few steps, but even so the search resulted in significantly better action selections.

We should not overlook the most obvious way in which heuristic search focuses updates: on the current state. Much of the effectiveness of heuristic search is due to its search tree being tightly focused on the states and actions that might immediately follow the current state. You may spend more of your life playing chess than checkers, but when you play checkers, it pays to think about checkers and about your particular checkers position, your likely next moves, and successor positions. No matter how you select actions, it is these states and actions that are of highest priority for updates and where you most urgently want your approximate value function to be accurate. Not only should your computation be preferentially devoted to imminent events, but so should your limited memory resources. In chess, for example, there are far too many possible positions to store distinct value estimates for each of them, but chess programs based on heuristic search can easily store distinct estimates for the millions of positions they encounter looking ahead from a single position. This great focusing of memory and computational resources on the current decision is presumably the reason why heuristic search can be so effective.

The distribution of updates can be altered in similar ways to focus on the current state and its likely successors. As a limiting case we might use exactly the methods of heuristic search to construct a search tree, and then perform the individual, one-step updates from bottom up, as suggested by Figure 8.10. If the updates are ordered in this way and a tabular representation is used, then exactly the same overall update would be achieved as in depth-first heuristic search. Any state-space search can be viewed in this way as the piecing together of a large number of individual one-step updates. Thus, the performance improvement observed with deeper searches is not due to the use of multistep updates as such. Instead, it is due to the focus and concentration of updates on states and actions immediately downstream from the current state. By devoting a large amount of computation specifically relevant to the candidate actions, decision-time planning can produce better decisions than can be produced by relying on unfocused updates.

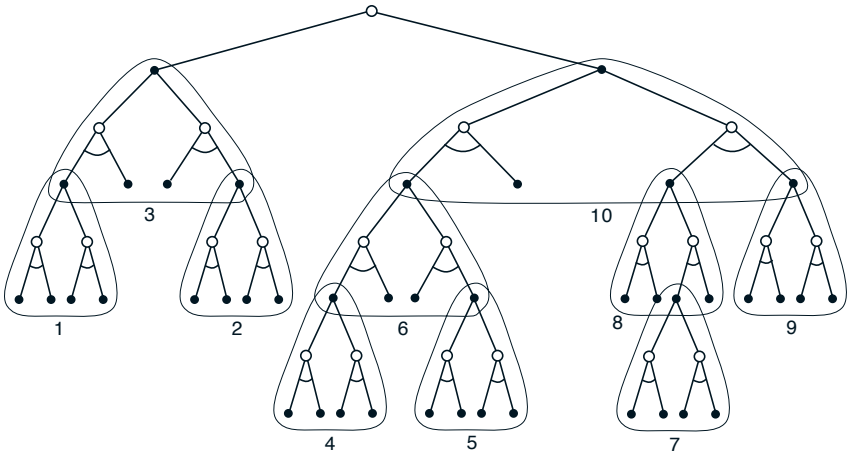


Figure 8.10: The deep updates of heuristic search can be implemented as a sequence of one-step updates (shown here outlined). The ordering shown is for a selective depth-first search.

8.10 Rollout Algorithms

Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories that all begin at the current environment state. They estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action and then follow the given policy. When the action-value estimates are considered to be accurate enough, the action (or one of the actions) having the highest estimated value is executed, after which the process is carried out anew from the resulting next state. As explained by Tesauro and Galperin (1997), who experimented with rollout algorithms for playing backgammon, the term "rollout" comes from estimating the value of a backgammon position by playing out, i.e., "rolling out," the position many times to the game's end with randomly generated sequences of dice rolls, where the moves of both players are made by some fixed policy.

Unlike the Monte Carlo control algorithms described in Chapter 5, the goal of a rollout algorithm is not to estimate a complete optimal action-value function, q_*, or a complete action-value function, q_π, for a given policy π. Instead, they produce Monte Carlo estimates of action values only for each current state and for a given policy usually called the *rollout policy*. As decision-time planning algorithms, rollout algorithms make immediate use of these action-value estimates, then discard them. This makes rollout algorithms relatively simple to implement because there is no need to sample outcomes for every state-action pair, and there is no need to approximate a function over either the state space or the state-action space.

What then do rollout algorithms accomplish? The policy improvement theorem described in Section 4.2 tells us that given any two policies π and π' that are identical except that $\pi'(s) = a \neq \pi(s)$ for some state s, if $q_\pi(s, a) \geq v_\pi(s)$, then policy π' is as good as, or better, than π. Moreover, if the inequality is strict, then π' is in fact better than π. This applies to rollout algorithms where s is the current state and π is the rollout policy. Averaging the returns of the simulated trajectories produces estimates of $q_\pi(s, a')$ for each action $a' \in \mathcal{A}(s)$. Then the policy that selects an action in s that maximizes these estimates and thereafter follows π is a good candidate for a policy that improves over π. The result is like one step of the policy-iteration algorithm of dynamic programming discussed in Section 4.3 (though it is more like one step of *asynchronous* value iteration described in Section 4.5 because it changes the action for just the current state).

In other words, the aim of a rollout algorithm is to improve upon the default policy; not to find an optimal policy. Experience has shown that rollout algorithms can be surprisingly effective. For example, Tesauro and Galperin (1997) were surprised by the dramatic improvements in backgammon playing ability produced by the rollout method. In some applications, a rollout algorithm can produce good performance even if the rollout policy is completely random. But clearly, the performance of the improved policy depends on the performance of the rollout policy and the accuracy of the Monte Carlo value estimates: the better the rollout policy and the more accurate the value estimates, the better the policy produced by a rollout algorithm is likely be.

This involves important tradeoffs because better rollout policies typically mean that more time is needed to simulate enough trajectories to obtain good value estimates. As decision-time planning methods, rollout algorithms usually have to meet strict time constraints. The computation time needed by a rollout algorithm depends on the number of actions that have to be evaluated for each decision, the number of time steps in the simulated trajectories needed to obtain useful sample returns, the time it takes the rollout policy to make decisions, and the number of simulated trajectories needed to obtain good Monte Carlo action-value estimates.

Balancing these factors is important in any application of rollout methods, though there are several ways to ease the challenge. Because the Monte Carlo trials are independent of one another, it is possible to run many trials in parallel on separate processors. Another tact is to truncate the simulated trajectories short of complete episodes, correcting the truncated returns by means of a stored evalu-

ation function (which brings into play all that we have said about truncated returns and updates in the preceding chapters). It is also possible, as Tesauro and Galperin (1997) suggest, to monitor the Monte Carlo simulations and prune away candidate actions that are unlikely to turn out to be the best, or whose values are close enough to that of the current best that choosing them instead would make no real difference (though Tesauro and Galperin point out that this would complicate a parallel implementation).

We do not ordinarily think of rollout algorithms as *learning* algorithms because they do not maintain long-term memories of values or policies. However, these algorithms take advantage of some of the features of reinforcement learning that we have emphasized in this book. As instances of Monte Carlo control, they estimate action values by averaging the returns of a collection of sample trajectories, in this case trajectories of simulated interactions with a sample model of the environment. In this way they are like reinforcement learning algorithms in avoiding the exhaustive sweeps of dynamic programming by trajectory sampling, and in avoiding the need for distribution models by relying on sample, instead of expected, updates. Finally, rollout algorithms take advantage of the policy improvement property by acting greedily with respect to the estimated action values.

8.11 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a recent and strikingly successful example of decision-time planning. At is base, MCTS is a rollout algorithm as described above, but enhanced by the addition of a means for accumulating value estimates obtained from the Monte Carlo simulations in order to successively direct simulations toward more highly-rewarding trajectories. MCTS is largely responsible for the improvement in computer Go from a weak amateur level in 2005 to a grandmaster level (6 dan or more) in 2015. Many variations of the basic algorithm have been developed, including a variant that we discuss in Section 16.7 that was critical for the stunning 2016 victories of the program AlphaGo over an 18-time world champion Go player. MCTS has proved to be effective in a wide variety of competitive settings, including general game playing (e.g., see Finnsson and Björnsson, 2008; Genesereth and Thielscher, 2014), but it is not limited to games; it can be effective for single-agent sequential decision problems if there is an environment model simple enough for fast multistep simulation.

MCTS is executed after encountering each new state to select the agent's action for that state; it is executed again to select the action for the next state, and so on. As in a rollout algorithm, each execution is an iterative process that simulates many trajectories starting from the current state and running to a terminal state (or until discounting makes any further reward negligible as a contribution to the return). The core idea of MCTS is to successively focus multiple simulations starting at the current state by extending the initial portions of trajectories that have received high evaluations from earlier simulations. MCTS does not have to retain approximate value functions or policies from one action selection to the next, though in many implementations it retains selected action values likely to be useful for its next execution.

For the most part, the actions in the simulated trajectories are generated using a simple policy, usually called a rollout policy as it is for simpler rollout algorithms. When both the rollout policy and the model do not require a lot of computation, many simulated trajectories can be generated in a short period of time. As in any tabular Monte Carlo method, the value of a state–action pair is estimated as the average of the (simulated) returns from that pair. Monte Carlo value estimates are maintained only for the subset of state–action pairs that are most likely to be reached in a few steps, which form a tree rooted at the current state, as illustrated in Figure 8.11. MCTS incrementally extends the tree by adding nodes representing states that look promising based on the results of the simulated trajectories. Any simulated trajectory will pass through the tree and then exit it at some leaf node. Outside the tree and at the leaf nodes the rollout policy is used for action selections, but at the states inside the tree something better is possible. For these states we have value estimates for of at least some of the actions,

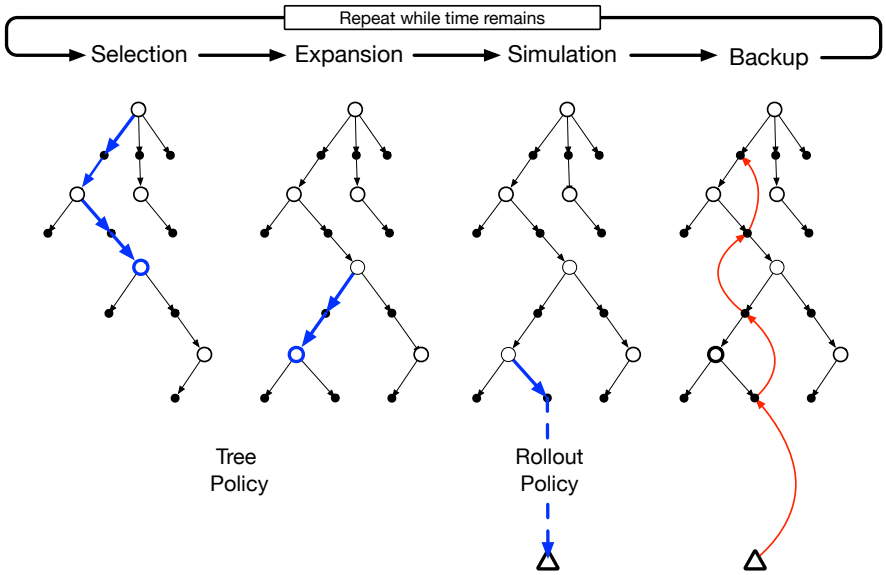


Figure 8.11: Monte Carlo Tree Search. When the environment changes to a new state, MCTS executes as many iterations as possible before an action needs to be selected, incrementally building a tree whose root node represents the current state. Each iteration consists of the four operations **Selection**, **Expansion** (though possibly skipped on some iterations), **Simulation**, and **Backup**, as explained in the text and illustrated by the bold arrows in the trees.

so we can pick among them using an informed policy, called the *tree policy*, that balances exploration and exploitation. For example, the tree policy could select actions using an ε-greedy or UCB selection rule (Chapter 2).

In more detail, each iteration of a basic version of MCTS consists of the following four steps as illustrated in Figure 8.11:

1. **Selection.** Starting at the root node, a *tree policy* based on the action values attached to the edges of the tree traverses the tree to select a leaf node.

2. **Expansion.** On some iterations (depending on details of the application), the tree is expanded from the selected leaf node by adding one or more child nodes reached from the selected node via unexplored actions.

3. **Simulation.** From the selected node, or from one of its newly-added child nodes (if any), simulation of a complete episode is run with actions selected by the rollout policy. The result is a Monte Carlo trial with actions selected first by the tree policy and beyond the tree by the rollout policy.

4. **Backup.** The return generated by the simulated episode is backed up to update, or to initialize, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS. No values are saved for the states and actions visited by the rollout policy beyond the tree. Figure 8.11 illustrates this by showing a backup from the terminal state of the simulated trajectory directly to the state–action node in the tree where the rollout policy began (though in general, the entire return over the simulated trajectory is backed up to this state–action node).

MCTS continues executing these four steps, starting each time at the tree's root node, until no more time is left, or some other computational resource is exhausted. Then, finally, an action from the root node (which still represents the current state of the environment) is selected according to some mechanism that depends on the accumulated statistics in the tree; for example, it may be an action having the largest action value of all the actions available from the root state, or perhaps the action with the largest visit count to avoid selecting outliers. This is the action MCTS actually selects. After the environment transitions to a new state, MCTS is run again, sometimes starting with a tree of a single root node representing the new state, but often starting with a tree containing any descendants of this node left over from the tree constructed by the previous execution of MCTS; all the remaining nodes are discarded, along with the action values associated with them.

MCTS was first proposed to select moves in programs playing two-person competitive games, such as Go. For game playing, each simulated episode is one complete play of the game in which both players select actions by the tree and rollout policies. Section 16.7 describes an extension of MCTS used in the AlphaGo program that combines the Monte Carlo evaluations of MCTS with action values learned by a deep ANN via self-play reinforcement learning.

Relating MCTS to the reinforcement learning principles we describe in this book provides some insight into how it achieves such impressive results. At its base, MCTS is a decision-time planning algorithm based on Monte Carlo control applied to simulations that start from the root state; that is, it is a kind of rollout algorithm as described in the previous section. It therefore benefits from online, incremental, sample-based value estimation and policy improvement. Beyond this, it saves action-value estimates attached to the tree edges and updates them using reinforcement learning's sample updates. This has the effect of focusing the Monte Carlo trials on trajectories whose initial segments are common to high-return trajectories previously simulated. Further, by incrementally expanding the tree, MCTS effectively grows a lookup table to store a partial action-value function, with memory allocated to the estimated values of state–action pairs visited in the initial segments of high-yielding sample trajectories. MCTS thus avoids the problem of globally approximating an action-value function while it retrains the benefit of using past experience to guide exploration.

The striking success of decision-time planning by MCTS has deeply influenced artificial intelligence, and many researchers are studying modifications and extensions of the basic procedure for use in both games and single-agent applications.

8.12 Summary of the Chapter

Planning requires a model of the environment. A *distribution model* consists of the probabilities of next states and rewards for possible actions; a sample model produces single transitions and rewards generated according to these probabilities. Dynamic programming requires a distribution model because it uses *expected updates*, which involve computing expectations over all the possible next states and rewards. A *sample model*, on the other hand, is what is needed to simulate interacting with the environment during which *sample updates*, like those used by many reinforcement learning algorithms, can be used. Sample models are generally much easier to obtain than distribution models.

We have presented a perspective emphasizing the surprisingly close relationships between planning optimal behavior and learning optimal behavior. Both involve estimating the same value functions, and in both cases it is natural to update the estimates incrementally, in a long series of small backing-up operations. This makes it straightforward to integrate learning and planning processes simply by allowing both to update the same estimated value function. In addition, any of the learning methods can be converted into planning methods simply by applying them to simulated (model-generated) experience rather than to real experience. In this case learning and planning become even more similar; they are possibly identical algorithms operating on two different sources of experience.

It is straightforward to integrate incremental planning methods with acting and model-learning. Planning, acting, and model-learning interact in a circular fashion (Figure 8.1), each producing what the other needs to improve; no other interaction among them is either required or prohibited. The most natural approach is for all processes to proceed asynchronously and in parallel. If the processes must share computational resources, then the division can be handled almost arbitrarily—by whatever organization is most convenient and efficient for the task at hand.

In this chapter we have touched upon a number of dimensions of variation among state-space planning methods. One dimension is the variation in the size of updates. The smaller the updates, the more incremental the planning methods can be. Among the smallest updates are one-step sample updates, as in Dyna. Another important dimension is the distribution of updates, that is, of the focus of search. Prioritized sweeping focuses backward on the predecessors of states whose values have recently changed. On-policy trajectory sampling focuses on states or state–action pairs that the agent is likely to encounter when controlling its environment. This can allow computation to skip over parts of the state space that are irrelevant to the prediction or control problem. Real-time dynamic programming, an on-policy trajectory sampling version of value iteration, illustrates some of the advantages this strategy has over conventional sweep-based policy iteration.

Planning can also focus forward from pertinent states, such as states actually encountered during an agent-environment interaction. The most important form of this is when planning is done at decision time, that is, as part of the action-selection process. Classical heuristic search as studied in artificial intelligence is an example of this. Other examples are rollout algorithms and Monte Carlo Tree Search that benefit from online, incremental, sample-based value estimation and policy improvement.

8.13 Summary of Part I: Dimensions

This chapter concludes Part I of this book. We have tried to present reinforcement learning not as a collection of individual methods, but as a coherent set of ideas cutting across methods. Each idea can be viewed as a dimension along which methods vary. The set of such dimensions spans a large space of possible methods. By exploring this space at the level of dimensions we hope to obtain the broadest and most lasting understanding. In this section we use the concept of dimensions in method space to recapitulate the view of reinforcement learning developed so far in this book.

All of the methods we have explored so far in this book have three key ideas in common: first, they all seek to estimate value functions; second, they all operate by backing up values along actual or possible state trajectories; and third, they all follow the general strategy of generalized policy iteration (GPI), meaning that they maintain an approximate value function and an approximate policy, and they continually try to improve each on the basis of the other. These three ideas are central to the subjects covered in this book. We suggest that value functions, backing-up value updates, and GPI are powerful organizing principles potentially relevant to any model of intelligence, whether artificial or natural.

Two of the most important dimensions along which the methods vary are shown in Figure 8.12. These dimensions have to do with the kind of update used to improve the value function. The horizontal dimension is whether they are sample updates (based on a sample trajectory) or expected updates (based on a distribution of possible trajectories). Expected updates require a distribution model, whereas sample updates need only a sample model, or can be done from actual experience with no model at all (another dimension of variation). The vertical dimension of Figure 8.12 corresponds to the depth of updates, that is, to the degree of bootstrapping. At three of the four corners of the space are the three primary methods for estimating values: DP, TD, and Monte Carlo. Along the left edge of the space are the sample-update methods, ranging from one-step TD updates to full-return Monte Carlo updates. Between these is a spectrum including methods based on n-step updates (and in Chapter 12 we will extend this to mixtures of n-step updates such as the λ-updates implemented by eligibility traces).

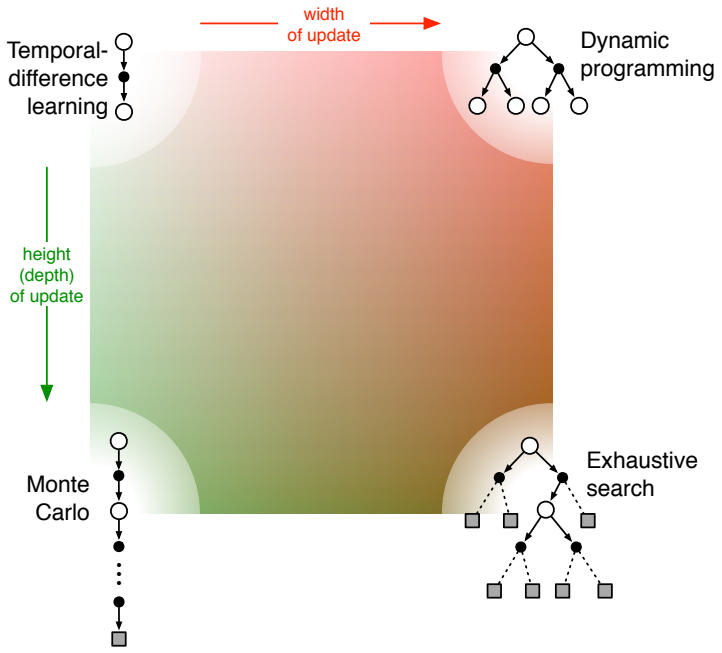


Figure 8.12: A slice through the space of reinforcement learning methods.

DP methods are shown in the extreme upper-right corner of the space because they involve one-step expected updates. The lower-right corner is the extreme case of expected updates so deep that they run all the way to terminal states (or, in a continuing task, until discounting has reduced the contribution of any further rewards to a negligible level). This is the case of exhaustive search. Intermediate methods along this dimension include heuristic search and related methods that search and update up to a limited depth, perhaps selectively. There are also methods that are intermediate along the horizontal dimension. These include methods that mix expected and sample updates, as well as the possibility of methods that mix samples and distributions within a single update. The interior of the square is filled in to represent the space of all such intermediate methods.

A third dimension that we have emphasized in this book is the binary distinction between on-policy and off-policy methods. In the former case, the agent learns the value function for the policy it is currently following, whereas in the latter case it learns the value function for the policy for a different policy, often the one that the agent currently thinks is best. The policy generating behavior is typically different from what is currently thought best because of the need to explore. This third dimension might be visualized as perpendicular to the plane of the page in Figure 8.12.

In addition to the three dimensions just discussed, we have identified a number of others throughout the book:

Definition of return Is the task episodic or continuing, discounted or undiscounted?

Action values vs. state values vs. afterstate values What kind of values should be estimated? If only state values are estimated, then either a model or a separate policy (as in actor–critic methods) is required for action selection.

Action selection/exploration How are actions selected to ensure a suitable trade-off between exploration and exploitation? We have considered only the simplest ways to do this: ε-greedy,

optimistic initialization of values, softmax, and upper confidence bound.

Synchronous vs. asynchronous Are the updates for all states performed simultaneously or one by one in some order?

Real vs. simulated Should one update based on real experience or simulated experience? If both, how much of each?

Location of updates What states or state–action pairs should be updated? Model-free methods can choose only among the states and state–action pairs actually encountered, but model-based methods can choose arbitrarily. There are many possibilities here.

Timing of updates Should updates be done as part of selecting actions, or only afterward?

Memory for updates How long should updated values be retained? Should they be retained permanently, or only while computing an action selection, as in heuristic search?

Of course, these dimensions are neither exhaustive nor mutually exclusive. Individual algorithms differ in many other ways as well, and many algorithms lie in several places along several dimensions. For example, Dyna methods use both real and simulated experience to affect the same value function. It is also perfectly sensible to maintain multiple value functions computed in different ways or over different state and action representations. These dimensions do, however, constitute a coherent set of ideas for describing and exploring a wide space of possible methods.

The most important dimension not mentioned here, and not covered in Part I of this book, is that of function approximation. Function approximation can be viewed as an orthogonal spectrum of possibilities ranging from tabular methods at one extreme through state aggregation, a variety of linear methods, and then a diverse set of nonlinear methods. This dimension is explored in Part II.

Bibliographical and Historical Remarks

8.1 The overall view of planning and learning presented here has developed gradually over a number of years, in part by the authors (Sutton, 1990, 1991a, 1991b; Barto, Bradtke, and Singh, 1991, 1995; Sutton and Pinette, 1985; Sutton and Barto, 1981b); it has been strongly influenced by Agre and Chapman (1990; Agre 1988), Bertsekas and Tsitsiklis (1989), Singh (1993), and others. The authors were also strongly influenced by psychological studies of latent learning (Tolman, 1932) and by psychological views of the nature of thought (e.g., Galanter and Gerstenhaber, 1956; Craik, 1943; Campbell, 1960; Dennett, 1978). In the Part III of the book, Section 14.6 relates model-based and model-free methods to psychological theories of learning and behavior, and Section 15.11 discusses ideas about how the brain might implement these types of methods.

8.2 The terms *direct* and *indirect*, which we use to describe different kinds of reinforcement learning, are from the adaptive control literature (e.g., Goodwin and Sin, 1984), where they are used to make the same kind of distinction. The term *system identification* is used in adaptive control for what we call *model-learning* (e.g., Goodwin and Sin, 1984; Ljung and Söderstrom, 1983; Young, 1984). The Dyna architecture is due to Sutton (1990), and the results in this and the next section are based on results reported there. Barto and Singh (1991) consider some of the issues in comparing direct and indirect reinforcement learning methods.

8.3 There have been several works with model-based reinforcement learning that take the idea of exploration bonuses and optimistic initialization to its logical extreme, in which all incompletely explored choices are assumed maximally rewarding and optimal paths are computed to test

them. The E^3 algorithm of Kearns and Singh (2002) and the R-max algorithm of Brafman and Tennenholtz (2003) are guaranteed to find a near-optimal solution in time polynomial in the number of states and actions. This is usually too slow for practical algorithms but is probably the best that can be done in the worst case.

8.4 Prioritized sweeping was developed simultaneously and independently by Moore and Atkeson (1993) and Peng and Williams (1993). The results in Example 8.4 are due to Peng and Williams (1993). The results in Example 8.5 are due to Moore and Atkeson. Key subsequent work in this area includes that by McMahan and Gordon (2005) and by van Seijen and Sutton (2013).

8.5 This section was strongly influenced by the experiments of Singh (1993).

8.6–7 Trajectory sampling has implicitly been a part of reinforcement learning from the outset, but it was most explicitly emphasized by Barto, Bradtke, and Singh (1995) in their introduction of RTDP. They recognized that Korf's (1990) *learning real-time A** (LRTA*) algorithm is an asynchronous DP algorithm that applies to stochastic problems as well as the deterministic problems on which Korf focused. Beyond LRTA*, RTDP includes the option of updating the values of many states in the time intervals between the execution of actions. Barto et al. (1995) proved the convergence result described here by combining Korf's (1990) convergence proof for LRTA* with the result of Bertsekas (1982) (also Bertsekas and Tsitsiklis, 1989) ensuring convergence of asynchronous DP for stochastic shortest path problems in the undiscounted case. Combining model-learning with RTDP is called *Adaptive* RTDP, also presented by Barto et al. (1995) and discussed by Barto (2011).

8.9 For further reading on heuristic search, the reader is encouraged to consult texts and surveys such as those by Russell and Norvig (2009) and Korf (1988). Peng and Williams (1993) explored a forward focusing of updates much as is suggested in this section.

8.10 Abramson's (1990) expected-outcome model is a rollout algorithm applied to two-person games in which the play of both simulated players is random. He argued that even with random play, it is a "powerful heuristic" that is "precise, accurate, easily estimable, efficiently calculable, and domain-independent." Tesauro and Galperin (1997) demonstrated the effectiveness of rollout algorithms for improving the play of backgammon programs, adopting the term "rollout" from its use in evaluating backgammon positions by playing out positions with different randomly generating sequences of dice rolls. Bertsekas, Tsitsiklis, and Wu (1997) examine rollout algorithms applied to combinatorial optimization problems, and Bertsekas (2013) surveys their use in discrete deterministic optimization problems, remarking that they are "often surprisingly effective."

8.11 The central ideas of MCTS were introduced by Coulom (2006) and by Kocsis and Szepesvári (2006). They built upon previous research with Monte Carlo planning algorithms as reviewed by these authors. Browne, Powley, Whitehouse, Lucas, Cowling, Rohlfshagen, Tavener, Perez, Samothrakis, and Colton (2012) is an excellent survey of MCTS methods and their applications. David Silver contributed to the ideas and presentation in this section.

Part II: Approximate Solution Methods

In the second part of the book we extend the tabular methods presented in Part I to apply to problems with arbitrarily large state spaces. In many of the tasks to which we would like to apply reinforcement learning the state space is combinatorial and enormous; the number of possible camera images, for example, is much larger than the number of atoms in the universe. In such cases we cannot expect to find an optimal policy or the optimal value function even in the limit of infinite time and data; our goal instead is to find a good approximate solution using limited computational resources. In this part of the book we explore such approximate solution methods.

The problem with large state spaces is not just the memory needed for large tables, but the time and data needed to fill them accurately. In many of our target tasks, almost every state encountered will never have been seen before. To make sensible decisions in such states it is necessary to generalize from previous encounters with different states that are in some sense similar to the current one. In other words, the key issue is that of *generalization*. How can experience with a limited subset of the state space be usefully generalized to produce a good approximation over a much larger subset?

Fortunately, generalization from examples has already been extensively studied, and we do not need to invent totally new methods for use in reinforcement learning. To some extent we need only combine reinforcement learning methods with existing generalization methods. The kind of generalization we require is often called *function approximation* because it takes examples from a desired function (e.g., a value function) and attempts to generalize from them to construct an approximation of the entire function. Function approximation is an instance of *supervised learning*, the primary topic studied in machine learning, artificial neural networks, pattern recognition, and statistical curve fitting. In theory, any of the methods studied in these fields can be used in the role of function approximator within reinforcement learning algorithms, although in practice some fit more easily into this role than others.

Reinforcement learning with function approximation involves a number of new issues that do not normally arise in conventional supervised learning, such as nonstationarity, bootstrapping, and delayed targets. We introduce these and other issues successively over the five chapters of this part. Initially we restrict attention to on-policy training, treating in Chapter 9 the prediction case, in which the policy is given and only its value function is approximated, and then in Chapter 10 the control case, in which an approximation to the optimal policy is found. The challenging problem of off-policy learning with function approximation is treated in Chapter 11. In each of these three chapters we will have to return to first principles and re-examine the objectives of the learning to take into account function approximation. Chapter 12 introduces and analyzes the algorithmic mechanism of *eligibility traces*, which dramatically improves the computational properties of multi-step reinforcement learning methods in many cases. The final chapter of this part explores a different approach to control, *policy-gradient methods*, which approximate the optimal policy directly and need never form an approximate value function (although they may be much more efficient if they do approximate a value function as well the policy).

Chapter 9

On-policy Prediction with Approximation

In this chapter, we begin our study of function approximation in reinforcement learning by considering its use in estimating the state-value function from on-policy data, that is, in approximating v_π from experience generated using a known policy π. The novelty in this chapter is that the approximate value function is represented not as a table but as a parameterized functional form with weight vector $\mathbf{w} \in \mathbb{R}^d$. We will write $\hat{v}(s,\mathbf{w}) \approx v_\pi(s)$ for the approximate value of state s given weight vector $\mathbf{w}$. For example, $\hat{v}$ might be a linear function in features of the state, with $\mathbf{w}$ the vector of feature weights. More generally, $\hat{v}$ might be the function computed by a multi-layer artificial neural network, with $\mathbf{w}$ the vector of connection weights in all the layers. By adjusting the weights, any of a wide range of different functions can be implemented by the network. Or $\hat{v}$ might be the function computed by a decision tree, where $\mathbf{w}$ is all the numbers defining the split points and leaf values of the tree. Typically, the number of weights (the dimensionality of $\mathbf{w}$) is much less than the number of states ($d \ll |\mathcal{S}|$), and changing one weight changes the estimated value of many states. Consequently, when a single state is updated, the change generalizes from that state to affect the values of many other states. Such *generalization* makes the learning potentially more powerful but also potentially more difficult to manage and understand.

9.1 Value-function Approximation

All of the prediction methods covered in this book have been described as updates to an estimated value function that shift its value at particular states toward a "backed-up value," or *update target*, for that state. Let us refer to an individual update by the notation $s \mapsto u$, where s is the state updated and u is the update target that s's estimated value is shifted toward. For example, the Monte Carlo update for value prediction is $S_t \mapsto G_t$, the TD(0) update is $S_t \mapsto R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t)$, and the n-step TD update is $S_t \mapsto G_{t:t+n}$. In the DP (dynamic programming) policy-evaluation update, $s \mapsto \mathbb{E}_\pi[R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t) \mid S_t = s]$, an arbitrary state s is updated, whereas in the other cases the state encountered in actual experience, S_t, is updated.

It is natural to interpret each update as specifying an example of the desired input–output behavior of the value function. In a sense, the update $s \mapsto u$ means that the estimated value for state s should be more like the update target u. Up to now, the actual update has been trivial: the table entry for s's estimated value has simply been shifted a fraction of the way toward u, and the estimated values of all other states were left unchanged. Now we permit arbitrarily complex and sophisticated methods to implement the update, and updating at s generalizes so that the estimated values of many other states

are changed as well. Machine learning methods that learn to mimic input–output examples in this way are called *supervised learning* methods, and when the outputs are numbers, like u, the process is often called *function approximation*. Function approximation methods expect to receive examples of the desired input–output behavior of the function they are trying to approximate. We use these methods for value prediction simply by passing to them the $s \mapsto g$ of each update as a training example. We then interpret the approximate function they produce as an estimated value function.

Viewing each update as a conventional training example in this way enables us to use any of a wide range of existing function approximation methods for value prediction. In principle, we can use any method for supervised learning from examples, including artificial neural networks, decision trees, and various kinds of multivariate regression. However, not all function approximation methods are equally well suited for use in reinforcement learning. The most sophisticated neural network and statistical methods all assume a static training set over which multiple passes are made. In reinforcement learning, however, it is important that learning be able to occur on-line, while interacting with the environment or with a model of the environment. To do this requires methods that are able to learn efficiently from incrementally acquired data. In addition, reinforcement learning generally requires function approximation methods able to handle nonstationary target functions (target functions that change over time). For example, in control methods based on GPI (generalized policy iteration) we often seek to learn q_π while π changes. Even if the policy remains the same, the target values of training examples are nonstationary if they are generated by bootstrapping methods (DP and TD learning). Methods that cannot easily handle such nonstationarity are less suitable for reinforcement learning.

9.2 The Prediction Objective (MSVE)

Up to now we have not specified an explicit objective for prediction. In the tabular case a continuous measure of prediction quality was not necessary because the learned value function could come to equal the true value function exactly. Moreover, the learned values at each state were decoupled—an update at one state affected no other. But with genuine approximation, an update at one state affects many others, and it is not possible to get all states exactly correct. By assumption we have far more states than weights, so making one state's estimate more accurate invariably means making others' less accurate. We are obligated then to say which states we care most about. We must specify a *state weighting* or distribution $\mu(s) \geq 0$, $\sum_s \mu(s) = 1$, representing how much we care about the error in each state s. By the error in a state s we mean the square of the difference between the approximate value $\hat{v}(s,\mathbf{w})$ and the true value $v_\pi(s)$. Weighting this over the state space by μ, we obtain a natural objective function, the *Mean Squared Value Error*, or MSVE:

$$\text{MSVE}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \Big[v_\pi(s) - \hat{v}(s,\mathbf{w}) \Big]^2. \tag{9.1}$$

The square root of this measure, the root MSVE or RMSVE, gives a rough measure of how much the approximate values differ from the true values and is often used in plots. Often $\mu(s)$ is chosen to be the fraction of time spent in s. Under on-policy training this is called the *on-policy distribution*; we focus entirely on this case in this chapter. In continuing tasks, the on-policy distribution is the stationary distribution under π.

The on-policy distribution in episodic tasks

In an episodic task, the on-policy distribution is a little different in that it depends on how the initial states of episodes are chosen. Let $h(s)$ denote the probability that an episode begins in each state s, and let $n(s)$ denote the number of time steps spent, on average, in state s in a single episode. Time is spent in a state s if episodes start in s, or if transitions are made into s from a preceding state $\bar{s}$ in which time is spent:

$$n(s) = h(s) + \sum_{\bar{s}} n(\bar{s}) \sum_{a} \pi(a|\bar{s}) p(s|\bar{s}, a), \quad \text{for all } s \in \mathcal{S}. \tag{9.2}$$

This system of equations can be solved for the expected number of visits $n(s)$. The on-policy distribution is then the fraction of time spent in each state normalized to sum to one:

$$\mu(s) = \frac{n(s)}{\sum_s n(s)}, \quad \text{for all } s \in \mathcal{S}. \tag{9.3}$$

This is the natural choice without discounting. If there is discounting ($\gamma < 1$), then it should be treated as a form of termination, which can be done simply by including a factor of γ in the second term of (9.2). Although this is more general, it would complicate the following presentation of algorithms, and concerns a rare case that we don't treat in this chapter, so we omit it here.

The two cases, continuing and episodic, behave similarly, but with approximation they must be treated separately in formal analyses, as we will see repeatedly in this part of the book. This completes the specification of the learning objective.

It is not completely clear that the MSVE is the right performance objective for reinforcement learning. Remember that our ultimate purpose—the reason we are learning a value function—is to find a better policy. The best value function for this purpose is not necessarily the best for minimizing MSVE. Nevertheless, it is not yet clear what a more useful alternative goal for value prediction might be. For now, we will focus on MSVE.

An ideal goal in terms of MSVE would be to find a *global optimum*, a weight vector $\mathbf{w}^*$ for which $\text{MSVE}(\mathbf{w}^*) \leq \text{MSVE}(\mathbf{w})$ for all possible $\mathbf{w}$. Reaching this goal is sometimes possible for simple function approximators such as linear ones, but is rarely possible for complex function approximators such as artificial neural networks and decision trees. Short of this, complex function approximators may seek to converge instead to a *local optimum*, a weight vector $\mathbf{w}^*$ for which $\text{MSVE}(\mathbf{w}^*) \leq \text{MSVE}(\mathbf{w})$ for all $\mathbf{w}$ in some neighborhood of $\mathbf{w}^*$. Although this guarantee is only slightly reassuring, it is typically the best that can be said for nonlinear function approximators, and often it is enough. Still, for many cases of interest in reinforcement learning there is no guaranteed of convergence to an optimum, or even to within a bounded distance of an optimum. Some methods may in fact diverge, with their MSVE approaching infinity in the limit.

In the last two sections we have outlined a framework for combining a wide range of reinforcement learning methods for value prediction with a wide range of function approximation methods, using the updates of the former to generate training examples for the latter. We have also described a MSVE performance measure which these methods may aspire to minimize. The range of possible function approximation methods is far too large to cover all, and anyway too little is known about most of them to make a reliable evaluation or recommendation. Of necessity, we consider only a few possibilities. In the rest of this chapter we focus on function approximation methods based on gradient principles, and on linear gradient-descent methods in particular. We focus on these methods in part because we consider them to be particularly promising and because they reveal key theoretical issues, but also because they are simple and our space is limited.

9.3 Stochastic-gradient and Semi-gradient Methods

We now develop in detail one class of learning methods for function approximation in value prediction, those based on stochastic gradient descent (SGD). SGD methods are among the most widely used of all function approximation methods and are particularly well suited to online reinforcement learning.

In gradient-descent methods, the weight vector is a column vector with a fixed number of real valued components, $\mathbf{w} \doteq (w_1, w_2, \ldots, w_d)^\top$,[1] and the approximate value function $\hat{v}(s,\mathbf{w})$ is a differentiable function of $\mathbf{w}$ for all $s \in \mathcal{S}$. We will be updating $\mathbf{w}$ at each of a series of discrete time steps, $t = 0, 1, 2, 3, \ldots$, so we will need a notation $\mathbf{w}_t$ for the weight vector at each step. For now, let us assume that, on each step, we observe a new example $S_t \mapsto v_\pi(S_t)$ consisting of a (possibly randomly selected) state S_t and its true value under the policy. These states might be successive states from an interaction with the environment, but for now we do not assume so. Even though we are given the exact, correct values, $v_\pi(S_t)$ for each S_t, there is still a difficult problem because our function approximator has limited resources and thus limited resolution. In particular, there is generally no $\mathbf{w}$ that gets all the states, or even all the examples, exactly correct. In addition, we must generalize to all the other states that have not appeared in examples.

We assume that states appear in examples with the same distribution, μ, over which we are trying to minimize the MSVE as given by (9.1). A good strategy in this case is to try to minimize error on the observed examples. *Stochastic gradient-descent* (SGD) methods do this by adjusting the weight vector after each example by a small amount in the direction that would most reduce the error on that example:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t - \frac{1}{2}\alpha\nabla\Big[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\Big]^2 \tag{9.4}$$

$$= \mathbf{w}_t + \alpha\Big[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\Big]\nabla\hat{v}(S_t,\mathbf{w}_t), \tag{9.5}$$

where α is a positive step-size parameter, and $\nabla f(\mathbf{w})$, for any scalar expression $f(\mathbf{w})$, denotes the vector of partial derivatives with respect to the components of the weight vector:

$$\nabla f(\mathbf{w}) \doteq \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \ldots, \frac{\partial f(\mathbf{w})}{\partial w_d}\right)^\top. \tag{9.6}$$

This derivative vector is the *gradient* of f with respect to $\mathbf{w}$. SGD methods are "gradient descent" methods because the overall step in $\mathbf{w}_t$ is proportional to the negative gradient of the example's squared error (9.4). This is the direction in which the error falls most rapidly. Gradient descent methods are called "stochastic" when the update is done, as here, on only a single example, which might have been selected stochastically. Over many examples, making small steps, the overall effect is to minimize an average performance measure such as the MSVE.

It may not be immediately apparent why SGD takes only a small step in the direction of the gradient. Could we not move all the way in this direction and completely eliminate the error on the example? In many cases this could be done, but usually it is not desirable. Remember that we do not seek or expect to find a value function that has zero error for all states, but only an approximation that balances the errors in different states. If we completely corrected each example in one step, then we would not find such a balance. In fact, the convergence results for SGD methods assume that α decreases over time. If it decreases in such a way as to satisfy the standard stochastic approximation conditions (2.7), then the SGD method (9.5) is guaranteed to converge to a local optimum.

We turn now to the case in which the target output, here denoted $U_t \in \mathbb{R}$, of the tth training example, $S_t \mapsto U_t$, is not the true value, $v_\pi(S_t)$, but some, possibly random, approximation to it. For example,

[1] The $^\top$ denotes transpose, needed here to turn the horizontal row vector in the text into a vertical column vector; in this book vectors are generally taken to be column vectors unless explicitly written out horizontally, as here, or transposed.

U_t might be a noise-corrupted version of $v_\pi(S_t)$, or it might be one of the bootstrapping targets using $\hat{v}$ mentioned in the previous section. In these cases we cannot perform the exact update (9.5) because $v_\pi(S_t)$ is unknown, but we can approximate it by substituting U_t in place of $v_\pi(S_t)$. This yields the following general SGD method for state-value prediction:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[U_t - \hat{v}(S_t,\mathbf{w}_t)\Big]\nabla\hat{v}(S_t,\mathbf{w}_t). \tag{9.7}$$

If U_t is an *unbiased* estimate, that is, if $\mathbb{E}[U_t] = v_\pi(S_t)$, for each t, then $\mathbf{w}_t$ is guaranteed to converge to a local optimum under the usual stochastic approximation conditions (2.7) for decreasing α.

For example, suppose the states in the examples are the states generated by interaction (or simulated interaction) with the environment using policy π. Because the true value of a state is the expected value of the return following it, the Monte Carlo target $U_t \doteq G_t$ is by definition an unbiased estimate of $v_\pi(S_t)$. With this choice, the general SGD method (9.7) converges to a locally optimal approximation to $v_\pi(S_t)$. Thus, the gradient-descent version of Monte Carlo state-value prediction is guaranteed to find a locally optimal solution. Pseudocode for a complete algorithm is shown in the box below.

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$

Initialize value-function weights $\mathbf{w}$ as appropriate (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat forever:
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  For $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

One does not obtain the same guarantees if a bootstrapping estimate of $v_\pi(S_t)$ is used as the target U_t in (9.7). Bootstrapping targets such as n-step returns $G_{t:t+n}$ or the DP target $\sum_{a,s',r}\pi(a|S_t)p(s',r\,|\,S_t,a)[r+\gamma\hat{v}(s',\mathbf{w}_t)]$ all depend on the current value of the weight vector $\mathbf{w}_t$, which implies that they will be biased and that they will not produce a true gradient-descent method. One way to look at this is that the key step from (9.4) to (9.5) relies on the target being independent of $\mathbf{w}_t$. This step would not be valid if a bootstrapping estimate was used in place of $v_\pi(S_t)$. Bootstrapping methods are not in fact instances of true gradient descent (Barnard, 1993). They take into account the effect of changing the weight vector $\mathbf{w}_t$ on the estimate, but ignore its effect on the target. They include only a part of the gradient and, accordingly, we call them *semi-gradient methods*.

Although semi-gradient (bootstrapping) methods do not converge as robustly as gradient methods, they do converge reliably in important cases such as the linear case discussed in the next section. Moreover, they offer important advantages which makes them often clearly preferred. One reason for this is that they are typically significantly faster to learn, as we have seen in Chapters 6 and 7. Another is that they enable learning to be continual and online, without waiting for the end of an episode. This enables them to be used on continuing problems and provides computational advantages. A prototypical semi-gradient method is semi-gradient TD(0), which uses $U_t \doteq R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w})$ as its target. Complete pseudocode for this method is given in the box below.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$

Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S' is terminal

State aggregation is a simple form of generalizing function approximation in which states are grouped together, with one estimated value (one component of the weight vector $\mathbf{w}$) for each group. The value of a state is estimated as its group's component, and when the state is updated, that component alone is updated. State aggregation is a special case of SGD (9.7) in which the gradient, $\nabla\hat{v}(S_t,\mathbf{w}_t)$, is 1 for S_t's group's component and 0 for the other components.

Example 9.1: State Aggregation on the 1000-state Random Walk Consider a 1000-state version of the random walk task (Examples 6.2 and 7.1 on pages 102 and 118). The states are numbered from 1 to 1000, left to right, and all episodes begin near the center, in state 500. State transitions are from the current state to one of the 100 neighboring states to its left, or to one of the 100 neighboring states to its right, all with equal probability. Of course, if the current state is near an edge, then there may be fewer than 100 neighbors on that side of it. In this case, all the probability that would have gone into those missing neighbors goes into the probability of terminating on that side (thus, state 1 has a 0.5 chance of terminating on the left, and state 950 has a 0.25 chance of terminating on the right). As usual, termination on the left produces a reward of -1, and termination on the right produces a reward of $+1$. All other transitions have a reward of zero. We use this task as a running example throughout this section.

Figure 9.1 shows the true value function v_π for this task. It is nearly a straight line, but tilted slightly toward the horizontal and curving further in this direction for the last 100 states at each end. Also shown is the final approximate value function learned by the gradient Monte-Carlo algorithm with state aggregation after 100,000 episodes with a step size of $\alpha = 2 \times 10^{-5}$. For the state aggregation, the 1000 states were partitioned into 10 groups of 100 states each (i.e., states 1–100 were one group, states 101–200 were another, and so on). The staircase effect shown in the figure is typical of state aggregation; within each group, the approximate value is constant, and it changes abruptly from one group to the next. These approximate values are close to the global minimum of the MSVE (9.1).

Some of the details of the approximate values are best appreciated by reference to the state distribution μ for this task, shown in the lower portion of the figure with a right-side scale. State 500, in the center, is the first state of every episode, but it is rarely visited again. On average, about 1.37% of the time steps are spent in the start state. The states reachable in one step from the start state are the second most visited, with about 0.17% of the time steps being spent in each of them. From there μ falls off almost linearly, reaching about 0.0147% at the extreme states 1 and 1000. The most visible effect of the distribution is on the leftmost groups, whose values are clearly shifted higher than the unweighted average of the true values of states within the group, and on the rightmost groups, whose values are clearly shifted lower. This is due to the states in these areas having the greatest asymmetry

in their weightings by μ. For example, in the leftmost group, state 99 is weighted more than 3 times more strongly than state 0. Thus the estimate for the group is biased toward the true value of state 99, which is higher than the true value of state 0.

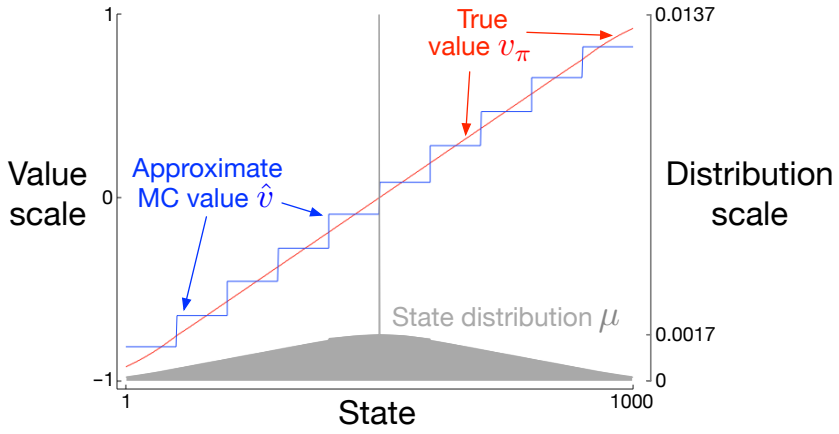


Figure 9.1: Function approximation by state aggregation on the 1000-state random walk task, using the gradient Monte Carlo algorithm (page 165). ■

9.4 Linear Methods

One of the most important special cases of function approximation is that in which the approximate function, $\hat{v}(\cdot,\mathbf{w})$, is a linear function of the weight vector, $\mathbf{w}$. Corresponding to every state s, there is a real-valued vector of features $\mathbf{x}(s) \doteq (x_1(s), x_2(s), \ldots, x_d(s))^\top$, with the same number of components as $\mathbf{w}$. The features may be constructed from the states in many different ways; we cover a few possibilities in the next sections. However the features are constructed, the approximate state-value function is given by the inner product between $\mathbf{w}$ and $\mathbf{x}(s)$:

$$\hat{v}(s,\mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s). \tag{9.8}$$

In this case the approximate value function is said to be *linear in the weights*, or simply *linear*. The individual functions $x_i : \mathcal{S} \to \mathbb{R}$ are called *basis functions* because they form a linear basis for the set of approximate functions of this form. Constructing n-dimensional feature vectors to represent states is the same as selecting a set of n basis functions.

It is natural to use SGD updates with linear function approximation. The gradient of the approximate value function with respect to $\mathbf{w}$ in this case is

$$\nabla \hat{v}(s,\mathbf{w}) = \mathbf{x}(s).$$

Thus, the general SGD update (9.7) reduces to a particularly simple form in the linear case.

Because it is so simple, the linear SGD case is one of the most favorable for mathematical analysis. Almost all useful convergence results for learning systems of all kinds are for linear (or simpler) function approximation methods.

In particular, in the linear case there is only one optimum (or, in degenerate cases, one set of equally good optima), and thus any method that is guaranteed to converge to or near a local optimum is

automatically guaranteed to converge to or near the global optimum. For example, the gradient Monte Carlo algorithm presented in the previous section converges to the global optimum of the MSVE under linear function approximation if α is reduced over time according to the usual conditions.

The semi-gradient TD(0) algorithm presented in the previous section also converges under linear function approximation, but this does not follow from general results on SGD; a separate theorem is necessary. The weight vector converged to is also not the global optimum, but rather a point near the local optimum. It is useful to consider this important case in more detail, specifically for the continuing case. The update at each time t is

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha\Big(R_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t\Big)\mathbf{x}_t \\ &= \mathbf{w}_t + \alpha\Big(R_{t+1}\mathbf{x}_t - \mathbf{x}_t\big(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}\big)^\top \mathbf{w}_t\Big),\end{aligned} \tag{9.9}$$

where here we have used the notational shorthand $\mathbf{x}_t = \mathbf{x}(S_t)$. Once the system has reached steady state, for any given $\mathbf{w}_t$, the expected next weight vector can be written

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t), \tag{9.10}$$

where

$$\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d \quad \text{and} \quad \mathbf{A} \doteq \mathbb{E}\Big[\mathbf{x}_t\big(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}\big)^\top\Big] \in \mathbb{R}^d \times \mathbb{R}^d \tag{9.11}$$

From (9.10) it is clear that, if the system converges, it must converge to the weight vector $\mathbf{w}_{TD}$ at which

$$\begin{aligned} & \mathbf{b} - \mathbf{A}\mathbf{w}_{TD} = \mathbf{0} \\ \Rightarrow \quad & \mathbf{b} = \mathbf{A}\mathbf{w}_{TD} \\ \Rightarrow \quad & \mathbf{w}_{TD} \doteq \mathbf{A}^{-1}\mathbf{b}. \end{aligned} \tag{9.12}$$

This quantity is called the *TD fixed point*. In fact linear semi-gradient TD(0) converges to this point. Some of the theory proving its convergence, and the existence of the inverse above, is given in the box.

Proof of Convergence of Linear TD(0)

What properties assure convergence of the linear TD(0) algorithm (9.9)? Some insight can be gained by rewriting (9.10) as

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = (\mathbf{I} - \alpha\mathbf{A})\mathbf{w}_t + \alpha\mathbf{b}. \tag{9.13}$$

Note that the matrix $\mathbf{A}$ multiplies the weight vector $\mathbf{w}_t$ and not $\mathbf{b}$; only $\mathbf{A}$ is important to convergence. To develop intuition, consider the special case in which $\mathbf{A}$ is a diagonal matrix. If any of the diagonal elements are negative, then the corresponding diagonal element of $\mathbf{I} - \alpha\mathbf{A}$ will be greater than one, and the corresponding component of $\mathbf{w}_t$ will be amplified, which will lead to divergence if continued. On the other hand, if the diagonal elements of $\mathbf{A}$ are all positive, then α can be chosen smaller than one over the largest of them, such that $\mathbf{I} - \alpha\mathbf{A}$ is diagonal with all diagonal elements between 0 and 1. In this case the first term of the update tends to shrink $\mathbf{w}_t$, and stability is assured. In general case, $\mathbf{w}_t$ will be reduced toward zero whenever $\mathbf{A}$ is *positive definite*, meaning $y^\top \mathbf{A} y > 0$ for real vector y. Positive definiteness also ensures that the inverse $\mathbf{A}^{-1}$ exists.

For linear TD(0), in the continuing case with $\gamma < 1$, the $\mathbf{A}$ matrix (9.11) can be written

$$\begin{aligned}
\mathbf{A} &= \sum_s \mu(s) \sum_a \pi(a|s) \sum_{r,s'} p(r, s' \,|\, s, a) \mathbf{x}(s) \big(\mathbf{x}(s) - \gamma \mathbf{x}(s')\big)^\top \\
&= \sum_s \mu(s) \sum_{s'} p(s' \,|\, s) \mathbf{x}(s) \big(\mathbf{x}(s) - \gamma \mathbf{x}(s')\big)^\top \\
&= \sum_s \mu(s) \mathbf{x}(s) \Big(\mathbf{x}(s) - \gamma \sum_{s'} p(s' \,|\, s) \mathbf{x}(s')\Big)^\top \\
&= \mathbf{X}^\top \mathbf{D}(\mathbf{I} - \gamma \mathbf{P}) \mathbf{X},
\end{aligned}$$

where $\mu(s)$ is the stationary distribution under π, $p(s' \,|\, s)$ is the probability of transition from s to s' under policy π, $\mathbf{P}$ is the $|\mathcal{S}| \times |\mathcal{S}|$ matrix of these probabilities, $\mathbf{D}$ is the $|\mathcal{S}| \times |\mathcal{S}|$ diagonal matrix with the $\mu(s)$ on its diagonal, and $\mathbf{X}$ is the $|\mathcal{S}| \times d$ matrix with $\mathbf{x}(s)$ as its rows. From here it is clear that the inner matrix $\mathbf{D}(\mathbf{I} - \gamma \mathbf{P})$ is key to determining the positive definiteness of $\mathbf{A}$.

For a key matrix of this type, positive definiteness is assured if all of its columns sum to a nonnegative number. This was shown by Sutton (1988, p. 27) based on two previously established theorems. One theorem says that any matrix $\mathbf{M}$ is positive definite if and only if the symmetric matrix $\mathbf{S} = \mathbf{M} + \mathbf{M}^\top$ is positive definite (Sutton 1988, appendix). The second theorem says that any symmetric real matrix $\mathbf{S}$ is positive definite if all of its diagonal entries are positive and greater than the sum of the corresponding off-diagonal entries (Varga 1962, p. 23). For our key matrix, $\mathbf{D}(\mathbf{I} - \gamma \mathbf{P})$, the diagonal entries are positive and the off-diagonal entries are negative, so all we have to show is that each row sum plus the corresponding column sum is positive. The row sums are all positive because $\mathbf{P}$ is a stochastic matrix and $\gamma < 1$. Thus it only remains to show that the column sums are nonnegative. Note that the row vector of the column sums of any matrix $\mathbf{M}$ can be written as $\mathbf{1}^\top \mathbf{M}$, where $\mathbf{1}$ is the column vector with all components equal to 1. Let $\boldsymbol{\mu}$ denote the $|\mathcal{S}|$-vector of the $\mu(s)$, where $\boldsymbol{\mu} = \mathbf{P}^\top \boldsymbol{\mu}$ by virtue of μ being the stationary distribution. The column sums of our key matrix, then, are:

$$\begin{aligned}
\mathbf{1}^\top \mathbf{D}(\mathbf{I} - \gamma \mathbf{P}) &= \boldsymbol{\mu}^\top (\mathbf{I} - \gamma \mathbf{P}) \\
&= \boldsymbol{\mu}^\top - \gamma \boldsymbol{\mu}^\top \mathbf{P} \\
&= \boldsymbol{\mu}^\top - \gamma \boldsymbol{\mu}^\top && \text{(because } \boldsymbol{\mu} \text{ is the stationary distribution)} \\
&= (1 - \gamma) \boldsymbol{\mu},
\end{aligned}$$

all components of which are positive. Thus, the key matrix and its $\mathbf{A}$ matrix are positive definite, and on-policy TD(0) is stable. (Additional conditions and a schedule for reducing α over time are needed to prove convergence with probability one.)

At the TD fixed point, it has also been proven (in the continuing case) that the MSVE is within a bounded expansion of the lowest possible error:

$$\text{MSVE}(\mathbf{w}_{TD}) \leq \frac{1}{1 - \gamma} \min_{\mathbf{w}} \text{MSVE}(\mathbf{w}). \tag{9.14}$$

That is, the asymptotic error of the TD method is no more than $\frac{1}{1-\gamma}$ times the smallest possible error, that attained in the limit by the Monte Carlo method. Because γ is often near one, this expansion factor can be quite large, so there is substantial potential loss in asymptotic performance with the TD method. On the other hand, recall that the TD methods are often of vastly reduced variance compared to Monte Carlo methods, and thus faster, as we saw in Chapters 6 and 7. Which method will be best

depends on the nature of the approximation and problem, and on how long learning contiunues.

A bound analogous to (9.14) applies to other on-policy bootstrapping methods as well. For example, linear semi-gradient DP (Eq. 9.7 with $U_t \doteq \sum_a \pi(a|S_t) \sum_{s',r} p(s',r\,|\,S_t,a)[r+\gamma\hat{v}(s',\mathbf{w}_t)]$) with updates according to the on-policy distribution will also converge to the TD fixed point. One-step semi-gradient *action-value* methods, such as semi-gradient Sarsa(0) covered in the next chapter converge to an analogous fixed point and an analogous bound. For episodic tasks, there is a slightly different but related bound (see Bertsekas and Tsitsiklis, 1996). There are also a few technical conditions on the rewards, features, and decrease in the step-size parameter, which we have omitted here. The full details can be found in the original paper (Tsitsiklis and Van Roy, 1997).

Critical to the these convergence results is that states are updated according to the on-policy distribution. For other update distributions, bootstrapping methods using function approximation may actually diverge to infinity. Examples of this and a discussion of possible solution methods are given in Chapter 11.

Example 9.2: Bootstrapping on the 1000-state Random Walk State aggregation is a special case of linear function approximation, so let's return to the 1000-state random walk to illustrate some of the observations made in this chapter. The left panel of Figure 9.2 shows the final value function learned by the semi-gradient TD(0) algorithm (page 166) using the same state aggregation as in Example 9.1. We see that the near-asymptotic TD approximation is indeed farther from the true values than the Monte Carlo approximation shown in Figure 9.1.

Nevertheless, TD methods retain large potential advantages in learning rate, and generalize Monte Carlo methods, as we investigated fully with the multi-step TD methods of Chapter 7. The right panel of Figure 9.2 shows results with an n-step semi-gradient TD method using state aggregation and the 1000-state random walk that are strikingly similar to those we obtained earlier with tabular methods and the 19-state random walk. To obtain such quantitatively similar results we switched the state aggregation to 20 groups of 50 states each. The 20 groups are then quantitatively close to the 19 states of the tabular problem. In particular, the state transitions of at-most 100 states to the right or left, or 50 states on average, were quantitively analogous to the single-state state transitions of the tabular system. To complete the match, we use here the same performance measure—an unweighted average of the RMS error over all states and over the first 10 episodes—rather than a MSVE objective as is otherwise more appropriate when using function approximation.

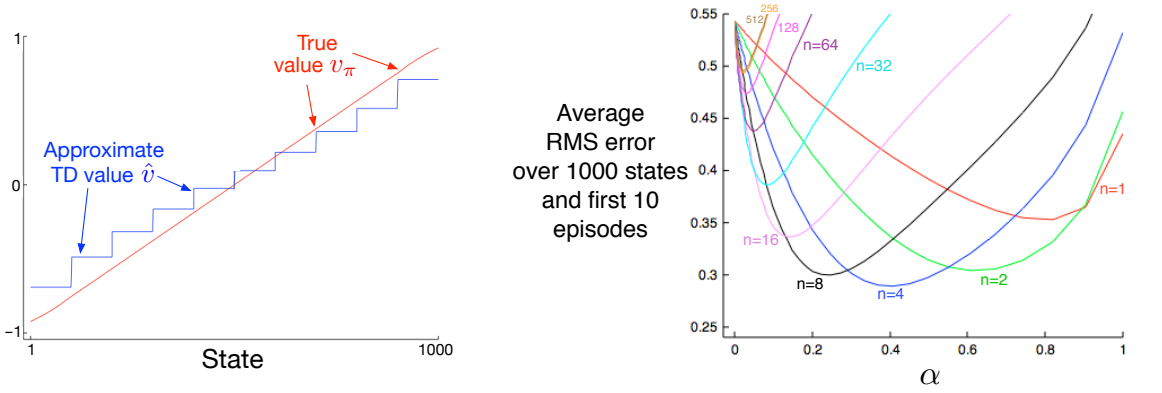


Figure 9.2: Bootstrapping with state aggregation on the 1000-state random walk task. Left: Asymptotic values of semi-gradient TD are worse than the asymptotic Monte Carlo values in Figure 9.1. Right: Performance of n-step methods with state-aggregation are strikingly similar to those with tabular representations (cf. Figure 7.2). ■

The semi-gradient n-step TD algorithm we used in this example is the natural extension of the tabular n-step TD algorithm presented in Chapter 7 to semi-gradient function approximation. The key equation, analogous to (7.2), is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha \left[G_{t:t+n} - \hat{v}(S_t, \mathbf{w}_{t+n-1}) \right] \nabla \hat{v}(S_t, \mathbf{w}_{t+n-1}), \qquad 0 \leq t < T, \tag{9.15}$$

where the n-step return is generalized from (7.1) to

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}, \mathbf{w}_{t+n-1}), \quad 0 \leq t \leq T - n. \tag{9.16}$$

Pseudocode for the complete algorithm is given in the box below.

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal}, \cdot) = 0$
Parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations (S_t and R_t) can take their index mod n

Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$ $(G_{\tau:\tau+n})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G - \hat{v}(S_\tau, \mathbf{w}) \right] \nabla \hat{v}(S_\tau, \mathbf{w})$
  Until $\tau = T - 1$

9.5 Feature Construction for Linear Methods

Linear methods are interesting because of their convergence guarantees, but also because in practice they can be very efficient in terms of both data and computation. Whether or not this is so depends critically on how the states are represented in terms of the features, which we investigate in this large section. Choosing features appropriate to the task is an important way of adding prior domain knowledge to reinforcement learning systems. Intuitively, the features should correspond to the natural features of the task, those along which generalization is most appropriate. If we are valuing geometric objects, for example, we might want to have features for each possible shape, color, size, or function. If we are valuing states of a mobile robot, then we might want to have features for locations, degrees of remaining battery power, recent sonar readings, and so on.

In general, we also need features for combinations of these natural qualities. This is because the linear form prohibits the representation of interactions between features, such as the presence of feature i being good only in the absence of feature j. For example, in the pole-balancing task (Example 3.4),

a high angular velocity may be either good or bad depending on the angular position. If the angle is high, then high angular velocity means an imminent danger of falling—a bad state—whereas if the angle is low, then high angular velocity means the pole is righting itself—a good state. In cases with such interactions one needs to introduce features for combinations of feature values when using linear function approximation methods. In the following subsections we consider a variety of general ways of doing this.

9.5.1 Polynomials

For multi-dimensional continuous state spaces, function approximation for reinforcement learning has much in common with the familiar tasks of interpolation and regression, which aim to define functions between and/or beyond given samples of function values. Various families of polynomials commonly used for these tasks can also be used in reinforcement learning. Here we discuss only the most basic polynomial family.

Suppose a reinforcement learning problem's state space is two-dimensional so that each state is a real vector $s = (s_1, s_2)^\top$. You might choose to represent each s with the feature vector $(1, s_1, s_2, s_1 s_2)^\top$ in order to take the interaction of the state variables into account by weighting the product $s_1 s_2$ in an appropriate way. Or you might choose to use feature vectors like $(1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1 s_2^2, s_1^2 s_2, s_1^2 s_2^2)^\top$ to take more complex interactions into account. Using these features means that functions are approximated as multi-dimensional quadratic functions—even though the approximation is still linear in the weights that have to be learned.

These example feature vectors are the result of selecting sets of polynomial basis functions, which are defined for any dimension and can encompass highly-complex interactions among the state variables:

> For d state variables taking real values, every state s is a d-dimensional vector $(s_1, s_2, \ldots, s_d)^\top$ of real numbers. Each d-dimensional polynomial basis function x_i can be written as
>
> $$x_i(s) = \Pi_{j=1}^{d} s_j^{c_{i,j}}, \tag{9.17}$$
>
> where each $c_{i,j}$ is an integer in the set $\{0, 1, \ldots, N\}$ for an integer $N \geq 0$. These functions make up the order-N polynomial basis, which contains $(N+1)^d$ different functions.

Higher-order polynomial bases allow for more accurate approximations of more complicated functions. But because the number of functions in an order-N polynomial basis grows exponentially with the state space dimension (for $N > 0$), it is generally necessary to select a subset of them for function approximation. This can be done using prior beliefs about the nature of the function to be approximated, and some automated selection methods developed for polynomial regression can be adapted to deal with the incremental and nonstationary nature of reinforcement learning.

Exercise 9.1 Why does (9.17) define $(N+1)^d$ distinct functions for dimension d? □

Exercise 9.2 Give N and the $c_{i,j}$ defining the basis functions that produce feature vectors $(1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1 s_2^2, s_1^2 s_2, s_1^2 s_2^2)^\top$. □

9.5.2 Fourier Basis

Another linear function approximation method is based on the time-honored Fourier series, which expresses periodic functions as a weighted sum of sine and cosine basis functions of different frequencies. (A function f is periodic if $f(x) = f(x+T)$ for all x and some period T.) The Fourier series and the more general Fourier transform are widely used in applied sciences because—among many other reasons—if a function to be approximated is known, then the basis function weights are given by simple formulae

and, further, with enough basis functions essentially any function can be approximated as accurately as desired. In reinforcement learning, where the functions to be approximated are unknown, Fourier basis functions are of interest because they are easy to use and can perform well in a range of reinforcement learning problems. Konidaris, Osentoski, and Thomas (2011) presented the Fourier basis in a simple form suitable for reinforcement learning problems with multi-dimensional continuous state spaces and functions that do not have to be periodic.

First consider the one-dimensional case. The usual Fourier series representation of a function of one dimension having period T represents the function as a linear combination of sine and cosine functions that are each periodic with periods that evenly divide T (in other words, whose frequencies are integer multiples of a fundamental frequency $1/T$). But if you are interested in approximating an aperiodic function defined over a bounded interval, you can use these Fourier basis functions with T set to the length the interval. The function of interest is then just one period of the periodic linear combination of the sine and cosine basis functions.

Furthermore, if you set T to twice the length of the interval of interest and restrict attention to the approximation over the half interval $[0, T/2]$, you can use just the cosine basis functions. This is possible because you can represent any *even* function, that is, any function that is symmetric about the origin, with just the cosine basis functions. So any function over the half-period $[0, T/2]$ can be approximated as closely as desired with enough cosine basis functions. (Saying "any function" is not exactly correct because the function has to be mathematically well-behaved, but we skip this technicality here.) Alternatively, it is possible to use just the sine basis functions, linear combinations of which are always *odd* functions, that is functions that are anti-symmetric about the origin. But it is generally better to keep just the cosine basis functions because "half-even" functions tend to be easier to approximate than "half-odd" functions since the latter are often discontinuous at the origin.

Following this logic and letting $T = 2$ so that the functions are defined over the half-T interval $[0, 1]$, the one-dimensional order-N Fourier cosine basis consists of the $N + 1$ functions

$$x_i(s) = \cos(i\pi s), \quad s \in [0, 1],$$

for $i = 0, \ldots, N$. Figure 9.3 shows one-dimensional Fourier cosine basis functions x_i, for $i = 1, 2, 3, 4$; x_0 is a constant function.

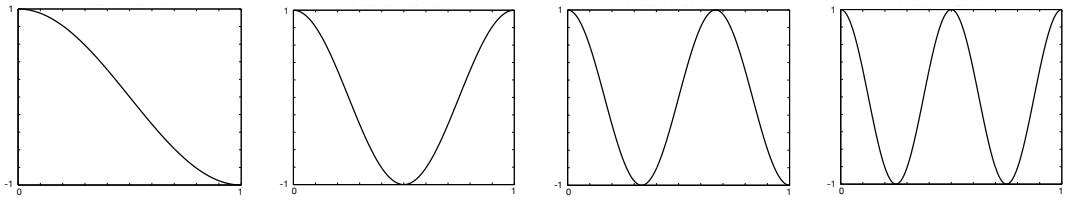


Figure 9.3: One-dimensional Fourier cosine basis functions x_i, $i = 1, 2, 3, 4$, for approximating functions over the interval $[0, 1]$; x_0 is a constant function. After Konidaris et al. (2011).

This same reasoning applies to the Fourier cosine series approximation in the multi-dimensional case as described in the box below.

For a state space that is the d-dimensional unit hypercube with the origin in one corner, states are real vectors $s = (s_1, \ldots, s_d)^\top$, $s_i \in [0, 1]$. Each function in the order-N Fourier cosine basis can be written

$$x_i(s) = \cos(\pi c^i \cdot s), \tag{9.18}$$

where $c^i = (c^i_1, \ldots, c^i_d)^\top$, with $c^i_j \in \{0, \ldots, N\}$ for $j = 1, \ldots, d$ and $i = 0, \ldots, (N + 1)^d$. This defines a function for each of the $(N + 1)^d$ possible integer vectors c^i. The dot-product $c^i \cdot s$ has

the effect of assigning an integer in $\{0, \ldots, N\}$ to each dimension. As in the one-dimensional case, this integer determines the function's frequency along that dimension. The basis functions can of course be shifted and scaled to suit the bounded state space of a particular application.

As an example, consider the $d = 2$ case in which $s = (s_1, s_2)$, where each $c^i = (c_1^i, c_2^i)^\top$. Figure 9.4 shows a selection of six Fourier cosine basis functions, each labeled by the vector c^i that defines it (s_1 is the horizontal axis and c^i is shown as a row vector with the index i omitted). Any zero in c means the function is constant along that dimension. So if $c = (0, 0)$, the function is constant over both dimensions; if $c = (c_1, 0)$ the function is constant over the second dimension and varies over the first with frequency depending on c_1; and similarly, for $c = (0, c_2)$. When $c = (c_1, c_2)$ with neither $c_j = 0$, the basis function varies along both dimensions and represents an interaction between the two state variables. The values of c_1 and c_2 determine the frequency along each dimension, and their ratio gives the direction of the interaction.

Konidaris et al. (2011) found that when using Fourier cosine basis functions with a learning algorithm such as (9.7), semi-gradient TD(0), or semi-gradient Sarsa, it is helpful to use a different step-size parameter for each basis function. If α is the basic step-size parameter, they suggest setting the step-size parameter for basis function x_i to $\alpha_i = \alpha/\sqrt{(c_1^i)^2 + \cdots + (c_d^i)^2}$ (except when each $c_j^i = 0$, in which case $\alpha_i = \alpha$).

Fourier cosine basis functions with Sarsa were found to produce good performance compared to several other collections of basis functions, including polynomial and radial basis functions, on several reinforcement learning tasks. Not surprisingly, however, Fourier basis functions have trouble with discontinuities because it is difficult to avoid "ringing" around points of discontinuity unless very high frequency basis functions are included.

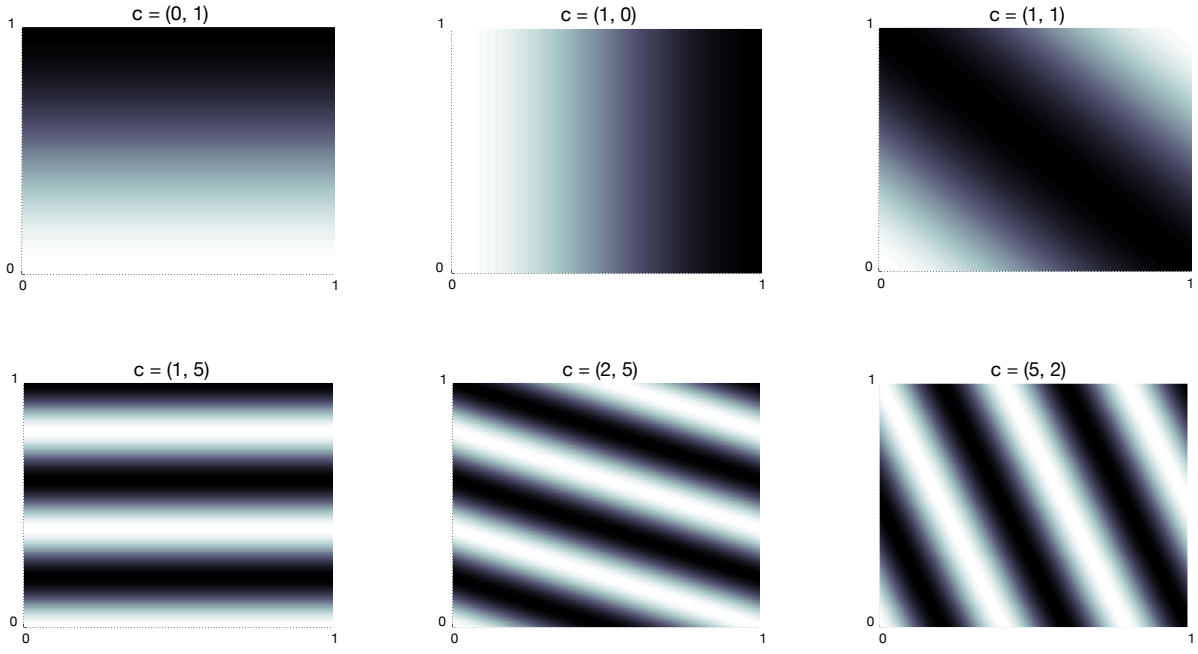


Figure 9.4: A selection of six two-dimensional Fourier cosine basis functions, each labeled by the vector c^i that defines it (s_1 is the horizontal axis, and c^i is shown as a row vector with the index i omitted). After Konidaris et al. (2011).

As is true for polynomial approximation, the number of basis functions in the order-N Fourier cosine basis grows exponentially with the state space dimension. This makes it necessary to select a subset of these functions if the state space has high dimension (e.g., $d > 5$). This can be done using prior beliefs about the nature of the function to be approximated, and some automated selection methods can be adapted to deal with the incremental and nonstationary nature of reinforcement learning. Advantages of Fourier basis functions in this regard are that it is easy to select functions by setting the c^i vectors to account for suspected interactions among the state variables, and by limiting the values in the c^j vectors so that the approximation can filter out high frequency components considered to be noise.

Figure 9.5 shows learning curves comparing the Fourier and polynomial bases on the 1000-state random walk example. In general, we do not recommend using the polynomial basis for online learning.

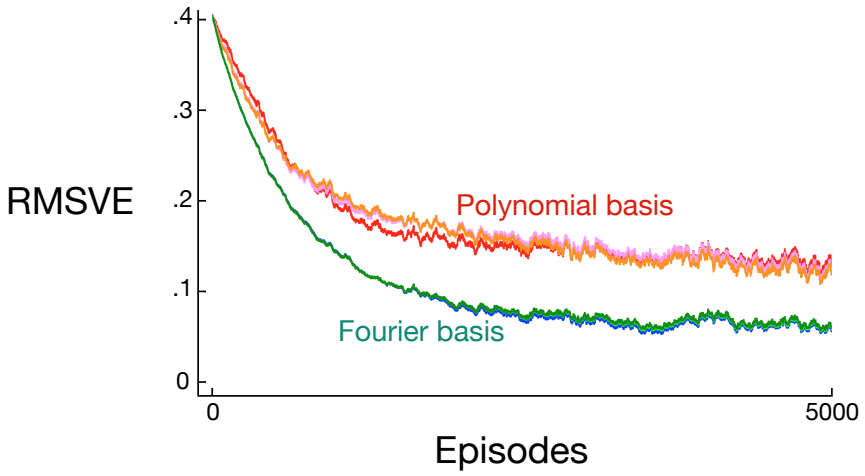


Figure 9.5: Fourier basis vs polynomials on the 1000-state random walk. Shown are learning curves for the gradient Monte Carlo method with Fourier and polynomial bases of order 5, 10, and 20. The step-size parameters were roughly optimized for each case: $\alpha = 0.0001$ for the polynomial basis and $\alpha = 0.00005$ for the Fourier basis.

Exercise 9.3 Why does (9.18) define $(N+1)^d$ distinct functions for dimension d? □

9.5.3 Coarse Coding

Consider a task in which the state set is continuous and two-dimensional. A state in this case is a point in 2-space, a vector with two real components. One kind of feature for this case is those corresponding to *circles* in state space, as shown in Figure 9.6. If the state is inside a circle, then the corresponding feature has the value 1 and is said to be *present*; otherwise the feature is 0 and is said to be *absent*. This kind of 1–0-valued feature is called a *binary feature*. Given a state, which binary features are present indicate within which circles the state lies, and thus coarsely code for its location. Representing a state with features that overlap in this way (although they need not be circles or binary) is known as *coarse coding*.

Assuming linear gradient-descent function approximation, consider the effect of the size and density of the circles. Corresponding to each circle is a single weight (a component of $\mathbf{w}$) that is affected by learning. If we train at one state, a point in the space, then the weights of all circles intersecting that state will be affected. Thus, by (9.8), the approximate value function will be affected at all states within the union of the circles, with a greater effect the more circles a point has "in common" with the state, as shown in Figure 9.6. If the circles are small, then the generalization will be over a short distance, as

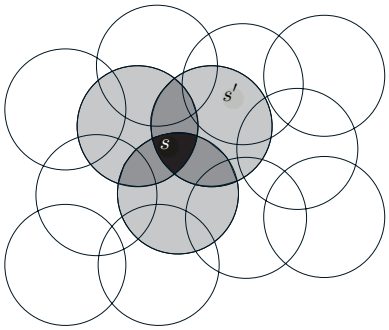


Figure 9.6: Coarse coding. Generalization from state s to state s' depends on the number of their features whose receptive fields (in this case, circles) overlap. These states have one feature in common, so there will be slight generalization between them.

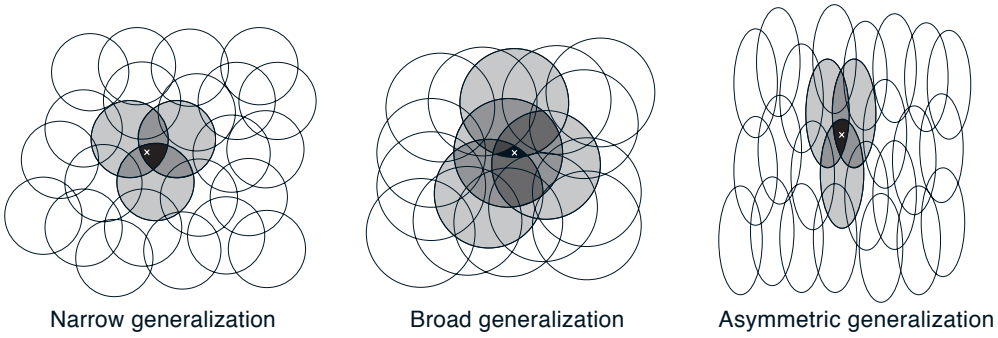


Figure 9.7: Generalization in linear function approximation methods is determined by the sizes and shapes of the features' receptive fields. All three of these cases have roughly the same number and density of features.

in Figure 9.7a, whereas if they are large, it will be over a large distance, as in Figure 9.7b. Moreover, the shape of the features will determine the nature of the generalization. For example, if they are not strictly circular, but are elongated in one direction, then generalization will be similarly affected, as in Figure 9.7c.

Features with large receptive fields give broad generalization, but might also seem to limit the learned function to a coarse approximation, unable to make discriminations much finer than the width of the receptive fields. Happily, this is not the case. Initial generalization from one point to another is indeed controlled by the size and shape of the receptive fields, but acuity, the finest discrimination ultimately possible, is controlled more by the total number of features.

Example 9.3: Coarseness of Coarse Coding This example illustrates the effect on learning of the size of the receptive fields in coarse coding. Linear function approximation based on coarse coding and (9.7) was used to learn a one-dimensional square-wave function (shown at the top of Figure 9.8). The values of this function were used as the targets, U_t. With just one dimension, the receptive fields were intervals rather than circles. Learning was repeated with three different sizes of the intervals: narrow, medium, and broad, as shown at the bottom of the figure. All three cases had the same density of features, about 50 over the extent of the function being learned. Training examples were generated uniformly at random over this extent. The step-size parameter was $\alpha = \frac{0.2}{m}$, where m is the number of features that were present at one time. Figure 9.8 shows the functions learned in all three cases over the course of learning. Note that the width of the features had a strong effect early in learning. With broad features, the generalization tended to be broad; with narrow features, only the close neighbors of

each trained point were changed, causing the function learned to be more bumpy. However, the final function learned was affected only slightly by the width of the features. Receptive field shape tends to have a strong effect on generalization but little effect on asymptotic solution quality.

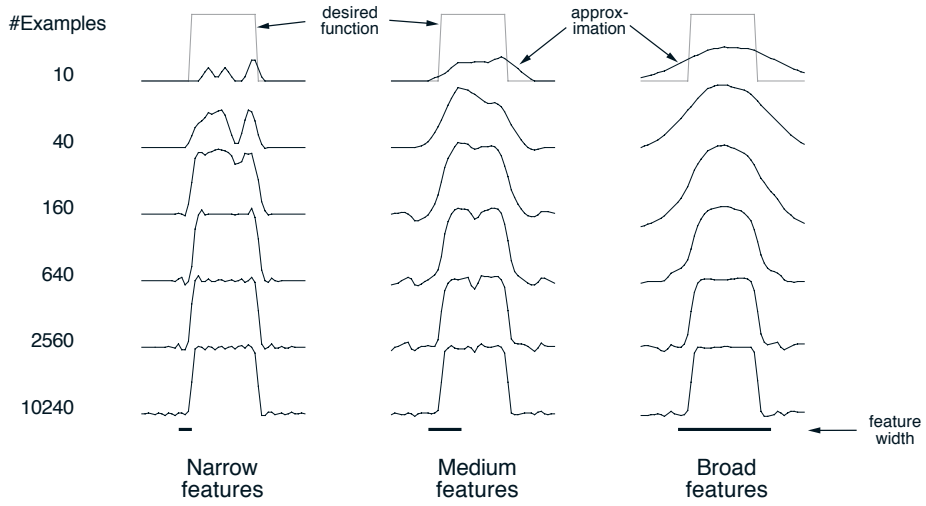


Figure 9.8: Example of feature width's strong effect on initial generalization (first row) and weak effect on asymptotic accuracy (last row). ■

9.5.4 Tile Coding

Tile coding is a form of coarse coding for multi-dimensional continuous spaces that is flexible and computationally efficient. It may be the most practical feature representation for modern sequential digital computers. Open-source software is available for many kinds of tile coding.

In tile coding the receptive fields of the features are grouped into partitions of the input space. Each such partition is called a *tiling*, and each element of the partition is called a *tile*. For example, the simplest tiling of a two-dimensional state space is a uniform grid such as that shown on the left side of Figure 9.9. The tiles or receptive field here are squares rather than the circles in Figure 9.6. If just this

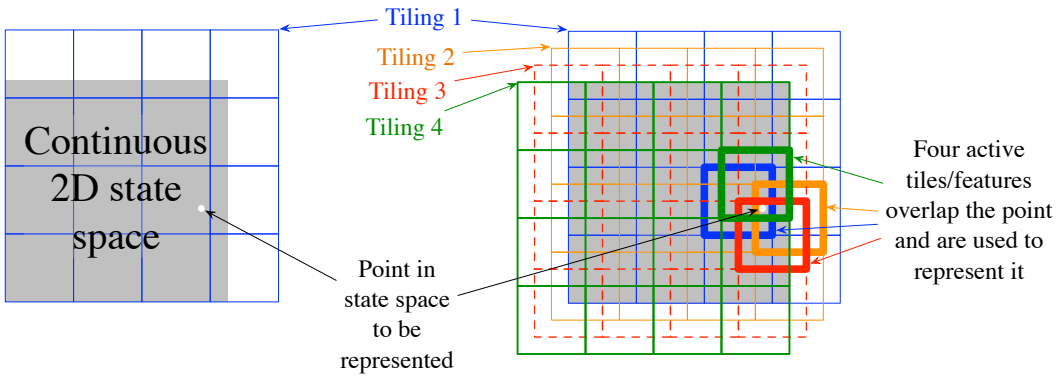


Figure 9.9: Multiple, overlapping grid-tilings on a limited two-dimensional space. These tilings are offset from one another by a uniform amount in each dimension.

single tiling were used, then the state indicated by the white spot would be represented by the single feature whose tile it falls within; generalization would be complete to all states within the same tile and nonexistent to states outside it. With just one tiling, we would not have coarse coding by just a case of state aggregation.

To get the strengths of coarse coding requires overlapping receptive fields, and by definition the tiles of a partition do not overlap. To get true coarse coding with tile coding, multiple tilings are used, each offset by a fraction of a tile width. A simple case with four tilings is shown on the right side of Figure 9.9. Every state, such as that indicated by the white spot, falls in exactly one tile in each of the four tilings. These four tiles correspond to four features that become active when the state occurs. Specifically, the feature vector $\mathbf{x}(s)$ has one component for each tile in each tiling. In this example there are $4 \times 4 \times 4 = 64$ components, all of which will be 0 except for the four corresponding to the tiles that s falls within. Figure 9.10 shows the advantage of multiple offset tilings (coarse coding) over a single tiling on the 1000-state random walk example.

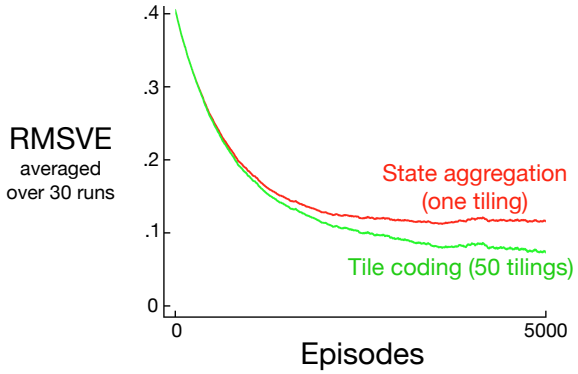


Figure 9.10: Why we use coarse coding. Shown are learning curves on the 1000-state random walk example for the gradient Monte Carlo algorithm with a single tiling and with multiple tilings. The space of 1000 states was treated as a single continuous dimension, covered with tiles each 200 states wide. The multiple tilings were offset from each other by 4 states. The step-size parameter was set so that the initial learning rate in the two cases was the same, $\alpha = 0.0001$ for the single tiling and $\alpha = 0.0001/50$ for the 50 tilings.

An immediate practical advantage of tile coding is that, because it works with partitions, the overall number of features that are active at one time is the same for any state. Exactly one feature is present in each tiling, so the total number of features present is always the same as the number of tilings. This allows the step-size parameter, α, to be set in an easy, intuitive way. For example, choosing $\alpha = \frac{1}{m}$, where m is the number of tilings, results in exact one-trial learning. If the example $s \mapsto v$ is trained on, then whatever the prior estimate, $\hat{v}(s,\mathbf{w}_t)$, the new estimate will be $\hat{v}(s,\mathbf{w}_{t+1}) = v$. Usually one wishes to change more slowly than this, to allow for generalization and stochastic variation in target outputs. For example, one might choose $\alpha = \frac{1}{10m}$, in which case the estimate for the trained state would move one-tenth of the way to the target in one update, and neighboring states will be moved less, proportional to the number of tiles they have in common.

Tile coding also gains computational advantages from its use of binary feature vectors. Because each component is either 0 or 1, the weighted sum making up the approximate value function (9.8) is almost trivial to compute. Rather than performing n multiplications and additions, one simply computes the indices of the $m \ll n$ active features and then adds up the m corresponding components of the weight vector.

Generalization occurs to states other than the one trained if those states fall within any of the same tiles, proportional to the number of tiles in common. Even the choice of how to offset the tilings from each other affects generalization. If they are offset uniformly in each dimension, as they were in

Figure 9.9, then different states can generalize in qualitatively different ways, as shown below in the upper half of Figure 9.11. Each of the eight subfigures show the pattern of generalization from a trained state to nearby points. In this example their are eight tilings, thus 64 subregions within a tile that generalize distinctly, but all according to one of these eight patterns. Note how uniform offsets result in a strong effect along the diagonal in many patterns. These artifacts can be avoided if the tilings are offset asymmetrically, as shown in the lower half of the figure. These lower generalization patterns are better because they are all well centered on the trained state with no obvious asymmetries.

Tilings in all cases are offset from each other by a fraction of a tile width in each dimension. If w denotes the tile width and k the number of tilings, then $\frac{w}{k}$ is a fundamental unit. Within small squares $\frac{w}{k}$ on a side, all states activate the same tiles, have the same feature representation, and the same approximated value. If a state is moved by $\frac{w}{k}$ in any cartesian direction, the feature representation changes by one component/tile. Uniformly offset tilings are offset from each other by exactly this unit distance. For a two-dimensional space, we say that each tiling is offset by the displacement vector $(1, 1)$, meaning that it is offset from the previous tiling by $\frac{w}{k}$ times this vector. In these terms, the asymmetrically offset tilings shown in the lower part of Figure 9.11 are offset by a displacement vector of $(1, 3)$.

Extensive studies have been made of the effect of different displacement vectors on the generalization of tile coding (Parks and Militzer, 1991; An, 1991; An, Miller and Parks, 1991; Miller, Glanz and Carter, 1991), assessing their homegeneity and tendency toward diagonal artifacts like those seen for the $(1, 1)$ displacement vectors. Based on this work, Miller and Glanz (1996) recommend using displacement vectors consisting of the first odd integers. In particular, for a continuous space of dimension d, a good

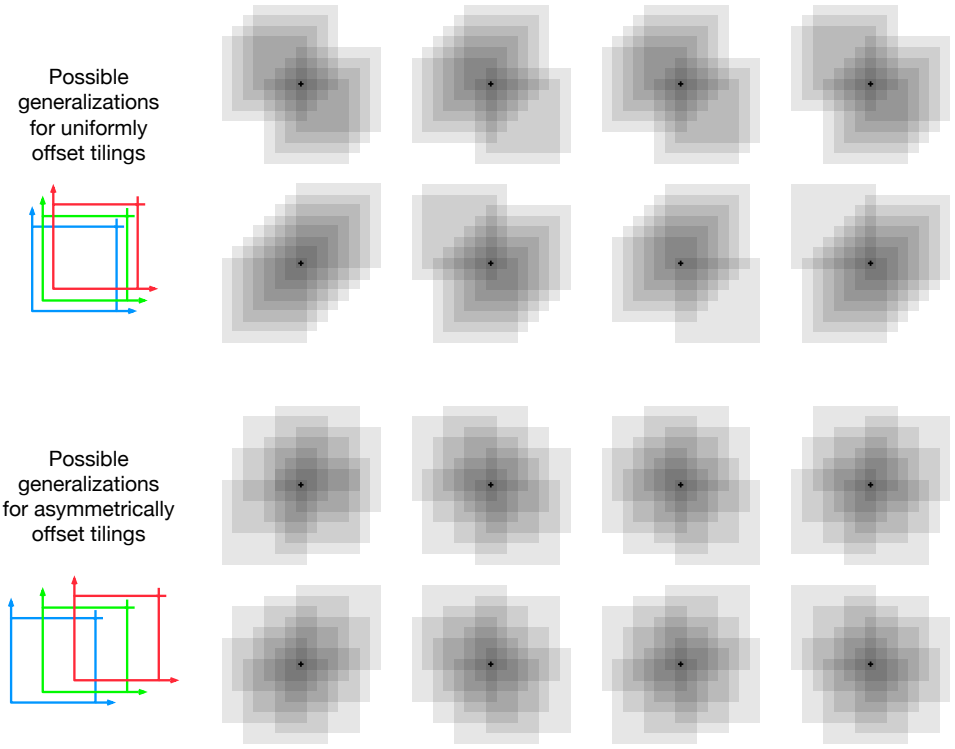


Figure 9.11: Why tile asymmetrical offsets are preferred in tile coding. Shown is the strength of generalization from a trained state, indicated by the small black plus, to nearby states, for the case of eight tilings. If the tilings are uniformly offset (above), then there are diagonal artifacts and substantial variations in the generalization, whereas with asymmetrically offset tilings the generalization is more spherical and homogeneous.

choice is to use the first odd integers $(1, 3, 5, 7, \ldots, 2d-1)$, with k (the number of tilings) set to an integer power of 2 greater than or equal to $4d$. This is what we have done to produce the tilings in the lower half of Figure 9.11, in which $d = 2$, $k = 2^3 \geq 4d$, and the displacement vector is $(1, 3)$. In a three-dimensional case, the first four tilings would be offset in total from a base position by $(0, 0, 0)$, $(1, 3, 5)$, $(2, 6, 10)$, and $(3, 9, 15)$. Open-source software that can efficiently make tilings like this for any d is readily available.

In choosing a tiling strategy, one has to pick the number of the tilings and the shape of the tiles. The number of tilings, along with the size of the tiles, determines the resolution or fineness of the asymptotic approximation, as in general coarse coding and illustrated in Figure 9.8. The shape of the tiles will determine the nature of generalization as in Figure 9.7. Square tiles will generalize roughly equally in each dimension as indicated in Figure 9.11 (lower). Tiles that are elongated along one dimension, such as the stripe tilings in Figure 9.12b, will promote generalization along that dimension. The tilings in Figure 9.12b are also denser and thinner on the left, promoting discrimination along the horizonal dimension at lower values along that dimension. The diagonal stripe tiling in Figure 9.12c will promote generalization along one diagonal. In higher dimensions, axis-aligned stripes correspond to ignoring some of the dimensions in some of the tilings, that is, to hyperplanar slices. Irregular tilings such as shown in Figure 9.12a are also possible, though rare in practice and beyond the standard software.

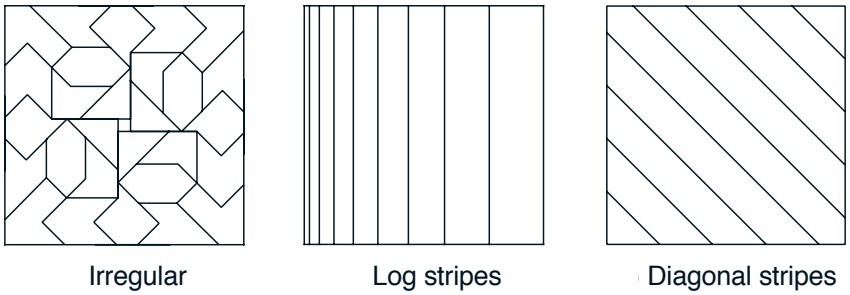
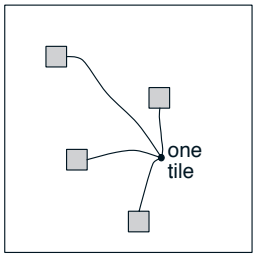


Figure 9.12: Tilings need not be grids. They can be arbitrarily shaped and non-uniform, while still in many cases being computationally efficient to compute.

In practice, it is often desirable to use different shaped tiles in different tilings. For example, one might use some vertical stripe tilings and some horizontal stripe tilings. This would encourage generalization along either dimension. However, with stripe tilings alone it is not possible to learn that a particular conjunction of horizontal and vertical coordinates has a distinctive value (whatever is learned for it will bleed into states with the same horizontal and vertical coordinates). For this one needs the conjunctive rectangular tiles such as originally shown in Figure 9.9. With multiple tilings—some horizontal, same vertical, and some conjunctive—one can get everything: a preference for generalizing along each dimension, yet the ability to learn specific values for conjunctions (see Section 16.3 for a case study using this). The choice of tilings determines generalization, and until this choice can be effectively automated, it is important that tile coding enables the choice to be made flexibly and in a way that makes sense to people.

Another useful trick for reducing memory requirements is *hashing*—a consistent pseudo-random collapsing of a large tiling into a much smaller set of tiles. Hashing produces tiles consisting of noncontiguous, disjoint regions randomly spread throughout the state space, but that still form an exhaustive partition. For example, one tile might consist of the four subtiles shown to the right. Through hashing, memory requirements are often reduced by large factors with little loss of performance. This is possible because high resolution is needed in only a small fraction of the state space. Hashing frees us from the



curse of dimensionality in the sense that memory requirements need not be exponential in the number of dimensions, but need merely match the real demands of the task. Good open-source implementations of tile coding, including hashing, are widely available.

Exercise 9.4 Suppose we believe that one of two state dimensions is more likely to have an effect on the value function than is the other, that generalization should be primarily across this dimension rather than along it. What kind of tilings could be used to take advantage of this prior knowledge? □

9.5.5 Radial Basis Functions

Radial basis functions (RBFs) are the natural generalization of coarse coding to continuous-valued features. Rather than each feature being either 0 or 1, it can be anything in the interval $[0, 1]$, reflecting various *degrees* to which the feature is present. A typical RBF feature, i, has a Gaussian (bell-shaped) response $x_i(s)$ dependent only on the distance between the state, s, and the feature's prototypical or center state, c_i, and relative to the feature's width, σ_i:

$$x_i(s) \doteq \exp\left(-\frac{||s - c_i||^2}{2\sigma_i^2}\right).$$

The norm or distance metric of course can be chosen in whatever way seems most appropriate to the states and task at hand. Figure 9.13 shows a one-dimensional example with a Euclidean distance metric.

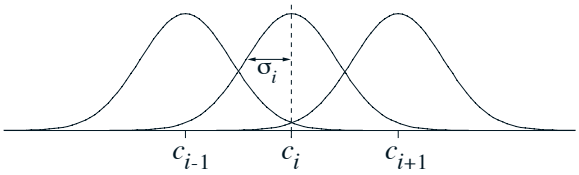


Figure 9.13: One-dimensional radial basis functions.

The primary advantage of RBFs over binary features is that they produce approximate functions that vary smoothly and are differentiable. Although this is appealing, in most cases it has no practical significance. Nevertheless, extensive studies have been made of graded response functions such as RBFs in the context of tile coding (An, 1991; Miller et al., 1991; An, Miller and Parks, 1991; Lane, Handelman and Gelfand, 1992). All of these methods require substantial additional computational complexity (over tile coding) and often reduce performance when there are more than two state dimensions. In high dimensions the edges of tiles are much more important, and it has proven difficult to obtain well controlled graded tile activations near the edges.

An *RBF network* is a linear function approximator using RBFs for its features. Learning is defined by equations (9.7) and (9.8), exactly as in other linear function approximators. In addition, some learning methods for RBF networks change the centers and widths of the features as well, bringing them into the realm of nonlinear function approximators. Nonlinear methods may be able to fit target functions much more precisely. The downside to RBF networks, and to nonlinear RBF networks especially, is greater computational complexity and, often, more manual tuning before learning is robust and efficient.

9.6 Nonlinear Function Approximation: Artificial Neural Networks

Artificial neural networks (ANNs) are widely used for nonlinear function approximation. An ANN is a network of interconnected units that have some of the properties of neurons, main component of nervous systems. ANNs have a long history, with latest advances in training deeply-layered ANNs being responsible for some of the most impressive abilities of machine learning systems, including reinforcement learning systems. In Chapter 16 we describe several stunning examples of reinforcement learning systems that use ANN function approximation.

Figure 9.14 shows a generic feedforward ANN, meaning that there are no loops in the network, that is, there are no paths within the network by which a unit's output can influence its input. The network in the figure has an output layer consisting of two output units, an input layer with four input units, and two hidden layers: layers that are neither input nor output layers. A real-valued weight is associated with each link. A weight roughly corresponds to the efficacy of a synaptic connection in a real neural network (see Section 15.1). If an ANN has at least one loop in its connections, it is a recurrent rather than a feedforward ANN. Although both feedforward and recurrent ANNs have been used in reinforcement learning, here we look only at the simpler feedforward case.

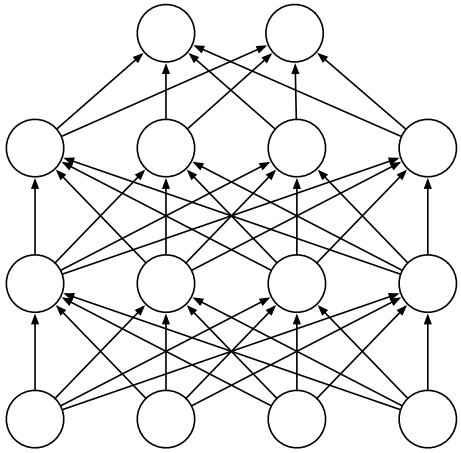


Figure 9.14: A generic feedforward neural network with four input units, two output units, and two hidden layers.

The units (the circles in Figure 9.14) are typically semi-linear units, meaning that they compute a weighted sum of their input signals and then apply to the result a nonlinear function, called the *activation function*, to produce the unit's output, or activation. Many different activation functions are used, but they are typically S-shaped, or sigmoid, functions such as the logistic function $f(x) = 1/(1 + e^{-x})$, though sometimes the rectifier nonlinearity $f(x) = \max(0, x)$ is used. A step function like $f(x) = 1$ if $x \geq \theta$, and 0 otherwise, results in a binary unit with threshold θ. It is often useful for units in different layers to use different activation functions.

The activation of each output unit of a feedforward ANN is a nonlinear function of the activation patterns over the network's input units. The functions are parameterized by the network's connection weights. An ANN with no hidden layers can represent only a very small fraction of the possible input-output functions. However an ANN with a single hidden layer having a large enough finite number of sigmoid units can approximate any continuous function on a compact region of the network's input space to any degree of accuracy (Cybenko, 1989). This is also true for other nonlinear activation functions

that satisfy mild conditions, but nonlinearity is essential: if all the units in a multi-layer feedforward ANN have linear activation functions, the entire network is equivalent to a network with no hidden layers (because linear functions of linear functions are themselves linear).

Despite this "universal approximation" property of one-hidden-layer ANNs, both experience and theory show that approximating the complex functions needed for many artificial intelligence tasks is made easier—indeed may require—abstractions that are hierarchical compositions of many layers of lower-level abstractions, that is, abstractions produced by deep architectures such as ANNs with many hidden layers. (See Bengio, 2009, for a thorough review.) The successive layers of a deep ANN compute increasingly abstract representations of the network's "raw" input, with each unit providing a feature contributing to a hierarchical representation of the overall input-output function of the network.

Creating these kinds of hierarchical representations without relying exclusively on hand-crafted features has been an enduring challenge for artificial intelligence. This is why learning algorithms for ANNs with hidden layers have received so much attention over the years. ANNs typically learn by a stochastic gradient method (Section 9.3). Each weight is adjusted in a direction aimed at improving the network's overall performance as measured by an objective function to be either minimized or maximized. In the most common supervised learning case, the objective function is the expected error, or loss, over a set of labeled training examples. In reinforcement learning, ANNs can use TD errors to learn value functions, or they can aim to maximize expected reward as in a gradient bandit (Section 2.8) or a policy-gradient algorithm (Chapter 13). In all of these cases it is necessary to estimate how a change in each connection weight would influence the network's overall performance, in other words, to estimate the partial derivative of an objective function with respect to each weight, given the current values of all the network's weights. The gradient is the vector of these partial derivatives.

The most successful way to do this for ANNs with hidden layers (provided the units have differentiable activation functions) is the backpropagation algorithm, which consists of alternating forward and backward passes through the network. Each forward pass computes the activation of each unit given the current activations of the network's input units. After each forward pass, a backward pass efficiently computes a partial derivative for each weight. (As in other stochastic gradient learning algorithms, the vector of these partial derivatives is an estimate of the true gradient.) In Section 15.10 we discuss methods for training ANNs with hidden layers that use reinforcement learning principles instead of backpropagation. These methods are less efficient than the backpropagation algorithm, but they may be closer to how real neural networks learn.

The backpropagation algorithm can produce good results for shallow networks having 1 or 2 hidden layers, but it does not work well for deeper ANNs. In fact, training a network with $k + 1$ hidden layers can actually result in poorer performance than training a network with k hidden layers, even though the deeper network can represent all the functions that the shallower network can (Bengio, 2009). Explaining results like these is not easy, but several factors are important. First, the large number of weights in a typical deep ANN makes it difficult to avoid the problem of overfitting, that is, the problem of failing to generalize correctly to cases on which the network has not been trained. Second, backpropagation does not work well for deep ANNs because the partial derivatives computed by its backward passes either decay rapidly toward the input side of the network, making learning by deep layers extremely slow, or the partial derivatives grow rapidly toward the input side of the network, making learning unstable. Methods for dealing with these problems are largely responsible for many impressive results achieved by systems that use deep ANNs.

Overfitting is a problem for any function approximation method that adjusts functions with many degrees of freedom on the basis of limited training data. It is less of a problem for on-line reinforcement learning that does not rely on limited training sets, but generalizing effectively is still an important issue. Overfitting is a problem for ANNs in general, but especially so for deep ANNs because they tend to have very large numbers of weights. Many methods have been developed for reducing overfitting. These include stopping training when performance begins to decrease on validation data different from

the training data (cross validation), modifying the objective function to discourage complexity of the approximation (regularization), and introducing dependencies among the weights to reduce the number of degrees of freedom (e.g., weight sharing).

A particularly effective method for reducing overfitting by deep ANNs is the dropout method introduced by Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014). During training, units are randomly removed from the network (dropped out) along with their connections. This can be thought of as training a large number of "thinned" networks. Combining the results of these thinned networks at test time is a way to improve generalization performance. The dropout method efficiently approximates this combination by multiplying each outgoing weight of a unit by the probability that that unit was retained during training. Srivastava et al. found that this method significantly improves generalization performance. It encourages individual hidden units to learn features that work well with random collections of other features. This increases the versatility of the features formed by the hidden units so that the network does not overly specialize to rarely-occurring cases.

Hinton, Osindero, and Teh (2006) took a major step toward solving the problem of training the deep layers of a deep ANN in their work with deep belief networks, layered networks closely related to the deep ANNs discussed here. In their method, the deepest layers are trained one at a time using an unsupervised learning algorithm. Without relying on the overall objective function, unsupervised learning can extract features that capture statistical regularities of the input stream. The deepest layer is trained first, then with input provided by this trained layer, the next deepest layer is trained, and so on, until the weights in all, or many, of the network's layers are set to values that now act as initial values for supervised learning. The network is then fine-tuned by backpropagation with respect to the overall objective function. Studies show that this approach generally works much better than backpropagation with weights initialized with random values. The better performance of networks trained with weights initialized this way could be due to many factors, but one idea is that this method places the network in a region of weight space from which a gradient-based algorithm can make good progress.

A type of deep ANN that has proven to be very successful in applications, including impressive reinforcement learning applications (Chapter 16) is the *deep convolutional network*. This type of network is specialized for processing high-dimensional data arranged in spatial arrays, such as images. It was inspired by how early visual processing works in the brain (LeCun, Bottou, Bengio and Haffner, 1998). Because of its special architecture, a deep convolutional network can be trained by backpropagation without resorting to methods like those described above to train the deep layers.

Among other techniques that make it easier to train deep ANNs, including deep convolutional networks, are *batch normalization* (Ioffe and Szegedy, 2015) and *deep residual learning* (He, Zhang, Ren, and Sun, 2016). These were used in application of reinforcement learning to playing the game of Go that we describe in Chapter 16. It has long been known that ANN learning is easier if the network input is normalized, for example, by adjusting each input variable to have zero mean and unit variance. Batch normalization for training deep ANNs normalizes the output of deep layers before they feed into the following layer. Ioffe and Szegedy (2015) used statistics from subsets, or "mini-batches," of training examples to normalize these between-layer signals to improve the learning rate of deep ANNs. Deep residual learning is simple method that can also be effective. Sometimes it is easier to learn how a function differs from the identity function than to learn the function itself. Then adding this difference, or residual function, to the input produces the desired function. In deep ANNs a block of layers can be made to learn a residual function simply by adding shortcut, or skip, connections around the block. These connections add the input to the block to its output, and no additional weights are needed. He et al. (2016) evaluated this method using deep convolutional networks with skip connections around every pair of adjacent layers, finding substantial improvement over networks without the skip connections on benchmark image classification tasks.

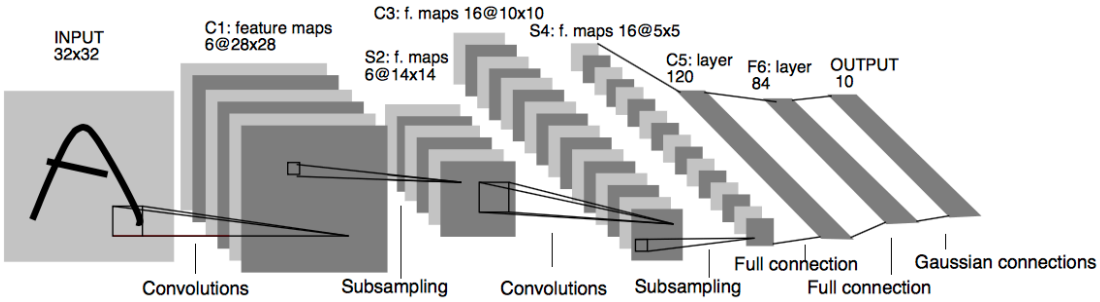


Figure 9.15: Deep Convolutional Network. Republished with permission of Proceedings of the IEEE, from Gradient-based learning applied to document recognition, LeCun, Bottou, Bengio, and Haffner, volume 86, 1998; permission conveyed through Copyright Clearance Center, Inc.

Figure 9.15 illustrates the architecture of a deep convolutional network. This instance, from LeCun et al. (1998), was designed to recognize hand-written characters. It consists of alternating convolutional and subsampling layers, followed by several fully connected final layers. Each convolutional layer produces a number of feature maps. A feature map is a pattern of activity over an array of units, where each unit performs the same operation on data in its receptive field, which is the part of the data it "sees" from the preceding layer (or from the external input in the case of the first convolutional layer). The units of a feature map are identical to one another except that their receptive fields, which are all the same size and shape, are shifted to different locations on the arrays of incoming data. Units in the same feature map share the same weights. This means that a feature map detects the same feature no matter where it is located in the input array. In the network in Figure 9.15, for example, the first convolutional layer produces 6 feature maps, each consisting of 28×28 units. Each unit in each feature map has a 5×5 receptive field, and these receptive fields overlap (in this case by four columns and five rows). Consequently, each of the 6 feature maps is specified by just 25 adjustable weights.

The subsampling layers of a deep convolutional network reduce the spatial resolution of the feature maps. Each feature map in a subsampling layer consists of units that average over a receptive field of units in the feature maps of the preceding convolutional layer. For example, each unit in each of the 6 feature maps in the first subsampling layer of the network of Figure 9.15 averages over a 2×2 non-overlapping receptive fields of a feature map produced by the first convolutional layer, resulting in six 14×14 feature maps. The subsampling layers reduce the network's sensitivity to the spatial locations of the features detected, that is, they help make the network's responses spatially invariant. This is useful because a feature detected at one place in an image is likely to be useful at other places as well.

Advances in the design and training of ANNs—of which we have only mentioned a few—all contribute to reinforcement learning. Although current reinforcement learning theory is mostly limited to methods using tabular or linear function approximation methods, the impressive performances of notable reinforcement learning applications owe much of their success to nonlinear function approximation by ANNs, in particular, by deep ANNs. The case studies we discuss in Chapter 16 that use ANNs all use deep convolutional networks, which are well suited for the spatial arrays that represent states in these problems. Other network architectures are appropriate for other types of problems, and one of the challenges of using ANNs for function approximation is finding a network architecture that works well for the problem of interest.

9.7 Least-Squares TD

In Section 9.4 we established that TD(0) with linear function approximation converges asymptotically, for appropriately decreasing step sizes, to the TD fixed point:

$$\mathbf{w}_{TD} = \mathbf{A}^{-1}\mathbf{b},$$

where

$$\mathbf{A} \doteq \mathbb{E}\big[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\big] \quad \text{and} \quad \mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t]\,.$$

Why, we might ask, must we compute this solution iteratively? This is wasteful of data! Could one not do better by computing estimates of $\mathbf{A}$ and $\mathbf{b}$, and then directly computing the TD fixed point? The Least-Squares TD algorithm, commonly known as LSTD, does exactly this. It forms the natural estimates

$$\widehat{\mathbf{A}}_t \doteq \sum_{k=0}^{t} \mathbf{x}_k(\mathbf{x}_k - \gamma\mathbf{x}_{k+1})^\top + \varepsilon\mathbf{I} \quad \text{and} \quad \widehat{\mathbf{b}}_t \doteq \sum_{k=0}^{t} R_{t+1}\mathbf{x}_k \tag{9.19}$$

(where $\varepsilon\mathbf{I}$, for some small $\varepsilon > 0$, ensures that $\widehat{\mathbf{A}}_t$ is always invertible) and then estimates the TD fixed point as

$$\mathbf{w}_{t+1} \doteq \widehat{\mathbf{A}}_t^{-1}\widehat{\mathbf{b}}_t. \tag{9.20}$$

This algorithm is the most data efficient form of linear TD(0), but it is also much more expensive computationally. Recall that semi-gradient TD(0) requires memory and per-step computation that is only $O(d)$.

How complex is LSTD? As it is written above the complexity seems to increase with t, but the two approximations in (9.19) could be implemented incrementally using the techniques we have covered earlier (e.g., in Chapter 2) so that they can be done in constant time per step. Even so, the update for $\widehat{\mathbf{A}}_t$ would involve an outer product (a column vector times a row vector) and thus would be a matrix update; its computational complexity would be $O(d^2)$, and of course the memory required to hold the $\widehat{\mathbf{A}}_t$ matrix would be $O(d^2)$.

A potentially greater problem is that our final computation (9.20) uses the inverse of $\widehat{\mathbf{A}}_t$, and the computational complexity of a general inverse computation is $O(d^3)$. Fortunately, an inverse of a matrix of our special form—a sum of outer products—can also be updated incrementally with only $O(d^2)$ computations, as

$$\begin{aligned}
\widehat{\mathbf{A}}_t^{-1} &= \Big(\widehat{\mathbf{A}}_{t-1} + \mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\Big)^{-1} && \text{(from (9.19))} \\
&= \widehat{\mathbf{A}}_{t-1}^{-1} - \frac{\widehat{\mathbf{A}}_{t-1}^{-1}\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\widehat{\mathbf{A}}_{t-1}^{-1}}{1 + (\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\widehat{\mathbf{A}}_{t-1}^{-1}\mathbf{x}_t}, && (9.21)
\end{aligned}$$

with $\widehat{\mathbf{A}}_{-1} \doteq \varepsilon\mathbf{I}$. Although the identity (9.21), known as *the Sherman-Morrison formula*, is superficially complicated, it involves only vector-matrix and vector-vector multiplications and thus is only $O(d^2)$. Thus we can store and maintain the inverse matrix $\widehat{\mathbf{A}^{-1}}_t$, and then use it in (9.20), all with only $O(d^2)$ memory and per-step computation. The complete algorithm is given in the box below.

Of course, $O(d^2)$ is still significantly more expensive than the $O(d)$ of semi-gradient TD. Whether the greater data efficiency of LSTD is worth this computational expense depends on how large d is, how important it is to learn quickly, and the expense of other parts of the system. The fact that LSTD requires no step-size parameter is sometimes also touted, but the advantage of this is probably

LSTD for estimating $\hat{v} \approx v_\pi$ ($O(d^2)$ version)

Input: feature representation $\mathbf{x}(s) \in \mathbb{R}^d$, for all $s \in \mathcal{S}, \mathbf{x}(\text{terminal}) \doteq \mathbf{0}$

$\widehat{\mathbf{A}^{-1}} \leftarrow \varepsilon^{-1}\mathbf{I}$ — An $d \times d$ matrix
$\widehat{\mathbf{b}} \leftarrow \mathbf{0}$ — An d-dimensional vector
Repeat (for each episode):
  Initialize S; obtain corresponding $\mathbf{x}$
  Repeat (for each step of episode):
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'; obtain corresponding $\mathbf{x}'$
    $\mathbf{v} \leftarrow \widehat{\mathbf{A}^{-1}}^\top(\mathbf{x} - \gamma\mathbf{x}')$
    $\widehat{\mathbf{A}^{-1}} \leftarrow \widehat{\mathbf{A}^{-1}} - \big(\widehat{\mathbf{A}^{-1}}\mathbf{x}\big)\mathbf{v}^\top / \big(1 + \mathbf{v}^\top\mathbf{x}\big)$
    $\widehat{\mathbf{b}} \leftarrow \widehat{\mathbf{b}} + R\mathbf{x}$
    $\boldsymbol{\theta} \leftarrow \widehat{\mathbf{A}^{-1}}\widehat{\mathbf{b}}$
    $S \leftarrow S'$; $\mathbf{x} \leftarrow \mathbf{x}'$
  until S' is terminal

overstated. LSTD does not require a step size, but it does requires ε; if ε is chosen too small the sequence of inverses can vary wildly, and if ε is chosen too large then learning is slowed. In addition, LSTD's lack of a step size parameter means that it never forgets. This is sometimes desirable, but it is problematic if the target policy π changes as it does in reinforcement learning and GPI. In control applications, LSTD typically has to be combined with some other mechanism to induce forgeting, mooting any initial advantage of not requiring a step size parameter.

9.8 Memory-based Function Approximation

So far we have discussed the *parametric* approach to approximating value functions. In this approach, a learning algorithm adjusts the parameters of a functional form intended to approximate the value function over a problem's entire state space. Each update, $s \mapsto g$, is a training example used by the learning algorithm to change the parameters with the aim of reducing the approximation error. After the update, the training example can be discarded (although it might be saved to be used again). When an approximate value of a state (which we will call the *query state*) is needed, the function is simply evaluated at that state using the latest parameters produced by the learning algorithm.

Memory-based function approximation methods are very different. They simply save training examples in memory as they arrive (or at least save a subset of the examples) without updating any parameters. Then, whenever a query state's value estimate is needed, a set of examples is retrieved from memory and used to compute a value estimate for the query state. This approach is sometimes called *lazy learning* because processing training examples is postponed until the system is queried to provide an output.

Memory-based function approximation methods are prime examples of *nonparametric* methods. Unlike parametric methods, the approximating function's form is not limited to a fixed parameterized class of functions, such as linear functions or polynomials, but is instead determined by the training examples themselves, together with some means for combining them to output estimated values for query states. As more training examples accumulate in memory, one expects nonparametric methods to produce increasingly accurate approximations of any target function.

There are many different memory-based methods depending on how the stored training examples are selected and how they are used to respond to a query. Here, we focus on *local-learning* methods that approximate a value function only locally in the neighborhood of the current query state. These methods retrieve a set of training examples from memory whose states are judged to be the most relevant to the query state, where relevance usually depends on the distance between states: the closer a training example's state is to the query state, the more relevant it is considered to be, where distance can be defined in many different ways. After the query state is given a value, the local approximation is discarded.

The simplest example of the memory-based approach is the *nearest neighbor* method, which simply finds the example in memory whose state is closest to the query state and returns that example's value as the approximate value of the query state. In other words, if the query state is s, and $s' \mapsto g$ is the example in memory in which s' is the closest state to s, then g is returned as the approximate value of s. Slightly more complicated are *weighted average* methods that retrieve a set of nearest neighbor examples and return a weighted average of their target values, where the weights generally decrease with increasing distance between their states and the query state. *Locally weighted regression* is similar, but it fits a surface to the values of a set of nearest states by means of a parametric approximation method that minimizes a weighted error measure like (9.1), where the weights depend on distances from the query state. The value returned is the evaluation of the locally-fitted surface at the query state, after which the local approximation surface is discarded.

Being nonparametric, memory-based methods have the advantage over parametric methods of not limiting approximations to pre-specified functional forms. This allows accuracy to improve as more data accumulates. Memory-based *local* approximation methods have other properties that make them well suited for reinforcement learning. Because trajectory sampling is of such importance in reinforcement learning, as discussed in Section 8.6, memory-based local methods can focus function approximation on local neighborhoods of states (or state–action pairs) visited in real or simulated trajectories. There may be no need for global approximations because many areas of the state space will never (or almost never) be reached. In addition, memory-based methods allow an agent's experience to have a relatively immediate affect on value estimates in the neighborhood if its environment's current state, in contrast with a parametric method's need to incrementally adjust parameters of a global approximation.

Avoiding global approximations is also a way to address the curse of dimensionality. For example, for a state space with d dimensions, a tabular method storing a global approximation requires memory exponential in d. On the other hand, in storing examples for a memory-based method, each example requires memory proportional to d, and the memory required to store, say, n examples is linear in n. Nothing is exponential in d or n. Of course, the critical remaining issue is whether a memory-based method can answer queries quickly enough to be useful to an agent. A related concern is how speed degrades as the size of the memory grows. Finding nearest neighbors in a large database can take too long to be practical in many applications.

Proponents of memory-based methods have developed ways to accelerate the nearest neighbor search. Using parallel computers or special purpose hardware is one approach; another is the use of special multi-dimensional data structures to store the training data. One data structure studied for this application is the k-d tree (short for k-dimensional tree), which recursively splits a k-dimensional space into regions arranged as nodes of a binary tree. Depending on the amount of data and how it is distributed over the state space, nearest-neighbor search using k-d trees can quickly eliminate large regions of the space in the search for neighbors, making the searches feasible in some problems where naive searches would take too long.

Locally weighted regression additionally requires fast ways to do the local regression computations which have to be repeated to answer each query. Researchers have developed many ways to address these problems, including methods for forgetting entries in order to keep the size of the database within bounds. The Bibliographic and Historical Comments section at the end of this chapter points to some of

the relevant literature, including a selection of papers describing applications of memory-based learning to reinforcement learning.

9.9 Kernel-based Function Approximation

Memory-baed methods such as the weighted average and locally weighted regression methods described above depend on assigning weights to examples $s' \mapsto g$ in the database depending on the distance between s' and a query states s. The function that assigns these weights is called a *kernel function*, or simply a *kernel*. In the weighted average and locally weighted regressions methods, for example, a kernel function $k : \mathbb{R} \to \mathbb{R}$ assigns weights to distances between states. More generally, weights do not have to depend on distances; they can depend on some other measure of similarity between states. In this case, $k : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$, so that $k(s, s')$ is the weight given to data about s' in its influence on answering queries about s.

Viewed slightly differently, $k(s, s')$ is a measure of the strength of generalization from s' to s. Kernel functions numerically express how *relevant* knowledge about any state is to any other state. As an example, the strengths of generalization for tile coding shown in Figure 9.11 correspond to different kernel functions resulting from uniform and asymmetrical tile offsets. Although tile coding does not explicitly use a kernel function in its operation, it generalizes according to one. In fact, as we discuss more below, the strength of generalization resulting from linear parametric function approximation can always be described by a kernel function.

Kernel regression is the memory-based method that computes a kernel weighted average of the targets of *all* examples stored in memory, assigning the result to the query state. If D is the set of stored examples, and $g(s')$ denotes the target for state s' in a stored example, then kernel regression approximates the target function, in this case a value function depending on D, as

$$\hat{v}(s,D) = \sum_{s' \in D} k(s, s')g(s'). \tag{9.22}$$

The weighted average method described above is a special case in which $k(s, s')$ is non-zero only when s and s' are close to one another so that the sum need not be computed over all of D.

A common kernel is the Gaussian radial basis function (RBF) used in RBF function approximation as described in Section 9.5.5. In the method described there, RBFs are features whose centers and widths are either fixed from the start, with centers presumably concentrated in areas where many examples are expected to fall, or are adjusted in some way during learning. Barring methods that adjust centers and widths, this is a linear parametric method whose parameters are the weights of each RBF, which are typically learned by stochastic gradient, or semi-gradient, descent. The form of the approximation is a linear combination of the pre-determined RBFs. Kernel regression with an RBF kernel differs from this in two ways. First, it is memory-based: the RBFs are centered on the states of the stored examples. Second, it is nonparametric: there are no parameters to learn; the response to a query is given by (9.22).

Of course, many issues have to be addressed for practical implementation of kernel regression, issues that are beyond the scope or our brief discussion. However, it turns out that any linear parametric regression method like those we described in Section 9.4, with states represented by feature vectors $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_n(s))^\top$, can be recast as kernel regression where $k(s, s')$ is the inner product of the feature vector representations of s and s'; that is

$$k(s, s') = \mathbf{x}(s)^\top \mathbf{x}(s'). \tag{9.23}$$

Kernel regression with this kernel function produces the same approximation that a linear parametric method would if it used these feature vectors and learned with the same training data.

We skip the mathematical justification for this, which can be found in any modern machine learning text, such as Bishop (2006), and simply point out an important implication. Instead of constructing features for linear parametric function approximators, one can instead construct kernel functions directly without referring at all to feature vectors. Not all kernel functions can be expressed as inner products of feature vectors as in (9.23), but a kernel function that can be expressed like this can offer significant advantages over the equivalent parametric method. For many sets of feature vectors, (9.23) has a compact functional form that can be evaluated without any computation taking place in the n-dimensional feature space. In these cases, kernel regression is much less complex than directly using a linear parametric method with states represented by these feature vectors. This is the so-called "kernel trick" that allows effectively working in the high-dimension of an expansive feature space while actually working only with the set of stored training examples. The kernel trick is the basis of many machine learning methods, and researchers have shown how it can sometimes benefit reinforcement learning.

9.10 Looking Deeper at On-policy Learning: Interest and Emphasis

The algorithms we have considered so far in this chapter have treated all the states encountered equally, as if they were all equally important. In some cases, however, we are more interested in some states than others. In discounted episodic problems, for example, we may be more interested in accurately valuing early states in the episode than in later states where discounting may have made the rewards much less important to the value of the start state. Or, if an action-value function is being learned, it may be less important to accurately value poor actions whose value is much less than the greedy action. Function approximation resources are always limited, and if they were used in a more targeted way, then performance could be improved.

One reason we have treated all states encountered equally is that then we are updating according to the on-policy distribution, for which stronger theoretical results are available for semi-gradient methods. Recall that the on-policy distribution was defined as the distribution of states encountered in an MDP while following the target policy. Now we will generalize this concept significantly. Rather than having one on-policy distribution for the MDP, we will have many. All of them will have in common that they are a distribution of states encountered in trajectories while following the target policy, but they will vary in how the trajectories are, in a sense, initiated.

We now introduce some new concepts. First be introduce a non-negative scalar measure, a random variable I_t called *interest*, indicating the degree to which we are interested in accurately valuing the state (or state–action pair) at time t. If we don't care at all about the state, then the interest should be zero; if we fully care, it might be one, though it's formally allowed take any non-negative value. The interest can be set in any causal way; for example, it may depend on the trajectory up to time t or the learned parameters at time t. The distribution μ in the MSVE (9.1) is then defined as the distribution of states encountered while following the target policy, weighted by the interest. Second, we introduce another non-negative scalar random variable, the *emphasis* M_t. This scalar multiplies the learning update and thus emphasizes or de-emphasizes the learning done at time t. The general n-step learning rule, replacing (9.15), is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha M_t \left[G_{t:t+n} - \hat{v}(S_t,\mathbf{w}_{t+n-1})\right] \nabla \hat{v}(S_t,\mathbf{w}_{t+n-1}), \qquad 0 \leq t < T, \tag{9.24}$$

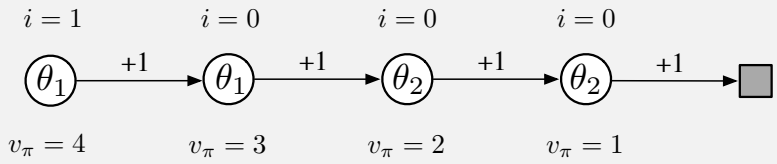
with the n-step return given by (9.16) and the emphasis determined recursively from the interest by:

$$M_t = I_t + \gamma^n M_{t-n}, \qquad 0 \leq t < T, \tag{9.25}$$

with $M_t \doteq 0$, for all $t < 0$. These equations are taken to include the Monte Carlo case, for which $G_{t:t+n} = G_t$, all the updates are taken at episode's end, $n = T - t$, and $M_t = I_t$.

Example 9.4: Interest and Emphasis

To see the potential benefits of using interest and emphasis, consider the four-state Markov reward process shown below:



Episodes start in the leftmost state, then transition one state to the right, with a reward of +1, on each step until the terminal state is reached. The true value of the first state is thus 4, of the second state 3, and so on as shown below each state. These are the true values; the estimated values can only approximate these because they are constrained by the parameterization. There are two components to the parameter vector $\mathbf{w} = (w_1, w_2)^\top$, and the parameterization is as written inside each state. The estimated values of the first two states are given by w_1 alone and thus must be the same even though their true values are different. Similarly, the estimated values of the third and fourth states are given by w_2 alone and must be the same even though their true values are different. Suppose that we are interested in accurately valuing only the leftmost state; we assign it an interest of 1 while all the other states are assigned an interest of 0, as indicated above the states.

First consider applying gradient Monte Carlo algorithms to this problem. The algorithms presented earlier in this chapter that do not take into account interest and emphasis (in (9.7) and the box on page 165) will converge (for decreasing step sizes) to the parameter vector $\mathbf{w}_\infty = (3.5, 1.5)$, which gives the first state—the only one we are interested in—a value of 3.5 (i.e., intermediate between the true values of the first and second states). The methods presented in this section that do use interest and emphasis, on the other hand, will learn the value of the first state exactly correctly; w_1 will converge to 4 while w_2 will never be updated because the emphasis is zero in all states save the leftmost.

Now consider applying two-step semi-gradient TD methods. The methods from earlier in this chapter without interest and emphasis (in (9.15) and (9.16) and the box on page 171) will again converge to $\mathbf{w}_\infty = (3.5, 1.5)$, while the methods with interest and emphasis converge to $\mathbf{w}_\infty = (4, 2)$. The latter produces the exactly correct values for the first state and for the third state (which the first state bootstraps from) while never making any updates corresponding to the second or fourth states.

9.11 Summary

Reinforcement learning systems must be capable of *generalization* if they are to be applicable to artificial intelligence or to large engineering applications. To achieve this, any of a broad range of existing methods for *supervised-learning function approximation* can be used simply by treating each update as a training example.

Perhaps the most suitable supervised learning methods are those using *parameterized function approximation*, in which the policy is parameterized by a weight vector $\mathbf{w}$. Although the weight vector has many components, the state space is much larger still and we must settle for an approximate solution. We defined MSVE($\mathbf{w}$) as a measure of the error in the values $v_{\pi_\mathbf{w}}(s)$ for a weight vector $\mathbf{w}$ under the *on-policy distribution*, μ. The MSVE gives us a clear way to rank different value-function approximations in the on-policy case.

To find a good weight vector, the most popular methods are variations of *stochastic gradient descent* (SGD). In this chapter we have focused on the *on-policy* case with a *fixed policy*, also known as policy evaluation or prediction; a natural learning algorithm for this case is *n-step semi-gradient TD*, which includes gradient Monte Carlo and semi-gradient TD(0) algorithms as the special cases when $n = \infty$ and $n = 1$ respectively. Semi-gradient TD methods are not true gradient methods. In such bootstrapping methods (including DP), the weight vector appears in the update target, yet this is not taken into account in computing the gradient—thus they are *semi*-gradient methods. As such, they cannot rely on classical SGD results.

Nevertheless, good results can be obtained for semi-gradient methods in the special case of *linear* function approximation, in which the value estimates are sums of features times corresponding weights. The linear case is the most well understood theoretically and works well in practice when provided with appropriate features. Choosing the features is one of the most important ways of adding prior domain knowledge to reinforcement learning systems. They can be chosen as polynomials, but this case generalizes poorly in the online learning setting typically considered in reinforcement learning. Better is to choose features according the Fourier basis, or according to some form of coarse coding with sparse overlapping receptive fields. Tile coding is a form of coarse coding that is particularly computationally efficient and flexible. Radial basis functions are useful for one- or two-dimensional tasks in which a smoothly varying response is important. LSTD is the most data-efficient linear TD prediction method, but requires computation proportional to the square of the number of weights, whereas all the other methods are of complexity linear in the number of weights. Nonlinear methods include artificial neural networks trained by backpropagation and variations of SGD; these methods have become very popular in recent years under the name *deep reinforcement learning*.

Linear semi-gradient n-step TD is guaranteed to converge under standard conditions, for all n, to a MSVE that is within a bound of the optimal error. This bound is always tighter for higher n and approaches zero as $n \to \infty$. However, in practice that choice results in very slow learning, and some degree of bootstrapping ($1 < n < \infty$) is usually preferrable.

Bibliographical and Historical Remarks

Generalization and function approximation have always been an integral part of reinforcement learning. Bertsekas and Tsitsiklis (1996), Bertsekas (2012), and Sugiyama et al. (2013) present the state of the art in function approximation in reinforcement learning. Some of the early work with function approximation in reinforcement learning is discussed at the end of this section.

9.3 Gradient-descent methods for minimizing mean-squared error in supervised learning are well known. Widrow and Hoff (1960) introduced the least-mean-square (LMS) algorithm, which is the prototypical incremental gradient-descent algorithm. Details of this and related algorithms are provided in many texts (e.g., Widrow and Stearns, 1985; Bishop, 1995; Duda and Hart, 1973).

Semi-gradient TD(0) was first explored by Sutton (1984, 1988), as part of the linear TD(λ) algorithm that we will treat in Chapter 12. The term "semi-gradient" to describe these bootstrapping methods is new to the second edition of this book.

The earliest use of state aggregation in reinforcement learning may have been Michie and Chambers's BOXES system (1968). The theory of state aggregation in reinforcement learning has been developed by Singh, Jaakkola, and Jordan (1995) and Tsitsiklis and Van Roy (1996). State aggregation has been used in dynamic programming from its earliest days (e.g., Bellman, 1957a).

9.4 Sutton (1988) proved convergence of linear TD(0) in the mean to the minimal MSVE solution for the case in which the feature vectors, $\{\mathbf{x}(s) : s \in \mathcal{S}\}$, are linearly independent. Convergence with probability 1 was proved by several researchers at about the same time (Peng, 1993; Dayan and Sejnowski, 1994; Tsitsiklis, 1994; Gurvits, Lin, and Hanson, 1994). In addition, Jaakkola, Jordan, and Singh (1994) proved convergence under on-line updating. All of these results assumed linearly independent feature vectors, which implies at least as many components to $\mathbf{w}_t$ as there are states. Convergence for the more important case of general (dependent) feature vectors was first shown by Dayan (1992). A significant generalization and strengthening of Dayan's result was proved by Tsitsiklis and Van Roy (1997). They proved the main result presented in this section, the bound on the asymptotic error of linear bootstrapping methods.

9.5 Our presentation of the range of possibilities for linear function approximation is based on that by Barto (1990).

9.5.3 The term *coarse coding* is due to Hinton (1984), and our Figure 9.6 is based on one of his figures. Waltz and Fu (1965) provide an early example of this type of function approximation in a reinforcement learning system.

9.5.4 Tile coding, including hashing, was introduced by Albus (1971, 1981). He described it in terms of his "cerebellar model articulator controller," or CMAC, as tile coding is sometimes known in the literature. The term "tile coding" was new to the first edition of this book, though the idea of describing CMAC in these terms is taken from Watkins (1989). Tile coding has been used in many reinforcement learning systems (e.g., Shewchuk and Dean, 1990; Lin and Kim, 1991; Miller, Scalera, and Kim, 1994; Sofge and White, 1992; Tham, 1994; Sutton, 1996; Watkins, 1989) as well as in other types of learning control systems (e.g., Kraft and Campagna, 1990; Kraft, Miller, and Dietz, 1992). This section draws heavily on the work of Miller and Glanz (1996).

9.5.5 Function approximation using radial basis functions (RBFs) has received wide attention ever since being related to neural networks by Broomhead and Lowe (1988). Powell (1987) reviewed

earlier uses of RBFs, and Poggio and Girosi (1989, 1990) extensively developed and applied this approach.

9.6 The introduction of the threshold logic unit as an abstract model neuron by McCulloch and Pitts (1943) was the beginning of artificial neural networks (ANNs). The history of ANNs as learning methods for classification or regression has passed through several stages: roughly, the Perceptron (Rosenblatt, 1962) and ADALINE (ADAptive LINear Element) (Widrow and Hoff, 1960) stage of learning by single-layer ANNs, the error-backpropagation stage (Werbos, 1974; LeCun, 1985; Parker, 1985; Rumelhart, Hinton, and Williams, 1986) of learning by multi-layer ANNs, and the current deep-learning stage with its emphasis on representation learning (e.g., Bengio, Courville, and Vincent, 2012; Goodfellow, Bengio, and Courville, 2016). Examples of the many books on ANNs are Haykin (1994), Bishop (1995), and Ripley (2007).

ANNs as function approximation for reinforcement learning goes back to the early neural networks of Farley and Clark (1954), who used reinforcement-like learning to modify the weights of linear threshold functions representing policies. Widrow, Gupta, and Maitra (1973) presented a neuron-like linear threshold unit implementing a learning process they called *learning with a critic* or *selective bootstrap adaptation*, a reinforcement-learning variant of the ADALINE algorithm. Werbos (1974, 1987, 1994) developed an approach to prediction and control that uses ANNs trained by error backpropation to learn policies and value functions using TD-like algorithms. Barto, Sutton, and Brouwer (1981) and Barto and Sutton (1981b) extended the idea of an associative memory network (e.g., Kohonen, 1977; Anderson, Silverstein, Ritz, and Jones, 1977) to reinforcement learning. Barto, Anderson, and Sutton (1982) used a two-layer ANN to learn a nonlinear control policy, and emphasized the first layer's role of learning a suitable representation. Hampson (1983, 1989) was an early proponent of multilayer ANNs for learning value functions. Barto, Sutton, and Anderson (1983) presented an actor–critic algorithm in the form of an ANN learning to balance a simulated pole (see Sections 15.7 and 15.8). Barto and Anandan (1985) introduced a stochastic version of Widrow, Gupta, and Maitra's (1973) selective bootstrap algorithm called the *associative reward-penalty* (A_{R-P}) *algorithm*. Barto (1985, 1986) and Barto and Jordan (1987) described multi-layer ANNs consisting of $\mathrm{A_{R-P}}$ units trained with a globally-broadcast reinforcement signal to learn classification rules that are not linearly separable. Barto (1985) discussed this approach to ANNs and how this type of learning rule is related to others in the literature at that time. (See Section 15.10 for additional discussion of this approach to training multi-layer ANNs.) Anderson (1986, 1987, 1989) evaluated numerous methods for training multilayer ANNs and showed that an actor–critic algorithm in which both the actor and critic were implemented by two-layer ANNs trained by error backpropagation outperformed single-layer ANNs in the pole-balancing and tower of Hanoi tasks. Williams (1988) described several ways that backpropagation and reinforcement learning can be combined for training ANNs. Gullapalli (1990) and Williams (1992) devised reinforcement learning algorithms for neuron-like units having continuous, rather than binary, outputs. Barto, Sutton, and Watkins (1990) argued that ANNs can play significant roles for approximating functions required for solving sequential decision problems. Williams (1992) related REINFORCE learning rules (Section 13.3) to the error backpropagation method for training multi-layer ANNs. Schmidhuber (2015) reviews applications of ANNs in reinforcement learning, including applications of recurrent ANNs.

9.7 LSTD is due to Bradtke and Barto (see Bradtke, 1993, 1994; Bradtke and Barto, 1996; Bradtke, Ydstie, and Barto, 1994), and was further developed by Boyan (1999, 2002) and Nedić and Bertsekas (2003). The incremental update of the inverse matrix has been known at least since 1949 (Sherman and Morrison, 1949). An extension of least-squares methods to control was introduced by Lagoudakis and Parr (2003).

9.8 Our discussion of memory-based function approximation is largely based on the review of locally weighted learning by Atkeson, Moore, and Schaal (1997). Atkeson (1992) discussed the use of locally weighted regression in memory-based robot learning and supplied an extensive bibliography covering the history of the idea. Stanfill and Waltz (1986) influentially argued for the importance of memory based methods in artificial intelligence, especially in light of parallel architectures then becoming available, such as the Connection Machine. Baird and Klopf (1993) introduced a novel memory-based approach and used it as the function approximation method for Q-learning applied to the pole-balancing task. Schaal and Atkeson (1994) applied locally weighted regression to a robot juggling control problem, where it was used to learn a system model. Ping (1995) used the pole-balancing task to experiment with several nearest-neighbor methods for approximating value functions, policies, and environment models. Tadepalli and Ok (1996) obtained promising results with locally-weighted linear regression to learn a value function for a simulated automatic guided vehicle task. Bottou and Vapnik (1996) demonstrated surprising efficiency of several local learning algorithms compared to non-local algorithms in some pattern recognition tasks, discussing the impact of local learning on generalization.

Bentley (1975) introduced k-d trees and reported observing average running time of O(log n) for nearest neighbor search over n records. Friedman, Bentley, and Finkel (1977) clarified the algorithm for nearest neighbor search with k-d trees. Omohundro (1987) discussed efficiency gains possible with hierarchical data structures such as k-d-trees. Moore, Schneider, and Deng (1997) introduced the use of k-d trees for efficient locally weighted regression.

9.9 The origin of kernel regression is the *method of potential functions* of Aizerman, Braverman, and Rozonoer (1964). They likened the data to point electric charges of various signs and magnitudes distributed over space. The resulting electric potential over space produced by summing the potentials of the point charges corresponded to the interpolated surface. In this analogy, the kernel function is the potential of a point charge, which falls off as the reciprocal of the distance from the charge. Connell and Utgoff (1987) applied an actor–critic method to the pole-balancing task in which the critic approximated the value function using kernel regression with inverse distance weighting given by "Shepard's function." Predating widespread interest in kernel regression in machine learning, these authors did not use the term kernel, but referred to "Shepard's method' (Shepard, 1968). Their system could learn to balance the pole in about 16 episodes. Other kernel-based approaches to reinforcement learning include those of Ormoneit and Sen (2002), Dietterich and Wang (2002), Xu, Xie, Hu, Nu, and Lu (2005), Taylor and Parr (2009), Barreto, Precup, and Pineau (2011), and Bhat, Farias, and Moallemi (2012).

9.10 See the bibliographical notes for Emphatic-TD methods in Section 11.8.

The earliest example we know of in which function approximation methods were used for learning value functions was Samuel's checkers player (1959, 1967). Samuel followed Shannon's (1950) suggestion that a value function did not have to be exact to be a useful guide to selecting moves in a game and that it might be approximated by linear combination of features. In addition to linear function approximation, Samuel experimented with lookup tables and hierarchical lookup tables called signature tables (Griffith, 1966, 1974; Page, 1977; Biermann, Fairfield, and Beres, 1982).

At about the same time as Samuel's work, Bellman and Dreyfus (1959) proposed using function approximation methods with DP. (It is tempting to think that Bellman and Samuel had some influence on one another, but we know of no reference to the other in the work of either.) There is now a fairly extensive literature on function approximation methods and DP, such as multigrid methods and methods using splines and orthogonal polynomials (e.g., Bellman and Dreyfus, 1959; Bellman, Kalaba, and Kotkin, 1973; Daniel, 1976; Whitt, 1978; Reetz, 1977; Schweitzer and Seidmann, 1985; Chow and Tsitsiklis, 1991; Kushner and Dupuis, 1992; Rust, 1996).

Holland's (1986) classifier system used a selective feature-match technique to generalize evaluation information across state–action pairs. Each classifier matched a subset of states having specified values for a subset of features, with the remaining features having arbitrary values ("wild cards"). These subsets were then used in a conventional state-aggregation approach to function approximation. Holland's idea was to use a genetic algorithm to evolve a set of classifiers that collectively would implement a useful action-value function. Holland's ideas influenced the early research of the authors on reinforcement learning, but we focused on different approaches to function approximation. As function approximators, classifiers are limited in several ways. First, they are state-aggregation methods, with concomitant limitations in scaling and in representing smooth functions efficiently. In addition, the matching rules of classifiers can implement only aggregation boundaries that are parallel to the feature axes. Perhaps the most important limitation of conventional classifier systems is that the classifiers are learned via the genetic algorithm, an evolutionary method. As we discussed in Chapter 1, there is available during learning much more detailed information about how to learn than can be used by evolutionary methods. This perspective led us to instead adapt supervised learning methods for use in reinforcement learning, specifically gradient-descent and neural network methods. These differences between Holland's approach and ours are not surprising because Holland's ideas were developed during a period when neural networks were generally regarded as being too weak in computational power to be useful, whereas our work was at the beginning of the period that saw widespread questioning of that conventional wisdom. There remain many opportunities for combining aspects of these different approaches.

Christensen and Korf (1986) experimented with regression methods for modifying coefficients of linear value function approximations in the game of chess. Chapman and Kaelbling (1991) and Tan (1991) adapted decision-tree methods for learning value functions. Explanation-based learning methods have also been adapted for learning value functions, yielding compact representations (Yee, Saxena, Utgoff, and Barto, 1990; Dietterich and Flann, 1995).

Chapter 10

On-policy Control with Approximation

In this chapter we return to the control problem, now with parametric approximation of the action-value function $\hat{q}(s, a, \mathbf{w}) \approx q_*(s, a)$, where $\mathbf{w} \in \mathbb{R}^d$ is a finite-dimensional weight vector. We continue to restrict attention to the on-policy case, leaving off-policy methods to Chapter 11. The present chapter features the semi-gradient Sarsa algorithm, the natural extension of semi-gradient TD(0) (last chapter) to action values and to on-policy control. In the episodic case, the extension is straightforward, but in the continuing case we have to take a few steps backward and re-examine how we have used discounting to define an optimal policy. Surprisingly, once we have genuine function approximation we have to give up discounting and switch to a new "average-reward" formulation of the control problem, with new "differential" value functions.

Starting first in the episodic case, we extend the function approximation ideas presented in the last chapter from state values to action values. Then we extend them to control following the general pattern of on-policy GPI, using ε-greedy for action selection. We show results for n-step linear Sarsa on the Mountain Car problem. Then we turn to the continuing case and repeat the development of these ideas for the average-reward case with differential values.

10.1 Episodic Semi-gradient Control

The extension of the semi-gradient prediction methods of Chapter 9 to action values is straightforward. In this case it is the approximate action-value function, $\hat{q} \approx q_\pi$, that is represented as a parameterized functional form with weight vector $\mathbf{w}$. Whereas before we considered random training examples of the form $S_t \mapsto U_t$, now we consider examples of the form $S_t, A_t \mapsto U_t$. The update target U_t can be any approximation of $q_\pi(S_t, A_t)$, including the usual backed-up values such as the full Monte Carlo return, G_t, or any of the n-step Sarsa returns (7.4). The general gradient-descent update for action-value prediction is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[U_t - \hat{q}(S_t, A_t, \mathbf{w}_t)\Big]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t). \tag{10.1}$$

For example, the update for the one-step Sarsa method is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)\Big]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t). \tag{10.2}$$

We call this method *episodic semi-gradient one-step Sarsa*. For a constant policy, this method converges in the same way that TD(0) does, with the same kind of error bound (9.14).

To form control methods, we need to couple such action-value prediction methods with techniques for policy improvement and action selection. Suitable techniques applicable to continuous actions, or to actions from large discrete sets, are a topic of ongoing research with as yet no clear resolution. On the other hand, if the action set is discrete and not too large, then we can use the techniques already developed in previous chapters. That is, for each possible action a available in the current state S_t, we can compute $\hat{q}(S_t, a, \mathbf{w}_t)$ and then find the greedy action $A_t^* = \arg\max_a \hat{q}(S_t, a, \mathbf{w}_t)$. Policy improvement is then done (in the on-policy case treated in this chapter) by changing the estimation policy to a soft approximation of the greedy policy such as the ε-greedy policy. Actions are selected according to this same policy. Pseudocode for the complete algorithm is given in the box.

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

Example 10.1: Mountain Car Task Consider the task of driving an underpowered car up a steep mountain road, as suggested by the diagram in the upper left of Figure 10.1. The difficulty is that gravity is stronger than the car's engine, and even at full throttle the car cannot accelerate up the steep slope. The only solution is to first move away from the goal and up the opposite slope on the left. Then, by applying full throttle the car can build up enough inertia to carry it up the steep slope even though it is slowing down the whole way. This is a simple example of a continuous control task where things have to get worse in a sense (farther from the goal) before they can get better. Many control methodologies have great difficulties with tasks of this kind unless explicitly aided by a human designer.

The reward in this problem is -1 on all time steps until the car moves past its goal position at the top of the mountain, which ends the episode. There are three possible actions: full throttle forward ($+1$), full throttle reverse (-1), and zero throttle (0). The car moves according to a simplified physics. Its position, x_t, and velocity, $\dot{x}_t$, are updated by

$$x_{t+1} \doteq \mathit{bound}\big[x_t + \dot{x}_{t+1}\big]$$

$$\dot{x}_{t+1} \doteq \mathit{bound}\big[\dot{x}_t + 0.001A_t - 0.0025\cos(3x_t)\big],$$

where the *bound* operation enforces $-1.2 \leq x_{t+1} \leq 0.5$ and $-0.07 \leq \dot{x}_{t+1} \leq 0.07$. In addition, when x_{t+1} reached the left bound, $\dot{x}_{t+1}$ was reset to zero. When it reached the right bound, the goal was reached and the episode was terminated. Each episode started from a random position $x_t \in [-0.6, -0.4)$ and zero velocity. To convert the two continuous state variables to binary features, we used grid-tilings as in Figure 9.9. We used 8 tilings, with each tile covering 1/8th of the bounded distance in each dimension, and asymmetrical offsets as described in Section 9.5.4.[1] The feature vectors $\mathbf{x}(s, a)$ created

[1] In particular, we used the tile-coding software, available on the web, version 3 (Python), with `iht=IHT(4096)` and

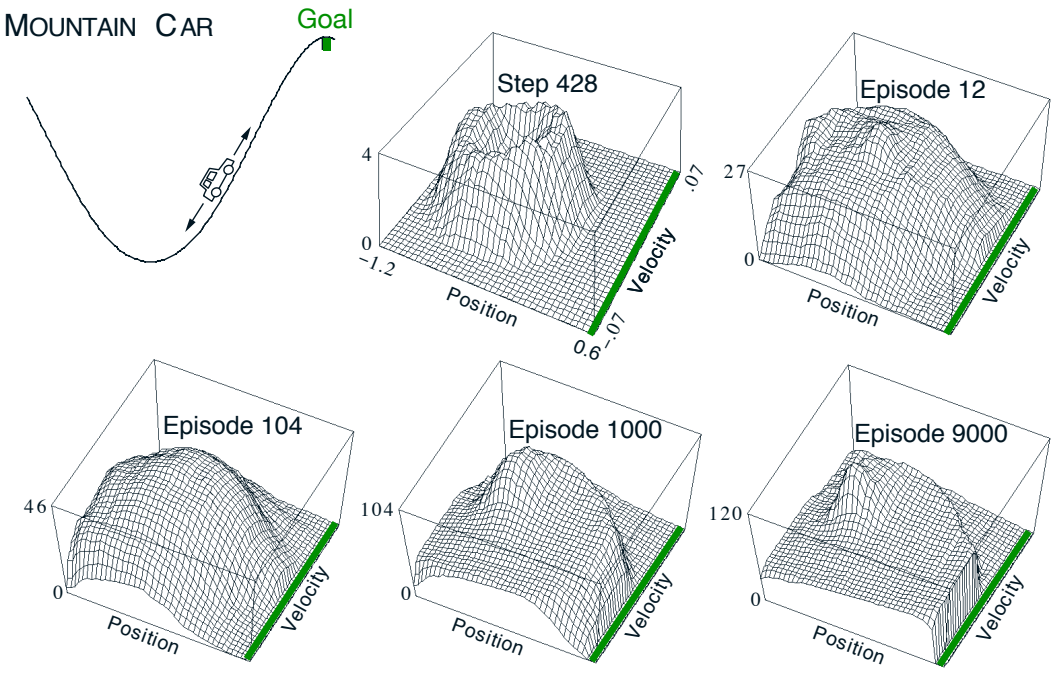


Figure 10.1: The Mountain Car task (upper left panel) and the cost-to-go function ($-\max_a \hat{q}(s, a, \mathbf{w})$) learned during one run.

by tile coding were then combined linearly with the parameter vector to approximate the action-value function:

$$\hat{q}(s, a, \mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s, a) = \sum_i w_i \cdot x_i(s, a), \tag{10.3}$$

for each pair of state, s, and action, a.

Figure 10.1 shows what typically happens while learning to solve this task with this form of function approximation.[2] Shown is the negative of the value function (the *cost-to-go* function) learned on a single run. The initial action values were all zero, which was optimistic (all true values are negative in this task), causing extensive exploration to occur even though the exploration parameter, ε, was 0. This can be seen in the middle-top panel of the figure, labeled "Step 428". At this time not even one episode had been completed, but the car has oscillated back and forth in the valley, following circular trajectories in state space. All the states visited frequently are valued worse than unexplored states, because the actual rewards have been worse than what was (unrealistically) expected. This continually drives the agent away from wherever it has been, to explore new states, until a solution is found.

Figure 10.2 shows several learning curves for semi-gradient Sarsa on this problem, with various step sizes.

Exercise 10.1 Why have we not considered Monte Carlo methods in this chapter? □

`tiles(iht, 8, [8*x/(0.5+1.2), 8*xdot/(0.07+0.07)], A)` to get the indices of the ones in the feature vector for state (`x`, `xdot`) and action `A`.

[2]This data is actually from the "semi-gradient Sarsa(λ)" algorithm that we will not meet until Chapter 12, but semi-gradient Sarsa behaves similarly.

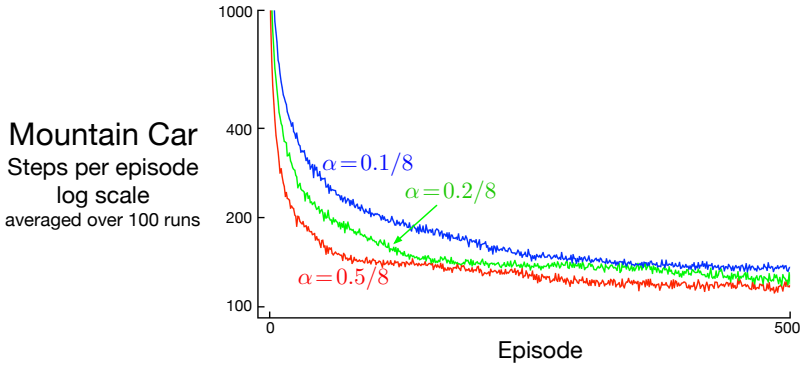


Figure 10.2: Mountain Car learning curves for the semi-gradient Sarsa method with tile-coding function approximation and ε-greedy action selection. ■

10.2 n-step Semi-gradient Sarsa

We can obtain an n-step version of episodic semi-gradient Sarsa by using an n-step return as the update target in the semi-gradient Sarsa update equation (10.1). The n-step return immediately generalizes from its tabular form (7.4) to a function approximation form:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \quad n \geq 1, 0 \leq t < T-n, \tag{10.4}$$

with $G_{t:t+n} \doteq G_t$ if $t+n \geq T$, as usual. The n-step update equation is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha \left[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}) \right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}), \qquad 0 \leq t < T. \tag{10.5}$$

Episodic semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_*$, or $\hat{q} \approx q_\pi$

Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$, possibly π
Initialize value-function weight vector $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Parameters: step size $\alpha > 0$, small $\varepsilon > 0$, a positive integer n
All store and access operations (S_t, A_t, and R_t) can take their index mod n

Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$ or ε-greedy wrt $\hat{q}(S_0, \cdot, \mathbf{w})$
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
    If $t < T$, then:
      Take action A_t
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
      If S_{t+1} is terminal, then:
        $T \leftarrow t+1$
      else:
        Select and store $A_{t+1} \sim \pi(\cdot|S_{t+1})$ or ε-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
    $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$
      If $\tau + n < T$, then $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$ $(G_{\tau:\tau+n})$
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G - \hat{q}(S_\tau, A_\tau, \mathbf{w}) \right] \nabla \hat{q}(S_\tau, A_\tau, \mathbf{w})$
  Until $\tau = T - 1$

As we have seen before, performance is best if an intermediate level of bootstrapping is used, corresponding to an n larger than 1. Figure 10.3 shows how this algorithm tends to learn faster and obtain a better asymptotic performance at $n=8$ than at $n=1$ on the Mountain Car task. Figure 10.4 shows the results of a more detailed study of the effect of the parameters α and n on the rate of learning on this task.

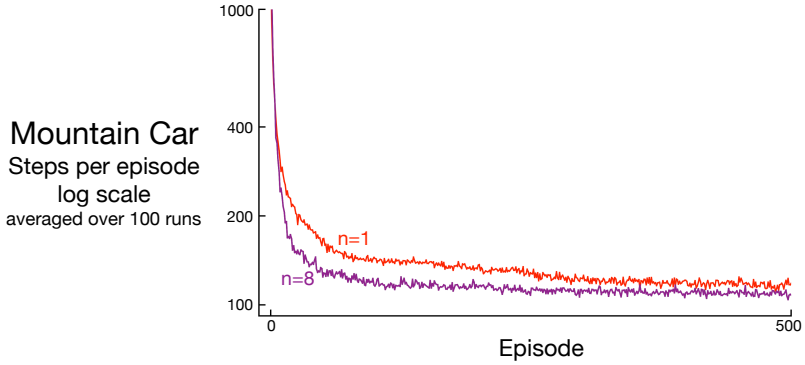


Figure 10.3: One-step vs multi-step performance of n-step semi-gradient Sarsa on the Mountain Car task. Good step sizes were used: $\alpha = 0.5/8$ for $n = 1$ and $\alpha = 0.3/8$ for $n = 8$.

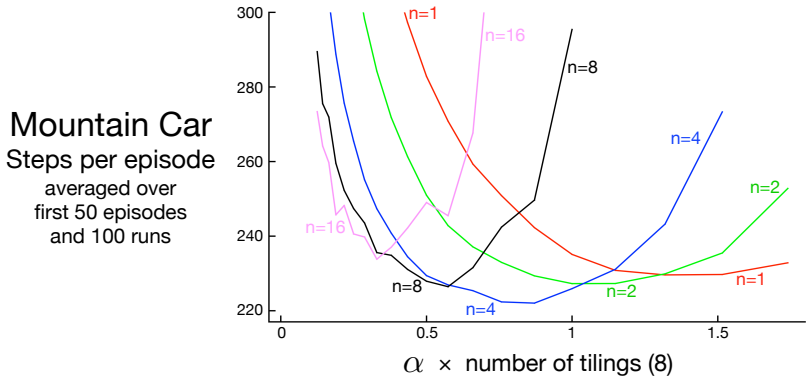


Figure 10.4: Effect of the α and n on early performance of n-step semi-gradient Sarsa and tile-coding function approximation on the Mountain Car task. As usual, an intermediate level of bootstrapping ($n = 4$) performed best. These results are for selected α values, on a log scale, and then connected by straight lines. The standard errors ranged from 0.5 (less than the line width) for $n = 1$ to about 4 for $n = 16$, so the main effects are all statistically significant.

Exercise 10.2 Give pseudocode for semi-gradient one-step *Expected* Sarsa for control. □

Exercise 10.3 Why do the results shown in Figure 10.4 have higher standard errors at large n than at low n? □

10.3 Average Reward: A New Problem Setting for Continuing Tasks

We now introduce a third classical setting—alongside the episodic and discounted settings—for formulating the goal in Markov decision problems (MDPs). Like the discounted setting, the *average reward* setting applies to continuing problems, problems for which the interaction between agent and environment goes on and on forever without termination or start states. Unlike that setting, however, there is no discounting—the agent cares just as much about delayed rewards as it does about immediate reward. The average-reward setting is one of the major settings considered in the classical theory of dynamic programming and, though less often, in reinforcement learning. As we discuss in the next section, the discounted setting is problematic with function approximation, and thus the average-reward setting is needed to replace it.

In the average-reward setting, the quality of a policy π is defined as the average rate of reward while following that policy, which we denote as $r(\pi)$:

$$\begin{aligned} r(\pi) &\doteq \lim_{T\to\infty} \frac{1}{T} \sum_{t=1}^{T} \mathbb{E}[R_t \mid A_{0:t-1} \sim \pi] \\ &= \lim_{t\to\infty} \mathbb{E}[R_t \mid A_{0:t-1} \sim \pi], \\ &= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a) r, \end{aligned} \tag{10.6}$$

where the expectations are conditioned on the prior actions, $A_0, A_1, \ldots, A_{t-1}$, being taken according to π, and μ_π is the steady-state distribution, $\mu_\pi(s) \doteq \lim_{t\to\infty} \Pr\{S_t = s | A_{0:t-1} \sim \pi\}$, which is assumed to exist and to be independent of S_0. This property is known as *ergodicity*. It means that where the MDP starts or any early decision made by the agent can have only a temporary effect; in the long run your expectation of being in a state depends only on the policy and the MDP transition probabilities. Ergodicity is sufficient to guarantee the existence of the limits in the equations above.

There are subtle distinctions that can be drawn between different kinds of optimality in the undiscounted continuing case. Nevertheless, for most practical purposes it may be adequate simply to order policies according to their average reward per time step, in other words, according to their $r(\pi)$. This quantity is essentially the average reward under π, as suggested by (10.6). In particular, we consider all policies that attain the maximal value of $r(\pi)$ to be optimal.

Note that the steady state distribution is the special distribution under which, if you select actions according to π, you remain in the same distribution. That is, for which

$$\sum_s \mu_\pi(s) \sum_a \pi(a|s, \mathbf{w}) p(s' \,|\, s, a) = \mu_\pi(s'). \tag{10.7}$$

In the average-reward setting, returns are defined in terms of differences between rewards and the average reward:

$$G_t \;\doteq\; R_{t+1} - r(\pi) \;+\; R_{t+2} - r(\pi) \;+\; R_{t+3} - r(\pi) \;+\; \cdots. \tag{10.8}$$

This is known as the *differential* return, and the corresponding value functions are known as *differential* value functions. They are defined in the same way and we will use the same notation for them as we have all along: $v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s]$ and $q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$ (similarly for v_* and q_*). Differential value functions also have Bellman equations, just slightly different from those we have seen earlier. We simply remove all γs and replace all rewards by the difference between the reward and the true average reward:

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

$$q_\pi(s,a) = \sum_{r,s'} p(s',r\,|\,s,a)\Big[r - r(\pi) + \sum_{a'} \pi(a'|s') q_\pi(s',a')\Big],$$

$$v_*(s) = \max_a \sum_{r,s'} p(s',r\,|\,s,a)\Big[r - r(\pi) + v_*(s')\Big], \text{ and}$$

$$q_*(s,a) = \sum_{r,s'} p(s',r\,|\,s,a)\Big[r - r(\pi) + \max_{a'} q_*(s',a')\Big]$$

(cf. Eqs. 3.13, 4.1, and 4.2).

There is also a differential form of the two TD errors:

$$\delta_t \doteq R_{t+1} - \bar{R}_{t+1} + \hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t), \text{ and} \tag{10.9}$$

$$\delta_t \doteq R_{t+1} - \bar{R}_{t+1} + \hat{q}(S_{t+1},A_{t+1},\mathbf{w}_t) - \hat{q}(S_t,A_t,\mathbf{w}_t), \tag{10.10}$$

where $\bar{R}_t$ is an estimate at time t of the average reward $r(\pi)$. With these alternate definitions, most of our algorithms and many theoretical results carry through to the average-reward setting.

For example, the average reward version of semi-gradient Sarsa is defined just as in (10.2) except with the differential version of the TD error. That is, by

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t \nabla\hat{q}(S_t,A_t,\mathbf{w}_t), \tag{10.11}$$

with δ_t given by (10.10). The pseudocode for the complete algorithm is given below.

Differential semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Parameters: step sizes $\alpha, \beta > 0$

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average reward estimate $\bar{R}$ arbitrarily (e.g., $\bar{R} = 0$)
Initialize state S, and action A

Repeat (for each step):
  Take action A, observe R, S'
  Choose A' as a function of $\hat{q}(S',\cdot,\mathbf{w})$ (e.g., ε-greedy)
  $\delta \leftarrow R - \bar{R} + \hat{q}(S',A',\mathbf{w}) - \hat{q}(S,A,\mathbf{w})$
  $\bar{R} \leftarrow \bar{R} + \beta\delta$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S,A,\mathbf{w})$
  $S \leftarrow S'$
  $A \leftarrow A'$

Example 10.2: An Access-Control Queuing Task This is a decision task involving access control to a set of k servers. Customers of four different priorities arrive at a single queue. If given access to a server, the customers pay a reward of 1, 2, 4, or 8 to the server, depending on their priority, with higher priority customers paying more. In each time step, the customer at the head of the queue is either accepted (assigned to one of the servers) or rejected (removed from the queue, with a reward of zero). In either case, on the next time step the next customer in the queue is considered. The queue never empties, and the priorities of the customers in the queue are equally randomly distributed. Of course a customer cannot be served if there is no free server; the customer is always rejected in this case. Each busy server becomes free with probability p on each time step. Although we have just described them for definiteness, let us assume the statistics of arrivals and departures are unknown. The task is to decide on each step whether to accept or reject the next customer, on the basis of his priority and the number of free servers, so as to maximize long-term reward without discounting.

In this example we consider a tabular solution to this problem. Although there is no generalization between states, we can still consider it in the general function approximation setting as this setting generalizes the tabular setting. Thus we have a differential action-value estimate for each pair of state (number of free servers and priority of the customer at the head of the queue) and action (accept or reject). Figure 10.5 shows the solution found by differential semi-gradient Sarsa for this task with $k = 10$ and $p = 0.06$. The algorithm parameters were $\alpha = 0.01$, $\beta = 0.01$, and $\epsilon = 0.1$. The initial action values and $\bar{R}$ were zero.

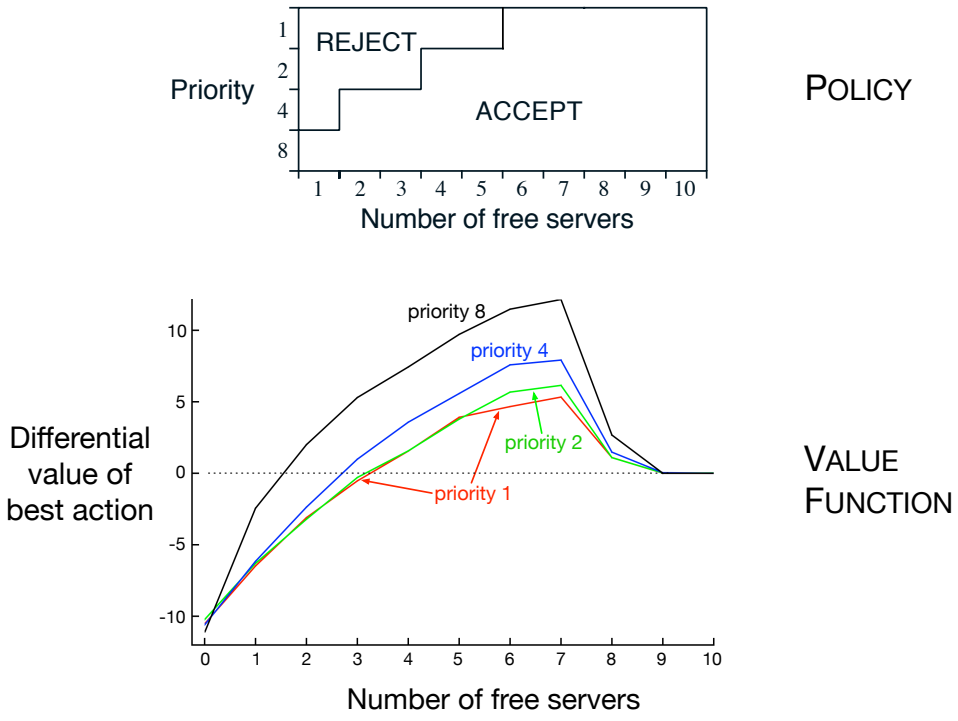


Figure 10.5: The policy and value function found by differential semi-gradient one-step Sarsa on the access-control queuing task after 2 million steps. The drop on the right of the graph is probably due to insufficient data; many of these states were never experienced. The value learned for $\bar{R}$ was about 2.31. ■

10.4 Deprecating the Discounted Setting

The continuing, discounted problem formulation has been very useful in the tabular case, in which the returns from each state can be separately identified and averaged. But in the approximate case it is questionable whether one should ever use this problem formulation.

To see why, consider an infinite sequence of returns with no beginning or end, and no clearly identified states. The states might be represented only by feature vectors, which may do little to distinguish the states from each other. As a special case, all of the feature vectors may be the same. Thus one really has only the reward sequence (and the actions), and performance has to be assessed purely from these. How could it be done? One way is by averaging the rewards over a long interval—this is the idea of the average-reward setting. How could discounting be used? Well, for each time step we could measure the discounted return. Some returns would be small and some big, so again we would have to average them over a sufficiently large time interval. In the continuing setting there are no starts and ends, and no special time steps, so there is nothing else that could be done. However, if you do this, it turns out that the average of the discounted returns is proportional to the average reward. In fact, for policy π, the average of the discounted returns is always $r(\pi)/(1-\gamma)$, that is, it is essentially the average reward, $r(\pi)$. In particular, the *ordering* of all policies in the average discounted return setting would be exactly the same as in the average-reward setting. The discount rate γ thus has no effect on the problem formulation. It could in fact be *zero* and the ranking would be unchanged.

This surprising fact is proven in the box, but the basic idea can be seen via a symmetry argument. Each time step is exactly the same as every other. With discounting, every reward will appear exactly once in each position in some return. The tth reward will appear undiscounted in the $t-1$st return, discounted once in the $t-2$nd return, and discounted 999 times in the $t-1000$th return. The weight on

The Futility of Discounting in Continuing Problems

Perhaps discounting can be saved by choosing an objective that sums discounted values over the distribution with which states occur under the policy:

$$
\begin{aligned}
J(\pi) &= \sum_s \mu_\pi(s) v_\pi^\gamma(s) && \text{(where } v_\pi^\gamma \text{ is the discounted value function)} \\
&= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \,|\, s, a) \left[r + \gamma v_\pi^\gamma(s') \right] && \text{(Bellman Eq.)} \\
&= r(\pi) + \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \,|\, s, a) \gamma v_\pi^\gamma(s') && \text{(from (10.6))} \\
&= r(\pi) + \gamma \sum_{s'} v_\pi^\gamma(s') \sum_s \mu_\pi(s) \sum_a \pi(a|s) p(s' \,|\, s, a) && \text{(from (3.3))} \\
&= r(\pi) + \gamma \sum_{s'} v_\pi^\gamma(s') \mu_\pi(s') && \text{(from (10.7))} \\
&= r(\pi) + \gamma J(\pi) \\
&= r(\pi) + \gamma r(\pi) + \gamma^2 J(\pi) \\
&= r(\pi) + \gamma r(\pi) + \gamma^2 r(\pi) + \gamma^3 r(\pi) + \cdots \\
&= \frac{1}{1-\gamma} r(\pi).
\end{aligned}
$$

The proposed discounted objective orders policies identically to the undiscounted (average reward) objective. We have failed to save discounting!

the tth reward is thus $1+\gamma+\gamma^2+\gamma^3+\cdots = 1/(1-\gamma)$. Since all states are the same, they are all weighted by this, and thus the average of the returns will be this times the average reward, or $r(\pi)/(1-\gamma)$.

So in this key case, what the discounted case was invented for, discounting is not applicable. The discounted case is still pertinent, or at least possible, for the episodic case.

10.5 n-step Differential Semi-gradient Sarsa

In order to generalize to *n-step* bootstrapping, we need an n-step version of the TD error. We begin by generalizing the n-step return (7.4) to its differential form, with function approximation:

$$G_{t:t+n} \doteq R_{t+1} - \bar{R}_{t+1} + R_{t+2} - \bar{R}_{t+2} + \cdots + R_{t+n} - \bar{R}_{t+n} + \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \quad (10.12)$$

where $\bar{R}$ is an estimate of $r(\pi)$, $n \geq 1$, and $t+n < T$. If $t+n \geq T$, then we define $G_{t:t+n} \doteq G_t$ as usual. The n-step TD error is then

$$\delta_t \doteq G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}), \quad (10.13)$$

after which we can apply our usual semi-gradient Sarsa update (10.11). Pseudocode for the complete algorithm is given in the box.

Differential semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_*$, or $\hat{q} \approx q_\pi$

Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^m \to \mathbb{R}$, a policy π
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^m$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average-reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)
Parameters: step size $\alpha, \beta > 0$, a positive integer n
All store and access operations (S_t, A_t, and R_t) can take their index mod n

Initialize and store S_0 and A_0
For $t = 0, 1, 2, \ldots$:
  Take action A_t
  Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$, or ε-greedy wrt $\hat{q}(S_0, \cdot, \mathbf{w})$
  $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
  If $\tau \geq 0$:
    $\delta \leftarrow \sum_{i=\tau+1}^{\tau+n} (R_i - \bar{R}) + \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w}) - \hat{q}(S_\tau, A_\tau, \mathbf{w})$
    $\bar{R} \leftarrow \bar{R} + \beta\delta$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S_\tau, A_\tau, \mathbf{w})$

10.6 Summary

In this chapter we have extended the ideas of parameterized function approximation and semi-gradient descent, introduced in the previous chapter, to control. The extension is immediate for the episodic case, but for the continuing case we have to introduce a whole new problem formulation based on maximizing the *average reward* per time step. Surprisingly, the discounted formulation cannot be carried over to control in the presence of approximations. In the approximate case most policies cannot be represented by a value function. The arbitrary policies that remain need to be ranked, and the scalar average reward $r(\pi)$ provides an effective way to do this.

The average reward formulation involves new *differential* versions of value functions, Bellman equations, and TD errors, but all of these parallel the old ones, and the conceptual changes are small. There is also a new parallel set of differential algorithms for the average-reward case. We illustrate this by developing differential versions of semi-gradient n-step Sarsa.

Bibliographical and Historical Remarks

10.1 Semi-gradient Sarsa with function approximation was first explored by Rummery and Niranjan (1994). Linear semi-gradient Sarsa with ε-greedy action selection does not converge in the usual sense, but does enter a bounded region near the best solution (Gordon, 1995). Precup and Perkins (2003) showed convergence in a differentiable action selection setting. See also Perkins and Pendrith (2002) and Melo, Meyn, and Ribiero (2008). The mountain–car example is based on a similar task studied by Moore (1990), but the exact form used here is from Sutton (1996).

10.2 Episodic n-step semi-gradient Sarsa is based on the forward Sarsa(λ) algorithm of van Seijen (2016). The empirical results shown here are new to the second edition of this text.

10.3 The average-reward formulation has been described for dynamic programming (e.g., Puterman, 1994) and from the point of view of reinforcement learning (Mahadevan, 1996; Tadepalli and Ok, 1994; Bertsekas and Tsitiklis, 1996; Tsitsiklis and Van Roy, 1999). The algorithm described here is the on-policy analog of the "R-learning" algorithm introduced by Schwartz (1993). The name R-learning was probably meant to be the alphabetic successor to Q-learning, but we prefer to think of it as a reference to the learning of differential or *relative* values. The access-control queuing example was suggested by the work of Carlström and Nordström (1997).

10.4 The recognition of the limitations of discounting as a formulation of the reinforcement learning problem with function approximation became apparent to the authors shortly after the publication of the first edition of this text. The second edition of this book may be the first publication of the demonstration of the futility of discounting in the box on page 205.

10.5 The differential version of n-step semi-gradient Sarsa is new to this text and has not been significantly studied.

Chapter 16

Applications and Case Studies

In this chapter we present a few case studies of reinforcement learning. Several of these are substantial applications of potential economic significance. One, Samuel's checkers player, is primarily of historical interest. Our presentations are intended to illustrate some of the trade-offs and issues that arise in real applications. For example, we emphasize how domain knowledge is incorporated into the formulation and solution of the problem. We also highlight the representation issues that are so often critical to successful applications. The algorithms used in some of these case studies are substantially more complex than those we have presented in the rest of the book. Applications of reinforcement learning are still far from routine and typically require as much art as science. Making applications easier and more straightforward is one of the goals of current research in reinforcement learning.

16.1 TD-Gammon

One of the most impressive applications of reinforcement learning to date is that by Gerald Tesauro to the game of backgammon (Tesauro, 1992, 1994, 1995, 2002). Tesauro's program, *TD-Gammon*, required little backgammon knowledge, yet learned to play extremely well, near the level of the world's strongest grandmasters. The learning algorithm in TD-Gammon was a straightforward combination of the TD(λ) algorithm and nonlinear function approximation using a multilayer neural network trained by backpropagating TD errors.

Backgammon is a major game in the sense that it is played throughout the world, with numerous tournaments and regular world championship matches. It is in part a game of chance, and it is a popular vehicle for waging significant sums of money. There are probably more professional backgammon players than there are professional chess players. The game is played with 15 white and 15 black pieces on a board of 24 locations, called *points*. Figure 16.1 shows a typical position early in the game, seen from the perspective of the white player.

In this figure, white has just rolled the dice and obtained a 5 and a 2. This means that he can move one of his pieces 5 steps and one (possibly the same piece) 2 steps. For example, he could move two pieces from the 12 point, one to the 17 point, and one to the 14 point. White's objective is to advance all of his pieces into the last quadrant (points 19–24) and then off the board. The first player to remove all his pieces wins. One complication is that the pieces interact as they pass each other going in different directions. For example, if it were black's move in Figure 16.1, he could use the dice roll of 2 to move a piece from the 24 point to the 22 point, "hitting" the white piece there. Pieces that have been hit are placed on the "bar" in the middle of the board (where we already see one previously hit black piece), from whence they reenter the race from the start. However, if there are two pieces on a point, then the opponent cannot move to that point; the pieces are protected from being hit. Thus, white cannot use

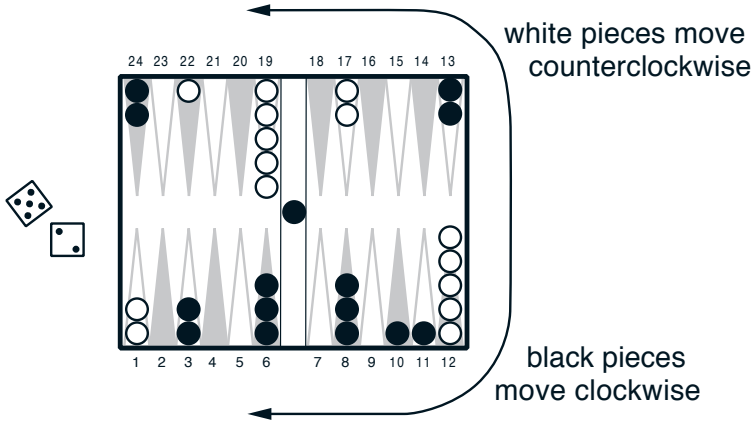


Figure 16.1: A backgammon position

his 5–2 dice roll to move either of his pieces on the 1 point, because their possible resulting points are occupied by groups of black pieces. Forming contiguous blocks of occupied points to block the opponent is one of the elementary strategies of the game.

Backgammon involves several further complications, but the above description gives the basic idea. With 30 pieces and 24 possible locations (26, counting the bar and off-the-board) it should be clear that the number of possible backgammon positions is enormous, far more than the number of memory elements one could have in any physically realizable computer. The number of moves possible from each position is also large. For a typical dice roll there might be 20 different ways of playing. In considering future moves, such as the response of the opponent, one must consider the possible dice rolls as well. The result is that the game tree has an effective branching factor of about 400. This is far too large to permit effective use of the conventional heuristic search methods that have proved so effective in games like chess and checkers.

On the other hand, the game is a good match to the capabilities of TD learning methods. Although the game is highly stochastic, a complete description of the game's state is available at all times. The game evolves over a sequence of moves and positions until finally ending in a win for one player or the other, ending the game. The outcome can be interpreted as a final reward to be predicted. On the other hand, the theoretical results we have described so far cannot be usefully applied to this task. The number of states is so large that a lookup table cannot be used, and the opponent is a source of uncertainty and time variation.

TD-Gammon used a nonlinear form of TD(λ). The estimated value, $\hat{v}(s,\mathbf{w})$, of any state (board position) s was meant to estimate the probability of winning starting from state s. To achieve this, rewards were defined as zero for all time steps except those on which the game is won. To implement the value function, TD-Gammon used a standard multilayer neural network, much as shown in Figure 16.2. (The real network had two additional units in its final layer to estimate the probability of each player's winning in a special way called a "gammon" or "backgammon.") The network consisted of a layer of input units, a layer of hidden units, and a final output unit. The input to the network was a representation of a backgammon position, and the output was an estimate of the value of that position.

In the first version of TD-Gammon, TD-Gammon 0.0, backgammon positions were represented to the network in a relatively direct way that involved little backgammon knowledge. It did, however, involve substantial knowledge of how neural networks work and how information is best presented to them. It is instructive to note the exact representation Tesauro chose. There were a total of 198 input units to the network. For each point on the backgammon board, four units indicated the number of white pieces on the point. If there were no white pieces, then all four units took on the value zero. If

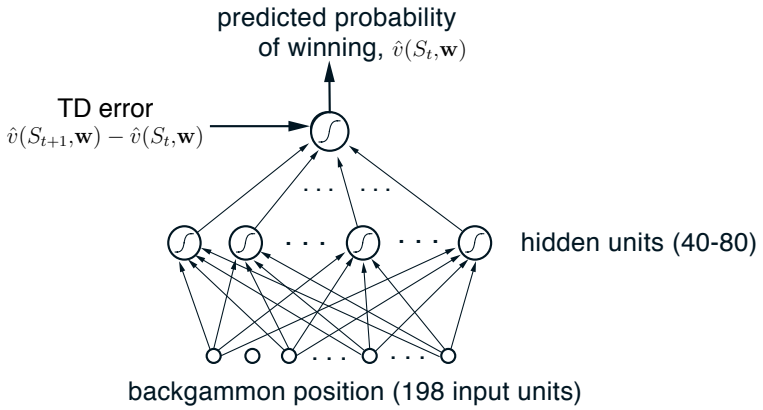


Figure 16.2: The neural network used in TD-Gammon

there was one piece, then the first unit took on the value 1. This encoded the elementary concept of a "blot," i.e., a piece that can be hit by the opponent. If there were two or more pieces, then the second unit was set to 1. This encoded the basic concept of a "made point" on which the opponent cannot land. If there were exactly three pieces on the point, then the third unit was set to 1. This encoded the basic concept of a "single spare," i.e., an extra piece in addition to the two pieces that made the point. Finally, if there were more than three pieces, the fourth unit was set to a value proportionate to the number of additional pieces beyond three. Letting n denote the total number of pieces on the point, if $n > 3$, then the fourth unit took on the value $(n-3)/2$. This encoded a linear representation of "multiple spares" at the given point.

With four units for white and four for black at each of the 24 points, that made a total of 192 units. Two additional units encoded the number of white and black pieces on the bar (each took the value $n/2$, where n is the number of pieces on the bar), and two more encoded the number of black and white pieces already successfully removed from the board (these took the value $n/15$, where n is the number of pieces already borne off). Finally, two units indicated in a binary fashion whether it was white's or black's turn to move. The general logic behind these choices should be clear. Basically, Tesauro tried to represent the position in a straightforward way, while keeping the number of units relatively small. He provided one unit for each conceptually distinct possibility that seemed likely to be relevant, and he scaled them to roughly the same range, in this case between 0 and 1.

Given a representation of a backgammon position, the network computed its estimated value in the standard way. Corresponding to each connection from an input unit to a hidden unit was a real-valued weight. Signals from each input unit were multiplied by their corresponding weights and summed at the hidden unit. The output, $h(j)$, of hidden unit j was a nonlinear sigmoid function of the weighted sum:

$$h(j) = \sigma\left(\sum_i w_{ij}x_i\right) = \frac{1}{1+e^{-\sum_i w_{ij}x_i}},$$

where x_i is the value of the ith input unit and w_{ij} is the weight of its connection to the jth hidden unit (all the weights in the network together make up the parameter vector $\mathbf{w}$). The output of the sigmoid is always between 0 and 1, and has a natural interpretation as a probability based on a summation of evidence. The computation from hidden units to the output unit was entirely analogous. Each connection from a hidden unit to the output unit had a separate weight. The output unit formed the weighted sum and then passed it through the same sigmoid nonlinearity.

TD-Gammon used the semi-gradient form of the TD(λ) algorithm described in Section 12.2, with the

gradients computed by the error backpropagation algorithm (Rumelhart, Hinton, and Williams, 1986). Recall that the general update rule for this case is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t)\Big]\mathbf{e}_t, \tag{16.1}$$

where $\mathbf{w}_t$ is the vector of all modifiable parameters (in this case, the weights of the network) and $\mathbf{e}_t$ is a vector of eligibility traces, one for each component of $\mathbf{w}_t$, updated by

$$\mathbf{e}_t \doteq \gamma\lambda\mathbf{e}_{t-1} + \nabla\hat{v}(S_t,\mathbf{w}_t),$$

with $\mathbf{e}_0 \doteq \mathbf{0}$. The gradient in this equation can be computed efficiently by the backpropagation procedure. For the backgammon application, in which $\gamma = 1$ and the reward is always zero except upon winning, the TD error portion of the learning rule is usually just $\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})$, as suggested in Figure 16.2.

To apply the learning rule we need a source of backgammon games. Tesauro obtained an unending sequence of games by playing his learning backgammon player against itself. To choose its moves, TD-Gammon considered each of the 20 or so ways it could play its dice roll and the corresponding positions that would result. The resulting positions are *afterstates* as discussed in Section 6.8. The network was consulted to estimate each of their values. The move was then selected that would lead to the position with the highest estimated value. Continuing in this way, with TD-Gammon making the moves for both sides, it was possible to easily generate large numbers of backgammon games. Each game was treated as an episode, with the sequence of positions acting as the states, $S_0, S_1, S_2, \ldots$. Tesauro applied the nonlinear TD rule (16.1) fully incrementally, that is, after each individual move.

The weights of the network were set initially to small random values. The initial evaluations were thus entirely arbitrary. Since the moves were selected on the basis of these evaluations, the initial moves were inevitably poor, and the initial games often lasted hundreds or thousands of moves before one side or the other won, almost by accident. After a few dozen games however, performance improved rapidly.

After playing about 300,000 games against itself, TD-Gammon 0.0 as described above learned to play approximately as well as the best previous backgammon computer programs. This was a striking result because all the previous high-performance computer programs had used extensive backgammon knowledge. For example, the reigning champion program at the time was, arguably, *Neurogammon*, another program written by Tesauro that used a neural network but not TD learning. Neurogammon's network was trained on a large training corpus of exemplary moves provided by backgammon experts, and, in addition, started with a set of features specially crafted for backgammon. Neurogammon was a highly tuned, highly effective backgammon program that decisively won the World Backgammon Olympiad in 1989. TD-Gammon 0.0, on the other hand, was constructed with essentially zero backgammon knowledge. That it was able to do as well as Neurogammon and all other approaches is striking testimony to the potential of self-play learning methods.

The tournament success of TD-Gammon 0.0 with zero expert backgammon knowledge suggested an obvious modification: add the specialized backgammon features but keep the self-play TD learning method. This produced TD-Gammon 1.0. TD-Gammon 1.0 was clearly substantially better than all previous backgammon programs and found serious competition only among human experts. Later versions of the program, TD-Gammon 2.0 (40 hidden units) and TD-Gammon 2.1 (80 hidden units), were augmented with a selective two-ply search procedure. To select moves, these programs looked ahead not just to the positions that would immediately result, but also to the opponent's possible dice rolls and moves. Assuming the opponent always took the move that appeared immediately best for him, the expected value of each candidate move was computed and the best was selected. To save computer time, the second ply of search was conducted only for candidate moves that were ranked highly after the first ply, about four or five moves on average. Two-ply search affected only the moves selected; the learning process proceeded exactly as before. The final versions of the program, TD-Gammon 3.0 and 3.1, used 160 hidden units and a selective three-ply search. TD-Gammon illustrates the combination of

Program	Hidden Units	Training Games	Opponents	Results
TD-Gam 0.0	40	300,000	other programs	tied for best
TD-Gam 1.0	80	300,000	Robertie, Magriel, ...	−13 pts / 51 games
TD-Gam 2.0	40	800,000	various Grandmasters	−7 pts / 38 games
TD-Gam 2.1	80	1,500,000	Robertie	−1 pt / 40 games
TD-Gam 3.0	80	1,500,000	Kazaros	+6 pts / 20 games

Table 16.1: Summary of TD-Gammon Results

learned value functions and decision-time search as in heuristic search and MCTS methods. In follow-on work, Tesauro and Galperin (1997) explored trajectory sampling methods as an alternative to full-width search, which reduced the error rate of live play by large numerical factors (4x–6x) while keeping the think time reasonable at $\sim$ 5–10 seconds per move.

During the 1990s, Tesauro was able to play his programs in a significant number of games against world-class human players. A summary of the results is given in Table 16.1. Based on these results and analyses by backgammon grandmasters (Robertie, 1992; see Tesauro, 1995), TD-Gammon 3.0 appeared to play at close to, or possibly better than, the playing strength of the best human players in the world. Tesauro reported in a subsequent article (Tesauro, 2002) the results of an extensive rollout analysis of the move decisions and doubling decisions of TD-Gammon relative to top human players. The conclusion was that TD-Gammon 3.1 had a "lopsided advantage" in piece-movement decisions, and a "slight edge" in doubling decisions, over top humans.

TD-Gammon had a significant impact on the way the best human players play the game. For example, it learned to play certain opening positions differently than was the convention among the best human players. Based on TD-Gammon's success and further analysis, the best human players now play these positions as TD-Gammon does (Tesauro, 1995). The impact on human play was greatly accelerated when several other self-teaching neural net backgammon programs inspired by TD-Gammon, such as Jellyfish, Snowie, and GNUBackgammon, became widely available. These programs enabled wide dissemination of new knowledge generated by the neural nets, resulting in great improvements in the overall caliber of human tournament play (Tesauro, 2002).

16.2 Samuel's Checkers Player

An important precursor to Tesauro's TD-Gammon was the seminal work of Arthur Samuel (1959, 1967) in constructing programs for learning to play checkers. Samuel was one of the first to make effective use of heuristic search methods and of what we would now call temporal-difference learning. His checkers players are instructive case studies in addition to being of historical interest. We emphasize the relationship of Samuel's methods to modern reinforcement learning methods and try to convey some of Samuel's motivation for using them.

Samuel first wrote a checkers-playing program for the IBM 701 in 1952. His first *learning* program was completed in 1955 and was demonstrated on television in 1956. Later versions of the program achieved good, though not expert, playing skill. Samuel was attracted to game-playing as a domain for studying machine learning because games are less complicated than problems "taken from life" while still allowing fruitful study of how heuristic procedures and learning can be used together. He chose to study checkers instead of chess because its relative simplicity made it possible to focus more strongly on learning.

Samuel's programs played by performing a lookahead search from each current position. They used what we now call heuristic search methods to determine how to expand the search tree and when to stop

searching. The terminal board positions of each search were evaluated, or "scored," by a value function, or "scoring polynomial," using linear function approximation. In this and other respects Samuel's work seems to have been inspired by the suggestions of Shannon (1950). In particular, Samuel's program was based on Shannon's minimax procedure to find the best move from the current position. Working backward through the search tree from the scored terminal positions, each position was given the score of the position that would result from the best move, assuming that the machine would always try to maximize the score, while the opponent would always try to minimize it. Samuel called this the "backed-up score" of the position. When the minimax procedure reached the search tree's root—the current position—it yielded the best move under the assumption that the opponent would be using the same evaluation criterion, shifted to its point of view. Some versions of Samuel's programs used sophisticated search control methods analogous to what are known as "alpha-beta" cutoffs (e.g., see Pearl, 1984).

Samuel used two main learning methods, the simplest of which he called *rote learning*. It consisted simply of saving a description of each board position encountered during play together with its backed-up value determined by the minimax procedure. The result was that if a position that had already been encountered were to occur again as a terminal position of a search tree, the depth of the search was effectively amplified since this position's stored value cached the results of one or more searches conducted earlier. One initial problem was that the program was not encouraged to move along the most direct path to a win. Samuel gave it a "a sense of direction" by decreasing a position's value a small amount each time it was backed up a level (called a ply) during the minimax analysis. "If the program is now faced with a choice of board positions whose scores differ only by the ply number, it will automatically make the most advantageous choice, choosing a low-ply alternative if winning and a high-ply alternative if losing" (Samuel, 1959, p. 80). Samuel found this discounting-like technique essential to successful learning. Rote learning produced slow but continuous improvement that was most effective for opening and endgame play. His program became a "better-than-average novice" after learning from many games against itself, a variety of human opponents, and from book games in a supervised learning mode.

Rote learning and other aspects of Samuel's work strongly suggest the essential idea of temporal-difference learning—that the value of a state should equal the value of likely following states. Samuel came closest to this idea in his second learning method, his "learning by generalization" procedure for modifying the parameters of the value function. Samuel's method was the same in concept as that used much later by Tesauro in TD-Gammon. He played his program many games against another version of itself and performed an update after each move. The idea of Samuel's update is suggested by the diagram in Figure 16.3. Each open circle represents a position where the program moves next, an *on-move* position, and each solid circle represents a position where the opponent moves next. An update was made to the value of each on-move position after a move by each side, resulting in a second on-move position. The update was toward the minimax value of a search launched from the second on-move position. Thus, the overall effect was that of a backing-up over one full move of real events and then a search over possible events, as suggested by Figure 16.3. Samuel's actual algorithm was significantly more complex than this for computational reasons, but this was the basic idea.

Samuel did not include explicit rewards. Instead, he fixed the weight of the most important feature, the *piece advantage* feature, which measured the number of pieces the program had relative to how many its opponent had, giving higher weight to kings, and including refinements so that it was better to trade pieces when winning than when losing. Thus, the goal of Samuel's program was to improve its piece advantage, which in checkers is highly correlated with winning.

However, Samuel's learning method may have been missing an essential part of a sound temporal-difference algorithm. Temporal-difference learning can be viewed as a way of making a value function consistent with itself, and this we can clearly see in Samuel's method. But also needed is a way of tying the value function to the true value of the states. We have enforced this via rewards and by discounting or giving a fixed value to the terminal state. But Samuel's method included no rewards and no special

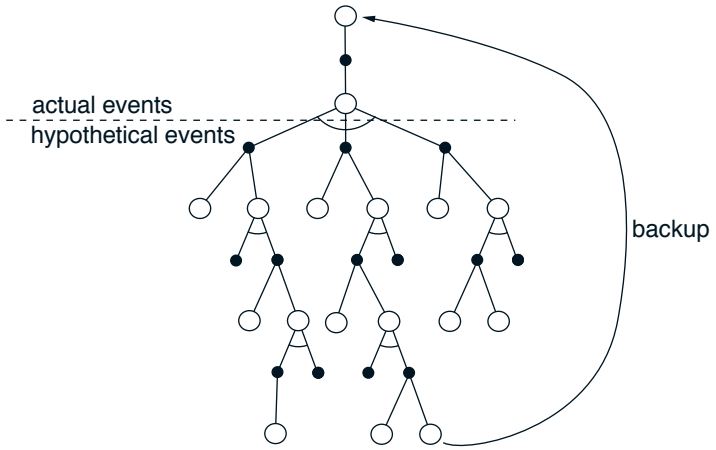


Figure 16.3: The update diagram for Samuel's checkers player.

treatment of the terminal positions of games. As Samuel himself pointed out, his value function could have become consistent merely by giving a constant value to all positions. He hoped to discourage such solutions by giving his piece-advantage term a large, nonmodifiable weight. But although this may decrease the likelihood of finding useless evaluation functions, it does not prohibit them. For example, a constant function could still be attained by setting the modifiable weights so as to cancel the effect of the nonmodifiable one.

Since Samuel's learning procedure was not constrained to find useful evaluation functions, it should have been possible for it to become worse with experience. In fact, Samuel reported observing this during extensive self-play training sessions. To get the program improving again, Samuel had to intervene and set the weight with the largest absolute value back to zero. His interpretation was that this drastic intervention jarred the program out of local optima, but another possibility is that it jarred the program out of evaluation functions that were consistent but had little to do with winning or losing the game.

Despite these potential problems, Samuel's checkers player using the generalization learning method approached "better-than-average" play. Fairly good amateur opponents characterized it as "tricky but beatable" (Samuel, 1959). In contrast to the rote-learning version, this version was able to develop a good middle game but remained weak in opening and endgame play. This program also included an ability to search through sets of features to find those that were most useful in forming the value function. A later version (Samuel, 1967) included refinements in its search procedure, such as alpha-beta pruning, extensive use of a supervised learning mode called "book learning," and hierarchical lookup tables called signature tables (Griffith, 1966) to represent the value function instead of linear function approximation. This version learned to play much better than the 1959 program, though still not at a master level. Samuel's checkers-playing program was widely recognized as a significant achievement in artificial intelligence and machine learning.

16.3 The Acrobot

Reinforcement learning has been applied to a wide variety of physical control tasks (e.g., for a collection of robotics applications, see Kober and Peters, 2012). One such task is the *acrobot*, a two-link, under-actuated robot roughly analogous to a gymnast swinging on a high bar (Figure 16.4). The first joint (corresponding to the gymnast's hands on the bar) cannot exert torque, but the second joint (corresponding to the gymnast bending at the waist) can. The system has four continuous state variables: two joint positions and two joint velocities. The equations of motion are given in Figure 16.5. This system

has been widely studied by control engineers (e.g., Spong, 1994) and machine-learning researchers (e.g., Dejong and Spong, 1994; Boone, 1997).

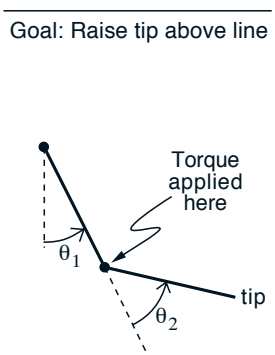


Figure 16.4: The acrobot.

One objective for controlling the acrobot is to swing the tip (the "feet") above the first joint by an amount equal to one of the links in minimum time. In this task, the torque applied at the second joint is limited to three choices: positive torque of a fixed magnitude, negative torque of the same magnitude, or no torque. A reward of -1 is given on all time steps until the goal is reached, which ends the episode. No discounting is used ($\gamma = 1$). Thus, the optimal value, $v_*(s)$, of any state, s, is the minimum time to reach the goal (an integer number of steps) starting from s.

Sutton (1996) addressed the acrobot swing-up task in an on-line, model-free context. Although the acrobot was simulated, the simulator was not available for use by the agent/controller in any way. The training and interaction were just as if a real, physical acrobot had been used. Each episode began with both links of the acrobot hanging straight down and at rest. Torques were applied by the reinforcement learning agent until the goal was reached, which always happened eventually. Then the acrobot was restored to its initial rest position and a new episode was begun.

The learning algorithm used was Sarsa(λ) with linear function approximation, tile coding, and replacing traces as in Section 12.8. With a small, discrete action set, it is natural to use a separate set of tilings for each action. The next choice is of the continuous variables with which to represent the state. A clever designer would probably represent the state in terms of the angular position and velocity of the center of mass and of the second link, which might make the solution simpler and consistent with broad generalization. But since this was just a test problem, a more naive, direct representation was used in terms of the positions and velocities of the links: $\theta_1, \dot{\theta}_1, \theta_2$, and $\dot{\theta}_2$. The two angles are restricted to a limited range by the physics of the acrobot (see Figure 16.5) and the two angles are naturally restricted to $[0, 2\pi]$. Thus, the state space in this task is a bounded rectangular region in four dimensions.

$$
\begin{aligned}
\ddot{\theta}_1 &= -d_1^{-1}(d_2\ddot{\theta}_2 + \phi_1) \\
\ddot{\theta}_2 &= \left(m_2 l_{c2}^2 + I_2 - \frac{d_2^2}{d_1}\right)^{-1} \left(\tau + \frac{d_2}{d_1}\phi_1 - m_2 l_1 l_{c2} \dot{\theta}_1^2 \sin\theta_2 - \phi_2\right),
\end{aligned}
$$

where

$$
\begin{aligned}
d_1 &= m_1 l_{c1}^2 + m_2(l_1^2 + l_{c2}^2 + 2 l_1 l_{c2} \cos\theta_2) + I_1 + I_2 \\
d_2 &= m_2(l_{c2}^2 + l_1 l_{c2} \cos\theta_2) + I_2 \\
\phi_1 &= -m_2 l_1 l_{c2} \dot{\theta}_2^2 \sin\theta_2 - 2 m_2 l_1 l_{c2} \dot{\theta}_2 \dot{\theta}_1 \sin\theta_2 \\
&\quad + (m_1 l_{c1} + m_2 l_1) g \cos(\theta_1 - \pi/2) + \phi_2 \\
\phi_2 &= m_2 l_{c2} g \cos(\theta_1 + \theta_2 - \pi/2).
\end{aligned}
$$

Figure 16.5: The equations of motions of the simulated acrobot. A time step of 0.05 seconds was used in the simulation, with actions chosen after every four time steps. The torque applied at the second joint is denoted by $\tau \in \{+1, -1, 0\}$. There were no constraints on the joint positions, but the angular velocities were limited to $\dot{\theta}_1 \in [-4\pi, 4\pi]$ and $\dot{\theta}_2 \in [-9\pi, 9\pi]$. The constants were $m_1 = m_2 = 1$ (masses of the links), $l_1 = l_2 = 1$ (lengths of links), $l_{c1} = l_{c2} = 0.5$ (lengths to center of mass of links), $I_1 = I_2 = 1$ (moments of inertia of links), and $g = 9.8$ (gravity).

This leaves the question of what tilings to use. There are many possibilities, as discussed in Chapter 9. One is to use a complete grid, slicing the four-dimensional space along all dimensions, and thus into many small four-dimensional tiles. Alternatively, one could slice along only one of the dimensions, making hyperplanar stripes. In this case one has to pick which dimension to slice along. And of course in all cases one has to pick the width of the slices, the number of tilings of each kind, and, if there are multiple tilings, how to offset them. One could also slice along pairs or triplets of dimensions to get other tilings. For example, if one expected the velocities of the two links to interact strongly in their effect on value, then one might make many tilings that sliced along both of these dimensions. If one thought the region around zero velocity was particularly critical, then the slices could be more closely spaced there.

Sutton used tilings that sliced in a variety of simple ways. Each of the four dimensions was divided into six equal intervals. A seventh interval was added to the angular velocities so that tilings could be offset by a random fraction of an interval in all dimensions (see Chapter 9, subsection "Tile Coding"). Of the total of 48 tilings, 12 sliced along all four dimensions as discussed above, dividing the space into $6 \times 7 \times 6 \times 7 = 1764$ tiles each. Another 12 tilings sliced along three dimensions (3 randomly offset tilings each for each of the 4 sets of three dimensions), and another 12 sliced along two dimensions (2 tilings for each of the 6 sets of two dimensions. Finally, a set of 12 tilings depended each on only one dimension (3 tilings for each of the 4 dimensions). This resulted in a total of approximately $25,000$ tiles for each action. This number is small enough that hashing was not necessary. All tilings were offset by a random fraction of an interval in all relevant dimensions.

The remaining parameters of the learning algorithm were $\alpha = 0.2/48$, $\lambda = 0.9$, $\epsilon = 0$, and $\mathbf{w}_0 = 0$. The use of a greedy policy ($\varepsilon = 0$) seemed preferable on this task because long sequences of correct actions are needed to do well. One exploratory action could spoil a whole sequence of good actions. Exploration was ensured instead by starting the action values optimistically, at the low value of 0. As discussed in Section 2.7 and Example 9.2, this makes the agent continually disappointed with whatever rewards it initially experiences, driving it to keep trying new things.

Figure 16.6 shows learning curves for the acrobot task and the learning algorithm described above. Note from the single-run curve that single episodes were sometimes extremely long. On these episodes, the acrobot was usually spinning repeatedly at the second joint while the first joint changed only slightly from vertical down. Although this often happened for many time steps, it always eventually ended as the action values were driven lower. All runs ended with an efficient policy for solving the problem, usually lasting about 75 steps. A typical final solution is shown in Figure 16.7. First the acrobot pumps back and forth several times symmetrically, with the second link always down. Then, once enough energy has been added to the system, the second link is swung upright and stabbed to the goal height.

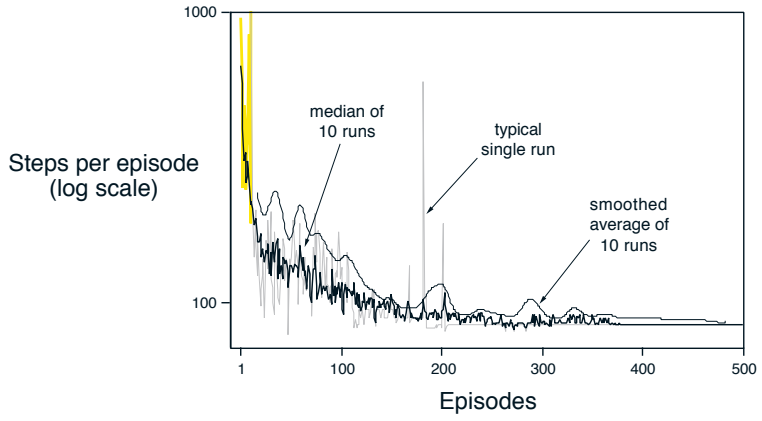


Figure 16.6: Learning curves for Sarsa(λ) on the acrobot task.

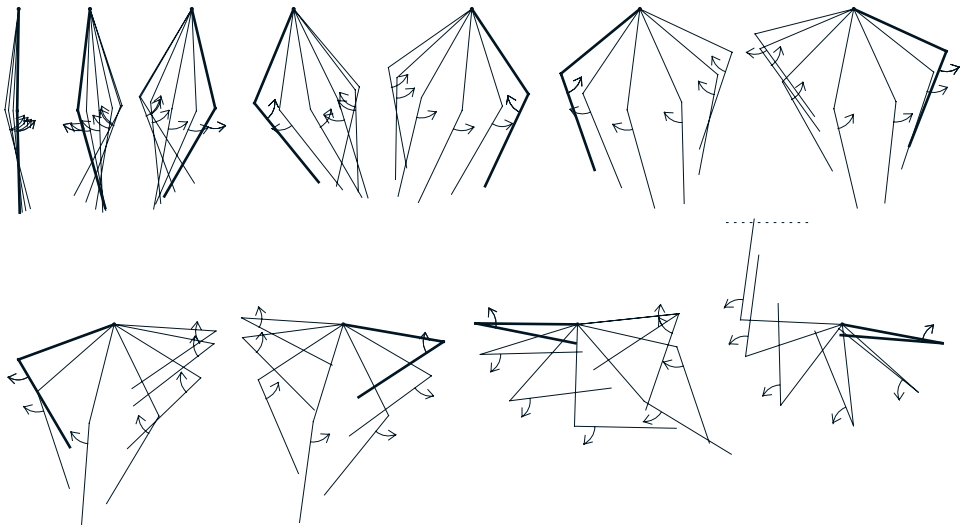


Figure 16.7: A typical learned behavior of the acrobot. Each group is a series of consecutive positions, the thicker line being the first. The arrow indicates the torque applied at the second joint.

16.4 Watson's Daily-Double Wagering

IBM Watson[1] is the system developed by a team of IBM researchers to play the popular TV quiz show *Jeopardy!*.[2] It gained fame in 2011 by winning first prize in an exhibition match against human champions. Although the main technical achievement demonstrated by Watson was its ability to quickly and accurately answer natural language questions over broad areas of general knowledge, its winning *Jeopardy!* performance also relied on sophisticated decision-making strategies for critical parts of the game. Tesauro, Gondek, Lechner, Fan, and Prager (2012, 2013) adapted Tesauro's TD-Gammon system described above to create the strategy used by Watson in "Daily-Double" (DD) wagering in its celebrated winning performance against human champions. These authors report that the effectiveness of this wagering strategy went well beyond what human players are able to do in live game play, and that it, along with other advanced strategies, was an important contributor to Watson's impressive winning performance. Here we focus only on DD wagering because it is the component of Watson that owes the most to reinforcement learning.

Jeopardy! is played by three contestants who face a board showing 30 squares, each of which hides a clue and has a dollar value. The squares are arranged in six columns, each corresponding to a different category. A contestant selects a square, the host reads the square's clue, and each contestant may choose to respond to the clue by sounding a buzzer ("buzzing in"). The first contestant to buzz in gets to try responding to the clue. If this contestant's response is correct, their score increases by the dollar value of the square; if their response is not correct, or if they do not respond within five seconds, their score decreases by that amount, and the other contestants get a chance to buzz in to respond to the same clue. One or two squares (depending on the game's current round) are special DD squares. A contestant who selects one of these gets an exclusive opportunity to respond to the square's clue and has to decide—before the clue is revealed—on how much to wager, or bet. The bet has to be greater than five dollars but not greater than the contestant's current score. If the contestant responds correctly to the DD clue, their score increases by the bet amount; otherwise it decreases by the bet amount. At the end of each game is a "Final Jeopardy" (FJ) round in which each contestant writes down a sealed

[1] Registered trademark of IBM Corp.

[2] Registered trademark of Jeopardy Productions Inc.

bet and then writes an answer after the clue is read. The contestant with the highest score after three rounds of play (where a round consists of revealing all 30 clues) is the winner. The game has many other details, but these are enough to appreciate the importance of DD wagering. Winning or losing often depends on a contestant's DD wagering strategy.

Whenever WATSON selected a DD square, it chose its bet by comparing action values, $\hat{q}(s, bet)$, that estimated the probability of a win from the current game state, s, for each round-dollar legal bet. Except for some risk-abatement measures described below, WATSON selected the bet with the maximum action value. Action values were computed whenever a betting decision was needed by using two types of estimates that were learned before any live game play took place. The first were estimated values of the afterstates (Section 6.8) that would result from selecting each legal bet. These estimates were obtained from a state-value function, $\hat{v}(\cdot,\mathbf{w})$, defined by parameters $\mathbf{w}$, that gave estimates of the probability of a win for WATSON from any game state. The second estimates used to compute action values gave the "in-category DD confidence," p_{DD}, which estimated the likelihood that WATSON would respond correctly to the as-yet unrevealed DD clue.

Tesauro et al. used the reinforcement learning approach of TD-Gammon described above to learn $\hat{v}(\cdot,\mathbf{w})$: a straightforward combination of nonlinear TD(λ) using a multilayer neural network with weights $\mathbf{w}$ trained by backpropagating TD errors during many simulated games. States were represented to the network by feature vectors specifically designed for *Jeopardy!*. Features included the current scores of the three players, how many DDs remained, the total dollar value of the remaining clues, and other information related to the amount of play left in the game. Unlike TD-Gammon, which learned by self-play, WATSON's $\hat{v}$ was learned over millions of simulated games against carefully-crafted models of human players. In-category confidence estimates were conditioned on the number of right responses r and wrong responses w that WATSON gave in previously-played clues in the current category. The dependencies on (r, w) were estimated from WATSON's actual accuracies over many thousands of historical categories.

With the previously learned value function $\hat{v}$ and in-category DD confidence p_{DD}, WATSON computed $\hat{q}(s, bet)$ for each legal round-dollar bet as follows:

$$\hat{q}(s, bet) = p_{DD} \times \hat{v}(S_W + bet,\ldots) + (1 - p_{DD}) \times \hat{v}(S_W - bet,\ldots), \tag{16.2}$$

where S_W is WATSON's current score, and $\hat{v}$ gives the estimated value for the game state after WATSON's response to the DD clue, which is either correct or incorrect. Computing an action value this way corresponds to the insight from Exercise 3.16 that an action value is the expected next state value given the action (except that here it is the expected next *afterstate* value because the full next state of the entire game depends on the next square selection).

Tesauro et al. found that selecting bets by maximizing action values incurred "a frightening amount of risk," meaning that if WATSON's response to the clue happened to be wrong, the loss could be disastrous for its chances of winning. To decrease the downside risk of a wrong answer, Tesauro et al. adjusted (16.2) by subtracting a small fraction of the standard deviation over WATSON's correct/incorrect afterstate evaluataions. They further reduced risk by prohibiting bets that would cause the wrong-answer afterstate value to decrease below a certain limit. These measures slightly reduced WATSON's expectation of winning, but they significantly reduced downside risk, not only in terms of average risk per DD bet, but even more so in extreme-risk scenarios where a risk-neutral WATSON would bet most or all of its bankroll.

Why was the TD-Gammon method of self-play not used to learn the critical value function $\hat{v}$? Learning from self-play in *Jeopardy!* would not have worked very well because WATSON was so different from any human contestant. Self-play would have led to exploration of state space regions that are not typical for play against human opponents, particularly human champions. In addition, unlike backgammon, *Jeopardy!* is a game of imperfect information because contestants do not have access to all the information influencing their opponents' play. In particular, *Jeopardy!* contestants do not know

how much confidence their opponents have for responding to clues in the various categories. Self-play would have been something like playing poker with someone who is holding the same cards that you hold.

As a result of these complications, much of the effort in developing WATSON's DD-wagering strategy was devoted to creating good models of human opponents. The models did not address the natural language aspect of the game, but were instead stochastic process models of events that can occur during play. Statistics were extracted from an extensive fan-created archive of game information from the beginning of the show to the present day. The archive includes information such as the ordering of the clues, right and wrong contestant answers, DD locations, and DD and FJ bets for nearly 300,000 clues. Three models were constructed: an Average Contestant model (based on all the data), a Champion model (based on statistics from games with the 100 best players), and a Grand Champion model (based on statistics from games with the 10 best players). In addition to serving as opponents during learning, the models were used to asses the benefits produced by the learned DD-wagering strategy. WATSON's win rate in simulation when it used a baseline heuristic DD-wagering strategy was 61%; when it used the learned values and a default confidence value, its win rate increased to 64%; and with live in-category confidence, it was 67%. Tesauro et al. regarded this as a significant improvement, given that the DD wagering was needed only about 1.5 to 2 times in each game.

Because WATSON had only a few seconds to bet, as well as to select squares and decide whether or not to buzz in, the computation time needed to make these decisions was a critical factor. The neural network implementation of $\hat{v}$ allowed DD bets to be made quickly enough to meet the time constraints of live play. However, once games could be simulated fast enough through improvements in the simulation software, near the end of a game it was feasible to estimate the value of bets by averaging over many Monte-Carlo trials in which the consequence of each bet was determined by simulating play to the game's end. Selecting endgame DD bets in live play based on Monte-Carlo trials instead of the neural network significantly improved WATSON's performance because errors in value estimates in endgames could seriously affect its chances of winning. Making all the decisions via Monte-Carlo trials might have led to better wagering decisions, but this was simply impossible given the complexity of the game and the time constraints of live play.

Although its ability to quickly and accurately answer natural language questions stands out as WATSON's major achievement, all of its sophisticated decision strategies contributed to its impressive defeat of human champions. According to Tesauro et al. (2012):

> ... it is plainly evident that our strategy algorithms achieve a level of quantitative precision and real-time performance that exceeds human capabilities. This is particularly true in the cases of DD wagering and endgame buzzing, where humans simply cannot come close to matching the precise equity and confidence estimates and complex decision calculations performed by Watson.

16.5 Optimizing Memory Control

Most computers use dynamic random access memory (DRAM) as their main memory because of its low cost and high capacity. The job of a DRAM memory controller is to efficiently use the interface between the processor chip and an off-chip DRAM system to provide the high-bandwidth and low-latency data transfer necessary for high-speed program execution. A memory controller needs to deal with dynamically changing patterns of read/write requests while adhering to a large number of timing and resource constraints required by the hardware. This is a formidable scheduling problem, especially with modern processors with multiple cores sharing the same DRAM.

İpek, Mutlu, Martínez, and Caruana (2008) (also Martínez and İpek, 2009) designed a reinforcement learning memory controller and demonstrated that it can significantly improve the speed of program

execution over what was possible with conventional controllers at the time of their research. They were motivated by limitations of existing state-of-the-art controllers that used policies that did not take advantage of past scheduling experience and did not account for long-term consequences of scheduling decisions. İpek et al.'s project was carried out by means of simulation, but they designed the controller at the detailed level of the hardware needed to implement it—including the learning algorithm—directly on a processor chip.

Accessing DRAM involves a number of steps that have to be done according to strict time constraints. DRAM systems consist of multiple DRAM chips, each containing multiple rectangular arrays of storage cells arranged in rows and columns. Each cell stores a bit as the charge on a capacitor. Since the charge decreases over time, each DRAM cell needs to be recharged—refreshed—every few milliseconds to prevent memory content from being lost. This need to refresh the cells is why DRAM is called "dynamic."

Each cell array has a row buffer that holds a row of bits that can be transferred into or out of one of the array's rows. An *activate* command "opens a row," which means moving the contents of the row whose address is indicated by the command into the row buffer. With a row open, the controller can issue *read* and *write* commands to the cell array. Each read command transfers a word (a short sequence of consecutive bits) in the row buffer to the external data bus, and each write command transfers a word in the external data bus to the row buffer. Before a different row can be opened, a *precharge* command must be issued which transfers the (possibly updated) data in the row buffer back into the addressed row of the cell array. After this, another activate command can open a new row to be accessed. Read and write commands are *column commands* because they sequentially transfer bits into or out of columns of the row buffer; multiple bits can be transferred without re-opening the row. Read and write commands to the currently-open row can be carried out more quickly than accessing a different row, which would involve additional *row commands*: precharge and activate; this is sometimes referred to as "row locality." A memory controller maintains a *memory transaction queue* that stores memory-access requests from the processors sharing the memory system. The controller has to process requests by issuing commands to the memory system while adhering to a large number of timing constraints.

A controller's policy for scheduling access requests can have a large effect on the performance of the memory system, such as the average latency with which requests can be satisfied and the throughput the system is capable of achieving. The simplest scheduling strategy handles access requests in the order in which they arrive by issuing all the commands required by the request before beginning to service the next one. But if the system is not ready for one of these commands, or executing a command would result in resources being underutilized (e.g., due to timing constraints arising from servicing that one command), it makes sense to begin servicing a newer request before finishing the older one. Policies can gain efficiency by reordering requests, for example, by giving priority to read requests over write requests, or by giving priority to read/write commands to already open rows. The policy called First-Ready, First-Come-First-Serve (FR-FCFS), gives priority to column commands (read and write) over row commands (activate and precharge), and in case of a tie gives priority to the oldest command. FR-FCFS was shown to outperform other scheduling policies in terms of average memory-access latency under conditions commonly encountered (Rixner, 2004).

Figure 16.8 is a high-level view of İpek et al.'s reinforcement learning memory controller. They modeled the DRAM access process as an MDP whose states are the contents of the transaction queue and whose actions are commands to the DRAM system: *precharge*, *activate*, *read*, *write*, and *NoOp*. The reward signal is 1 whenever the action is *read* or *write*, and otherwise it is 0. State transitions were considered to be stochastic because the next state of the system not only depends on the scheduler's command, but also on aspects of the system's behavior that the scheduler cannot control, such as the workloads of the processor cores accessing the DRAM system.

Critical to this MDP are constraints on the actions available in each state. Recall from Chapter 3 that the set of available actions can depend on the state: $A_t \in \mathcal{A}(S_t)$, where A_t is the action at time

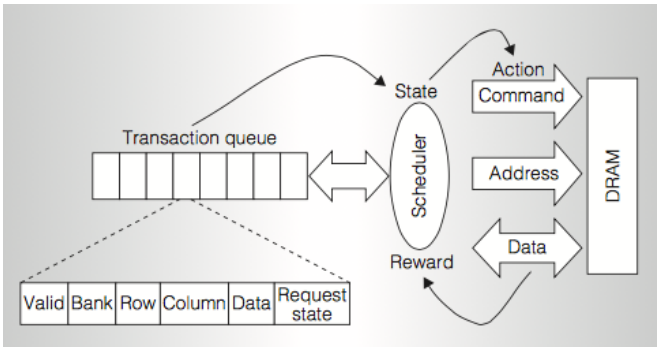


Figure 16.8: High-level view of the reinforcement learning DRAM controller. The scheduler is the reinforcement learning agent. Its environment is represented by features of the transaction queue, and its actions are commands to the DRAM system. ©2009 IEEE. Reprinted, with permission, from J. F. Martínez and E. İpek, Dynamic multicore resource management: A machine learning approach, Micro, IEEE, 29(5), p. 12.

step t and $\mathcal{A}(S_t)$ is the set of actions available in state S_t. In this application, the integrity of the DRAM system was assured by not allowing actions that would violate timing or resource constraints. Although İpek et al. did not make it explicit, they effectively accomplished this by pre-defining the sets $\mathcal{A}(S_t)$ for all possible states S_t.

These constraints explain why the MDP has a *NoOp* action and why the reward signal is 0 except when a *read* or *write* command is issued. *NoOp* is issued when it is the sole legal action in a state. To maximize utilization of the memory system, the controller's task is to drive the system to states in which either a *read* or a *write* action can be selected: only these actions result in sending data over the external data bus, so it is only these that contribute to the throughput of the system. Although *precharge* and *activate* produce no immediate reward, the agent needs to select these actions to make it possible to later select the rewarded *read* and *write* actions.

The scheduling agent used Sarsa (Figure 6.4) to learn an action-value function. States were represented by six integer-valued features. To approximate the action-value function, the algorithm used linear function approximation implemented by tile coding with hashing (Section 9.5.4). The tile coding had 32 tilings, each storing 256 action values as 16-bit fixed point numbers. Exploration was ϵ-greedy with $\epsilon = 0.05$.

State features included the number of read requests in the transaction queue, the number of write requests in the transaction queue, the number of write requests in the transaction queue waiting for their row to be opened, and the number of read requests in the transaction queue waiting for their row to be opened that are the oldest issued by their requesting processors. (The other features depended on how the DRAM interacts with cache memory, details we omit here.) The selection of the state features was based on İpek et al.'s understanding of factors that impact DRAM performance. For example, balancing the rate of servicing reads and writes based on how many of each are in the transaction queue can help avoid stalling the DRAM system's interaction with cache memory. The authors in fact generated a relatively long list of potential features, and then pared them down to a handful using simulations guided by stepwise feature selection.

An interesting aspect of this formulation of the scheduling problem as an MDP is that the features input to the tile coding for defining the action-value function were different from the features used to specify the action-constraint sets $\mathcal{A}(S_t)$. Whereas the tile coding input was derived from the contents of the transaction queue, the constraint sets depended on a host of other features related to timing and resource constraints that had to be satisfied by the hardware implementation of the entire system. In this way, the action constraints ensured that the learning algorithm's exploration could not endanger

the integrity of the physical system, while learning was effectively limited to a "safe" region of the much larger state space of the hardware implementation.

Since an objective of this work was that the learning controller could be implemented on a chip so that learning could occur on-line while a computer is running, hardware implementation details were important considerations. The design included two five-stage pipelines to calculate and compare two action values at every processor clock cycle, and to update the appropriate action value. This included accessing the tile coding which was stored on-chip in static RAM. For the configuration İpek et al. simulated, which was a 4GHz 4-core chip typical of high-end workstations at the time of their research, there were 10 processor cycles for every DRAM cycle. Considering the cycles needed to fill the pipes, up to 12 actions could be evaluated in each DRAM cycle. İpek et al. found that the number of legal commands for any state was rarely greater than this, and that performance loss was negligible if enough time was not always available to consider all legal commands. These and other clever design details made it feasible to implement the complete controller and learning algorithm on a multi-processor chip.

İpek et al. evaluated their learning controller in simulation by comparing it with three other controllers: 1) the FR-FCFS controller mentioned above that produces the best on-average performance, 2) a conventional controller that processes each request in order, and 3) an unrealizable ideal controller, called the Optimistic controller, able to sustain 100% DRAM throughput if given enough demand by ignoring all timing and resource constraints, but otherwise modeling DRAM latency (as row buffer hits) and bandwidth. They simulated nine memory-intensive parallel workloads consisting of scientific and data-mining applications. Figure 16.9 shows the performance (the inverse of execution time normalized to the performance of FR-FCFS) of each controller for the nine applications, together with the geometric mean of their performances over the applications. The learning controller, labeled RL in the figure, improved over that of FR-FCFS by from 7% to 33% over the nine applications, with an average improvement of 19%. Of course, no realizable controller can match the performance of Optimistic, which ignores all timing and resource constraints, but the learning controller's performance closed the gap with Optimistic's upper bound by an impressive 27%.

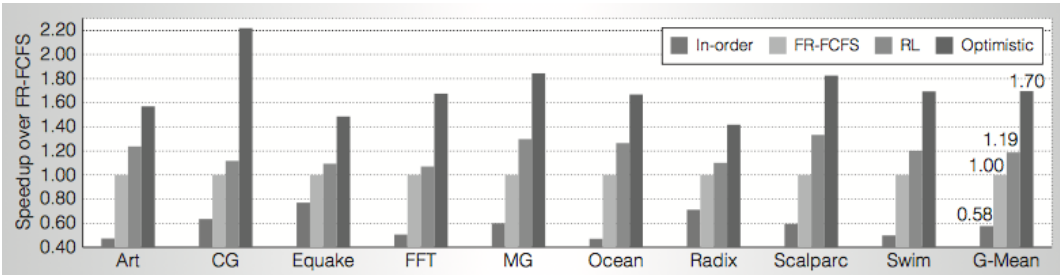


Figure 16.9: Performances of four controllers over a suite of 9 simulated benchmark applications. The controllers are: the simplest 'in-order' controller, FR-FCFS, the learning controller RL, and the unrealizable Optimistic controller which ignores all timing and resource constraints to provide a performance upper bound. Performance, normalized to that of FR-FCFS, is the inverse of execution time. At far right is the geometric mean of performances over the 9 benchmark applications for each controller. Controller RL comes closest to the ideal performance. ©2009 IEEE. Reprinted, with permission, from J. F. Martínez and E. İpek, Dynamic multicore resource management: A machine learning approach, Micro, IEEE, 29(5), p. 13.

Because the rationale for on-chip implementation of the learning algorithm was to allow the scheduling policy to adapt on-line to changing workloads, İpek et al. analyzed the impact of on-line learning compared to a previously-learned fixed policy. They trained their controller with data from all nine benchmark applications and then held the resulting action values fixed throughout the simulated execution of the applications. They found that the average performance of the controller that learned on-line was 8% better than that of the controller using the fixed policy, leading them to conclude that

on-line learning is an important feature of their approach.

This learning memory controller was never committed to physical hardware because of the large cost of fabrication. Nevertheless, İpek et al. could convincingly argue on the basis of their simulation results that a memory controller that learns on-line via reinforcement learning has the potential to improve performance to levels that would otherwise require more complex and more expensive memory systems, while removing from human designers some of the burden required to manually design efficient scheduling policies. Mukundan and Martínez (2012) took this project forward by investigating learning controllers with additional actions, other performance criteria, and more complex reward functions derived using genetic algorithms. They considered additional performance criteria related to energy efficiency. The results of these studies surpassed the earlier results described above and significantly surpassed the 2012 state-of-the-art for all of the performance criteria they considered. The approach is especially promising for developing sophisticated power-aware DRAM interfaces.

16.6 Human-level Video Game Play

One of the greatest challenges in applying reinforcement learning to real-world problems is deciding how to represent and store value functions and/or policies. Unless the state set is finite and small enough to allow exhaustive representation by a lookup table—as in many of our illustrative examples—one must use a parameterized function approximation scheme. Whether linear or non-linear, function approximation relies on features that have to be readily accessible to the learning system and able to convey the information necessary for skilled performance. Most successful applications of reinforcement learning owe much to sets of features carefully handcrafted based on human knowledge and intuition about the specific problem to be tackled.

A team of researchers at Google DeepMind developed an impressive demonstration that a deep multi-layer artificial neural network (ANN) can automate the feature design process (Mnih et al., 2015). Multi-layer ANNs have been used for function approximation in reinforcement learning ever since the 1986 popularization of the backpropagation algorithm as a method for learning internal representations (Rumelhart, Hinton, and Williams, 1986; see Section 9.6). Striking results have been obtained by coupling reinforcement learning with backpropagation. The results obtained by Tesauro and colleages with TD-Gammon and Watson discussed above are notable examples. These and other applications benefited from the ability of multi-layer ANNs to learn task-relevant features. However, in all the examples of which we are aware, the most impressive demonstrations required the network's input to be represented in terms of specialized features handcrafted for the given problem. This is vividly apparent in the TD-Gammon results. TD-Gammon 0.0, whose network input was essentially a "raw" representation of he backgammon board, meaning that it involved very little knowledge of backgammon, learned to play approximately as well as the best previous backgammon computer programs. Adding specialized backgammon features produced TD-Gammon 1.0 which was substantially better than all previous backgammon programs and competed well against human experts.

Mnih et al. developed a reinforcement learning agent called *deep Q-network* (DQN) that combined Q-learning with a *deep convolutional* ANN, a many-layered, or deep, ANN specialized for processing spatial arrays of data such as images. We describe deep convolutional ANNs in Section 9.6. By the time of Mnih et al.'s work with DQN, deep ANNs, including deep convolutional ANNs, had produced impressive results in many applications, but they had not been widely used in reinforcement learning.

Mnih et al. used DQN to show how a single reinforcement learning agent can achieve high levels of performance in many different problems without relying on different problem-specific feature sets. To demonstrate this, they let DQN learn to play 49 different Atari 2600 video games by interacting with a game emulator. For learning each game, DQN used the same raw input, the same network architecture, and the same parameter values (e.g., step-size, discount rate, exploration parameters, and many more specific to the implementation). DQN achieved levels of play at or beyond human level on

a large fraction of these games. Although the games were alike in being played by watching streams of video images, they varied widely in other respects. Their actions had different effects, they had different state-transition dynamics, and they needed different policies for earning high scores. The deep convolutional ANN learned to transform the raw input common to all the games into features specialized for representing the action values required for playing at the high level DQN achieved for most of the games.

The Atari 2600 is a home video game console that was sold in various versions by Atari Inc. from 1977 to 1992. It introduced or popularized many arcade video games that are now considered classics, such as Pong, Breakout, Space Invaders, and Asteroids. Although much simpler than modern video games, Atari 2600 games are still entertaining and challenging for human players, and they have been attractive as testbeds for developing and evaluating reinforcement learning methods (Diuk, Cohen, Littman, 2008; Naddaf, 2010; Cobo, Zang, Isbell, and Thomaz, 2011; Bellemare, Veness, and Bowling, 2012). Bellemare, Naddaf, Veness, and Bowling (2012) developed the publicly available Arcade Learning Environment (ALE) to encourage and simplify using Atari 2600 games to study learning and planning algorithms.

These previous studies and the availability of ALE made the Atari 2600 game collection a good choice for Mnih et al.'s demonstration, which was also influenced by the impressive human-level performance that TD-Gammon was able to achieve in backgammon. DQN is similar to TD-Gammon in using a multi-layer ANN as the function approximation method for a semi-gradient form of a TD algorithm, with the gradients computed by the backpropagation algorithm. However, instead of using TD(λ) as TD-Gammon did, DQN used the semi-gradient form of Q-learning. TD-Gammon estimated the values of afterstates, which were easily obtained from the rules for making backgammon moves. To use the same algorithm for the Atari games would have required generating the next states for each possible action (which would not have been afterstates in that case). This could have been done by using the game emulator to run single-step simulations for all the possible actions (which ALE makes possible). Or a model of each game's state-transition function could have been learned and used to predict next states (Oh, Guo, Lee, Lewis, and Singh, 2015). While these methods might have produced results comparable to DQN's, they would have been more complicated to implement and would have significantly increased the time needed for learning. Another motivation for using Q-learning was that DQN used the *experience replay* method, described below, which requires an off-policy algorithm. Being model-free and off-policy made Q-learning a natural choice.

Before describing the details of DQN and how the experiments were conducted, we look at the skill levels DQN was able to achieve. Mnih et al. compared the scores of DQN with the scores of the best performing learning system in the literature at the time, the scores of a professional human games tester, and the scores of an agent that selected actions at random. The best system from the literature used linear function approximation with features hand designed using some knowledge about Atari 2600 games (Bellemare, Naddaf, Veness, and Bowling, 2012). DQN learned on each game by interacting with the game emulator for 50 million frames, which corresponds to about 38 days of experience with the game. At the start of learning on each game, the weights of DQN's network were reset to random values. To evaluate DQN's skill level after learning, its score was averaged over 30 sessions on each game, each lasting up to 5 minutes and beginning with a random initial game state. The professional human tester played using the same emulator (with the sound turned off to remove any possible advantage over DQN which did not process audio). After 2 hours of practice, the human played about 20 episodes of each game for up to 5 minutes each and was not allowed to take any break during this time. DQN learned to play better than the best previous reinforcement learning systems on all but 6 of the games, and played better than the human player on 22 of the games. By considering any performance that scored at or above 75% of the human score to be comparable to, or better than, human-level play, Mnih et al. concluded that the levels of play DQN learned reached or exceeded human level on 29 of the 46 games. See Mnih et al. (2015) for a more detailed account of these results.

For an artificial learning system to achieve these levels of play would be impressive enough, but what

makes these results remarkable—and what many at the time considered to be breakthrough results for artificial intelligence—is that the very same learning system achieved these levels of play on widely varying games without relying on any game-specific modifications.

A human playing any of these 46 Atari games sees 210×160 pixel image frames with 128 colors at 60Hz. In principle, exactly these images could have formed the raw input to DQN, but to reduce memory and processing requirements, Mnih et al. preprocessed each frame to produce an 84×84 array of luminance values. Since the full states of many of the Atari games are not completely observable from the image frames, Mnih et al. "stacked" the four most recent frames so that the inputs to the network had dimension $84 \times 84 \times 4$. This did not eliminate partial observability for all of the games, but it was helpful in making many of them more Markovian.

An essential point here is that these preprocessing steps were exactly the same for all 46 games. No game-specific prior knowledge was involved beyond the general understanding that it should still be possible to learn good policies with this reduced dimension and that stacking adjacent frames should help with the partial observability of some of the games. Since no game-specific prior knowledge beyond this minimal amount was used in preprocessing the image frames, we can think of the $84 \times 84 \times 4$ input vectors as being "raw" input to DQN.

The basic architecture of DQN is similar to the deep convolutional ANN illustrated in Figure 9.15 (though unlike that network, subsampling in DQN is treated as part of each convolutional layer, with feature maps consisting of units having only a selection of the possible receptive fields). DQN has three hidden convolutional layers, followed by one fully connected hidden layer, followed by the output layer. The three successive hidden convolutional layers of DQN produce 32 20×20 feature maps, 64 9×9 feature maps, and 64 7×7 feature maps. The activation function of the units of each feature map is a rectifier nonlinearity ($\max(0, x)$). The 3,136 ($64 \times 7 \times 7$) units in this third convolutional layer all connect to each of 512 units in the fully connected hidden layer, which then each connect to all 18 units in the output layer, one for each possible action in an Atari game.

The activation levels of DQN's output units were the estimated optimal action values (optimal Q-values) of the corresponding state–action pairs, for the state represented by the network's input. The assignment of output units to a game's actions varied from game to game, and since the number of valid actions varied between 4 and 18 for the games, not all output units had functional roles in all of the games. It helps to think of the network as if it were 18 separate networks, one for estimating the optimal action value of each possible action. In reality, these networks shared their initial layers, but the output units learned to use the features extracted by these layers in different ways.

DQN's reward signal indicated how a games's score changed from one time step to the next: $+1$ whenever it increased, -1 whenever it decreased, and 0 otherwise. This standardized the reward signal across the games and made a single step-size parameter work well for all the games despite their varying ranges of scores. DQN used an ϵ-greedy policy, with ϵ decreasing linearly over the first million frames and remaining at a low value for the rest of the learning session. The values of the various other parameters, such as the learning step-size, discount rate, and others specific to the implementation, were selected by performing informal searches to see which values worked best for a small selection of the games. These values were then held fixed for all of the games.

After DQN selected an action, the action was executed by the game emulator, which returned a reward and the next video frame. The frame was preprocessed and added to the four-frame stack that became the next input to the network. Skipping for the moment the changes to the basic Q-learning procedure made by Mnih et al., DQN used the following semi-gradient form of Q-learning to update the network's weights:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \Big[R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t) \Big] \nabla_{\mathbf{w}_t} \hat{q}(S_t, A_t, \mathbf{w}_t), \tag{16.3}$$

where $\mathbf{w}_t$ is the vector of the network's weights, A_t is the action selected at time step t, and S_t and S_{t+1} are respectively the preprocessed image stacks input to the network at time steps t and $t+1$.

The gradient in (16.3) was computed by backpropagation. Imagining again that there was a separate network for each action, for the update at time step t, backpropagation was applied only to the network corresponding to A_t. Mnih et al. took advantage of techniques shown to improve the basic backpropagation algorithm when applied to large networks. They used a *mini-batch method* that updated weights only after accumulating gradient information over a small batch of images (here after 32 images). This yielded smoother sample gradients compared to the usual procedure that updates weights after each action. They also used a gradient-ascent algorithm called RMSProp (Tieleman and Hinton, 2012) that accelerates learning by adjusting the step-size parameter for each weight based on a running average of the magnitudes of recent gradients for that weight.

Mnih et al. modified the basic Q-learning procedure in three ways. First, they used a method called *experience replay* first studied by Lin (1992). This method stores the agent's experience at each time step in a replay memory that is accessed to perform the weight updates. It worked like this in DQN. After the game emulator executed action A_t in a state represented by the image stack S_t, and returned reward R_{t+1} and image stack S_{t+1}, it added the tuple $(S_t, A_t, R_{t+1}, S_{t+1})$ to the replay memory. This memory accumulated experiences over many plays of the same game. At each time step multiple Q-learning updates—a mini-batch—were performed based on experiences sampled uniformly at random from the replay memory. Instead of S_{t+1} becoming the new S_t for the next update as it would in the usual form of Q-learning, a new unconnected experience was drawn from the replay memory to supply data for the next update. Since Q-learning is an off-policy algorithm, it does not need to be applied along connected trajectories.

Q-learning with experience replay provided several advantages over the usual form of Q-learning. The ability to use each stored experience for many updates allowed DQN to learn more efficiently from its experiences. Experience replay reduced the variance of the updates because successive updates were not correlated with one another as they would be with standard Q-learning. And by removing the dependence of successive experiences on the current weights, experience replay eliminated one source of instability.

Mnih et al. modified standard Q-learning in a second way to improve its stability. As in other methods that bootstrap, the target for a Q-learning update depends on the current action-value function estimate. When a parameterized function approximation method is used to represent action values, the target is a function of the same parameters that are being updated. For example, the target in the update given by (16.3) is $\gamma \max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t)$. Its dependence on $\mathbf{w}_t$ complicates the process compared to the simpler supervised-learning situation in which the targets do not depend on the parameters being updated. As discussed in Chapter 11 this can lead to oscillations and/or divergence.

To address this problem Mnih et al. used a technique that brought Q-learning closer to the simpler supervised-learning case while still allowing it to bootstrap. Whenever a certain number, C, of updates had been done to the weights $\mathbf{w}$ of the action-value network, they inserted the network's current weights into another network and held these duplicate weights fixed for the next C updates of $\mathbf{w}$. The outputs of this duplicate network over the next C updates of $\mathbf{w}$ were used as the Q-learning targets. Letting $\tilde{q}$ denote the output of this duplicate network, then instead of (16.3) the update rule was:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\Big[R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)\Big]\nabla_{\mathbf{w}_t}\hat{q}(S_t, A_t, \mathbf{w}_t).$$

A final modification of standard Q-learning was also found to improve stability. They clipped the error term $R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)$ so that it remained in the interval $[-1, 1]$.

Mnih et al. conducted a large number of learning runs on 5 of the games to gain insight into the effect that various of DQN's design features had on its performance. They ran DQN with the four combinations of experience replay and the duplicate target network being included or not included. Although the results varied from game to game, each of these features alone significantly improved performance, and very dramatically improved performance when used together. Mnih et al. also studied the role played by the deep convolutional ANN in DQN's learning ability by comparing the deep

convolutional version of DQN with a version having a network of just one linear layer, both receiving the same stacked preprocessed video frames. Here, the improvement of the deep convolutional version over the linear version was particularly striking across all 5 of the test games.

Creating artificial agents that excel over a diverse collection of challenging tasks has been an enduring goal of artificial intelligence. The promise of machine learning as a means for achieving this has been frustrated by the need to craft problem-specific representations. DeepMind's DQN stands as a major step forward by demonstrating that a single agent can learn problem-specific features enabling it to acquire human-competitive skills over a range of tasks. But as Mnih et al. point out, DQN is not a complete solution to the problem of task-independent learning. Although the skills needed to excel on the Atari games were markedly diverse, all the games were played by observing video images, which made a deep convolutional ANN a natural choice for this collection of tasks. In addition, DQN's performance on some of the Atari 2600 games fell considerably short of human skill levels on these games. The games most difficult for DQN—especially Montezuma's Revenge on which DQN learned to perform about as well as the random player—require deep planning beyond what DQN was designed to do. Further, learning control skills through extensive practice, like DQN learned how to play the Atari games, is just one of the types of learning humans routinely accomplish. Despite these limitations, DQN advanced the state-of-the-art in machine learning by impressively demonstrating the promise of combining reinforcement learning with modern methods of deep learning.

16.7 Mastering the Game of Go

The ancient Chinese game of Go has challenged artificial intelligence researchers for many decades. Methods that achieve human-level skill, or even superhuman-level skill, in other games have not been successful in producing strong Go programs. Thanks to a very active community of Go programmers and international competitions, the level of Go program play has improved significantly over the years. Until recently, however, no Go program had been able to play anywhere near the level of a human Go master. Here we describe a program called AlphaGo developed by a Google DeepMind team (Silver et al., 2016) that broke this barrier by combining deep artificial neural networks (deep ANNs, Section 9.6), supervised learning, Monte Carlo tree search (MCTS, Section 8.11), and reinforcement learning. By the time of Silver et al.'s 2016 publication, AlphaGo had been shown to be decisively stronger than other current Go programs, and it had defeated the human European Go champion 5 games to 0. These were the first victories of a Go program over a human professional Go player without handicap in full Go games. Shortly thereafter, AlphaGo went on to stunning victories over an 18-time world champion Go player, winning 4 out of a 5 games in a challenge match, making worldwide headline news. Artificial intelligence researchers thought that it would be many more years, perhaps decades, for a program to reach this level of play.

In many ways, AlphaGo is a descendant of Tesauro's TD-Gammon (Section 16.1), itself a descendant of Samuel's checkers player (Section 16.2). AlphaGo, like these earlier programs, used reinforcement learning with function approximation over many simulated games. AlphaGo also built upon the progress made by Google DeepMind on playing Atari games with the program DQN (Section 16.6) by approximating value functions with deep convolutional ANNs. By using a novel variant of MCTS, AlphaGo extended the technology responsible for the impressive gains of the most successful preceding Go programs. But AlphaGo was not a simple combination of these technologies: it combined them in a highly-engineered way that was perhaps critical for AlphaGo's impressive performance. Another element of AlphaGo was its distributed architecture: many of its computations were executed in parallel on many processors so that it could select moves quickly enough to meet the time constraints of live play. Although this contributed to AlphaGo's success, most of its playing skill was due to algorithmic innovations, and here we neglect AlphaGo's distributed architecture.

Go is a game between two players who alternately place black and white 'stones' on unoccupied

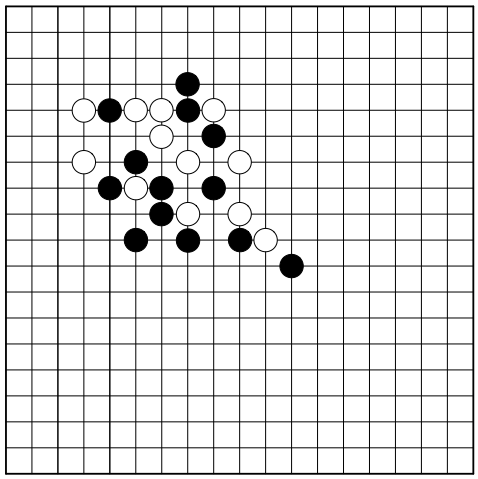


Figure 16.10: A Go board configuration.

intersections, or 'points,' on a board with a grid of 19 horizontal and 19 vertical lines (Figure 16.10). The game's goal is to capture an area of the board larger than that captured by the opponent. Stones are captured according to simple rules. A player's stones are captured if they are completely surrounded by the other player's stones, meaning that there is no horizontally or vertically adjacent point that is unoccupied. For example, Figure 16.11 shows on the left three white stones with an unoccupied adjacent point (labeled X). If player black places a stone on X, the three white stones are captured and taken off the board (Figure 16.11 middle). However, if player white were to place a stone on point X first, than the possibility of this capture would be blocked (Figure 16.11 right). Other rules are needed to prevent infinite capturing/re-capturing loops. The game ends when neither player wishes to place another stone. These rules are simple, but they produce a very complex game that has had wide appeal for thousands of years.

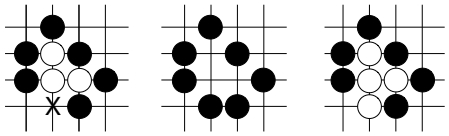


Figure 16.11: Go capturing rule. Left: the three white stones are not surrounded because point X is unoccupied. Middle: if black places a stone on X, the three white stones are captured and removed from the board. Right: if white places a stone on point X first, the capture is blocked.

Methods that produce strong play for other games, such as chess, have not worked as well for Go. The search space for Go is significantly larger than that of chess because Go has a larger number of legal moves per position than chess (≈ 250 versus ≈ 35) and Go games tend to involve more moves than chess games (≈ 150 versus ≈ 80). But the size of the search space is not the major factor that makes Go so difficult. Exhaustive search is infeasible for both chess and Go, and Go on smaller boards, e.g., 9×9, has proven to be exceedingly difficult as well. Experts agree that the major stumbling block to creating stronger-than-amateur Go programs is the difficulty of defining an adequate position evaluation function. A good evaluation function allows search to be truncated at a feasible depth by providing relatively easy-to-compute predictions of what deeper search would likely yield. According to Müller (2002): "No simple yet reasonable evaluation function will ever be found for Go." A major step forward was the introduction of MCTS to Go programs, which does not attempt to store an evaluation function, instead evaluating moves at decision time by running many Monte Carlo simulations of entire

games. The strongest programs at the time of AlphaGo's development all used MCTS, but master-level skill remained elusive.

The main innovation of AlphaGo is its use of a variant of MCTS called "asynchronous policy and value MCTS," or APV-MCTS. APV-MCTS selects moves via basic MCTS as described in Section 8.11 and illustrated in Figure 8.11, but with some twists in terms of how policies and value functions are computed and ultimately how each move is chosen. In addition to the *tree policy* for traversing the search tree, and a *rollout policy*, p_π, used in the Monte Carlo simulations, there is a *supervised learning policy* (SL policy), p_σ, a *reinforcement learning policy* (RL policy), p_ρ, and a *state-value function*, v_θ. These policies and value function were all learned off-line before actual play. The SL policy is used in the expansion phase of APV-MCTS iterations, the phase in which a node in the search tree is selected as a promising node from which to explore further. In contrast to basic MCTS, which expands the selected node by choosing an unexplored action based on stored action-values, APV-MCTS chooses an action according to prior probabilities supplied by the SL-policy. Both the rollout policy and the SL policy were learned before play via supervised learning using large databases of expert human moves. The RL policy was learned via self-play reinforcement learning and was used to derive the state-value function v_θ. Below we explain how this was done.

In contrast to basic MCTS, which evaluates the expanded node solely on the basis of the return of the simulation initiated from it, APV-MCTS evaluates the node in two ways: by this return of the simulation, but also by the value function v_θ: if s is the node selected for expansion, its value becomes

$$V(s) = (1 - \eta)v_\theta(s) + \eta G, \tag{16.4}$$

where G is the return of the simulation and η controls the mixing of the values resulting from these two evaluation methods.

It is in the formation of the rollout policy, the SL policy, the RL policy, and the value function that deep ANNs came into play. These policies were learned by deep convolutional ANNs, respectively called the *rollout policy network*, the *SL policy network*, the *RL policy network*, and the *value network*. Figure 16.12 illustrates the steps taken to train these networks in what the DeepMind team called the "AlphaGo pipeline." They were all trained before any live game play took place and their weights remained fixed throughout AlphaGo's live play.

The first stage of this pre-play process was to train the SL policy network. This network was similar to DQN's deep convolutional network described in Section 16.6 for playing Atari games, except that it

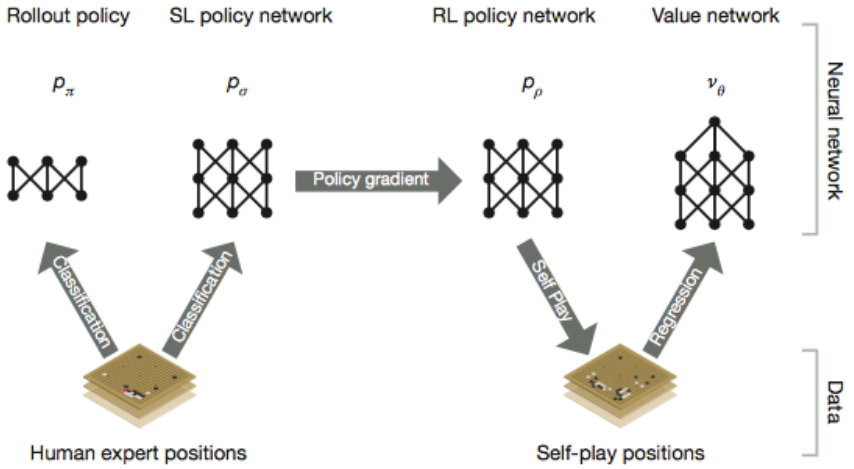


Figure 16.12: AlphaGo pipeline. Adapted with permission from Macmillan Publishers Ltd: *Nature*, vol. 529(7587), p. 485, copyright (2016).

had 13 layers with the final layer consisting of a softmax unit for each point on the 19×19 board. The network's input was a $19 \times 19 \times 48$ image stack in which each point on the Go board was represented by the values of 48 binary or integer-valued features. For example, for each point, one feature indicated if the point was occupied by one of AlphaGo's stones, one of its opponent's stones, or was unoccupied, thus providing the "raw" representation of the board configuration. Other features were based on the rules of Go, such as the number of adjacent points that were empty, the number of opponent stones that would be captured by placing a stone there, the number of turns since a stone was placed there, and other feature vectors that the design team considered to be important. The SL network was trained via supervised learning to predict moves contained in a data base of nearly 30 million expert moves. Its output, which was a probability distribution over all legal moves, defined the SL policy that supplied the action probabilities in the expansion step of MPV-MCTS. Training took approximately 3 weeks using a distributed implementation of stochastic gradient ascent on 50 processors. The SL network achieved 57% accuracy, compared to best accuracy achieved by other groups at the time of publication of 44.4%.

The goal for the rollout policy network was that it should select actions quickly while still being reasonably accurate. In principle, the SL policy could have served as the rollout policy, but the forward propagation through the SL policy network took too much time for this network to be used in rollout simulations, a great many of which had to be carried out for each move decision during live play. For this reason, the rollout policy network was less complex than the SL policy network, and its input features could be computed more quickly than the features used for the SL policy network. This network was trained by supervised learning on a corpus of 8 million human moves. The resulting rollout policy network allowed approximately 1,000 complete game simulations per second to be run on each of the processing threads that AlphaGo used.

Next, the value function, v_θ, needed by APV-MCTS had to be created. Ideally, v_θ would be the optimal state-value function. It might have been possible to approximate the optimal value function along the lines of TD-Gammon described above: self-play with nonlinear TD(λ) coupled to a deep convolutional ANN. But the DeepMind team took a different approach that held more promise for a game as complex as Go. They divided the process into two stages. In the first stage, they created the best policy they could by training a deep convolutional network by means of a policy-gradient reinforcement learning algorithm. This resulted in the RL policy network implementing the RL policy p_ρ.

The RL policy network had the same architecture as the SL policy network and was initialized with the final weights of the SL policy network that were learned via supervised learning. Reinforcement learning was then used to improve upon the SL policy. Learning was by means of simulated games between the network's current policy and opponents using policies randomly selected from policies produced by earlier iterations of the learning algorithm. Playing against a randomly selected collection of opponents prevented overfitting to the current policy. The reward signal was $+1$ if the current policy won, -1 if it lost, and zero otherwise. These games directly pitted two policies against one another without involving any search. By simulating many games in parallel on 50 processors, the DeepMind team trained the RL policy network on a million games in a single day. In testing the final RL policy p_ρ, they found that it won more than 80% of games played against the SL policy p_σ, and it won 85% of games played against a Go program using Monte Carlo search that simulated 100,000 games per move.

Finally, the value function v_θ was learned by the value network: a deep convolutional ANN whose structure was like that of the SL and RL policy networks except that its single output unit produced state values (the right-most network in Figure 16.12) instead of probability distributions over legal actions. The value network was trained by Monte Carlo policy evaluation on data obtained from a large number of simulated games in which each player used the RL policy p_ρ. The team found that the values produced by this network were more accurate than values produced by multiple simulations using the rollout policy p_π, and in fact compared well with values produced by simulations using the higher-performing RL policy p_ρ—even though they could be computed 15,000 times more quickly.

One may wonder why APV-MCTS used the SL policy p_σ instead of the better RL policy p_ρ to select actions in the expansion phase. These policies took the same amount of time to compute since they used the same network architecture. The team actually found that AlphaGo worked better against human opponents when APV-MCTS used as its SL policy p_σ instead of p_ρ. They conjectured that the reason for this was that the p_ρ was tuned to respond to optimal moves rather than to the broader set of moves characteristic of human play. Interestingly, the situation was reversed for the value function, v_θ, used by APV-MCTS. They found that when APV-MCTS used the value function derived from the RL policy, it performed better than if it used the value function derived from the SL policy.

Several methods worked together to produce AlphaGo's impressive playing skill. The DeepMind team evaluated different versions of AlphaGo in order to asses the contributions made by these various components. The parameter η in (16.4) controls the mixing of game state evaluations produced by the value network and by rollouts. With $\eta = 0$, AlphaGo used just the value network without rollouts, and with $\eta = 1$, evaluation relied just on rollouts. They found that AlphaGo using just the value network played better than the rollout-only AlphaGo, and in fact played better than the strongest of all other Go programs. The best play resulted from setting $\eta = 0.5$, indicating that combining the value network with rollouts was particularly important to AlphaGo's success. These evaluation methods complemented one another: the value network evaluated the high-performance policy p_ρ that was too slow to be used in live play, while rollouts using the weaker but much faster rollout policy p_π were able to add precision to the value network's evaluations for specific states that occurred during games.

Overall, AlphaGo's remarkable success helped fuel a new round of enthusiasm for the promise of artificial intelligence, specifically for systems combining reinforcement learning with deep ANNs, to address problems in many other challenging domains.

16.8 Personalized Web Services

Personalizing web services such as the delivery of news articles or advertisements is one approach to increasing users' satisfaction with a website or to increase the yield of a marketing campaign. A policy can recommend content considered to be the best for each particular user based on a profile of that user's interests and preferences inferred from their history of online activity. This is a natural domain for machine learning, and in particular, for reinforcement learning. A reinforcement learning system can improve a recommendation policy by making adjustments in response to user feedback. One way to obtain user feedback is by means of website satisfaction surveys, but for acquiring feedback in real time it is common to monitor user clicks as indicators of interest in a link.

A method long used in marketing called *A/B testing* is a simple type of reinforcement learning used to decide which of two versions, A or B, of a website users prefer. Because it is non-associative, like a two-armed bandit problem, this approach does not personalize content delivery. Adding context consisting of features describing individual users and the content to be delivered allows personalizing service. This has been formalized as a contextual bandit problem (or an associative reinforcement learning problem, Section 2.9) with the objective of maximizing the total number of user clicks. Li, Chu, Langford, and Schapire (2010) applied a contextual bandit algorithm to the problem of personalizing the Yahoo! Front Page Today webpage (one of the most visited pages on the internet at the time of their research) by selecting the news story to feature. Their objective was to maximize the *click-through rate* (CTR), which is the ratio of the total number of clicks all users make on a webpage to the total number of visits to the page. Their contextual bandit algorithm improved over a standard non-associative bandit algorithm by 12.5%.

Theocharous, Thomas, and Ghavamzadeh (2015) argued that better results are possible by formulating personalized recommendation as a Markov decision problem (MDP) with the objective of maximizing the total number of clicks users make over repeated visits to a website. Policies derived from the contextual bandit formulation are greedy in the sense that they do not take long-term effects

of actions into account. These policies effectively treat each visit to a website as if it were made by a new visitor uniformly sampled from the population of the website's visitors. By not using the fact that many users repeatedly visit the same websites, greedy policies do not take advantage of possibilities provided by long-term interactions with individual users.

As an example of how a marketing strategy might take advantage of long-term user interaction, Theocharous et al. contrasted a greedy policy with a longer-term policy for displaying ads for buying a product, say a car. The ad displayed by the greedy policy might offer a discount if the user buys the car immediately. A user either takes the offer or leaves the website, and if they ever return to the site, they would likely see the same offer. A longer-term policy, on the other hand, can transition the user "down a sales funnel" before presenting the final deal. It might start by describing the availability of favorable financing terms, then praise an excellent service department, and then, on the next visit, offer the final discount. This type of policy can result in more clicks by a user over repeated visits to the site, and if the policy is suitably designed, more eventual sales.

Working at Adobe Systems Incorporated, Theocharous et al. conducted experiments to see if policies designed to maximize clicks over the long term could in fact improve over short-term greedy policies. The Adobe Marketing Cloud, a set of tools that many companies use to to run digital marketing campaigns, provides infrastructure for automating user-targed advertising and fund-raising campaigns. Actually deploying novel policies using these tools entails significant risk because a new policy may end up performing poorly. For this reason, the research team needed to assess what a policy's performance would be if it were to be actually deployed, but to do so on the basis of data collected under the execution of other policies. A critical aspect of this research, then, was off-policy evaluation. Further, the team wanted to do this with high confidence to reduce the risk of deploying a new policy. Although high confidence off-policy evaluation was a central component of this research (see also Thomas, 2015; Thomas, Theocharous, and Ghavamzadeh, 2015), here we focus only on the algorithms and their results.

Theocharous et al. compared the results of two algorithms for learning ad recommendation policies. The first algorithm, which they called *greedy optimization*, had the goal of maximizing only the probability of immediate clicks. As in the standard contextual bandit formulation, this algorithm did not take the long-term effects of recommendations into account. The other algorithm, a reinforcement learning algorithm based on an MDP formulation, aimed at improving the number of clicks users made over multiple visits to a website. They called this latter algorithm *life-time value* (LTV) optimization. Both algorithms faced challenging problems because the reward signal in this domain is very sparse since users usually do not click on ads, and user clicking is very random so that returns have high variance.

Data sets from the banking industry were used for training and testing these algorithms. The data sets consisted of many complete trajectories of customer interaction with a bank's website that showed each customer one out of a collection of possible offers. If a customer clicked, the reward was 1, and otherwise it was 0. One data set contained approximately 200,000 interactions from a month of a bank's campaign that randomly offered one of 7 offers. The other data set from another bank's campaign contained 4,000,000 interactions involving 12 possible offers. All interactions included customer features such as the time since the customer's last visit to the website, the number of their visits so far, the last time the customer clicked, geographic location, one of a collection of interests, and features giving demographic information.

Greedy optimization was based on a mapping estimating the probability of a click as a function of user features. The mapping was learned via supervised learning from one of the data sets by means of a random forest (RF) algorithm (Breiman, 2001). RF algorithms have been widely used for large-scale applications in industry because they are effective predictive tools that tend not to overfit and are relatively insensitive to outliers and noise. Theocharous et al. then used the mapping to define an ϵ-greedy policy that selected with probability 1-ϵ the offer predicted by the RF algorithm to have the highest probability of producing a click, and otherwise selected from the other offers uniformly at random.

LTV optimization used a batch-mode reinforcement learning algorithm called *fitted Q iteration* (FQI). It is a variant of *fitted value iteration* (Gordon, 1999) adapted to Q-learning. Batch mode means that the entire data set for learning is available from the start, as opposed to the on-line mode of the algorithms we focus on in this book in which data are acquired sequentially while the learning algorithm executes. Batch-mode reinforcement learning algorithms are sometimes necessary when on-line learning is not practical, and they can use any batch-mode supervised learning regression algorithm, including algorithms known to scale well to high-dimensional spaces. The convergence of FQI depends on properties of the function approximation algorithm (Gordon, 1999). For their application to LTV optimization, Theocharous et al. used the same RF algorithm they used for the greedy optimization approach. Since in this case FQI convergence is not monotonic, Theocharous et al. kept track of the best FQI policy by off-policy evaluation using a validation training set. The final policy for testing the LTV approach was the ϵ-greedy policy based on the best policy produced by FQI with the initial action-value function set to the mapping produced by the RF for the greedy optimization approach.

To measure the performance of the policies produced by the greedy and LTV approaches, Theocharous et al. used the CTR metric and a metric they called the LTV metric. These metrics are similar, except that the LTV metric critically distinguishes between individual website visitors:

$$\text{CTR} = \frac{\text{Total \# of Clicks}}{\text{Total \# of Visits}},$$

$$\text{LTV} = \frac{\text{Total \# of Clicks}}{\text{Total \# of Visitors}}.$$

Figure 16.13 illustrates how these metrics differ. Each circle represents a user visit to the site; black circles are visits at which the user clicks. Each row represents visits by a particular user. By not distinguishing between visitors, the CTR for these sequences is 0.35, whereas the LTV is 1.5. Because LTV is larger than CTR to the extent that individual users revisit the site, it is an indicator of how successful a policy is in encouraging users to engage in extended interactions with the site.

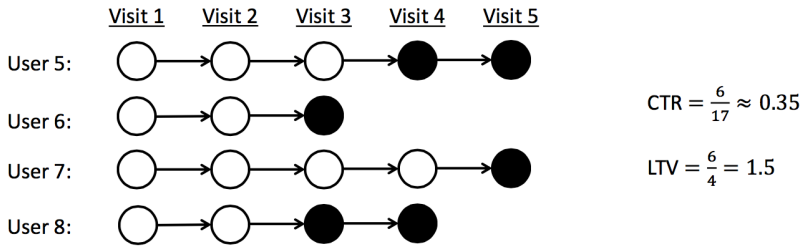


Figure 16.13: Click through rate (CTR) versus life-time value (LTV). Each circle represents a user visit; black circles are visits at which the user clicks. Adapted from Theocharous et al. (2015) permission pending.

Testing the policies produced by the greedy and LTV approaches was done using a high confidence off-policy evaluation method on a test data set consisting of real-world interactions with a bank website served by a random policy. As expected, results showed that greedy optimization performed best as measured by the CTR metric, while LTV optimization performed best as measured by the LTV metric. Furthermore—although we have omitted its details—the high confidence off-policy evaluation method provided probabilistic guarantees that the LTV optimization method would, with high probability, produce policies that improve upon policies currently deployed. Assured by these probabilistic guarantees, Adobe announced in 2016 that the new LTV algorithm would be a standard component of the Adobe Marketing Cloud so that a retailer could issue a sequence of offers following a policy likely to yield higher return than a policy that is insensitive to long-term results.

16.9 Thermal Soaring

Birds and gliders take advantage of upward air currents—thermals—to gain altitude in order maintain flight while expending little, or no, energy. Thermal soaring, as this behavior is called, is a complex skill requiring responding to subtle environmental cues to increase altitude by exploiting a rising column of air for as long as possible. Reddy, Celani, Sejnowski, and Vergassola (2016) used reinforcement learning to investigate thermal soaring policies that are effective in the strong atmospheric turbulence usually accompanying rising air currents. Their primary goal was to provide insight into the cues birds sense and how they use them to achieve their impressive thermal soaring performance, but the results also contribute to technology relevant to autonomous gliders. Reinforcement learning had previously been applied to the problem of navigating efficiently to the vicinity of a thermal updraft (Woodbury, Dunn, and Valasek, 2014) but not to the more challenging problem of soaring within the turbulence of the updraft itself.

Reddy et al. modeled the soaring problem as an MDP. The agent interacted with a detailed model of a glider flying in turbulent air. They devoted significant effort toward making the model generate realistic thermal soaring conditions, including investigating several different approaches to atmospheric modeling. For the learning experiments, air flow in a three-dimensional box with one kilometer sides, one of which was at ground level, was modeled by a sophisticated physics-based set of partial differential equations involving air velocity, temperature, and pressure. Introducing small random perturbations into the numerical simulation caused the model to produce analogs of thermal updrafts and accompanying turbulence (Figure 16.14 Left) Glider flight was modeled by aerodynamic equations involving velocity, lift, drag, and other factors governing powerless flight of a fixed-wing aircraft. Maneuvering the glider involved changing its angle of attack (the angle between the glider's wing and the direction of air flow) and its bank angle (Figure 16.14 Right).

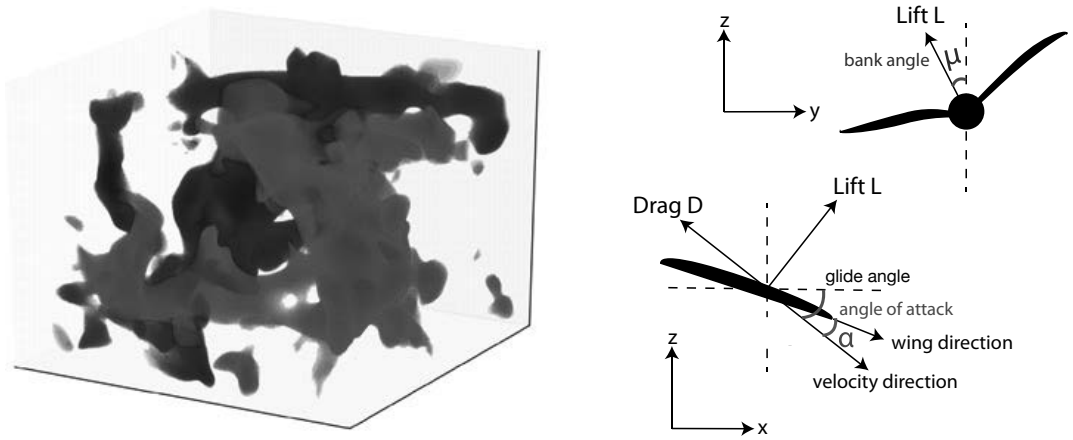


Figure 16.14: Thermal soaring model: Left: snapshot of the vertical velocity field of the simulated cube of air: in light (dark) grey is a region of large upward (downward) flow. Right: diagram of powerless flight showing bank angle μ and angle of attack α. Adapted with permission From PNAS vol. 113(22), p. E4879, 2016, Reddy, Celani, Sejnowski, and Vergassola, Learning to Soar in Turbulent Environments.

The interface between the agent and the environment required defining the agent's actions, the state information the agent receives from the environment, and the reward signal. By experimenting with various possibilities, Reddy et al. decided that three actions each for the angle of attack and the bank angle were enough for their purposes: increment or decrement the current bank angle and angle of attack by 5° and 2.5°, respectively, or leave them unchanged. This resulted in 3^2 possible actions. The bank angle was bounded to remain between $-15°$ and $+15°$.

Because a goal of their study was to try to determine what minimal set of sensory cues are necessary for effective soaring, both to shed light on the cues birds might use for soaring and to minimize the sensing complexity required for automated glider soaring, the authors tried various sets of signals as input to the reinforcement learning agent. They started by using state aggregation (Chapter 9) of a four-dimensional state space with dimensions giving local vertical wind speed, local vertical wind acceleration, torque depending on the difference between the vertical wind velocities at the left and right wing tips, and the local temperature. Each dimension was discretized into three bins: positive high, negative high, and small. Results, described below, showed that only two of these dimensions were critical for effective soaring behavior.

The overall objective of thermal soaring is to gain as much altitude as possible from each rising column of air. Reddy et al. tried a straightforward reward signal that rewarded the agent at the end of each episode based on the altitude gained over the episode, a large negative reward signal if the glider touched the ground, and zero otherwise. They found that learning was not successful with this reward signal for episodes of realistic duration and that eligibility traces did not help. By experimenting with various reward signals, they found that learning was best with a reward signal that at each time step linearly combined the vertical wind velocity and vertical wind acceleration observed on the previous time step.

Learning was by Sarsa with action selection using softmax applied to action values normalized to the interval $[0, 1]$. The temperature parameter was initialized to 2.0 and incrementally decreased to 0.2 during learning. The step-size and discount-rate parameters were fixed at 0.1 and 0.98 respectively. Each learning episode took place with the agent controlling simulated flight in an independently generated period of simulated turbulent air currents. Each episode lasted 2.5 minutes simulated with a 1 second time step. Learning effectively converged after a few hundred episodes. The left panel of Figure 16.15 shows a sample trajectory before learning where the agent selects actions randomly. Starting at the top of the volume shown, the glider's trajectory is in the direction indicated by the arrow and quickly loses altitude. Figure 16.15's right panel is a trajectory after learning. The glider starts at the same place (here appearing at the bottom of the volume) and gains altitude by spiraling within the rising column of air. Although Reddy at al. found that performance varied widely over different simulated periods of air flow, the number of times the glider touched the ground consistently decreased to nearly zero as learning progressed.

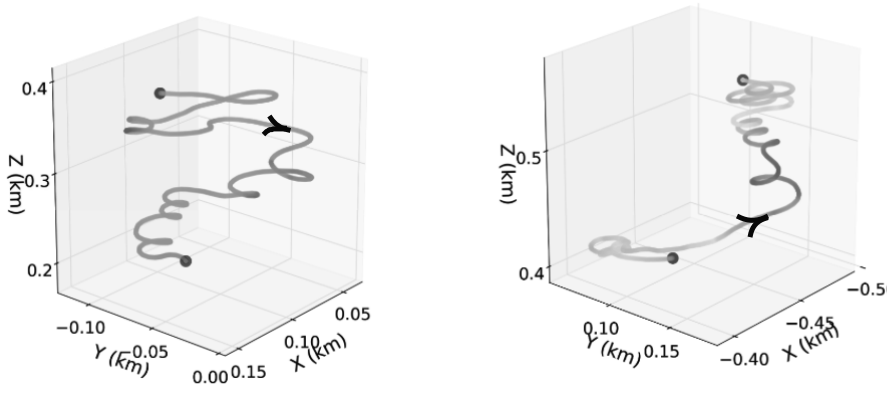


Figure 16.15: Sample thermal soaring trajectories, with arrows showing the direction of flight from the same starting point (note that the altitude scales are shifted). Left: before learning: the agent selects actions randomly and the glider descends. Right: after learning: the glider gains altitude by following a spiral trajectory. Adapted with permission from PNAS vol. 113(22), p. E4879, 2016, Reddy, Celani, Sejnowski, and Vergassola, Learning to Soar in Turbulent Environments.

After experimenting with different sets of features available to the learning agent, it turned out that the combination of just vertical wind acceleration and torques worked best. The authors conjectured that because these features give information about the gradient of vertical wind velocity in two different directions, they allow the controller to select between turning by changing the bank angle or continuing along the same course by leaving the bank angle alone. This allows the glider to stay within a rising column of air. Vertical wind velocity is indicative of the strength of the thermal but does not help in staying within the flow. They found that sensitivity to temperature was of little help. They also found that controlling the angle of attack is not helpful in staying within a particular thermal, being useful instead for traveling between thermals when covering large distances, as in cross-country gliding and bird migration.

Since soaring in different levels of turbulence requires different policies, training was done in conditions ranging from weak to strong turbulence. In strong turbulence the rapidly changing wind and glider velocities allowed less time for the controller to react. This reduced the amount of control possible compared to what was possible for maneuvering when fluctuations were weak. Reddy at al. examined the policies Sarsa learned under these different conditions. Common to policies learned in all regimes were these features: when sensing negative wind acceleration, bank sharply in the direction of the wing with the higher lift; when sensing large positive wind acceleration and no torque, do nothing. However, different levels of turbulence led to policy differences. Policies learned in strong turbulence were more conservative in that they preferred small bank angles, whereas in weak turbulence, the best action was to turn as much as possible by banking sharply. Systematic study of the bank angles preferred by the policies learned under the different conditions led the authors to suggest that by detecting when vertical wind acceleration crosses a certain threshold the controller can adjust its policy to cope with different turbulence regimes.

Reddy et al. also conducted experiments to investigate the effect of the discount-rate parameter γ on the performance of the learned policies. They found that the altitude gained in an episode increased as γ increased, reaching a maximum for $\gamma = .99$, suggesting that effective thermal soaring requires taking into account long-term effects of control decisions.

This computational study of thermal soaring illustrates how reinforcement learning can further progress toward different kinds of objectives. Learning policies having access to different sets of environmental cues and control actions contributes to both the engineering objective of designing autonomous gliders and the scientific objective of improving understanding of the soaring skills of birds. In both cases, hypotheses resulting from the learning experiments can be tested in the field by instrumenting real gliders and by comparing predictions with observed bird soaring behavior.

17.6 Reinforcement Learning and the Future of Artificial Intelligence

When we were writing the first edition of this book in the mid-1990s, artificial intelligence was making significant progress and was having an impact on society, though it was mostly still the *promise* of artificial intelligence that was inspiring developments. Machine learning was part of that outlook, but it had not yet become indispensable to artificial intelligence. By today that promise has transitioned to applications that are changing the lives of millions of people, and machine learning has come into its own as a key technology. As we write this second edition, some of the most remarkable developments in artificial intelligence have involved reinforcement learning, most notably "deep reinforcement learning"—reinforcement learning with function approximation by deep neural networks. We are at the beginning of a wave of real-world applications of artificial intelligence, many of which will include reinforcement learning, deep and otherwise, that will impact our lives in ways that are hard to predict.

But an abundance of successful real-world applications does not mean that true artificial intelligence has arrived. Despite great progress in many areas, the gulf between artificial intelligence and the intelligence of humans, and even of other animals, remains great. Superhuman performance can be achieved in some domains, even formidable domains like Go, but it remains a significant challenge to develop systems that are like us in being complete, interactive agents having general adaptability and problem-solving skills, emotional sophistication, creativity, and the ability to learn quickly from experience. With its focus on learning by interacting with dynamic environments, reinforcement learning, as it develops over the future, will be a critical component of agents with these abilities.

Reinforcement learning's connections to psychology and neuroscience (Chapters 14 and 15) underscore its relevance to another longstanding goal of artificial intelligence: shedding light on fundamental questions about the mind and how it emerges from the brain. Reinforcement learning theory is already contributing to our understanding of natural reward, motivation, and decision-making systems, understanding that can contribute to improving human abilities to learn, to remain motivated, and to make decisions. There is also good reason to believe that through its links to computational psychiatry, reinforcement learning theory will contribute to methods for treating mental disorders, including drug abuse and addiction.

Another contribution that reinforcement learning can make over the future is as an aid to human decision making. Policies derived by reinforcement learning in simulated environments can advise human decision makers in such areas as education, healthcare, transportation, energy, and public-sector resource allocation. Particularly relevant is the key feature of reinforcement learning that it takes long-term consequences of decisions into account. This is very clear in games like backgammon and Go, where some of the most impressive results of reinforcement learning have been demonstrated, but it is also a property of many high-stakes decisions that affect our lives and our planet. Reinforcement learning follows related methods for advising human decision making that have been developed in the past by decision analysts in many disciplines. With advanced function approximation methods and massive computational power, reinforcement learning methods have the potential to overcome some of the difficulties of scaling up traditional decision-support methods to larger and more complex problems.

The rapid pace of advances in artificial intelligence has led to warnings that artificial intelligence poses serious threats to our societies, even to humanity itself. The renowned scientist and artificial intelligence pioneer Herbert Simon anticipated the warnings we are hearing today in a presentation at the Earthware Symposium at CMU in 2000 (Simon, 2000). He spoke of the eternal conflict between the promise and perils of any new knowledge, reminding us of the Greek myths of Prometheus, the hero of modern science, who stole fire from the gods for the benefit of mankind, and Pandora, whose box could be opened by a small and innocent action to release untold perils on the world. While accepting that this conflict is inevitable, Simon urged us to recognize that as designers of our future and not simply as spectators, the decisions **we** make can tilt the scale in Prometheus' favor. This is certainly

true for reinforcement learning, which can benefit society but can also produce undesirable outcomes if it is carelessly deployed. Thus, the *safety* of artificial intelligence applications involving reinforcement learning is a topic that deserves careful attention.

A reinforcement learning agent can learn by interacting with either the real world or with a simulation of some piece of the real world, or by a mixture of these two sources of experience. Simulators provide safe environments in which an agent can explore and learn without risking real damage to itself or to its environment. In most current applications, policies are learned from simulated experience instead of direct interaction with the real world. In addition to avoiding undesirable real-world consequences, learning from simulated experience can make virtually unlimited data available for learning, generally at less cost than needed to obtain real experience, and since simulations typically run much faster than real time, learning can often occur more quickly than if it relied on real experience.

Nevertheless, the full potential of reinforcement learning requires reinforcement learning agents to be embedded into the flow of real-world experience, where they act, explore, and learn in *our* world, and not just in *their* worlds. After all, reinforcement learning algorithms—at least those upon which we focus in this book—are designed to learn online, and they emulate many aspects of how animals are able to survive in nonstationary and hostile environments. Embedding reinforcement learning agents in the real world can be transformative in realizing the promises of artificial intelligence to amplify and extend human abilities.

A major reason for wanting a reinforcement learning agent to act and learn in the real world is that it is often difficult, sometimes impossible, to simulate real-world experience with enough fidelity to make the resulting policies, whether derived by reinforcement learning or by other methods, work well—and safely—when directing real actions. This is especially true for environments whose dynamics depend on the behavior of humans, such as in education, healthcare, transportation, and public policy, domains that can surely benefit from improved decision making. However, it is for real-world embedded agents that warnings about potential dangers of artificial intelligence need to be heeded.

Reinforcement learning is a collection of optimization methods, so it inherits the pluses and minuses of all optimization methods. On the minus side is the problem we mentioned at several places above: how do you devise objective functions, or reward signals in the case of reinforcement learning, so that optimization produces the desired results while avoiding undesirable results? This is hardly a new problem with reinforcement learning; recognition of it has a long history in literature and engineering. The founder of cybernetics, Norbert Weiner, for one, warned of this more than half a century ago by relating the supernatural story of "The Monkey's Paw" (Weiner, 1964): wishes are granted but come with unacceptable cost. The problem has also been discussed at length in a modern context by Nick Bostrom (2014). Anyone having experience with reinforcement learning has likely seen their systems discover unexpected ways to obtain a lot of reward. Sometimes the unexpected behavior is good: it solves a problem in a nice new way. In other instances, what the agent learns violates considerations that the system designer may never have thought about. Careful design of reward signals is essential if an agent is to act in the real world with no opportunity for human vetting of its actions or means to easily interrupt its behavior.

Despite the possibility of unintended negative consequences, optimization has been used for hundreds of years by engineers, architects, and others whose designs have positively impacted the world. Many approaches have been developed to mitigate the risk of optimization, such as adding hard and soft constraints, restricting optimization to robust and risk-sensitive policies, and optimizing with multiple objective functions. Some of these have been adapted to reinforcement learning. We owe much that is good in our environment to the application of optimization methods. Still, the problem of ensuring that a reinforcement learning agent's goal is attuned to our own remains a challenge.

Another challenge if reinforcement learning agents are to act and learn in the real world is not just about what they might eventually learn, but about how they will behave while they are learning. How do you make sure that an agent gets enough experience to learn a high-performing policy, all the while

not harming its environment, other agents, or itself (or more realistically, while keeping the probability of harm acceptably low)? This problem is also not novel or unique to reinforcement learning. Risk management and mitigation for embedded reinforcement learning is similar to what control engineers have had to confront from the beginning of using automatic control in situations where a controller's behavior can have unacceptable, possibly catastrophic, consequences, as in the control of an aircraft or a delicate chemical process. Control applications rely on careful system modeling, model validation, and extensive testing, and there is a highly-developed body of theory aimed at ensuring convergence and stability of adaptive controllers designed for use when the dynamics of the system to be controlled are not fully known. Theoretical guarantees are never iron-clad because they depend on the validity of the assumptions underlying the mathematics, but without this theory, combined with risk-management and mitigation practices, automatic control—adaptive and otherwise—would not be as beneficial as it is today in improving the quality, efficiency, and cost-effectiveness of processes on which we have come to rely. Some of this theory has been adapted to reinforcement learning to help prevent unwanted behavior during, and after, learning, but many future applications of reinforcement learning are likely to be in domains that are less constrained than those to which control theory and practice readily apply. Developing methods to make it acceptably safe to fully embed reinforcement learning agents into physical environments is one of the most pressing areas for future research.

In closing, we return to Simon's call for us to recognize that we are designers of our future and not simply spectators. By decisions we make as individuals, and by the influence we can exert on how our societies are governed, we can work toward ensuring that the benefits made possible by a new technology outweigh the harm it can cause. There is ample opportunity to do this in the case of reinforcement learning, which can help improve the quality, fairness, and sustainability of life on our planet, but which can also release new perils. A threat already here is the displacement of jobs caused by applications of artificial intelligence. Still there are good reasons to believe that the benefits of artificial intelligence can outweigh the disruption it causes. As to safety, hazards possible with reinforcement learning are not completely different from those that have been managed successfully for related applications of optimization and control methods. As reinforcement learning moves out into the real world in future applications, developers have an obligation to follow best practices that have evolved for similar technologies, while at the same time extending them to make sure that Prometheus keeps the upper hand.

References

Abbeel, P., and Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*. ACM.

Abramson, B. (1990). Expected-outcome: A general model of static evaluation. *IEEE transactions on pattern analysis and machine intelligence 12*(2):182–193.

Adams, C. (1982). Variations in the sensitivity of instrumental responding to reinforcer devaluation. *The Quarterly Journal of Experimental Psychology*, 34(2):77–98.

Adams, C. D. and Dickinson, A. (1981). Instrumental responding following reinforcer devaluation. *The Quarterly Journal of Experimental Psychology 33*(2):109–121.

Adams, R. A., Huys, Q. J. M., and Roiser, J. P. (2015). Computational Psychiatry: towards a mathematically informed understanding of mental illness. *Journal of Neurology, Neurosurgery & Psychiatry*, doi:10.1136/jnnp-2015-310737.

Agrawal, R. (1995). Sample mean based index policies with $O(logn)$ regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27:1054–1078.

Agre, P. E. (1988). *The Dynamic Structure of Everyday Life*. Ph.D. thesis, Massachusetts Institute of Technology. AI-TR 1085, MIT Artificial Intelligence Laboratory.

Agre, P. E., Chapman, D. (1990). What are plans for? *Robotics and Autonomous Systems*, 6:17–34.

Aizerman, M. A., Braverman, E. Í., and Rozonoer, L. I. (1964). Probability problem of pattern recognition learning and potential functions method. *Avtomat. i Telemekh 25*(9):1307–1323.

Albus, J. S. (1971). A theory of cerebellar function. *Mathematical Biosciences*, 10:25–61.

Albus, J. S. (1981). *Brain, Behavior, and Robotics*. Byte Books, Peterborough, NH.

Aleksandrov, V. M., Sysoev, V. I., Shemeneva, V. V. (1968). Stochastic optimization of systems. *Izv. Akad. Nauk SSSR, Tekh. Kibernetika*, 14–19.

Amari, S. I. (1998). Natural gradient works efficiently in learning. *Neural Computation 10*(2), 251–276.

An, P.-C. E. (1991). *An Improved Multi-dimensional CMAC Neural network: Receptive Field Function and Placement* (Doctoral dissertation, PhD Thesis, Dept. Electrical and Computer Engineering, New Hampshire Univ., New Hampshire, USA).

An, P. C. E., Miller, W. T., Parks, P. C. (1991). Design improvements in associative memories for cerebellar model articulation controllers (CMAC). *Artificial Neural Networks*, pp. 1207–1210, Elsvier North-Holland.

Anderson, C. W. (1986). *Learning and Problem Solving with Multilayer Connectionist Systems*. Ph.D. thesis, University of Massachusetts, Amherst.

Anderson, C. W. (1987). Strategy learning with multilayer connectionist representations. *Proceedings of the Fourth International Workshop on Machine Learning*, pp. 103–114. Morgan Kaufmann, San Mateo, CA.

Anderson, C. W. (1989). Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine 9*(3):31–37.

Anderson, J. A., Silverstein, J. W., Ritz, S. A., Jones, R. S. (1977). Distinctive features, categorical perception, and probability learning: Some applications of a neural model. *Psychological Review*, 84:413–451.

Andreae, J. H. (1963). STELLA: A scheme for a learning machine. In *Proceedings of the 2nd IFAC Congress, Basle*, pp. 497–502. Butterworths, London.

Andreae, J. H. (1969a). A learning machine with monologue. *International Journal of Man–Machine Studies*, 1:1–20.

Andreae, J. H. (1969b). Learning machines—a unified view. In A. R. Meetham and R. A. Hudson (eds.), *Encyclopedia of Information, Linguistics, and Control*, pp. 261–270. Pergamon, Oxford.

Andreae, J. H. (1977). *Thinking with the Teachable Machine*. Academic Press, London.

Arthur, W. B. (1991). Designing economic agents that act like human agents: A behavioral approach to bounded rationality. *The American Economic Review 81*(2):353-359.

Atkeson, C. G. (1992). Memory-based approaches to approximating continuous functions. In *Sante Fe Institute Studies in the Sciences of Complexity*, Proceedings Vol. 12, pp. 521–521. Addison-Wesley Publishing Co.

Atkeson, C. G., Moore, A. W., and Schaal, S. (1997). Locally weighted learning. *Artificial Intelligence Review 11*:11–73.

Auer, P., Cesa-Bianchi, N., Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256.

Bae, J., Chhatbar, P., Francis, J. T., Sanchez, J. C., and Principe, J. C. (2011). Reinforcement learning via kernel temporal difference. In *Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pp. 5662–5665. IEEE.

Baird, L. C. (1995). Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 30–37. Morgan Kaufmann, San Francisco.

Baird, L. C., and Klopf, A. H. (1993). Reinforcement learning with high-dimensional, continuous actions. Wright Laboratory, Wright-Patterson Air Force Base, Tech. Rep. WL-TR-93-1147.

Baird, L., Moore, A. W. (1999). Gradient descent for general reinforcement learning. *Advances in Neural Information Processing Systems*, pp. 968–974.

Baldassarre, G. and Mirolli, M., editors (2013). *Intrinsically Motivated Learning in Natural and Artificial Systems*. Springer-Verlag, Berlin.

Balke, A., Pearl, J. (1994). Counterfactual probabilities: Computational methods, bounds and applications. In *Proceedings of the Tenth International Conference on Uncertainty in Artificial Intelligence* (pp. 46-54). Morgan Kaufmann.

Bao, G., Cassandras, C. G., Djaferis, T. E., Gandhi, A. D., Looze, D. P. (1994). Elevator dispatchers for down peak traffic. Technical report. ECE Department, University of Massachusetts, Amherst.

Baras, D. and Meir, R. (2007). Reinforcement learning, spike-time-dependent plasticity, and the BCM rule. *Neural Computation*, 19(8):2245–2279.

Barnard, E. (1993). Temporal-difference methods and Markov models. *IEEE Transactions on Systems, Man, and Cybernetics 23*:357–365.

Barnhill, R. E. (1977). Representation and approximation of surfaces. *Mathematical Software 3*:69–120.

Barreto, A. S., Precup, D., and Pineau, J. (2011). Reinforcement learning using kernel-based stochastic factorization. In *Advances in Neural Information Processing Systems*, pp. 720–728.

Bartlett, P. L. and Baxter, J. (1999). Hebbian synaptic modifications in spiking neurons that learn. Technical report, Research School of Information Sciences and Engineering, Australian National University.

Bartlett, P. L. and Baxter, J. (2000). A biologically plausible and locally optimal learning algorithm for spiking neurons. Rapport technique, Australian National University.

Barto, A. G. (1985). Learning by statistical cooperation of self-interested neuron-like computing elements. *Human Neurobiology*, 4:229–256.

Barto, A. G. (1986). Game-theoretic cooperativity in networks of self-interested units. In J. S. Denker (ed.), *Neural Networks for Computing*, pp. 41–46. American Institute of Physics, New York.

Barto, A. G. (1989). From chemotaxis to cooperativity: Abstract exercises in neuronal learning strategies. In Durbin, R., Maill, R., and Mitchison, G., editors, *The Computing Neuron*, pages 73–98. Addison-Wesley, Reading, MA.

Barto, A. G. (1990). Connectionist learning for control: An overview. In T. Miller, R. S. Sutton, and P. J. Werbos (eds.), *Neural Networks for Control*, pp. 5–58. MIT Press, Cambridge, MA.

Barto, A. G. (1991). Some learning tasks from a control perspective. In L. Nadel and D. L. Stein (eds.), *1990 Lectures in Complex Systems*, pp. 195–223. Addison-Wesley, Redwood City, CA.

Barto, A. G. (1992). Reinforcement learning and adaptive critic methods. In D. A. White and D. A. Sofge (eds.), *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pp. 469–491. Van Nostrand Reinhold, New York.

Barto, A. G. (1995a). Adaptive critics and the basal ganglia. In J. C. Houk, J. L. Davis, and D. G. Beiser (eds.), *Models of Information Processing in the Basal Ganglia*, pp. 215–232. MIT Press, Cambridge, MA.

Barto, A. G. (1995b). Reinforcement learning. In M. A. Arbib (ed.), *Handbook of Brain Theory and Neural Networks*, pp. 804–809. MIT Press, Cambridge, MA.

Barto, A. G. (2011). Adaptive real-time dynamic programming. In Sammut, C. and Webb, G. I. (Eds.) *Encyclopedia of machine learning*, pp. 19–22. Springer Science and Business Media.

Barto, A. G. (2013). Intrinsic motivation and reinforcement learning. In *Intrinsically Motivated Learning in Natural and Artificial Systems*, pp. 17–47. Springer Berlin Heidelberg

Barto, A. G., Anandan, P. (1985). Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15:360–375.

Barto, A. G., Anderson, C. W. (1985). Structural learning in connectionist systems. In *Program of the Seventh Annual Conference of the Cognitive Science Society*, pp. 43–54.

Barto, A. G., Anderson, C. W., Sutton, R. S. (1982). Synthesis of nonlinear control surfaces by a layered associative search network. *Biological Cybernetics*, 43:175–185.

Barto, A. G., Bradtke, S. J., Singh, S. P. (1991). Real-time learning and control using asynchronous dynamic programming. Technical Report 91-57. Department of Computer and Information Science, University of Massachusetts, Amherst.

Barto, A. G., Bradtke, S. J., Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138.

Barto, A. G., Duff, M. (1994). Monte Carlo matrix inversion and reinforcement learning. In J. D. Cohen, G. Tesauro, and J. Alspector (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 1993 Conference*, pp. 687–694. Morgan Kaufmann, San Francisco.

Barto, A. G., Jordan, M. I. (1987). Gradient following without back-propagation in layered networks. In M. Caudill and C. Butler (eds.), *Proceedings of the IEEE First Annual Conference on Neural Networks*, pp. II629–II636. SOS Printing, San Diego, CA.

Barto, A. G., and Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems 13*(4):341–379.

Barto, A. G., Singh, S., and Chentanez, N. (2004). Intrinsically motivated learning of hierarchical collections of skills. In *International Conference on Developmental Learning (ICDL)*, LaJolla, CA.

Barto, A. G., Sutton, R. S. (1981a). Goal seeking components for adaptive intelligence: An initial assessment. Technical Report AFWAL-TR-81-1070. Air Force Wright Aeronautical Laboratories/Avionics Laboratory, Wright-Patterson AFB, OH.

Barto, A. G., Sutton, R. S. (1981b). Landmark learning: An illustration of associative search. *Biological Cybernetics*, 42:1–8.

Barto, A. G., Sutton, R. S. (1982). Simulation of anticipatory responses in classical conditioning by a neuron-like adaptive element. *Behavioural Brain Research*, 4:221–235.

Barto, A. G., Sutton, R. S., Anderson, C. W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846. Reprinted in J. A. Anderson and E. Rosenfeld (eds.),

Neurocomputing: Foundations of Research, pp. 535–549. MIT Press, Cambridge, MA, 1988.

Barto, A. G., Sutton, R. S., Brouwer, P. S. (1981). Associative search network: A reinforcement learning associative memory. *Biological Cybernetics*, 40:201–211.

Barto, A. G., Sutton, R. S., and Watkins, C. J. C. H. (1990). Learning and sequential decision making. In M. Gabriel and J. Moore (eds.), *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pp. 539–602. MIT Press, Cambridge, MA.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2012a). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

Bellemare, M. G., Veness, J., and Bowling, M. (2012b). Investigating contingency awareness using Atari 2600 games. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2012)*, pages 864–871, Palo Alto, CA. The AAAI Press.

Bellman, R. E. (1956). A problem in the sequential design of experiments. *Sankhya*, 16:221–229.

Bellman, R. E. (1957a). *Dynamic Programming*. Princeton University Press, Princeton.

Bellman, R. E. (1957b). A Markov decision process. *Journal of Mathematical Mechanics*, 6:679–684.

Bellman, R. E., Dreyfus, S. E. (1959). Functional approximations and dynamic programming. *Mathematical Tables and Other Aids to Computation*, 13:247–251.

Bellman, R. E., Kalaba, R., Kotkin, B. (1973). Polynomial approximation—A new computational technique in dynamic programming: Allocation processes. *Mathematical Computation*, 17:155–161.

Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–27.

Bengio, Y., Courville, A. C., and Vincent, P. (2012). Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR 1*, arXiv 1206.5538.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*(9):509–517.

Berg, H. C. (1975). Chemotaxis in bacteria. *Annual review of biophysics and bioengineering*, 4(1):119–136.

Bernoulli, D. (1954). Exposition of a new theory on the measurement of risk. *Econometrica*, 22(1):23–36. English translation of the 1738 paper.

Berns, G. S., McClure, S. M., Pagnoni, G., and Montague, P. R. (2001). Predictability modulates human brain response to reward. *The journal of neuroscience*, 21(8):2793–2798.

Berridge, K. C. and Kringelbach, M. L. (2008). Affective neuroscience of pleasure: reward in humans and animals. *Psychopharmacology*, 199(3):457–480.

Berridge, K. C. and Robinson, T. E. (1998). What is the role of dopamine in reward: hedonic impact, reward learning, or incentive salience? *Brain Research Reviews*, 28(3):309–369.

Berry, D. A., Fristedt, B. (1985). *Bandit Problems*. Chapman and Hall, London.

Bertsekas, D. P. (1982). Distributed dynamic programming. *IEEE Transactions on Automatic Control*, 27:610–616.

Bertsekas, D. P. (1983). Distributed asynchronous computation of fixed points. *Mathematical Programming*, 27:107–120.

Bertsekas, D. P. (1987). *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Englewood Cliffs, NJ.

Bertsekas, D. P. (2005). *Dynamic Programming and Optimal Control, Volume 1,* third edition. Athena Scientific, Belmont, MA.

Bertsekas, D. P. (2012). *Dynamic Programming and Optimal Control, Volume 2: Approximate Dynamic Programming*, fourth edition. Athena Scientific, Belmont, MA.

Bertsekas, D. P. (2013). Rollout algorithms for discrete optimization: A survey. In *Handbook of Combinatorial Optimization*, pp. 2989–3013. Springer New York.

Bertsekas, D. P., Tsitsiklis, J. N. (1989). *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ.

Bertsekas, D. P., Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA.

Bertsekas, D. P., Tsitsiklis, J. N., and Wu, C. (1997). Rollout algorithms for combinatorial optimization. *Journal of Heuristics 3*(3):245–262.

Bertsekas, D. P., Yu, H. (2009). Projected equation methods for approximate solution of large linear systems. *Journal of Computational and Applied Mathematics*, 227(1):27–50.

Bhat, N., Farias, V., and Moallemi, C. C. (2012). Non-parametric approximate dynamic programming via the kernel method. In *Advances in Neural Information Processing Systems*, pp. 386–394.

Bhatnagar, S., Sutton, R., Ghavamzadeh, M., Lee, M. (2009). Natural actor–critic algorithms. *Automatica 45*(11).

Biermann, A. W., Fairfield, J. R. C., Beres, T. R. (1982). Signature table systems and learning. *IEEE Transactions on Systems, Man, and Cybernetics*, 12:635–648.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Clarendon, Oxford.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Blodgett, H. C. (1929). The effect of the introduction of reward upon the maze performance of rats. *University of California Publications in Psychology*, 4:113–134.

Boakes, R. A. and Costa, D. S. J. (2014). Temporal contiguity in associative learning: Iinterference and decay from an historical perspective. *Journal of Experimental Psychology: Animal Learning and Cognition*, 40(4):381–400.

Booker, L. B. (1982). *Intelligent Behavior as an Adaptation to the Task Environment*. Ph.D. thesis, University of Michigan, Ann Arbor.

Boone, G. (1997). Minimum-time control of the acrobot. In *1997 International Conference on Robotics and Automation*, pp. 3281–3287. IEEE Robotics and Automation Society.

Bottou, L., and Vapnik, V. (1992). Local learning algorithms. *Neural Computation 4*(6):888–900.

Boutilier, C., Dearden, R., Goldszmidt, M. (1995). Exploiting structure in policy construction. In *Proceedings of the Fourteenth*

International Joint Conference on Artificial Intelligence, pp. 1104–1111. Morgan Kaufmann.

Boyan, J. A., (1999). Least-squares temporal difference learning. *International Conference on Machine Learning 16*, pp. 49–56.

Boyan, J. (2002). Technical update: Least-squares temporal difference learning. *Machine Learning 49*:233–246.

Boyan, J. A., Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. Leen (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference*, pp. 369–376. MIT Press, Cambridge, MA.

Bradtke, S. J. (1993). Reinforcement learning applied to linear quadratic regulation. In S. J. Hanson, J. D. Cowan, and C. L. Giles (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 1992 Conference*, pp. 295–302. Morgan Kaufmann, San Mateo, CA.

Bradtke, S. J. (1994). *Incremental Dynamic Programming for On-Line Adaptive Optimal Control*. Ph.D. thesis, University of Massachusetts, Amherst. Appeared as CMPSCI Technical Report 94-62.

Bradtke, S. J., Barto, A. G. (1996). Linear least–squares algorithms for temporal difference learning. *Machine Learning*, 22:33–57.

Bradtke, S. J., Ydstie, B. E., Barto, A. G. (1994). Adaptive linear quadratic control using policy iteration. In *Proceedings of the American Control Conference*, pp. 3475–3479. American Automatic Control Council, Evanston, IL.

Bradtke, S. J., Duff, M. O. (1995). Reinforcement learning methods for continuous-time Markov decision problems. In G. Tesauro, D. Touretzky, and T. Leen (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference*, pp. 393–400. MIT Press, Cambridge, MA.

Brafman, R. I., Tennenholtz, M. (2003). R-max – a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3, 213–231.

Breiter, H. C., Aharon, I., Kahneman, D., Dale, A., and Shizgal, P. (2001). Functional imaging of neural responses to expectancy and experience of monetary gains and losses. *Neuron*, 30(2):619–639.

Breland, K. and Breland, M. (1961). The misbehavior of organisms. *American Psychologist*, 16(11):681–684.

Bridle, J. S. (1990). Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimates of parameters. In D. S. Touretzky (ed.), *Advances in Neural Information Processing Systems: Proceedings of the 1989 Conference*, pp. 211–217. Morgan Kaufmann, San Mateo, CA.

Broomhead, D. S., Lowe, D. (1988). Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2:321–355.

Bromberg-Martin, E. S., Matsumoto, M., Hong, S., and Hikosaka, O. (2010). A pallidus-habenula-dopamine pathway signals inferred stimulus values. *Journal of Neurophysiology*, 104(2):1068–1076.

Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games 4*(1):1–43.

Brown, J., Bullock, D., and Grossberg, S. (1999). How the basal ganglia use parallel excitatory and inhibitory learning pathways to selectively respond to unexpected rewarding cues. *The Journal of Neuroscience*, 19(23):10502–10511.

Bryson, A. E., Jr. (1996). Optimal control—1950 to 1985. *IEEE Control Systems*, 13(3):26–33.

Buchanan, B. G., Mitchell, T., Smith, R. G., and Jr., C. R. J. (1978). Models of learning systems. *Encyclopeadia of Computer Science and technology*, 11.

Buhusi, C. V. and Schmajuk, N. A. (1999). Timing in simple conditioning and occasion setting: A neural network approach. *Behavioural processes*, 45(1):33–57.

Burke, C. J., Dreher, J.-C., Seymour, B., and Tobler, P. N. (2014). State-dependent value representation: evidence from the stiatum. *Frontiers in Neuroscience*, 8.

Buşoniu, L., Lazaric, A., Ghavamzadeh, M., Munos, R., Babuška, R., and De Schutter, B. (2012). Least-squares methods for policy iteration. In Wiering and van Otterlo (Eds.) *Reinforcement Learning: State-of-the Art*, pp. 75–109. Springer Berlin Heidelberg.

Bush, R. R., Mosteller, F. (1955). *Stochastic Models for Learning*. Wiley, New York.

Byrne, J. H., Gingrich, K. J., Baxter, D. A. (1990). Computational capabilities of single neurons: Relationship to simple forms of associative and nonassociative learning in *aplysia*. In R. D. Hawkins and G. H. Bower (eds.), *Computational Models of Learning*, pp. 31–63. Academic Press, New York.

Calabresi, P., Picconi, B., Tozzi, A., and Filippo, M. D. (2007). Dopamine-mediated regulation of corticostriatal synaptic plasticity. *Trends in Neuroscience*, 30(5):211–219.

Camerer, C. (2003). *Behavioral game theory: Experiments in strategic interaction*. Princeton University Press.

Campbell, D. T. (1960). Blind variation and selective survival as a general strategy in knowledge-processes. In M. C. Yovits and S. Cameron (eds.), *Self-Organizing Systems*, pp. 205–231. Pergamon, New York.

Cao, X. R. (2009). Stochastic learning and optimization—A sensitivity-based approach. *Annual Reviews in Control 33*(1):11–24.

Carlström, J., Nordström, E. (1997). Control of self-similar ATM call traffic by reinforcement learning. In *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications 3*, pp. 54–62. Erlbaum, Hillsdale, NJ.

Chapman, D., Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the Twelfth International Conference on Artificial Intelligence*, pp. 726–731. Morgan Kaufmann, San Mateo, CA.

Chow, C.-S., Tsitsiklis, J. N. (1991). An optimal one-way multigrid algorithm for discrete-time stochastic control. *IEEE Transactions on Automatic Control*, 36:898–914.

Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 183–188. AAAI/MIT Press, Menlo Park, CA.

Christensen, J., Korf, R. E. (1986). A unified theory of heuristic evaluation functions and its application to learning. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pp. 148–152. Morgan Kaufmann, San Mateo, CA.

Cichosz, P. (1995). Truncating temporal differences: On the efficient implementation of TD(λ) for reinforcement learning. *Journal of Artificial Intelligence Research*, 2:287–318.

Claridge-Chang, A., Roorda, R. D., Vrontou, E., Sjulson, L., Li, H., Hirsh, J., and Miesenböck, G. (2009). Writing memories with light-addressable reinforcement circuitry. *Cell*, 139(2):405–415.

Clark, R. E. and Squire, L. R. (1998). Classical conditioning and brain systems: the role of awareness. *Science*, 280(5360):77–81.

Clark, W. A., Farley, B. G. (1955). Generalization of pattern recognition in a self-organizing system. In *Proceedings of the 1955 Western Joint Computer Conference*, pp. 86–91.

Clouse, J. (1996). *On Integrating Apprentice Learning and Reinforcement Learning TITLE2*. Ph.D. thesis, University of Massachusetts, Amherst. Appeared as CMPSCI Technical Report 96-026.

Clouse, J., Utgoff, P. (1992). A teaching method for reinforcement learning systems. In *Proceedings of the Ninth International Machine Learning Conference*, pp. 92–101. Morgan Kaufmann, San Mateo, CA.

Cobo, L. C., Zang, P., Isbell, C. L., and Thomaz, A. L. (2011). Automatic state abstraction from demonstration. In *IJCAAI Proceedings: International Joint Conference on Artificial Intelligence*, volume 22, page 1243.

Cohen, J. Y., Haesler, S., Vong, L., Lowell, B. B., and Uchida, N. (2012). Neuron-type-specific signals for reward and punishment in the ventral tegmental area. *Nature 482*(7383):85–88.

Colombetti, M., Dorigo, M. (1994). Training agent to perform sequential behavior. *Adaptive Behavior*, 2(3):247–275.

Connell, J. (1989). A colony architecture for an artificial creature. Technical Report AI-TR-1151. MIT Artificial Intelligence Laboratory, Cambridge, MA.

Connell, J., Mahadevan, S. (1993). *Robot Learning*. Kluwer Academic, Boston.

Connell, M. E., and Utgoff, P. E. (1987). Learning to control a dynamic physical system. *Computational intelligence 3*(1):330–337.

Contreras-Vidal, J. L. and Schultz, W. (1999). A predictive reinforcement model of dopamine neurons for learning approach behavior. *Journal of computational neuroscience*, 6(3):191–214.

Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search. In *Proceedings of the 5th International Conference on Computers and Games*, pp. 72–83.

Courville, A. C., Daw, N. D., and Touretzky, D. S. (2006). Bayesian theories of conditioning in a changing world. *Trends in Cognitive Science*, 10(7):294–300.

Craik, K. J. W. (1943). *The Nature of Explanation*. Cambridge University Press, Cambridge.

Crites, R. H. (1996). *Large-Scale Dynamic Optimization Using Teams of Reinforcement Learning Agents*. Ph.D. thesis, University of Massachusetts, Amherst.

Crites, R. H., Barto, A. G. (1996). Improving elevator performance using reinforcement learning. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pp. 1017–1023. MIT Press, Cambridge, MA.

Cross, J. G. (1973). A stochastic learning model of economic behavior. *The Quarterly Journal of Economics 87*(2):239-266.

Crow, T. J. (1968). Cortical synapses and reinforcement: a hypothesis. *Nature*, 219:736–737.

Curtiss, J. H. (1954). A theoretical comparison of the efficiencies of two classical methods and a Monte Carlo method for computing one component of the solution of a set of linear algebraic equations. In H. A. Meyer (ed.), *Symposium on Monte Carlo Methods*, pp. 191–233. Wiley, New York.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

Cziko, G. (1995). *Without Miracles: Universal Selection Theory and the Second Darvinian Revolution*. MIT Press, Cambridge, MA.

Daniel, J. W. (1976). Splines and efficiency in dynamic programming. *Journal of Mathematical Analysis and Applications*, 54:402–407.

Dann, C., Neumann, G., Peters, J. (2014). Policy evaluation with temporal differences: A survey and comparison. *Journal of Machine Learning Research 15*:809–883.

Daw, N. D., Courville, A. C., and Touretzky, D. S. (2003). Timing and partial observability in the dopamine system. In *Advances in neural information processing systems*, pages 99–106.

Daw, N. D., Courville, A. C., and Touretzky, D. S. (2006). Representation and timing in theories of the dopamine system. *Neural Computation*, 18(7):1637–1677.

Daw, N., Niv, Y., and Dayan, P. (2005). Uncertainty based competition between prefrontal and dorsolateral striatal systems for behavioral control. *Nature Neuroscience*, 8(12):1704–1711.

Daw, N. D. and Shohamy, D. (2008). The cognitive neuroscience of motivation and learning. *Social Cognition*, 26(5):593–620.

Dayan, P. (1991). Reinforcement comparison. In D. S. Touretzky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton (eds.), *Connectionist Models: Proceedings of the 1990 Summer School*, pp. 45–51. Morgan Kaufmann, San Mateo, CA.

Dayan, P. (1992). The convergence of TD(λ) for general λ. *Machine Learning*, 8:341–362.

Dayan, P. (2008). The role of value systems in decision making. In Engel, C. and Singer, W., editors, *Better Than Conscious?: Decision Making, the Human Mind, and Implications For Institutions (Strüngmann Forum Reports)*, pages 51–70. MIT Press, Cambridge, MA.

Dayan, P. and Abbott, L. F. (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press, Cambridge, MA.

Dayan, P., and Berridge, K. C. (2014). Model-based and model-free Pavlovian reward learning: Revaluation, revision, and revaluation. *Cognitive, Affective, & Behavioral Neuroscience*, 14(2):473–492.

Dayan, P., and Hinton, G. E. (1993). Feudal reinforcement learning. In S. J. Hanson, J. D. Cohen, and C. L. Giles (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 1992 Conference*, pp. 271–278. Morgan Kaufmann, San Mateo, CA.

Dayan, P. and Niv, Y. (2008). Reinforcement learning: the good, the bad and the ugly. *Current Opinion in Neurobiology*, 18(2):185–196.

Dayan, P., Niv, Y., Seymour, B., and Daw, N. D. (2006). The misbehavior of value and the discipline of the will. *Neural*

Networks 19(8):1153–1160.

Dayan, P., and Sejnowski, T. (1994). TD(λ) converges with probability 1. *Machine Learning*, 14:295–301.

De Asis, K., Hernandez-Garcia, J. F., Holland, G. Z., and Sutton, R. S. (2017). Multi-step Reinforcement Learning: A Unifying Algorithm. arXiv preprint arXiv:1703.01327.

Dean, T., Lin, S.-H. (1995). Decomposition techniques for planning in stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1121–1127. Morgan Kaufmann. See also Technical Report CS-95-10, Brown University, Department of Computer Science, 1995.

Degris, T., White, M., Sutton, R. S. (2012). Off-policy actor–critic. *Proceedings of the 29th International Conference on Machine Learning*.

DeJong, G., Spong, M. W. (1994). Swinging up the acrobot: An example of intelligent control. In *Proceedings of the American Control Conference*, pp. 2158–2162. American Automatic Control Council, Evanston, IL.

Denardo, E. V. (1967). Contraction mappings in the theory underlying dynamic programming. *SIAM Review*, 9:165–177.

Dennett, D. C. (1978). *Brainstorms*, pp. 71–89. Bradford/MIT Press, Cambridge, MA.

Derthick, M. (1984). Variations on the Boltzmann machine learning algorithm. Carnegie-Mellon University Department of Computer Science Technical Report No. CMU-CS-84-120.

Deutsch, J. A. (1953). A new type of behaviour theory. *British Journal of Psychology. General Section*, 44(4):304–317.

Deutsch, J. A. (1954). A machine with insight. *Quarterly Journal of Experimental Psychology*, 6(1):6–11.

Dick, T. (2015). *Policy Gradient Reinforcement Learning Without Regret*. MSc Thesis, University of Alberta.

Dickinson, A. (1980). *Contemporary Animal Learning Theory*. Cambridge University Press, Cambridge.

Dickinson, A. (1985). Actions and habits: the development of behavioral autonomy. *Phil. Trans. R. Soc. Lond. B*, 308(1135):67–78.

Dickinson, A. and Balleine, B. W. (2002). The role of learning in motivation. In Gallistel, C. R., editor, *Stevens handbook of experimental psychology*, volume 3, pages 497–533. Wiley, NY.

Dietterich, T. and Buchanan, B. G. (1984). The role of the critic in learning systems. In Selfridge, O. G., Rissland, E. L., and Arbib, M. A., editors, *Adaptive Control of Ill-Defined Systems*, pages 127–147. Plenum Press, NY. Proceedings of the NATO Advanced Research Institute on Adaptive Control of Ill-defined Systems, NATO Conference Series II, Systems Science, Vol. 16.

Dietterich, T. G., Flann, N. S. (1995). Explanation-based learning and reinforcement learning: A unified view. In A. Prieditis and S. Russell (eds.), *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 176–184. Morgan Kaufmann, San Francisco.

Dietterich, T. G. and Wang, X. (2002). Batch value function approximation via support vectors. In *Advances in Neural Information Processing Systems 14*, pp. 1491–1498. Cambridge, MA: MIT Press.

Diuk, C., Cohen, A., and Littman, M. L. (2008). An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th international conference on machine learning*, pages 240–247. ACM New York, NY.

Dolan, R. J. and Dayan, P. (2013). Goals and habits in the brain. *Neuron*, 80(2):312–325.

Doll, B. B., Simon, D. A., and Daw, N. D. (2012). The ubiquity of model-based reinforcement learning. *Current Opinion in Neurobiology*, 22:1–7.

Donahoe, J. W. and Burgos, J. E. (2000). Behavior analysis and revaluation. *Journal of the Experimental Analysis of Behavior*, 74(3):331–346.

Dorigo, M. and Colombetti, M. (1994). Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence*, 71(2):321–370.

Doya, K. (1996). Temporal difference learning in continuous time and space. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pp. 1073–1079. MIT Press, Cambridge, MA.

Doya, K. and Sejnowski, T. J. (1995). A novel reinforcement model of birdsong vocalization learning. In Tesauro, G., Touretzky, D. S., and Leen, T., editors, *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference*, pages 101–108, Cambridge, MA. MIT Press.

Doya, K. and Sejnowski, T. J. (1998). A computational model of birdsong learning by auditory experience and auditory feedback. In *Central auditory processing and neural modeling*, pages 77–88. Springer US.

Doyle, P. G., Snell, J. L. (1984). *Random Walks and Electric Networks*. The Mathematical Association of America. Carus Mathematical Monograph 22.

Dreyfus, S. E., Law, A. M. (1977). *The Art and Theory of Dynamic Programming*. Academic Press, New York.

Duda, R. O., Hart, P. E. (1973). *Pattern Classification and Scene Analysis*. Wiley, New York.

Duff, M. O. (1995). Q-learning for bandit problems. In A. Prieditis and S. Russell (eds.), *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 209–217. Morgan Kaufmann, San Francisco.

Egger, D. M. and Miller, N. E. (1962). Secondary reinforcement in rats as a function of information value and reliability of the stimulus. *Journal of Experimental Psychology*, 64:97–104.

Eshel, N., Bukwich, M., Rao, V., Hemmelder, V., Tian, J., and Uchida, N. (2015). Arithmetic and local circuitry underlying dopamine prediction errors. *Nature 525*(7568):243–246.

Eshel, N., Tian, J., Bukwich, M., and Uchida, N. (2016). Dopamine neurons share common response function for reward prediction error. *Nature Neuroscience 19*(3):479–486.

Estes, W. K. (1943). Discriminative conditioning. I. A discriminative property of conditioned anticipation. *Journal of Experimental Psychology 32*(2):150–155.

Estes, W. K. (1948). Discriminative conditioning. II. Effects of a Pavlovian conditioned stimulus upon a subsequently established operant response. *Journal of experimental psychology 38*(2):173–177.

Estes, W. K. (1950). Toward a statistical theory of learning. *Psycholollgical Review*, 57:94–107.

Farley, B. G., Clark, W. A. (1954). Simulation of self-organizing systems by digital computer. *IRE Transactions on Information*

Theory, 4:76–84.

Farries, M. A. and Fairhall, A. L. (2007). Reinforcement learning with modulated spike timingdependent synaptic plasticity. *Journal of neurophysiology*, 98(6):3648–3665.

Feldbaum, A. A. (1965). *Optimal Control Systems*. Academic Press, New York.

Finch, G., and Culler, E. (1934). Higher order conditioning with constant motivation. *The American Journal of Psychology*, 596–602.

Finnsson, H., Björnsson, Y. (2008). Simulation-based approach to general game playing. In *Proceedings of the Association for the Advancement of Artificial Intelligence*, 259–264.

Fiorillo, C. D., Tobler, P. N., and Schultz, W. (2003). Discrete coding of reward probability and uncertainty by dopamine neurons. *Science*, 299(5614):1898–1902.

Fiorillo, C. D., Yun, S. R., and Song, M. R. (2013). Diversity and homogeneity in responses of midbrain dopamine neurons. *The Journal of Neuroscience*, 33(11):4693–4709.

Florian, R. V. (2007). Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural Computation*, 19(6):1468–1502.

Fogel, L. J., Owens, A. J., Walsh, M. J. (1966). *Artificial intelligence through simulated evolution*. John Wiley and Sons.

Frey, U. and Morris, R. G. M. (1997). Synaptic tagging and long-term potentiation. *Nature*, 385(6616):533–536.

Friedman, J. H., Bentley, J. L., and Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software 3*(3):209–226.

Friston, K. J., Tononi, G., Reeke, G. N., Sporns, O., Edelman, G. M. (1994). Value-dependent selection in the brain: Simulation in a synthetic neural model. *Neuroscience*, 59:229–243.

Fu, K. S. (1970). Learning control systems—Review and outlook. *IEEE Transactions on Automatic Control*, 15:210–221.

Galanter, E., Gerstenhaber, M. (1956). On thought: The extrinsic theory. *Psychological Review*, 63:218–227.

Gallant, S. I. (1993). *Neural Network Learning and Expert Systems*. MIT Press, Cambridge, MA.

Gallistel, C. R. (2005). Deconstructing the law of effect. *Games and Economic Behavior 52*(2), 410-423.

Gällmo, O., Asplund, L. (1995). Reinforcement learning by construction of hypothetical targets. In J. Alspector, R. Goodman, and T. X. Brown (eds.), *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications 2*, pp. 300–307. Erlbaum, Hillsdale, NJ.

Gardner, M. (1973). Mathematical games. *Scientific American*, 228(1):108–115.

Geist, M., Scherrer, B. (2014). Off-policy learning with eligibility traces: A survey. *Journal of Machine Learning Research 15*:289–333.

Gelperin, A., Hopfield, J. J., Tank, D. W. (1985). The logic of *limax* learning. In A. Selverston (ed.), *Model Neural Networks and Behavior*, pp. 247–261. Plenum Press, New York.

Genesereth, M., Thielscher, M. (2014). General game playing. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(2), 1–229.

Gershman, S. J., Moustafa, A. A., and Ludvig, E. A. (2013). Time representation in reinforcement learning models of the basal ganglia. *Frontiers in computational neuroscience*, 7.

Gershman, S. J., Pesaran, B., and Daw, N. D. (2009). Human reinforcement learning subdivides structured action spaces by learning effector-specific values. *Journal of Neuroscience 29*(43):13524–13531.

Gershman, S. J. and Niv, Y. (2010). Learning latent structure: Carving nature at its joints. *Current Opinions in Neurobiology*, 20:251–256.

Ghiassian, S., Rafiee, B., Sutton, R. S. (2016). A first empirical study of emphatic temporal difference learning. Workshop on Continual Learning and Deep Learning at the Conference on Neural Information Processing Systems. ArXiv:1705.04185.

Gibbs, C. M., Cool, V., Land, T., Kehoe, E. J., and Gormezano, I. (1991). Second-order conditioning of the rabbits nictitating membrane response. *Integrative Physiological and Behavioral Science 26*(4):282–295.

Gittins, J. C., Jones, D. M. (1974). A dynamic allocation index for the sequential design of experiments. In J. Gani, K. Sarkadi, and I. Vincze (eds.), *Progress in Statistics*, pp. 241–266. North-Holland, Amsterdam–London.

Glimcher, P. W. (2011). Understanding dopamine and reinforcement learning: The dopamine reward prediction error hypothesis. *Proceedings of the National Academy of Sciences*, 108(Supplement 3):15647–15654.

Glimcher, P. W. (2003). *Decisions, uncertainty, and the brain: The science of neuroeconomics*. MIT Press, Cambridge, MA.

Glimcher, P. W. and Fehr, E., editors (2013). *Neuroeconomics: Decision making and the brain, Second Edition*. Academic Press.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA.

Goldstein, H. (1957). *Classical Mechanics*. Addison-Wesley, Reading, MA.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

Goodwin, G. C., Sin, K. S. (1984). *Adaptive Filtering Prediction and Control*. Prentice-Hall, Englewood Cliffs, NJ.

Gopnik, A., Glymour, C., Sobel, D., Schulz, L. E., Kushnir, T., and Danks, D. (2004). A theory of causal learning in children: Causal maps and Bayes nets. *Psychological Review*, 111(1):3–32.

Gordon, G. J. (1995). Stable function approximation in dynamic programming. In A. Prieditis and S. Russell (eds.), *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 261–268. Morgan Kaufmann, San Francisco. An expanded version was published as Technical Report CMU-CS-95-103. Carnegie Mellon University, Pittsburgh, PA, 1995.

Gordon, G. J. (1996a). Chattering in SARSA(λ). CMU learning lab internal report.

Gordon, G. J. (1996b). Stable fitted reinforcement learning. In D. S. Touretzky, M. C. Mozer, M. E. Hasselmo (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pp. 1052–1058. MIT Press, Cambridge, MA.

Gordon, G. J. (1999). *Approximate solutions to Markov decision processes*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Gordon, G. J. (2001). Reinforcement learning with function approximation converges to a region. *Advances in neural information processing systems*.

Graybiel, A. M. (2000). The basal ganglia. *Current Biology*, 10(14):R509–R511.

Greensmith, E., Bartlett, P. L., Baxter, J. (2001). Variance reduction techniques for gradient estimates in reinforcement learning. In *Advances in Neural Information Processing Systems: Proceedings of the 2000 Conference*, pp. 1507–1514.

Greensmith, E., Bartlett, P. L., Baxter, J. (2004). Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research 5*(Nov), 1471–1530.

Griffith, A. K. (1966). A new machine learning technique applied to the game of checkers. Technical Report Project MAC, Artificial Intelligence Memo 94. Massachusetts Institute of Technology, Cambridge, MA.

Griffith, A. K. (1974). A comparison and evaluation of three machine learning procedures as applied to the game of checkers. *Artificial Intelligence*, 5:137–148.

Grondman, I., Busoniu, L., Lopes, G. A., Babuska, R. (2012). A survey of actor–critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 42*(6), 1291–1307.

Grossberg, S. (1975). A neural model of attention, reinforcement, and discrimination learning. *International Review of Neurobiology*, 18:263–327.

Grossberg, S. and Schmajuk, N. A. (1989). Neural dynamics of adaptive timing and temporal discrimination during associative learning. *Neural Networks*, 2(2):79–102.

Gullapalli, V. (1990). A stochastic reinforcement algorithm for learning real-valued functions. *Neural Networks*, 3:671–692.

Gullapalli, V. and Barto, A. G. (1992). Shaping as a method for accelerating reinforcement learning. In *Proceedings of the 1992 IEEE International Symposium on Intelligent Control*, pages 554–559. IEEE.

Gurney, K., Prescott, T. J., and Redgrave, P. (2001). A computational model of action selection in the basal ganglia I. A new functional anatomy. *Biological cybernetics*, 84(6):401–410.

Gurvits, L., Lin, L.-J., Hanson, S. J. (1994). Incremental learning of evaluation functions for absorbing Markov chains: New methods and theorems. Preprint.

Hackman, L. (2012). *Faster Gradient-TD Algorithms* (MSc dissertation, University of Alberta).

Hallak, A., Tamar, A., Munos, R., Mannor, S. (2016). Generalized emphatic temporal difference learning: Bias-variance analysis. In *Thirtieth AAAI Conference on Artificial Intelligence*.

Hammer, M. (1997). The neural basis of associative reward learning in honeybees. *Trends in Neuroscience*, 20:245–252.

Hammer, M. and Menzel, R. (1995). Learning and memory in the honeybee. *Journal of Neuroscience*, 15(3):1617–1630.

Hampson, S. E. (1983). *A Neural Model of Adaptive Behavior*. Ph.D. thesis, University of California, Irvine.

Hampson, S. E. (1989). *Connectionist Problem Solving: Computational Aspects of Biological Learning*. Birkhauser, Boston.

Hare, T. A., O'Doherty, J., Camerer, C. F., Schultz, W., and Rangel, A. (2008). Dissociating the role of the orbitofrontal cortex and the striatum in the computation of goal values and prediction errors. *The Journal of Neuroscience*, 28(22):5623–5630.

Hassabis, D. and Maguire, E. A. (2007). Deconstructing episodic memory with construction. *Trends in cognitive sciences*, 11(7):299–306.

Hawkins, R. D., Kandel, E. R. (1984). Is there a cell-biological alphabet for simple forms of learning? *Psychological Review*, 91:375–391.

He, K., Huertas, M., Hong, S. Z., Tie, X., Hell, J. W., Shouval, H., and Kirkwood, A. (2015). Distinct eligibility traces for LTP and LTD in cortical synapses. *Neuron*, 88(3):528–538.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the 1992 IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778.

Hebb, D. O. (1949). *The organization of behavior: A neuropsychological theory*. John Wiley and Sons Inc., New York. Reissued by Lawrence Erlbaum Associates Inc., Mahwah NJ, 2002.

Hengst, B. (2012). Hierarchical approaches. In Wiering and van Otterlo (Eds.) *Reinforcement Learning: State-of-the Art*, pp. 293–323. Springer Berlin Heidelberg.

Herrnstein, R. J. (1970). On the Law of Effect. *Journal of the Experimental Analysis of Behavior 13*(2), 243-266.

Hersh, R., Griego, R. J. (1969). Brownian motion and potential theory. *Scientific American*, 220:66–74.

Hester, T., and Stone, P. (2012). Learning and using models. In Wiering and van Otterlo (Eds.) *Reinforcement Learning: State-of-the Art*, pp. 111–141. Springer Berlin Heidelberg.

Hesterberg, T. C. (1988), *Advances in importance sampling*, Ph.D. Dissertation, Statistics Department, Stanford University.

Hilgard, E. R. (1956). *Theories of Learning, Second Edition*. Appleton-Century-Cofts, Inc., New York.

Hilgard, E. R., Bower, G. H. (1975). *Theories of Learning*. Prentice-Hall, Englewood Cliffs, NJ.

Hinton, G. E. (1984). Distributed representations. Technical Report CMU-CS-84-157. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

Hinton, G. E., Osindero, S., and Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554.

Hochreiter, S., Schmidhuber, J. (1997). LTSM can solve hard time lag problems. In *Advances in Neural Information Processing Systems: Proceedings of the 1996 Conference*, pp. 473–479. MIT Press, Cambridge, MA.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor.

Holland, J. H. (1976). Adaptation. In R. Rosen and F. M. Snell (eds.), *Progress in Theoretical Biology*, vol. 4, pp. 263–293. Academic Press, New York.

Holland, J. H. (1986). Escaping brittleness: The possibility of general-purpose learning algorithms applied to rule-based systems. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (eds.), *Machine Learning: An Artificial Intelligence Approach*, vol. 2, pp. 593–623. Morgan Kaufmann, San Mateo, CA.

Hollerman, J. R. and Schultz, W. (1998). Dopmine neurons report an error in the temporal prediction of reward during learning.

Nature Neuroscience, 1:304–309.

Houk, J. C., Adams, J. L., Barto, A. G. (1995). A model of how the basal ganglia generates and uses neural signals that predict reinforcement. In J. C. Houk, J. L. Davis, and D. G. Beiser (eds.), *Models of Information Processing in the Basal Ganglia*, pp. 249–270. MIT Press, Cambridge, MA.

Howard, R. (1960). *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA.

Hull, C. L. (1932). The goal-gradient hypothesis and maze learning. *Psychological Review*, 39(1):25–43.

Hull, C. L. (1943). *Principles of Behavior*. Appleton-Century, New York.

Hull, C. L. (1952). *A Behavior System*. Wiley, New York.

Ioffe, S., and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv:1502.03167.

İpek, E., Mutlu, O., Martínez, J. F., and Caruana, R. (2008). Self-optimizing memory controllers: A reinforcement learning approach. In *35th International Symposium on Computer Architecture, ISCA'08*, pages 39–50. IEEE.

Izhikevich, E. M. (2007). Solving the distal reward problem through linkage of STDP and dopamine signaling. *Cerebral cortex*, 17(10):2443–2452.

Jaakkola, T., Jordan, M. I., Singh, S. P. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201.

Jaakkola, T., Singh, S. P., Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable Markov decision problems. In G. Tesauro, D. S. Touretzky, T. Leen (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference*, pp. 345–352. MIT Press, Cambridge, MA.

Joel, D., Niv, Y., and Ruppin, E. (2002). Actor–critic models of the basal ganglia: New anatomical and computational perspectives. *Neural networks*, 15(4):535–547.

Johanson, E. B., Killeen, P. R., Russell, V. A., Tripp, G., Wickens, J. R., Tannock, R., Williams, J., and Sagvolden, T. (2009). Origins of altered reinforcement effects in ADHD. *Behavioral and Brain Functions*, 5(7).

Johnson, A. and Redish, A. D. (2007). Neural ensembles in CA3 transiently encode paths forward of the animal at a decision point. *The Journal of neuroscience*, 27(45):12176–12189.

Kaelbling, L. P. (1993a). Hierarchical learning in stochastic domains: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 167–173. Morgan Kaufmann, San Mateo, CA.

Kaelbling, L. P. (1993b). *Learning in Embedded Systems*. MIT Press, Cambridge, MA.

Kaelbling, L. P. (Ed.) (1996). Special triple issue on reinforcement learning, *Machine Learning 22*(1/2/3).

Kaelbling, L. P., Littman, M. L., Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285.

Kahneman, D. and Tversky, A. (1979). Prospect theory: An analysis of decision under risk. *Econometrica: Journal of the Econometric Society*, 47:263–291.

Kakade, S. (2002). A natural policy gradient. *Advances in neural information processing systems 2*, 1531–1538.

Kakade, S. M. (2003). On the Sample Complexity of Reinforcement Learning (Doctoral dissertation, University of London).

Kakutani, S. (1945). Markov processes and the Dirichlet problem. *Proceedings of the Japan Academy*, 21:227–233.

Kalos, M. H., Whitlock, P. A. (1986). *Monte Carlo Methods*. Wiley, New York.

Kamin, L. J. (1968). "Attention-like" processes in classical conditioning. In Jones, M. R., editor, *Miami Symposium on the Prediction of Behavior, 1967: Aversive Stimulation*, pages 9–31. University of Miami Press, Coral Gables, Florida.

Kamin, L. J. (1969). Predictability, surprise, attention, and conditioning. In Campbell, B. A. and Church, R. M., editors, *Punishment and Aversive Behavior*, pages 279–296. Appleton-Century-Crofts, New York, NY.

Kandel, E. R., Schwartz, J. H., Jessell, T. M., Siegelbaum, S. A., and Hudspeth, A. J., editors (2013). *Principles of Neural Science, Fifth Edition*. McGraw-Hill Companies, Inc.

Kanerva, P. (1988). *Sparse Distributed Memory*. MIT Press, Cambridge, MA.

Kanerva, P. (1993). Sparse distributed memory and related models. In M. H. Hassoun (ed.), *Associative Neural Memories: Theory and Implementation*, pp. 50–76. Oxford University Press, New York.

Karampatziakis, N., and Langford, J. (2010). Online importance weight aware updates. ArXiv:1011.1576.

Kashyap, R. L., Blaydon, C. C., Fu, K. S. (1970). Stochastic approximation. In J. M. Mendel and K. S. Fu (eds.), *Adaptive, Learning, and Pattern Recognition Systems: Theory and Applications*, pp. 329–355. Academic Press, New York.

Kearns, M., Singh, S. (2002). Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2-3), 209–232.

Keerthi, S. S., Ravindran, B. (1997). Reinforcement learning. In E. Fiesler and R. Beale (eds.), *Handbook of Neural Computation*, C3. Oxford University Press, New York.

Kehoe, E. J. (1982). Conditioning with serial compound stimuli: Theoretical and empirical issues. *Experimental Animal Behavior*, 1:30–65.

Kehoe, E. J., Schreurs, B. G., and Graham, P. (1987). Temporal primacy overrides prior training in serial compound conditioning of the rabbits nictitating membrane response. *Animal Learning & Behavior*, 15(4):455–464.

Keiflin, R. and Janak, P. H. (2015). Dopamine prediction errors in reward learning and addiction: Ffrom theory to neural circuitry. *Neuron*, 88(2):247– 263.

Kimble, G. A. (1961). *Hilgard and Marquis' Conditioning and Learning*. Appleton-Century-Crofts, New York.

Kimble, G. A. (1967). *Foundations of Conditioning and Learning*. Appleton-Century-Crofts, New York.

Klopf, A. H. (1972). Brain function and adaptive systems—A heterostatic theory. Technical Report AFCRL-72-0164, Air Force Cambridge Research Laboratories, Bedford, MA. A summary appears in *Proceedings of the International Conference on Systems, Man, and Cybernetics*. IEEE Systems, Man, and Cybernetics Society, Dallas, TX, 1974.

Klopf, A. H. (1975). A comparison of natural and artificial intelligence. *SIGART Newsletter*, 53:11–13.

Klopf, A. H. (1982). *The Hedonistic Neuron: A Theory of Memory, Learning, and Intelligence*. Hemisphere, Washington, DC.

Klopf, A. H. (1988). A neuronal model of classical conditioning. *Psychobiology*, 16:85–125.

Kober, J. and Peters, J. (2012). Reinforcement learning in robotics: A survey. In Wiering, M. and van Otterlo, M., editors, *Reinforcement Learning: State-of-the-Art*, pages 579–610. Springer-Verlag, Berlin.

Kocsis, L., Szepesvári, Cs. (2006). Bandit based Monte-Carlo planning. In *Proceedings of the European Conference on Machine Learning*, 282–293. Springer Berlin Heidelberg.

Kohonen, T. (1977). *Associative Memory: A System Theoretic Approach*. Springer-Verlag, Berlin.

Koller, D., Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

Kolodziejski, C., Porr, B., and Wörgötter, F. (2009). On the asymptotic equivalence between differential Hebbian and temporal difference learning. *Neural computation*, 21(4):1173–1202.

Kolter, J. Z. (2011). The fixed points of off-policy TD. *Advances in Neural Information Processing Systems 24*, pp. 2169–2177.

Konidaris, G. D., Osentoski, S., Thomas, P. S. (2011). Value function approximation in reinforcement learning using the Fourier basis, *Proceedings of the Twenty-Fifth Conference of the Association for the Advancement of Artificial Intelligence*, pp. 380–385.

Korf, R. E. (1988). Optimal path finding algorithms. In L. N. Kanal and V. Kumar (eds.), *Search in Artificial Intelligence*, pp. 223–267. Springer Verlag, Berlin.

Korf, R. E. (1990). Real-time heuristic search. *Artificial Intelligence 42*(2–3), 189–211.

Koshland, D. E. (1980). *Bacterial Chemotaxis as a Model Bhavioral System*. Raven Press, New York.

Koza, J. R. (1992). *Genetic programming: On the programming of computers by means of natural selection* (Vol. 1). MIT press.

Kraft, L. G., Campagna, D. P. (1990). A summary comparison of CMAC neural network and traditional adaptive control systems. In T. Miller, R. S. Sutton, and P. J. Werbos (eds.), *Neural Networks for Control*, pp. 143–169. MIT Press, Cambridge, MA.

Kraft, L. G., Miller, W. T., Dietz, D. (1992). Development and application of CMAC neural network-based control. In D. A. White and D. A. Sofge (eds.), *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pp. 215–232. Van Nostrand Reinhold, New York.

Kumar, P. R., Varaiya, P. (1986). *Stochastic Systems: Estimation, Identification, and Adaptive Control*. Prentice-Hall, Englewood Cliffs, NJ.

Kumar, P. R. (1985). A survey of some results in stochastic adaptive control. *SIAM Journal of Control and Optimization*, 23:329–380.

Kumar, V., Kanal, L. N. (1988). The CDP: A unifying formulation for heuristic search, dynamic programming, and branch-and-bound. In L. N. Kanal and V. Kumar (eds.), *Search in Artificial Intelligence*, pp. 1–37. Springer-Verlag, Berlin.

Kushner, H. J., Dupuis, P. (1992). *Numerical Methods for Stochastic Control Problems in Continuous Time*. Springer-Verlag, New York.

Lagoudakis, M., Parr, R. (2003). Least squares policy iteration. *Journal of Machine Learning Research 4*:1107–1149.

Lai, T. L., Robbins, H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22.

Lakshmivarahan, S. and Narendra, K. S. (1982). Learning algorithms for two-person zero-sum stochastic games with incomplete information: A unified approach. *SIAM Journal of Control and Optimization*, 20:541–552.

Lammel, S., Lim, B. K., and Malenka, R. C. (2014). Reward and aversion in a heterogeneous midbrain dopamine system. *Neuropharmacology*, 76:353–359.

Lane, S. H., Handelman, D. A., Gelfand, J. J. (1992). Theory and development of higher-order CMAC neural networks. *IEEE Control Systems 12*(2):23–30.

Lang, K. J., Waibel, A. H., Hinton, G. E. (1990). A time-delay neural network architecture for isolated word recognition. *Neural Networks*, 3:33–43.

Lange, S., Gabel, T., and Riedmiller, M. (2012). Batch reinforcement learning. In Wiering and van Otterlo (Eds.) *Reinforcement Learning: State-of-the Art*, pp. 45–73. Springer Berlin Heidelberg.

LeCun, Y. (1985). Une procdure d'apprentissage pour rseau a seuil asymmetrique (a learning scheme for asymmetric threshold networks). In *Proceedings of Cognitiva 85*, Paris, France.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Legenstein, R. and andW. Maass, D. P. (2008). A learning theory for reward-modulated spike-timing-dependent plasticity with application to biofeedback. *PLoS Computational Biology*, 4(10).

Levy, W. B. and Steward, D. (1983). Temporal contiguity requirements for long-term associative potentiation/depression in thehippocampus. *Neuroscience*, 8:791–797.

Lewis, F. L., Liu, D. (Eds.). (2013). *Reinforcement Learning and Approximate Dynamic Programming for Feedback Control*. John Wiley and Sons.

Lewis, R. L., Howes, A., and Singh, S. (2014). Computational rationality: Linking mechanism and behavior through utility maximization. *Topics in Cognitive Science*, 6(2):279–311.

Li, L. (2012). Sample complexity bounds of exploration. In Wiering and van Otterlo (Eds.) *Reinforcement Learning: State-of-the Art*, pp. 175–204. Springer Berlin Heidelberg.

Li, L., Chu, W., Langford, J., and Schapire, R. E. (2010). A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th International Conference on World Wide Web*, pages 661–670. ACM.

Lin, C.-S., Kim, H. (1991). CMAC-based adaptive critic self-learning control. *IEEE Transactions on Neural Networks*, 2:530–533.

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321.

Lin, L.-J., Mitchell, T. (1992). Reinforcement learning with hidden states. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pp. 271–280. MIT Press, Cambridge, MA.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 157–163. Morgan Kaufmann, San Francisco.

Littman, M. L., Cassandra, A. R., Kaelbling, L. P. (1995). Learning policies for partially observable environments: Scaling up. In A. Prieditis and S. Russell (eds.), *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 362–370. Morgan Kaufmann, San Francisco.

Littman, M. L., Dean, T. L., Kaelbling, L. P. (1995). On the complexity of solving Markov decision problems. In *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence*, pp. 394–402.

Liu, J. S. (2001). *Monte Carlo strategies in scientific computing*. Berlin, Springer-Verlag.

Liu, W., Pokharel, P. P., and Principe, J. C. (2008). The kernel least-mean-square algorithm. *IEEE Transactions on Signal Processing 56*(2):543–554.

Ljung, L. (1998). System identification. In Procházka, A., Uhlíř, J., Rayner, P. W. J., and Kingsbury, N. G., editors, *Signal Analysis and Prediction*, pages 163–173. Springer Science + Business Media New York, LLC.

Ljung, L., Söderstrom, T. (1983). *Theory and Practice of Recursive Identification*. MIT Press, Cambridge, MA.

Ljungberg, T., Apicella, P., and Schultz, W. (1992). Responses of monkey dopamine neurons during learning of behavioral reactions. *Journal of Neurophysiology*, 67(1):145–163.

Lovejoy, W. S. (1991). A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, 28:47–66.

Luce, D. (1959). *Individual Choice Behavior*. Wiley, New York.

Ludvig, E. A., Bellemare, M. G., and Pearson, K. G. (2011). A primer on reinforcement learning in the brain: Psychological, computational, and neural perspectives. In Alonso, E. and Mondragón, E., editors, *Computational neuroscience for advancing artificial intelligence: Models, methods and applications*, pages 111–44. Medical Information Science Reference, Hershey PA.

Ludvig, E. A., Sutton, R. S., and Kehoe, E. J. (2008). Stimulus representation and the timing of reward-prediction errors in models of the dopamine system. *Neural Computation*, 20(12):3034–3054.

Ludvig, E. A., Sutton, R. S., and Kehoe, E. J. (2012). Evaluating the TD model of classical conditioning. *Learning & behavior*, 40(3):305–319.

Machado, A. (1997). Learning the temporal dynamics of behavior. *Psychological review*, 104(2):241–265.

Mackintosh, N. J. (1975). A theory of attention: Variations in the associability of stimuli with reinforcement. *Psychological Review*, 82(4):276–298.

Mackintosh, N. J. (1983). *Conditioning and Associative Learning*. Oxford: Clarendon Press.

Maclin, R., and Shavlik, J. W. (1994). Incorporating advice into agents that learn from reinforcements. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 694–699. AAAI Press, Menlo Park, CA.

Maei, H. R. (2011). *Gradient temporal-difference learning algorithms*. PhD thesis, University of Alberta.

Maei, H. R., and Sutton, R. S. (2010). GQ(λ): A general gradient algorithm for temporal-difference prediction learning with eligibility traces. In *Proceedings of the Third Conference on Artificial General Intelligence,* pp. 91–96.

Maei, H. R., Szepesvári, Cs., Bhatnagar, S., Precup, D., Silver, D., and Sutton, R. S. (2009). Convergent temporal-difference learning with arbitrary smooth function approximation. In *Advances in Neural Information Processing Systems*, pp. 1204–1212.

Maei, H. R., Szepesvári, Cs., Bhatnagar, S., and Sutton, R. S. (2010). Toward off-policy learning control with function approximation. In *Proceedings of the 27th International Conference on Machine Learning,* pp. 719–726).

Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22:159–196.

Mahadevan, S., Liu, B., Thomas, P., Dabney, W., Giguere, S., Jacek, N., Gemp, I., Liu, J. (2014). Proximal reinforcement learning: A new theory of sequential decision making in primal-dual spaces. ArXiv preprint arXiv:1405.6757.

Mahadevan, S., and Connell, J. (1992). Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55:311–365.

Mahmood, A. R. (2017). Incremental Off-policy Reinforcement Learning Algorithms. University of Alberta PhD thesis.

Mahmood, A. R., and Sutton, R. S. (2015). Off-policy learning based on weighted importance sampling with linear computational complexity. In *Proceedings of the 31st Conference on Uncertainty in Artificial Intelligence*, Amsterdam, Netherlands.

Mahmood, A. R., Sutton, R. S., Degris, T., and Pilarski, P. M. (2012). Tuning-free step-size adaptation. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on* (pp. 2121–2124). IEEE.

Mahmood, A. R., Yu, H, Sutton, R. S. (2017). Multi-step off-policy learning without importance sampling ratios. ArXiv 1702.03006.

Mahmood, A. R., van Hasselt, H., and Sutton, R. S. (2014). Weighted importance sampling for off-policy learning with linear function approximation. *Advances in Neural Information Processing Systems 27*.

Marbach, P., Tsitsiklis, J. N. (2001). Simulation-based optimization of Markov reward processes. *IEEE Transactions on Automatic Control 46*(2), 191–209. Also MIT Technical Report LIDS-P-2411 (1998).

Markey, K. L. (1994). Efficient learning of multiple degree-of-freedom control problems with quasi-independent Q-agents. In M. C. Mozer, P. Smolensky, D. S. Touretzky, J. L. Elman, and A. S. Weigend (eds.), *Proceedings of the 1990 Connectionist Models Summer School*. Erlbaum, Hillsdale, NJ.

Markram, H., Lübke, J., Frotscher, M., and Sakmann, B. (1997). Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs. *Science*, 275:213–215.

Martínez, J. F. and İpek, E. (2009). Dynamic multicore resource management: A machine learning approach. *Micro, IEEE*, 29(5):8–17.

Mataric, M. J. (1994). Reward functions for accelerated learning. In *Machine Learning: Proceedings of the Eleventh international conference*, pages 181–189.

Matsuda, W., Furuta, T., Nakamura, K. C., Hioki, H., Fujiyama, F., Arai, R., and Kaneko, T. (2009). Single nigrostriatal

dopaminergic neurons form widely spread and highly dense axonal arborizations in the neostriatum. *The Journal of Neuroscience*, 29(2):444–453.

Mazur, J. E. (1994). *Learning and Behavior*, 3rd ed. Prentice-Hall, Englewood Cliffs, NJ.

McCallum, A. K. (1993). Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 190–196. Morgan Kaufmann, San Mateo, CA.

McCallum, A. K. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. Ph.D. thesis, University of Rochester, Rochester, NY.

McCulloch, W. S., and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics 5*(4):115–133.

McMahan, H. B., Gordon, G. J. (2005). Fast Exact Planning in Markov Decision Processes. In *Proceedings of the International Conference on Automated Planning and Scheduling* (pp. 151-160).

Melo, F. S., Meyn, S. P., Ribeiro, M. I. (2008). An analysis of reinforcement learning with function approximation. In *Proceedings of the 25th international conference on Machine learning* (pp. 664–671).

Mendel, J. M. (1966). A survey of learning control systems. *ISA Transactions*, 5:297–303.

Mendel, J. M., McLaren, R. W. (1970). Reinforcement learning control and pattern recognition systems. In J. M. Mendel and K. S. Fu (eds.), *Adaptive, Learning and Pattern Recognition Systems: Theory and Applications*, pp. 287–318. Academic Press, New York.

Michie, D. (1961). Trial and error. In S. A. Barnett and A. McLaren (eds.), *Science Survey, Part 2*, pp. 129–145. Penguin, Harmondsworth.

Michie, D. (1963). Experiments on the mechanisation of game learning. 1. characterization of the model and its parameters. *Computer Journal*, 1:232–263.

Michie, D. (1974). *On Machine Intelligence*. Edinburgh University Press, Edinburgh.

Michie, D., Chambers, R. A. (1968). BOXES: An experiment in adaptive control. In E. Dale and D. Michie (eds.), *Machine Intelligence 2*, pp. 137–152. Oliver and Boyd, Edinburgh.

Miller, R. (1981). *Meaning and Purpose in the Intact Brain: A Philosophical, Psychological, and Biological Account of Conscious Process*. Clarendon Press, Oxford.

Miller, W. T., An, E., Glanz, F., Carter, M. (1990). The design of CMAC neural networks for control. *Adaptive and Learning Systems 1*:140–145.

Miller, W. T., Glanz, F. H. (1996). *UNH_CMAC verison 2.1: The University of New Hampshire Implementation of the Cerebellar Model Arithmetic Computer - CMAC*. Robotics Laboratory Technical Report, University of New Hampshire, Durham, New Hampshire.

Miller, S., Williams, R. J. (1992). Learning to control a bioreactor using a neural net Dyna-Q system. In *Proceedings of the Seventh Yale Workshop on Adaptive and Learning Systems*, pp. 167–172. Center for Systems Science, Dunham Laboratory, Yale University, New Haven.

Miller, W. T., Scalera, S. M., Kim, A. (1994). Neural network control of dynamic balance for a biped walking robot. In *Proceedings of the Eighth Yale Workshop on Adaptive and Learning Systems*, pp. 156–161. Center for Systems Science, Dunham Laboratory, Yale University, New Haven.

Minsky, M. L. (1954). *Theory of Neural-Analog Reinforcement Systems and Its Application to the Brain-Model Problem*. Ph.D. thesis, Princeton University.

Minsky, M. L. (1961). Steps toward artificial intelligence. *Proceedings of the Institute of Radio Engineers*, 49:8–30. Reprinted in E. A. Feigenbaum and J. Feldman (eds.), *Computers and Thought*, pp. 406–450. McGraw-Hill, New York, 1963.

Minsky, M. L. (1967). *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, NJ.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Modayil, J., and Sutton, R. S. (2014). Prediction driven behavior: Learning predictions that drive fixed responses. In *AAAI-14 Workshop on Artificial Intelligence and Robotics*, Quebec City, Canada.

Modayil, J., White, A., and Sutton, R. S. (2014). Multi-timescale nexting in a reinforcement learning robot. *Adaptive Behavior*, 22(2):146–160.

Montague, P. R., Dayan, P., Nowlan, S. J., Pouget, A., and Sejnowski, T. J. (1992). Using aperiodic reinforcement for directed self-organization during development. In *Advances in neural information processing systems 5*, pages 969–976.

Montague, P. R., Dayan, P., Person, C., and Sejnowski, T. J. (1995). Bee foraging in uncertain environments using predictive hebbian learning. *Nature*, 377(6551):725–728.

Montague, P. R., Dayan, P., Sejnowski, T. J. (1996). A framework for mesencephalic dopamine systems based on predictive Hebbian learning. *Journal of Neuroscience*, 16:1936–1947.

Montague, P. R., Dolan, R. J., Friston, K. J., and Dayan, P. (2012). Computational psychiatry. *Trends in Cognitive Sciences 16*(1):72–80.

Montague, P. R. and Sejnowski, T. J. (1994). The predictive brain: Temporal coincidence and temporal order in synaptic learningmechanisms. *Learning & Memory*, 1:1–33.

Moore, A. W. (1990). *Efficient Memory-Based Learning for Robot Control*. Ph.D. thesis, University of Cambridge.

Moore, A. W. (1994). The parti-game algorithm for variable resolution reinforcement learning in multidimensional spaces. In J. D. Cohen, G. Tesauro and J. Alspector (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 1993 Conference*, pp. 711–718. Morgan Kaufmann, San Francisco.

Moore, A. W., Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130.

Moore, A. W., Schneider, J., and Deng, K. (1997). Efficient locally weighted polynomial regression predictions. In *Proceedings of the 1997 International Machine Learning Conference*. Morgan Kaufmann.

Moore, J. W. and Blazis, D. E. J. (1989). Simulation of a classically conditioned response: A cerebellar implementation of

the sutton-barto-desmond model. In Byrne, J. H. and Berry, W. O., editors, *Neural Models of Plasticity*, pages 187–207. Academic Press, San Diego, CA.

Moore, J. W., Choi, J.-S., and Brunzell, D. H. (1998). Predictive timing under temporal uncertainty: The time derivative model of the conditioned response. In Rosenbaum, D. A. and Collyer, C. E., editors, *Timing of Behavior*, pages 3–34. MIT Press, Cambridge, MA.

Moore, J. W., Desmond, J. E., Berthier, N. E., Blazis, E. J., Sutton, R. S., and Barto, A. G. (1986). Simulation of the classically conditioned nictitating membrane response by a neuron-like adaptive element: I. Response topography, neuronal firing, and interstimulus intervals. *Behavioural Brain Research*, 21:143–154.

Moore, J. W., Marks, J. S., Castagna, V. E., and Polewan, R. J. (2001). Parameter stability in the TD model of complex CR topographies. Society for Neuroscience Abstract 642.2.

Moore, J. W. and Schmajuk, N. A. (2008). Kamin blocking. *Scholarpedia*, 3(5):3542.

Moore, J. W. and Stickney, K. J. (1980). Formation of attentional-associative networks in real time:Role of the hippocampus and implications for conditioning. *Physiological Psychology*, 8(2):207–217.

Mukundan, J. and Martínez, J. F. (2012). MORSE: Multi-objective reconfigurable self-optimizing memory scheduler. In *IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12.

Müller, M. (2002). Computer Go. *Artificial Intelligence*, 134(1):145–179.

Munos, R., Stepleton, T., Harutyunyan, A., and Bellemare, M. (2016). Safe and efficient off-policy reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 1046–1054.

Naddaf, Y. (2010). *Game-independent AI agents for playing Atari 2600 console games*. PhD thesis, University of Alberta.

Narendra, K. S., Thathachar, M. A. L. (1974). Learning automata—A survey. *IEEE Transactions on Systems, Man, and Cybernetics*, 4:323–334.

Narendra, K. S., Thathachar, M. A. L. (1989). *Learning Automata: An Introduction*. Prentice-Hall, Englewood Cliffs, NJ.

Narendra, K. S. and Wheeler, R. M. (1983). An n-player sequential stochastic game with identical payoffs. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:1154–1158.

Narendra, K. S., Wheeler, R. M. (1986). Decentralized learning in finite Markov chains. *IEEE Transactions on Automatic Control*, AC31(6):519–526.

Nedić, A., Bertsekas, D. P. (2003). Least squares policy evaluation algorithms with linear function approximation. *Discrete Event Dynamic Systems 13*(1-2):79–110.

Ng, A. Y. (2003). *Shaping and policy search in reinforcement learning*. PhD thesis, University of California, Berkeley, Berkeley, CA.

Ng, A. Y., Harada, D., and Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In Bratko, I. and Dzeroski, S., editors, *Proceedings of the Sixteenth International Conference on Machine Learning (ICML 1999)*, volume 99, pp. 278–287.

Ng, A. Y., and Russell, S. J. (2000). Algorithms for inverse reinforcement learning. In *International Conference on Machine Learning*, pp. 663–670.

Nie, J., Haykin, S. (1996). A dynamic channel assignment policy through Q-learning. CRL Report 334. Communications Research Laboratory, McMaster University, Hamilton, Ontario.

Niv, Y. (2009). Reinforcement learning in the brain. *Journal of Mathematical Psychology*, 53(3):139–154.

Niv, Y., Daw, N. D., and Dayan, P. (2005). How fast to work: Response vigor, motivation and tonic dopamine. In Yeiss, Y., Schölkopft, B., and Platt, J., editors, *Advances in Neural Information Processing Systems 18 (NIPS 2005)*, pages 1019–1026. MIT Press, Cambridge, MA.

Niv, Y., Daw, N. D., Joel, D., and Dayan, P. (2007). Tonic dopamine: opportunity costs and the control of response vigor. *Psychopharmacology*, 191(3):507–520.

Niv, Y., Joel, D., and Dayan, P. (2006). A normative perspective on motivation. *Trends in Cognitive Sciences*, 10(8):375–381.

Nowé, A., Vrancx, P., and Hauwere, Y.-M. D. (2012). Game theory and multi-agent reinforcement learning. In Wiering, M. and van Otterlo, M., editors, *Reinforcement Learning: State-of-the-Art*, pages 441–467. Springer-Verlag, Berlin.

Nutt, D. J., Lingford-Hughes, A., Erritzoe, D., and Stokes, P. R. A. (2015). The dopamine theory of addiction: 40 years of highs and lows. *Nature Reviews Neuroscience*, 16:305–312.

O'Doherty, J. P., Dayan, P., Friston, K., Critchley, H., and Dolan, R. J. (2003). Temporal difference models and reward-related learning in the human brain. *Neuron*, 38(2):329–337.

O'Doherty, J. P., Dayan, P., Schultz, J., Deichmann, R., Friston, K., and Dolan, R. J. (2004). Dissociable roles of ventral and dorsal striatum in instrumental conditioning. *Science*, 304(5669):452–454.

Ólafsdóttir, H. F., Barry, C., Saleem, A. B., Hassabis, D., and Spiers, H. J. (2015). Hippocampal place cells construct reward related sequences through unexplored space. *Elife*, 4:e06063.

Oh, J., Guo, X., Lee, H., Lewis, R. L., and Singh, S. (2015). Action-conditional video prediction using deep networks in Atari games. In *Advances in Neural Information Processing Systems*, pages 2845–2853.

Olds, J. and Milner, P. (1954). Positive reinforcement produced by electrical stimulation of the septal area and other regions of rat brain. *Journal of Comparative and Physiological Psychology*, 47(6):419–427.

Oliehoek, F. A. (2012). Decentralized POMDPs. In Wiering and van Otterlo (Eds.) *Reinforcement Learning: State-of-the Art*, pp. 471–503. Springer Berlin Heidelberg.

O'Reilly, R. C. and Frank, M. J. (2006). Making working memory work: A computational model of learning in the prefrontal cortex and basal ganglia. *Neural Computation*, 18(2):283–328.

O'Reilly, R. C., Frank, M. J., Hazy, T. E., and Watz, B. (2007). PVLV: the primary value and learned value Pavlovian learning algorithm. *Behavioral neuroscience*, 121(1):31–49.

Omohundro, S. M. (1987). Efficient algorithms with neural network behavior. Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign.

Orenstein, J. A. (1982). Multidimensional tries used for associative searching. *Information Processing Letters 14*(4):150–157.

Ormoneit, D., and Sen, Ś. (2002). Kernel-based reinforcement learning. *Machine learning 49*(2–3):161–178.

Oudeyer, P.-Y. and Kaplan, F. (2007). What is intrinsic motivation? A typology of computational approaches. *Frontiers in Neurorobotics*, 1.

Oudeyer, P.-Y., Kaplan, F., and Hafner, V. V. (2007). Intrinsic motivation systems for autonomous mental development. *IEEE Transactions on Evolutionary Computation*, 11(2):265–286.

Padoa-Schioppa, C., and Assad, J. A. (2006). Neurons in the orbitofrontal cortex encode economic value. *Nature 441*(7090):223–226.

Page, C. V. (1977). Heuristics for signature table analysis as a pattern recognition technique. *IEEE Transactions on Systems, Man, and Cybernetics*, 7:77–86.

Pagnoni, G., Zink, C. F., Montague, P. R., and Berns, G. S. (2002). Activity in human ventral striatum locked to errors of reward prediction. *Nature neuroscience*, 5(2):97–98.

Pan, W.-X., Schmidt, R., Wickens, J. R., and Hyland, B. I. (2005). Dopamine cells respond to predicted events during classical conditioning: Evidence for eligibility traces in the reward-learning network. *The Journal of Neuroscience*, 25(26):6235–6242.

Park, J., Kim, J., Kang, D. (2005). An RLS-based natural actor–critic algorithm for locomotion of a two-linked robot arm. *Computational Intelligence and Security*, 65–72.

Parker, D. B. (1985). *Learning Logic*. ???

Parks, P. C., Militzer, J. (1991). Improved allocation of weights for associative memory storage in learning control systems. *IFAC Design Methods of Control Systems*, Zurich, Switzerland, 507–512.

Parr, R., Russell, S. (1995). Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1088–1094. Morgan Kaufmann.

Pavlov, P. I. (1927). *Conditioned Reflexes*. Oxford University Press, London.

Pawlak, V. and Kerr, J. N. D. (2008). Dopamine receptor activation is required for corticostriatal spike-timing-dependent plasticity. *The Journal of Neuroscience*, 28(10):2435–2446.

Pawlak, V., Wickens, J. R., Kirkwood, A., and Kerr, J. N. D. (2010). Timing is not everything: neuromodulation opens the STDP gate. *Frontiers in synaptic neuroscience*, 2.

Pearce, J. M. and Hall, G. (1980). A model for Pavlovian learning: Variation in the effectiveness of conditioning but not unconditioned stimuli. *Psychological Review*, 87(6):532–552.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, MA.

Pearl, J. (1995). Causal diagrams for empirical research. *Biometrika*, 82(4), 669-688.

Pecevski, D., Maass, W., and Legenstein, R. A. (2007). Theoretical analysis of learning with reward-modulated spike-timing-dependent plasticity. In *Advances in Neural Information Processing Systems*, pp. 881–888.

Peng, J. (1993). *Efficient Dynamic Programming-Based Learning for Control*. Ph.D. thesis, Northeastern University, Boston.

Peng, J. (1995). Efficient memory-based dynamic programming. In *12th International Conference on Machine Learning*, pp. 438–446.

Peng, J., Williams, R. J. (1993). Efficient learning and planning within the Dyna framework. *Adaptive Behavior*, 1(4):437–454.

Peng, J., Williams, R. J. (1994). Incremental multi-step Q-learning. In W. W. Cohen and H. Hirsh (eds.), *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 226–232. Morgan Kaufmann, San Francisco.

Peng, J., Williams, R. J. (1996). Incremental multi-step Q-learning. *Machine Learning*, 22:283–290.

Perkins, T. J., Pendrith, M. D. (2002). On the existence of fixed points for Q-learning and Sarsa in partially observable domains. In *Proceedings of the International Conference on Machine Learning*, pp. 490–497.

Perkins, T. J., Precup, D. (2003). A convergent form of approximate policy iteration. In *Advances in neural information processing systems, proceedings of the 2002 conference*, pp. 1595–1602.

Peters, J. and Büchel, C. (2010). Neural representations of subjective reward value. *Behavioral brain research*, 213(2):135–141.

Peters, J., Schaal, S. (2008). Natural actor–critic. *Neurocomputing 71*(7), 1180–1190.

Peters, J., Vijayakumar, S., Schaal, S. (2005). Natural actor–critic. In *European Conference on Machine Learning* (pp. 280–291). Springer Berlin Heidelberg.

Peterson, G. B. (2004). A day of great illumination: B.F. Skinner's discovery of shaping. *Journal of the Experimental Analysis of Behavior*, 82(3):317–28.

Pezzulo, G., van der Meer, M. A. A., Lansink, C. S., and Pennartz, C. M. A. (2014). Internally generated sequences in learning and executing goal-directed behavior. *Trends in Cognitive Science*, 18(12):647–657.

Pfeiffer, B. E. and Foster, D. J. (2013). Hippocampal place-cell sequences depict future paths to remembered goals. *Nature*, 497(7447):74–79.

Phansalkar, V. V., Thathachar, M. A. L. (1995). Local and global optimization algorithms for generalized learning automata. *Neural Computation*, 7:950–973.

Poggio, T., Girosi, F. (1989). A theory of networks for approximation and learning. A.I. Memo 1140. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Poggio, T., Girosi, F. (1990). Regularization algorithms for learning that are equivalent to multilayer networks. *Science*, 247:978–982.

Polyak, B. T. (1990). New stochastic approximation type procedures. *Automat. i Telemekh 7*(98-107), 2 (in Russian).

Polyak, B. T., Juditsky, A. B. (1992). Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization 30*(4), 838-855.

Powell, M. J. D. (1987). Radial basis functions for multivariate interpolation: A review. In J. C. Mason and M. G. Cox (eds.), *Algorithms for Approximation*, pp. 143–167. Clarendon Press, Oxford.

Powell, W. B. (2011). *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, Second edition. John Wiley and Sons.

Powers, W. T. (1973). *Behavior: The Control of Perception*. Aldine de Gruyter, Chicago. 2nd expanded edition 2005.

Precup, D., Sutton, R. S., Dasgupta, S. (2001). Off-policy temporal-difference learning with function approximation. In *Proceedings of the 18th International Conference on Machine Learning*.

Precup, D., Sutton, R. S., Paduraru, C., Koop, A., and Singh, S. (2005). Off-policy learning with options and recognizers. In *Advances in Neural Processing Systems*, pp. 1097–1104.

Precup, D., Sutton, R. S., Singh, S. (2000). Eligibility traces for off-policy policy evaluation. In *Proceedings of the 17th International Conference on Machine Learning*, pp. 759–766. Morgan Kaufmann.

Puterman, M. L. (1994). *Markov Decision Problems*. Wiley, New York.

Puterman, M. L., Shin, M. C. (1978). Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24:1127–1137.

Quartz, S., Dayan, P., Montague, P. R., and Sejnowski, T. J. (1992). Expectation learning in the brain using diffuse ascending connections. In *Society for Neuroscience Abstracts*, volume 18, page 1210.

Randløv, J. and Alstrøm, P. (1998). Learning to drive a bicycle using reinforcement learning and shaping. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 463–471.

Rangel, A., Camerer, C., and Montague, P. R. (2008). A framework for studying the neurobiology of value-based decision making. *Nature Reviews Neuroscience*, 9(7):545–556.

Rangel, A. and Hare, T. (2010). Neural computations associated with goal-directed choice. *Current opinion in neurobiology*, 20(2):262–270.

Reddy, G., Celani, A., Sejnowski, T. J., and Vergassola, M. (2016). Learning to soar in turbulent environments. *Proceedings of the National Academy of Sciences*, 113(33):E4877–E4884.

Redgrave, P. and Gurney, K. (2006). The short-latency dopamine signal: a role in discovering novel actions? *Nature Reviews Neuroscience*, 7:967–975.

Redish, D. A. (2004). Addiction as a computational process gone awry. *Science*, 306(5703):1944–1947.

Reetz, D. (1977). Approximate solutions of a discounted Markovian decision process. *Bonner Mathematische Schriften*, 98:77–92.

Rescorla, R. A. and Wagner, A. R. (1972). A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement. In Black, A. H. and Prokasy, W. F., editors, *Classical Conditioning II*, pages 64–99. Appleton-Century-Crofts, New York.

Revusky, S. and Garcia, J. (1970). Learned associations over long delays. In Bower, G., editor, *The psychology of learning and motivation*, volume 4, pages 1–84. Academic Press, Inc., New York.

Reynolds, J. N. J. and Wickens, J. R. (2002). Dopamine-dependent plasticity of corticostriatal synapses. *Neural Networks*, 15(4):507–521.

Ring, M. B. (1994). *Continual Learning in Reinforcement Environments*. Ph.D. thesis, University of Texas, Austin.

Ripley, B. D. (2007). *Pattern Recognition and Neural Networks*. Cambridge University Press.

Rivest, R. L., Schapire, R. E. (1987). Diversity-based inference of finite automata. In *Proceedings of the Twenty-Eighth Annual Symposium on Foundations of Computer Science*, pp. 78–87. Computer Society Press of the IEEE, Washington, DC.

Rixner, S. (2004). Memory controller optimizations for web servers. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 355–366. IEEE Computer Society.

Robbins, H. (1952). Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58:527–535.

Robertie, B. (1992). Carbon versus silicon: Matching wits with TD-Gammon. *Inside Backgammon*, 2:14–22.

Roesch, M. R., Calu, D. J., and Schoenbaum, G. (2007). Dopamine neurons encode the better option in rats deciding between differently delayed or sized rewards. *Nature Neuroscience*, 10(12):1615–1624.

Romo, R. and Schultz, W. (1990). Dopamine neurons of the monkey midbrain: Contingencies of responses to active touch during self-initiated arm movements. *Journal of Neurophysiology*, 63(3):592–624.

Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, Washington, DC.

Ross, S. (1983). *Introduction to Stochastic Dynamic Programming*. Academic Press, New York.

Ross, T. (1933). Machines that think. *Scientific American*, pages 206–208.

Rubinstein, R. Y. (1981). *Simulation and the Monte Carlo Method*. Wiley, New York.

Rumelhart, D. E., Hinton, G. E., Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland (eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. I, *Foundations*. Bradford/MIT Press, Cambridge, MA.

Rummery, G. A. (1995). *Problem Solving with Reinforcement Learning*. Ph.D. thesis, Cambridge University.

Rummery, G. A., Niranjan, M. (1994). On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166. Engineering Department, Cambridge University.

Ruppert, D. (1988). Efficient estimations from a slowly convergent Robbins-Monro process. Cornell University Operations Research and Industrial Engineering Technical Report No. 781.

Russell, S., Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*, 3rd edition. Prentice-Hall, Englewood Cliffs, NJ.

Rust, J. (1996). Numerical dynamic programming in economics. In H. Amman, D. Kendrick, and J. Rust (eds.), *Handbook of Computational Economics*, pp. 614–722. Elsevier, Amsterdam.

Ryan, R. M. and Deci, E. L. (2000). Intrinsic and extrinsic motivations: Classic definitions and new directions. *Contemporary Educational Psychology*, 25(1):54–67.

Saddoris, M. P., Cacciapaglia, F., Wightmman, R. M., and Carelli, R. M. (2015). Differential dopamine release dynamics in the nucleus accumbens core and shell reveal complementary signals for error prediction and incentive motivation. *The Journal of Neuroscience*, 35(33):11572–11582.

Saksida, L. M., Raymond, S. M., and Touretzky, D. S. (1997). Shaping robot behavior using principles from instrumental

conditioning. *Robotics and Autonomous Systems*, 22(3):231–249.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3:211–229. Reprinted in E. A. Feigenbaum and J. Feldman (eds.), *Computers and Thought*, pp. 71–105. McGraw-Hill, New York, 1963.

Samuel, A. L. (1967). Some studies in machine learning using the game of checkers. II—Recent progress. *IBM Journal on Research and Development*, 11:601–617.

Schaal, S., and Atkeson, C. G. (1994). Robot juggling: Implementation of memory-based learning. *IEEE Control Systems 14*(1):57–71.

Schmajuk, N. A. (2008). Computational models of classical conditioning. *Scholarpedia*, 3(3):1664.

Schmidhuber, J. (1991a). Adaptive confidence and adaptive curiosity. Technical Report FKI-149-91, Institut für Informatik, Technische Universität München, Arcisstr. 21, 800 München 2, Germany.

Schmidhuber, J. (1991b). A possibility for implementing curiosity and boredom in model-building neural controllers. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 222–227, Cambridge, MA. MIT Press.

Schmidhuber, J. (2009). Driven by compression progress: A simple principle explains essential aspects of subjective beauty, novelty, surprise, interestingness, attention, curiosity, creativity, art, science, music, jokes. In Pezzulo, G., Butz, M. V., Sigaud, O., and Baldassarre, G., editors, *Anticipatory Behavior in Adaptive Learning Systems. From Psychological Theories to Artificial Cognitive Systems*, pages 48–76. Springer, Berlin.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks 61*, 85–117.

Schmidhuber, J., Storck, J., and Hochreiter, S. (1994). Reinforcement driven information acquisition in nondeterministic environments. Technical report, Fakultät für Informatik, Technische Universität München, München, Germany.

Schultz, D. G., Melsa, J. L. (1967). *State Functions and Linear Control Systems*. McGraw-Hill, New York.

Schultz, W. (1998). Predictive reward signal of dopamine neurons. *Journal of Neurophysiology*, 80:1–27.

Schultz, W., Apicella, P., and Ljungberg, T. (1993). Responses of monkey dopamine neurons to reward and conditioned stimuli during successive steps of learning a delayed response task. *Journal of Neuroscience 13*(3):900–913.

Schultz, W., Dayan, P., Montague, P. R. (1997). A neural substrate of prediction and reward. *Science*, 275:1593–1598.

Schultz, W. and Romo, R. (1990). Dopamine neurons of the monkey midbrain: contingencies of responses to stimuli eliciting immediate behavioral reactions. *Journal of Neurophysiology*, 63(3):607–624.

Schultz, W., Romo, R., Ljungberg, T., Mirenowicz, J., Hollerman, J. R., and Dickinson, A. (1995). Reward-related signals carried by dopamine neurons. In Houk, Davis, and Beiser (Eds.) *Models of Information Processing in the Basal Ganglia*, pp. 233–248. MIT Press Cambridge MA.

Schumaker, L. L. (1976). *Fitting Surfaces to Scattered Data*. University of Texas at Austin, Dept. of Mathematics.

Schwartz, A. (1993). A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 298–305. Morgan Kaufmann, San Mateo, CA.

Schweitzer, P. J., Seidmann, A. (1985). Generalized polynomial approximations in Markovian decision processes. *Journal of Mathematical Analysis and Applications*, 110:568–582.

Selfridge, O. G. (1978). Tracking and trailing: Adaptation in movement strategies. Technical report, Bolt Beranek and Newman, Inc. Unpublished report.

Selfridge, O. G. (1984). Some themes and primitives in ill-defined systems. In Selfridge, O. G., Rissland, E. L., and Arbib, M. A., editors, *Adaptive Control of Ill-Defined Systems*, pages 21–26. Plenum Press, NY. Proceedings of the NATO Advanced Research Institute on Adaptive Control of Ill-defined Systems, NATO Conference Series II, Systems Science, Vol. 16.

Selfridge, O. J., Sutton, R. S., Barto, A. G. (1985). Training and tracking in robotics. In A. Joshi (ed.), *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 670–672. Morgan Kaufmann, San Mateo, CA.

Seo, H., Barraclough, D., and Lee, D. (2007). Dynamic signals related to choices and outcomes in the dorsolateral prefrontal cortex. *Cerebral Cortex*, 17(suppl 1):110–117.

Seung, H. S. (2003). Learning in spiking neural networks by reinforcement of stochastic synaptic transmission. *Neuron*, 40(6):1063–1073.

Shah, A. (2012). Psychological and neuroscientific connections with reinforcement learning. In Wiering, M. and van Otterlo, M., editors, *Reinforcement Learning: State of the Art*, pages 507–537. Springer-Verlag, Berlin.

Shannon, C. E. (1950). Programming a computer for playing chess. *Philosophical Magazine*, 41:256–275.

Shannon, C. E. (1951). Presentation of a maze-solving machine. In Forester, H. V., editor, *Cybernetics. Transactions of the Eighth Conference*, pages 173–180. Josiah Macy Jr. Foundation.

Shannon, C. E. (1952). "Theseus" maze-solving mouse. http://cyberneticzoo.com/mazesolvers/1952--theseus-maze-solving-mouse--claude-shannon-american/.

Shelton, C. R. (2001). *Importance Sampling for Reinforcement Learning with Multiple Objectives*. PhD thesis, Massachusetts Institute of Technology.

Shepard, D. (1968). A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 23rd ACM National Conference*, pp. 517–524. ACM.

Sherman, J., Morrison, W. J. (1949). Adjustment of an inverse matrix corresponding to changes in the elements of a given column or a given row of the original matrix (abstract). *Annals of Mathematical Statistics 20*:621.

Shewchuk, J., Dean, T. (1990). Towards learning time-varying functions with high input dimensionality. In *Proceedings of the Fifth IEEE International Symposium on Intelligent Control*, pp. 383–388. IEEE Computer Society Press, Los Alamitos, CA.

Shimansky, Y. P. (2009). Biologically plausible learning in neural networks: a lesson from bacterial chemotaxis. *Biological Cybernetics*, 101(5-6):379–385.

Si, J., Barto, A., Powell, W., Wunsch, D. (Eds.). (2004). *Handbook of learning and approximate dynamic programming*. John Wiley and Sons.

Silver, D. (2009). *Reinforcement learning and simulation based search in the game of Go*. University of Alberta Doctoral dissertation.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning* (ICML-14) (pp. 387–395).

Şimşek, Ö., Algórta, S., and Kothiyal, A. (2016). Why most decisions are easy in tetris-and perhaps in other sequential decision problems, as well. *Proceedings of 33rd International Conference on Machine Learning*.

Singh, S. P. (1992a). Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 202–207. AAAI/MIT Press, Menlo Park, CA.

Singh, S. P. (1992b). Scaling reinforcement learning algorithms by learning variable temporal resolution models. In *Proceedings of the Ninth International Machine Learning Conference*, pp. 406–415. Morgan Kaufmann, San Mateo, CA.

Singh, S. P. (1993). *Learning to Solve Markovian Decision Processes*. Ph.D. thesis, University of Massachusetts, Amherst. Appeared as CMPSCI Technical Report 93-77.

Singh, S. P. (Ed.) (2002). Special double issue on reinforcement learning, *Machine Learning 49*(2/3).

Singh, S., Barto, A. G., and Chentanez, N. (2005). Intrinsically motivated reinforcement learning. In *Advances in Neural Information Processing Systems 17: Proceedings of the 2004 Conference*, pages 1281–1288, Cambridge MA. MIT Press.

Singh, S. P., Bertsekas, D. (1997). Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems: Proceedings of the 1996 Conference*, pp. 974–980. MIT Press, Cambridge, MA.

Singh, S. P., Jaakkola, T., Jordan, M. I. (1994). Learning without state-estimation in partially observable Markovian decision problems. In W. W. Cohen and H. Hirsch (eds.), *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 284–292. Morgan Kaufmann, San Francisco.

Singh, S. P., Jaakkola, T., Jordan, M. I. (1995). Reinforcement learing with soft state aggregation. In G. Tesauro, D. S. Touretzky, T. Leen (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference*, pp. 359–368. MIT Press, Cambridge, MA.

Singh, S., Lewis, R. L., and Barto, A. G. (2009). Where do rewards come from? In Taatgen, N. and van Rijn, H., editors, *Proceedings of the 31st Annual Conference of the Cognitive Science Society*, pages 2601–2606. Cognitive Science Society.

Singh, S., Lewis, R. L., Barto, A. G., and Sorg, J. (2010). Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Transactions on Autonomous Mental Development*, 2(2):7082. Special issue on Active Learning and Intrinsically Motivated Exploration in Robots: Advances and Challenges.

Singh, S. P., Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158.

Sivarajan, K. N., McEliece, R. J., Ketchum, J. W. (1990). Dynamic channel assignment in cellular radio. In *Proceedings of the 40th Vehicular Technology Conference*, pp. 631–637.

Skinner, B. F. (1938). *The Behavior of Organisms: An Experimental Analysis*. Appleton-Century, New York.

Skinner, B. F. (1958). Reinforcement today. *American Psychologist*, 13(3):94–99.

Skinner, B. F. (1981). Selection by consequences. *Science 213*(4507):501–504.

Smith, K. S. and Greybiel, A. M. (2013). A dual operator view of habitual behavior reflecting cortical and striatal dynamics. *Neuron*, 79(2):361–374.

Sofge, D. A., White, D. A. (1992). Applied learning: Optimal control for manufacturing. In D. A. White and D. A. Sofge (eds.), *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pp. 259–281. Van Nostrand Reinhold, New York.

Sorg, J. D. (2011). *The Optimal Reward Problem:Designing Effective Reward for Bounded Agents*. PhD thesis, Computer Science and Engineering, The University of Michigan.

Sorg, J., Lewis, R. L., and Singh, S. P. (2010). Reward design via online gradient ascent. In *Advances in Neural Information Processing Systems*, pp. 2190–2198.

Sorg, J., Singh, S., and Lewis, R. (2010). Internal rewards mitigate agent boundedness. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 1007–1014.

Spaan, M. T. (2012). Partially observable Markov decision processes. In Wiering and van Otterlo (Eds.) *Reinforcement Learning: State-of-the Art*, pp. 387–414. Springer Berlin Heidelberg.

Spence, K. W. (1947). The role of secondary reinforcement in delayed reward learning. *Psychological Review*, 54(1):1–8.

Spong, M. W. (1994). Swing up control of the acrobot. In *Proceedings of the 1994 IEEE Conference on Robotics and Automation*, pp. 2356-2361. IEEE Computer Society Press, Los Alamitos, CA.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Staddon, J. E. R. (1983). *Adaptive Behavior and Learning*. Cambridge University Press, Cambridge.

Stanfill, C., and Waltz, D. (1986). Toward memory-based reasoning. *Communications of the ACM 29*(12):1213–1228.

Steinberg, E. E., Keiflin, R., Boivin, J. R., Witten, I. B., Deisseroth, K., and Janak, P. H. (2013). A causal link between prediction errors, dopamine neurons and learning. *Nature Neuroscience*, 16(7):966–973.

Sterling, P. and Laughlin, S. (2015). *Principles of Neural Design*. MIT Press, Cambridge, MA.

Storck, J., Hochreiter, S., and Schmidhuber, J. (1995). Reinforcement-driven information acquisition in non-deterministic environments. In *Proceedings of ICANN'95, Paris, France*, volume 2, pages 159–164.

Sugiyama, M., Hachiya, H., Morimura, T. (2013). *Statistical Reinforcement Learning: Modern Machine Learning Approaches*. Chapman & Hall/CRC.

Suri, R. E., Bargas, J., and Arbib, M. A. (2001). Modeling functions of striatal dopamine modulation in learning and planning.

Neuroscience, 103(1):65–85.

Suri, R. E. and Schultz, W. (1998). Learning of sequential movements by neural network model with dopamine-like reinforcement signal. *Experimental Brain Research*, 121(3):350–354.

Suri, R. E. and Schultz, W. (1999). A neural network model with dopamine-like reinforcement signal that learns a spatial delayed response task. *Neuroscience*, 91(3):871–890.

Sutton, R. S. (1978a). Learning theory support for a single channel theory of the brain. Unpublished report.

Sutton, R. S. (1978b). Single channel theory: A neuronal theory of learning. *Brain Theory Newsletter*, 4:72–75. Center for Systems Neuroscience, University of Massachusetts, Amherst, MA.

Sutton, R. S. (1978c). A unified theory of expectation in classical and instrumental conditioning. Bachelors thesis, Stanford University.

Sutton, R. S. (1984). *Temporal Credit Assignment in Reinforcement Learning.* Ph.D. thesis, University of Massachusetts, Amherst.

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pp. 216–224. Morgan Kaufmann, San Mateo, CA.

Sutton, R. S. (1991a). Dyna, an integrated architecture for learning, planning, and reacting. *SIGART Bulletin*, 2:160–163. ACM Press.

Sutton, R. S. (1991b). Planning by incremental dynamic programming. In L. A. Birnbaum and G. C. Collins (eds.), *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 353–357. Morgan Kaufmann, San Mateo, CA.

Sutton, R. S. (Ed.) (1992). *Reinforcement Learning*. Kluwer Academic Press. Reprinting of a special double issue on reinforcement learning, *Machine Learning 8*(3/4).

Sutton, R. S. (1995a). TD models: Modeling the world at a mixture of time scales. In A. Prieditis and S. Russell (eds.), *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 531–539. Morgan Kaufmann, San Francisco.

Sutton, R. S. (1995). On the virtues of linear learning and trajectory distributions. *Proceedings of the Workshop on Value Function Approximation* at the International Conference on Machine Learning.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. S. Touretzky, M. C. Mozer and M. E. Hasselmo (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pp. 1038–1044. MIT Press, Cambridge, MA.

Sutton, R. S. (2009). The grand challenge of predictive empirical abstract knowledge. *Working Notes of the IJCAI-09 Workshop on Grand Challenges for Reasoning from Experiences.*

Sutton, R. S. (2015a) Introduction to reinforcement learning with function approximation. Tutorial at the Conference on Neural Information Processing Systems, Montreal, December 7, 2015.

Sutton, R. S. (2015b) True online Emphatic TD(λ): Quick reference and implementation guide. ArXiv:1507.07147. Code is available in Python and C++ by downloading the source files of this arXiv paper as a zip archive.

Sutton, R. S., Barto, A. G. (1981a). Toward a modern theory of adaptive networks: Expectation and prediction. *Psychological Review*, 88:135–170.

Sutton, R. S., Barto, A. G. (1981b). An adaptive network that constructs and uses an internal model of its world. *Cognition and Brain Theory*, 3:217–246.

Sutton, R. S., Barto, A. G. (1987). A temporal-difference model of classical conditioning. In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, pp. 355-378. Erlbaum, Hillsdale, NJ.

Sutton, R. S., Barto, A. G. (1990). Time-derivative models of Pavlovian reinforcement. In M. Gabriel and J. Moore (eds.), *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pp. 497–537. MIT Press, Cambridge, MA.

Sutton, R. S., Maei, H. R., Precup, D., Bhatnagar, S., Silver, D., Szepesvári, Cs., and Wiewiora, E. (2009). Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 993–1000. ACM.

Sutton, R. S., Maei, H. R., and Szepesvári, Cs. (2009). A convergent $O(d^2)$ temporal-difference algorithm for off-policy learning with linear function approximation. In *Advances in Neural Information Processing Systems*, pp. 1609–1616.

Sutton, R. S., Mahmood, A. R., Precup, D., van Hasselt, H. (2014). A new Q(λ) with interim forward view and Monte Carlo equivalence. *International Conference on Machine Learning 31. JMLR W&CP 32*(2).

Sutton, R. S., Mahmood, A. R., White, M. (2016). An emphatic approach to the problem of off-policy temporal-difference learning. *Journal of Machine Learning Research 17*(73):1-29.

Sutton, R. S., McAllester, D. A., Singh, S. P., Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 99*, pp. 1057–1063.

Sutton, R. S., Modayil, J., Delp, M., Degris, T., Pilarski, P. M., White, A., Precup, D. (2011). Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *Proceedings of the Tenth International Conference on Autonomous Agents and Multiagent Systems*, pp. 761–768, Taipei, Taiwan.

Sutton, R. S., Pinette, B. (1985). The learning of world models by connectionist networks. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, pp. 54–64.

Sutton, R. S., Singh, S. (1994). On bias and step size in temporal-difference learning. In *Proceedings of the Eighth Yale Workshop on Adaptive and Learning Systems*, pp. 91–96. Center for Systems Science, Dunham Laboratory, Yale University, New Haven.

Sutton, R. S., Whitehead, D. S. (1993). Online learning with random representations. In *Proceedings of the Tenth International Machine Learning Conference*, pp. 314–321. Morgan Kaufmann, San Mateo, CA.

Szepesvári, C. (2010). Algorithms for reinforcement learning. Synthesis Lectures on Artificial Intelligence and Machine Learning 4(1), 1–103.

Szita, I. (2012). Reinforcement learning in games. In *Reinforcement Learning* (pp. 539-577). Springer Berlin Heidelberg.

Tadepalli, P., Ok, D. (1994). H-learning: A reinforcement learning method to optimize undiscounted average reward. Technical Report 94-30-01. Oregon State University, Computer Science Department, Corvallis.

Tadepalli, P., and Ok, D. (1996). Scaling up average reward reinforcement learning by approximating the domain models and the value function. In *International Conference on Machine Learning*, pp. 471–479.

Takahashi, Y., Schoenbaum, G., and Niv, Y. (2008). Silencing the critics: understanding the effects of cocaine sensitization on dorsolateral and ventral striatum in the context of an actor/critic model. *Frontiers in Neuroscience*, 2(1):86–99.

Tan, M. (1991). Learning a cost-sensitive internal representation for reinforcement learning. In L. A. Birnbaum and G. C. Collins (eds.), *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 358–362. Morgan Kaufmann, San Mateo, CA.

Tan, M. (1993). Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 330–337. Morgan Kaufmann, San Mateo, CA.

Taylor, G., and Parr, R. (2009). Kernelized value function approximation for reinforcement learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 1017–1024. ACM.

Taylor, M. E., and Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research 10*:1633–1685.

Tesauro, G. J. (1986). Simple neural models of classical conditioning. *Biological Cybernetics*, 55:187–200.

Tesauro, G. J. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8:257–277.

Tesauro, G. J. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219.

Tesauro, G. J. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38:58–68.

Tesauro, G. (2002). Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1):181–199.

Tesauro, G. J., Galperin, G. R. (1997). On-line policy improvement using Monte-Carlo search. In *Advances in Neural Information Processing Systems: Proceedings of the 1996 Conference*, pp. 1068–1074. MIT Press, Cambridge, MA.

Tesauro, G., Gondek, D. C., Lechner, J., Fan, J., and Prager, J. M. (2012). Simulation, learning, and optimization techniques in watson's game strategies. *IBM Journal of Research and Development*, 56(3.4):16–1–16–11.

Tesauro, G., Gondek, D. C., Lenchner, J., Fan, J., and Prager, J. M. (2013). Analysis of Watson's strategies for playing Jeopardy! *Journal of Artificial Intelligence Research*, 21:205–251.

Tham, C. K. (1994). *Modular On-Line Function Approximation for Scaling up Reinforcement Learning*. PhD thesis, Cambridge University.

Thathachar, M. A. L. and Sastry, P. S. (1985). A new approach to the design of reinforcement schemes for learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15:168–175.

Thathachar, M. and Sastry, P. S. (2002). Varieties of learning automata: an overview. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 36(6):711–722.

Thathachar, M. and Sastry, P. S. (2011). *Networks of learning automata: Techniques for online stochastic optimization*. Springer Science & Business Media.

Theocharous, G., Thomas, P. S., and Ghavamzadeh, M. (2015). Personalized ad recommendation for life-time value optimization guarantees. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI-15)*.

Thistlethwaite, D. (1951). A critical review of latent learning and related experiments. *Psychological Bulletin*, 48(2):97–129.

Thomas, P. (2014). Bias in natural actor–critic algorithms. *International Conference on Machine Learning 31. JMLR W&CP 32*(1):441–448.

Thomas, P. S. (2015). *Safe Reinforcement Learning*. PhD thesis, University of Massachusetts Amherst.

Thomas, P. S., Theocharous, G., and Ghavamzadeh, M. (2015). High-confidence off-policy evaluation. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 3000–3006. The AAAI Press, Palo Alto, CA.

Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25:285–294.

Thompson, W. R. (1934). On the theory of apportionment. *American Journal of Mathematics*, 57:450–457.

Thorndike, E. L. (1898). Animal intelligence: An experimental study of the associative processes in animals. *The Psychological Review, Series of Monograph Supplements*, II(4).

Thorndike, E. L. (1911). *Animal Intelligence*. Hafner, Darien, CT.

Thorp, E. O. (1966). *Beat the Dealer: A Winning Strategy for the Game of Twenty-One*. Random House, New York.

Tian, T. (in preparation) An Empirical Study of Sliding-Step Methods in Temporal Difference Learning. University of Alberta MSc thesis.

Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop. COURSERA: Neural networks for machine learning.

Tobler, P. N., Fiorillo, C. D., and Schultz, W. (2005). Adaptive coding of reward value by dopamine neurons. *Science*, 307(5715):1642–1645.

Tolman, E. C. (1932). *Purposive Behavior in Animals and Men*. Century, New York.

Tolman, E. C. (1948). Cognitive maps in rats and men. *Psychological Review*, 55(4):189–208.

Tsai, H.-S., Zhang, F., Adamantidis, A., Stuber, G. D., Bonci, A., de Lecea, L., and Deisseroth, K. (2009). Phasic firing in dopaminergic neurons is sufficient for behavioral conditioning. *Science*, 324(5930):1080–1084.

Tsetlin, M. L. (1973). *Automaton Theory and Modeling of Biological Systems*. Academic Press, New York.

Tsitsiklis, J. N. (1994). Asynchronous stochastic approximation and Q-learning. *Machine Learning*, 16:185–202.

Tsitsiklis, J. N. (2002). On the convergence of optimistic policy iteration. *Journal of Machine Learning Research*, 3:59–72.

Tsitsiklis, J. N. and Van Roy, B. (1996). Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94.

Tsitsiklis, J. N., Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42:674–690.

Tsitsiklis, J. N., Van Roy, B. (1999). Average cost temporal-difference learning. *Automatica*, 35:1799–1808.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind* 433–460.

Turing, A. M. (1948). Intelligent Machinery, A Heretical Theory. *The Turing Test: Verbal Behavior as the Hallmark of Intelligence*, 105.

Ungar, L. H. (1990). A bioreactor benchmark for adaptive network-based process control. In W. T. Miller, R. S. Sutton, and P. J. Werbos (eds.), *Neural Networks for Control*, pp. 387–402. MIT Press, Cambridge, MA.

Urbanczik, R. and Senn, W. (2009). Reinforcement learning in populations of spiking neurons. *Nature neuroscience*, 12(3):250–252.

Urbanowicz, R. J., Moore, J. H. (2009). Learning classifier systems: A complete introduction, review, and roadmap. *Journal of Artificial Evolution and Applications*.

Valentin, V. V., Dickinson, A., and O'Doherty, J. P. (2007). Determining the neural substrates of goal-directed learning in the human brain. *The Journal of Neuroscience*, 27(15):4019–4026.

van Hasselt, H. (2010). Double Q-learning. In *Advances in Neural Information Processing Systems*, pp. 2613–2621.

van Hasselt, H. (2011). *Insights in Reinforcement Learning: Formal Analysis and Empircal Evaluation of Temporal-difference Learning*. SIKS dissertation series number 2011-04.

van Hasselt, H. (2012). Reinforcement learning in continuous state and action spaces. In Wiering and van Otterlo (Eds.) *Reinforcement Learning: State-of-the Art*, pp. 207–251. Springer Berlin Heidelberg.

van Hasselt, H., and Sutton, R. S. (2015). Learning to predict independent of span. ArXiv 1508.04582.

van Otterlo, M. (2009). *The Logic of Adaptive Behavior*. IOS Press.

van Otterlo, M. (2012). Solving relational and first-order logical markov decision processes: A survey. In Wiering and van Otterlo (Eds.) *Reinforcement Learning: State-of-the Art*, pp. 253–292. Springer Berlin Heidelberg.

Van Roy, B., Bertsekas, D. P., Lee, Y., Tsitsiklis, J. N. (1997). A neuro-dynamic programming approach to retailer inventory management. In *Proceedings of the 36th IEEE Conference on Decision and Control*, Vol. 4, pp. 4052–4057.

van Seijen, H. (2016). Effective multi-step temporal-difference learning for non-linear function approximation. arXiv preprint arXiv:1608.05151.

van Seijen, H., and Sutton, R. S. (2014). True online TD(λ). In *Proceedings of the 31st International Conference on Machine Learning*. JMLR W&CP 32(1):692–700.

Van Seijen, H., Mahmood, A. R., Pilarski, P. M., Machado, M. C., and Sutton, R. S. (2016). True online temporal-difference learning. *Journal of Machine Learning Research 17*(145), 1–40.

van Seijen, H., Van Hasselt, H., Whiteson, S., Wiering, M. (2009). A theoretical and empirical analysis of Expected Sarsa. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pp. 177–184.

Varga, R. S. (1962). *Matrix Iterative Analysis*. Englewood Cliffs, NJ: Prentice-Hall.

Vasilaki, E., Frémaux, N., Urbanczik, R., Senn, W., and Gerstner, W. (2009). Spike-based reinforcement learning in continuous state and action space: when policy gradient methods fail. *PLoS Computational Biology*, 5(12).

Viswanathan, R. and Narendra, K. S. (1974). Games of stochastic automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 4:131–135.

Vlassis, N., Ghavamzadeh, M., Mannor, S., and Poupart, P. (2012). Bayesian reinforcement learning. In Wiering and van Otterlo (Eds.) *Reinforcement Learning: State-of-the Art*, pp. 359–386. Springer Berlin Heidelberg.

Walter, W. G. (1950). An imitation of life. *Scientific American*, pages 42–45.

Walter, W. G. (1951). A machine that learns. *Scientific American*, 185(2):60–63.

Waltz, M. D., Fu, K. S. (1965). A heuristic approach to reinforcement learning control systems. *IEEE Transactions on Automatic Control*, 10:390–398.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. Ph.D. thesis, Cambridge University.

Watkins, C. J. C. H., Dayan, P. (1992). Q-learning. *Machine Learning*, 8:279–292.

Wiering, M., Van Otterlo, M. (2012). *Reinforcement Learning*. Springer Berlin Heidelberg.

Werbos, P. (1974). Beyond regression: New tools for prediction and analysis in the behavioral sciences. Phd Thesis, Harvard University, Cambridge, Massachusetts.

Werbos, P. J. (1977). Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook*, 22:25–38.

Werbos, P. J. (1982). Applications of advances in nonlinear sensitivity analysis. In R. F. Drenick and F. Kozin (eds.), *System Modeling and Optimization*, pp. 762–770. Springer-Verlag, Berlin.

Werbos, P. J. (1987). Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics*, 17:7–20.

Werbos, P. J. (1988). Generalization of back propagation with applications to a recurrent gas market model. *Neural Networks*, 1:339–356.

Werbos, P. J. (1989). Neural networks for control and system identification. In *Proceedings of the 28th Conference on Decision and Control*, pp. 260–265. IEEE Control Systems Society.

Werbos, P. J. (1990). Consistency of HDP applied to a simple reinforcement learning problem. *Neural Networks*, 3:179–189.

Werbos, P. J. (1992). Approximate dynamic programming for real-time control and neural modeling. In D. A. White and D. A. Sofge (eds.), *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pp. 493–525. Van Nostrand Reinhold, New York.

Werbos, P. J. (1994). *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting* (Vol. 1). John Wiley and Sons.

White, A. (2015). *Developing a Predictive Approach to Knowledge*. Phd thesis, University of Alberta.

White, D. J. (1969). *Dynamic Programming*. Holden-Day, San Francisco.

White, D. J. (1985). Real applications of Markov decision processes. *Interfaces*, 15:73–83.

White, D. J. (1988). Further real applications of Markov decision processes. *Interfaces*, 18:55–61.

White, D. J. (1993). A survey of applications of Markov decision processes. *Journal of the Operational Research Society*, 44:1073–1096.

White, A., and White, M. (2016). Investigating practical linear temporal difference learning. In *Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems*, pp. 494–502.

Whitehead, S. D., Ballard, D. H. (1991). Learning to perceive and act by trial and error. *Machine Learning*, 7:45–83.

Whitt, W. (1978). Approximations of dynamic programs I. *Mathematics of Operations Research*, 3:231–243.

Whittle, P. (1982). *Optimization over Time*, vol. 1. Wiley, New York.

Whittle, P. (1983). *Optimization over Time*, vol. 2. Wiley, New York.

Wickens, J. and Kötter, R. (1995). Cellular models of reinforcement. In Houk, J. C., Davis, J. L., and Beiser, D. G., editors, *Models of Information Processing in the Basal Ganglia*, pages 187–214. MIT Press, Cambridge, MA.

Widrow, B., Gupta, N. K., Maitra, S. (1973). Punish/reward: Learning with a critic in adaptive threshold systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 3:455–465.

Widrow, B., Hoff, M. E. (1960). Adaptive switching circuits. In *1960 WESCON Convention Record Part IV*, pp. 96–104. Institute of Radio Engineers, New York. Reprinted in J. A. Anderson and E. Rosenfeld, *Neurocomputing: Foundations of Research*, pp. 126–134. MIT Press, Cambridge, MA, 1988.

Widrow, B., Smith, F. W. (1964). Pattern-recognizing control systems. In J. T. Tou and R. H. Wilcox (eds.), *Computer and Information Sciences*, pp. 288–317. Spartan, Washington, DC.

Widrow, B., Stearns, S. D. (1985). *Adaptive Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ.

Wiewiora, E. (2003). Potential-based shaping and Q-value initialization are equivalent. *Journal of Artificial Intelligence Research 19*:205–208.

Williams, R. J. (1986). Reinforcement learning in connectionist networks: A mathematical analysis. Technical Report ICS 8605. Institute for Cognitive Science, University of California at San Diego, La Jolla.

Williams, R. J. (1987). Reinforcement-learning connectionist systems. Technical Report NU-CCS-87-3. College of Computer Science, Northeastern University, Boston.

Williams, R. J. (1988). On the use of backpropagation in associative reinforcement learning. In *Proceedings of the IEEE International Conference on Neural Networks*, pp. I263–I270. IEEE San Diego section and IEEE TAB Neural Network Committee.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256.

Williams, R. J., Baird, L. C. (1990). A mathematical analysis of actor–critic architectures for learning optimal controls through incremental dynamic programming. In *Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems*, pp. 96–101. Center for Systems Science, Dunham Laboratory, Yale University, New Haven.

Wilson, R. C., Takahashi, Y. K., Schoenbaum, G., and Niv, Y. (2014). Orbitofrontal cortex as a cognitive map of task space. *Neuron*, 81(2):267–279.

Wilson, S. W. (1994). ZCS: A zeroth order classifier system. *Evolutionary Computation*, 2:1–18.

Wise, R. A. (2004). Dopamine, learning, and motivation. *Nature Reviews Neuroscience*, 5(6):1–12.

Witten, I. H. (1976). The apparent conflict between estimation and control—A survey of the two-armed problem. *Journal of the Franklin Institute*, 301:161–189.

Witten, I. H. (1977). An adaptive optimal controller for discrete-time Markov environments. *Information and Control*, 34:286–295.

Witten, I. H., Corbin, M. J. (1973). Human operators and automatic adaptive controllers: A comparative study on a particular control task. *International Journal of Man–Machine Studies*, 5:75–104.

Woodbury, T., Dunn, C., and Valasek, J. (2014). Autonomous soaring using reinforcement learning for trajectory generation. In *52nd Aerospace Sciences Meeting*, page 0990.

Woodworth, R. S., Schlosberg, H. (1938). *Experimental psychology*. New York: Henry Holt and Company.

Xie, X. and Seung, H. S. (2004). Learning in neural networks by reinforcement of irregular spiking. *Physical Review E*, 69(4).

Xu, X., Xie, T., Hu, D., and Lu, X. (2005). Kernel least-squares temporal difference learning. *International Journal of Information Technology 11*(9):54–63.

Yagishita, S., Hayashi-Takagi, A., Ellis-Davies, G. C. R., Urakubo, H., Ishii, S., and Kasai, H. (2014). A critical time window for dopamine actions on the structural plasticity of dendritic spines. *Science*, 345(6204):1616–1619.

Yee, R. C., Saxena, S., Utgoff, P. E., Barto, A. G. (1990). Explaining temporal differences to create useful concepts for evaluating states. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 882–888. AAAI Press, Menlo Park, CA.

Yin, H. H. and Knowlton, B. J. (2006). The role of the basal ganglia in habit formation. *Nature Reviews Neuroscience*, 7(6):464–476.

Young, P. (1984). *Recursive Estimation and Time-Series Analysis*. Springer-Verlag, Berlin.

Yu, H. (2010). Convergence of least squares temporal difference methods under general conditions. *International Conference on Machine Learning 27*, pp. 1207–1214.

Yu, H. (2012). Least squares temporal difference methods: An analysis under general conditions. *SIAM Journal on Control and Optimization*, 50(6), 3310–3343.

Yu, H. (2015a). On convergence of emphatic temporal-difference learning. ArXiv:1506.02582. A shorter version appeared in *Conference on Learning Theory 18*, *JMLR W&CP 40*.

Yu, H. (2015b). Weak convergence properties of constrained emphatic temporal-difference learning with constant and slowly diminishing stepsize. ArXiv:1511.07471.

Zhang, M., Yum, T. P. (1989). Comparisons of channel-assignment strategies in cellular mobile telephone systems. *IEEE Transactions on Vehicular Technology*, 38:211-215.

Zhang, W. (1996). *Reinforcement Learning for Job-shop Scheduling*. Ph.D. thesis, Oregon State University. Technical Report CS-96-30-1.

Zhang, W., Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1114–1120. Morgan Kaufmann.

Zhang, W., Dietterich, T. G. (1996). High-performance job-shop scheduling with a time–delay TD(λ) network. In D. S. Touretzky, M. C. Mozer, M. E. Hasselmo (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pp. 1024–1030. MIT Press, Cambridge, MA.

Zweben, M., Daun, B., Deale, M. (1994). Scheduling and rescheduling with iterative repair. In M. Zweben and M. S. Fox (eds.), *Intelligent Scheduling*, pp. 241–255. Morgan Kaufmann, San Francisco.