

S E C O N D EDITION

Approximate Dynamic Programming

Solving the Curses of Dimensionality

Warren B. Powell



Wiley Series in Probability and Statistics

 WILEY

WWW.
LINK AVAILABLE

Approximate Dynamic Programming

WILEY SERIES IN PROBABILITY AND STATISTICS

Established by WALTER A. SHEWHART and SAMUEL S. WILKS

Editors: *David J. Balding, Noel A. C. Cressie, Garrett M. Fitzmaurice,
Harvey Goldstein, Iain M. Johnstone, Geert Molenberghs, David W. Scott,
Adrian F. M. Smith, Ruey S. Tsay, Sanford Weisberg*

Editors Emeriti: *Vic Barnett, J. Stuart Hunter, Joseph B. Kadane, Jozef L. Teugels*

A complete list of the titles in this series appears at the end of this volume.

Approximate Dynamic Programming

Solving the Curses of Dimensionality

Second Edition

Warren B. Powell

Princeton University

The Department of Operations Research and Financial Engineering
Princeton, NJ



A JOHN WILEY & SONS, INC., PUBLICATION

Copyright © 2011 by John Wiley & Sons, Inc. All rights reserved

Published by John Wiley & Sons, Inc., Hoboken, New Jersey
Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permission>.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data:

Powell, Warren B., 1955–

Approximate dynamic programming : solving the curses of dimensionality / Warren B. Powell.
– 2nd ed.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-470-60445-8 (cloth)

1. Dynamic programming. I. Title.

T57.83.P76 2011

519.7'03–dc22

2010047227

Printed in the United States of America

eBook ISBN: 978-1-118-02917-6

ePDF ISBN: 978-1-118-02915-2

ePub ISBN: 978-1-118-02916-9

10 9 8 7 6 5 4 3 2 1

Contents

Preface to the Second Edition	xi
Preface to the First Edition	xv
Acknowledgments	xvii
1 The Challenges of Dynamic Programming	1
1.1 A Dynamic Programming Example: A Shortest Path Problem, 2	
1.2 The Three Curses of Dimensionality, 3	
1.3 Some Real Applications, 6	
1.4 Problem Classes, 11	
1.5 The Many Dialects of Dynamic Programming, 15	
1.6 What Is New in This Book?, 17	
1.7 Pedagogy, 19	
1.8 Bibliographic Notes, 22	
2 Some Illustrative Models	25
2.1 Deterministic Problems, 26	
2.2 Stochastic Problems, 31	
2.3 Information Acquisition Problems, 47	
2.4 A Simple Modeling Framework for Dynamic Programs, 50	
2.5 Bibliographic Notes, 54	
Problems, 54	
3 Introduction to Markov Decision Processes	57
3.1 The Optimality Equations, 58	
3.2 Finite Horizon Problems, 65	

3.3	Infinite Horizon Problems,	66
3.4	Value Iteration,	68
3.5	Policy Iteration,	74
3.6	Hybrid Value-Policy Iteration,	75
3.7	Average Reward Dynamic Programming,	76
3.8	The Linear Programming Method for Dynamic Programs,	77
3.9	Monotone Policies*,	78
3.10	Why Does It Work?**,	84
3.11	Bibliographic Notes, Problems,	103
4	Introduction to Approximate Dynamic Programming	111
4.1	The Three Curses of Dimensionality (Revisited),	112
4.2	The Basic Idea,	114
4.3	<i>Q</i> -Learning and SARSA,	122
4.4	Real-Time Dynamic Programming,	126
4.5	Approximate Value Iteration,	127
4.6	The Post-Decision State Variable,	129
4.7	Low-Dimensional Representations of Value Functions,	144
4.8	So Just What Is Approximate Dynamic Programming?,	146
4.9	Experimental Issues,	149
4.10	But Does It Work?,	155
4.11	Bibliographic Notes, Problems,	156
5	Modeling Dynamic Programs	167
5.1	Notational Style,	169
5.2	Modeling Time,	170
5.3	Modeling Resources,	174
5.4	The States of Our System,	178
5.5	Modeling Decisions,	187
5.6	The Exogenous Information Process,	189
5.7	The Transition Function,	198
5.8	The Objective Function,	206
5.9	A Measure-Theoretic View of Information**,	211
5.10	Bibliographic Notes, Problems,	213

6 Policies	221
6.1 Myopic Policies, 224	
6.2 Lookahead Policies, 224	
6.3 Policy Function Approximations, 232	
6.4 Value Function Approximations, 235	
6.5 Hybrid Strategies, 239	
6.6 Randomized Policies, 242	
6.7 How to Choose a Policy?, 244	
6.8 Bibliographic Notes, 247	
Problems, 247	
7 Policy Search	249
7.1 Background, 250	
7.2 Gradient Search, 253	
7.3 Direct Policy Search for Finite Alternatives, 256	
7.4 The Knowledge Gradient Algorithm for Discrete Alternatives, 262	
7.5 Simulation Optimization, 270	
7.6 Why Does It Work?**, 274	
7.7 Bibliographic Notes, 285	
Problems, 286	
8 Approximating Value Functions	289
8.1 Lookup Tables and Aggregation, 290	
8.2 Parametric Models, 304	
8.3 Regression Variations, 314	
8.4 Nonparametric Models, 316	
8.5 Approximations and the Curse of Dimensionality, 325	
8.6 Why Does It Work?**, 328	
8.7 Bibliographic Notes, 333	
Problems, 334	
9 Learning Value Function Approximations	337
9.1 Sampling the Value of a Policy, 337	
9.2 Stochastic Approximation Methods, 347	
9.3 Recursive Least Squares for Linear Models, 349	
9.4 Temporal Difference Learning with a Linear Model, 356	
9.5 Bellman's Equation Using a Linear Model, 358	

9.6	Analysis of TD(0), LSTD, and LSPE Using a Single State,	364
9.7	Gradient-Based Methods for Approximate Value Iteration*,	366
9.8	Least Squares Temporal Differencing with Kernel Regression*,	371
9.9	Value Function Approximations Based on Bayesian Learning*,	373
9.10	Why Does It Work*,	376
9.11	Bibliographic Notes, Problems,	379 381
10	Optimizing While Learning	383
10.1	Overview of Algorithmic Strategies,	385
10.2	Approximate Value Iteration and <i>Q</i> -Learning Using Lookup Tables,	386
10.3	Statistical Bias in the Max Operator,	397
10.4	Approximate Value Iteration and <i>Q</i> -Learning Using Linear Models,	400
10.5	Approximate Policy Iteration,	402
10.6	The Actor–Critic Paradigm,	408
10.7	Policy Gradient Methods,	410
10.8	The Linear Programming Method Using Basis Functions,	411
10.9	Approximate Policy Iteration Using Kernel Regression*,	413
10.10	Finite Horizon Approximations for Steady-State Applications,	415
10.11	Bibliographic Notes, Problems,	416 418
11	Adaptive Estimation and Stepsizes	419
11.1	Learning Algorithms and Stepsizes,	420
11.2	Deterministic Stepsize Recipes,	425
11.3	Stochastic Stepsizes,	433
11.4	Optimal Stepsizes for Nonstationary Time Series,	437
11.5	Optimal Stepsizes for Approximate Value Iteration,	447
11.6	Convergence,	449
11.7	Guidelines for Choosing Stepsize Formulas,	451
11.8	Bibliographic Notes, Problems,	452 453

12 Exploration Versus Exploitation	457
12.1 A Learning Exercise: The Nomadic Trucker,	457
12.2 An Introduction to Learning,	460
12.3 Heuristic Learning Policies,	464
12.4 Gittins Indexes for Online Learning,	470
12.5 The Knowledge Gradient Policy,	477
12.6 Learning with a Physical State,	482
12.7 Bibliographic Notes,	492
Problems,	493
13 Value Function Approximations for Resource Allocation Problems	497
13.1 Value Functions versus Gradients,	498
13.2 Linear Approximations,	499
13.3 Piecewise-Linear Approximations,	501
13.4 Solving a Resource Allocation Problem Using Piecewise-Linear Functions,	505
13.5 The SHAPE Algorithm,	509
13.6 Regression Methods,	513
13.7 Cutting Planes*,	516
13.8 Why Does It Work?**,	528
13.9 Bibliographic Notes,	535
Problems,	536
14 Dynamic Resource Allocation Problems	541
14.1 An Asset Acquisition Problem,	541
14.2 The Blood Management Problem,	547
14.3 A Portfolio Optimization Problem,	557
14.4 A General Resource Allocation Problem,	560
14.5 A Fleet Management Problem,	573
14.6 A Driver Management Problem,	580
14.7 Bibliographic Notes,	585
Problems,	586
15 Implementation Challenges	593
15.1 Will ADP Work for Your Problem?,	593
15.2 Designing an ADP Algorithm for Complex Problems,	594
15.3 Debugging an ADP Algorithm,	596

15.4	Practical Issues,	597
15.5	Modeling Your Problem,	602
15.6	Online versus Offline Models,	604
15.7	If It Works, Patent It!,	606
Bibliography		607
Index		623

Preface to the Second Edition

The writing for the first edition of this book ended around 2005, followed by a year of editing before it was submitted to the publisher in 2006. As with everyone who works in this very rich field, my understanding of the models and algorithms was strongly shaped by the projects I had worked on. While I was very proud of the large industrial applications that were the basis of my success, at the time I had a very limited understanding of many other important problem classes that help to shape the algorithms that have evolved (and continue to evolve) in this field.

In the five years that passed before this second edition went to the publisher, my understanding of the field and my breadth of applications have grown dramatically. Reflecting my own personal growth, I realized that the book needed a fundamental restructuring along several dimensions. I came to appreciate that approximate dynamic programming is much more than approximating value functions. After writing an article that included a list of nine types of policies, I realized that every policy I had encountered could be broken down into four fundamental classes: myopic policies, lookahead policies, policy function approximations, and policies based on value function approximations. Many other policies can be created by combining these four fundamental classes into different types of hybrids.

I also realized that methods for approximating functions (whether they be policy function approximations or value function approximations) could be usefully organized into three fundamental strategies: lookup tables, parametric models, and nonparametric models. Of course, these can also be combined in different forms.

In preparing the second edition, I came to realize that the nature of the decision variable plays a critical role in the design of an algorithm. In the first edition, one of my goals was to create a bridge between dynamic programming (which tended to focus on small action spaces) and math programming, with its appreciation of vector-valued decisions. As a result I had adopted x as my generic decision variable. In preparing the new edition, I had come to realize that small action spaces cover a very important class of problems, and these are also the problems that a beginner is most likely to start with to learn the field. Also action “ a ” pervades the reinforcement learning community (along with portions of the operations research community), to the point that it is truly part of the language. As a result the second edition now uses action “ a ” for most of its presentation, but reverts to x specifically

for problems where the decisions are continuous and/or (more frequently) vectors. The challenges of vector-valued decisions has been largely overlooked in the reinforcement learning community, while the operations research community that works on these problems has largely ignored the power of dynamic programming.

The second edition now includes a new chapter (Chapter 6) devoted purely to a discussion of different types of policies, a summary of some hybrid strategies, and a discussion of problems that are well suited to each of the different strategies. This is followed by a chapter (Chapter 7) that focuses purely on the issue of policy search. This chapter brings together fields such as stochastic search and simulation optimization. The chapter also introduces a new class of optimal learning strategies based on the concept of the knowledge gradient, an idea that was developed originally to address the exploration–exploitation problem before realizing that it had many other applications.

I also acquired a much better understanding of the different methods for approximating value functions. I found that the best way to communicate the rich set of strategies that have evolved was to divide the material into three chapters. The first of these (Chapter 8) focuses purely on different statistical procedures for approximating value functions. While this can be viewed partly as a tutorial into statistics and machine learning, the focus is on strategies that have been used in the approximate dynamic programming/reinforcement learning literature. ADP imposes special demands on statistical learning algorithms, including the importance of recursive estimation, and the need to start with a small number of observations (which works better with a low-dimensional model) and transition to a larger number of observations with models that are high-dimensional in certain regions. Next, Chapter 9 summarizes different methods for estimating the value of being in a state using sample information, with the goal of estimating the value function for a fixed policy. Since I have found that a number of papers focus on a single policy without making this apparent, this chapter makes this very explicit by indexing variables that depend on a policy with a superscript π . Finally, Chapter 10 addresses the very difficult problem of estimating the value of being in a state while simultaneously optimizing over policies.

Chapter 11 of this book is a refined version of the old Chapter 6, which focused on stepsize rules. Chapter 11 is streamlined, with a new discussion of the implications of algorithms based on policy iteration (including least squares policy evaluation (LSPE), least squares temporal differences) and algorithms based on approximate value iteration and Q -learning. Following some recent research, I use the setting of a single state to develop a much clearer understanding of the demands on a stepsize that are placed by these different algorithmic strategy. A new section has been added introducing a stepsize rule that is specifically optimized for approximate value iteration.

Chapter 12, on the famous exploration–exploitation problem in approximate dynamic programming, has been heavily revised to reflect a much more thorough understanding of the general field that is coming to be known as optimal learning. This chapter includes a recently developed method for doing active learning in the presence of a physical state, by way of the concept of the knowledge gradient.

While this method looks promising, the general area of doing active learning in the context of dynamic programs (with a physical state) is an active area of research at the time of this writing.

A major theme of the first edition was to bridge the gap between disciplines, primarily reinforcement learning (computer science), simulation, and math programming (operations research). This edition reinforces this theme first by adopting more broadly the notation and vocabulary of reinforcement learning (which has made most of the contributions to this field) while retaining the bridge to math programming, but now also including stochastic search and simulation optimization (primarily in the context of policy search).

The mathematical level of the book continues to require only an understanding of statistics and probability. A goal of the first edition was that the material would be accessible to an advanced undergraduate audience. With this second edition a more accurate description would be that the material is accessible to a highly motivated and well prepared undergraduate, but the breadth of the material is more suitable to a graduate audience.

*Princeton, New Jersey
October 2010*

WARREN B. POWELL

Preface to the First Edition

The path to completing this book began in the early 1980s when I first started working on dynamic models arising in the management of fleets of vehicles for the truckload motor carrier industry. It is often said that necessity is the mother of invention, and as with many of my colleagues in this field, the methods that emerged evolved out of a need to solve a problem. The initially ad hoc models and algorithms I developed to solve these complex industrial problems evolved into a sophisticated set of tools supported by an elegant theory within a field that is increasingly being referred to as *approximate dynamic programming*.

The methods in this book reflect the original motivating applications. I started with elegant models for which academic is so famous, but my work with industry revealed the need to handle a number of complicating factors that were beyond the scope of these models. One of these was a desire from one company to understand the effect of uncertainty on operations, requiring the ability to solve these large-scale optimization problems in the presence of various forms of randomness (but most notably customer demands). This question launched what became a multiple-decade search for a modeling and algorithmic strategy that would provide practical, but high-quality, solutions.

This process of discovery took me through multiple fields, including linear and nonlinear programming, Markov decision processes, optimal control, and stochastic programming. It is somewhat ironic that the framework of Markov decision processes, which originally appeared to be limited to toy problems (three trucks moving between five cities), turned out to provide the critical theoretical framework for solving truly industrial-strength problems (thousands of drivers moving between hundreds of locations, each described by complex vectors of attributes).

The ability to solve these problems required the integration of four major disciplines: dynamic programming (Markov decision processes), math programming (linear, nonlinear and integer programming), simulation, and statistics. My desire to bring together the fields of dynamic programming and math programming motivated some fundamental notational choices (in particular, the use of x as a decision variable). In this book there is as a result a heavy dependence on the Monte Carlo methods so widely used in simulation, but a knowledgeable reader will quickly see how much is missing. The book covers in some depth a number of important

techniques from statistics, but even this presentation only scratches the surface of tools and concepts available from fields such as nonparametric statistics, signal processing and approximation theory.

Audience

The book is aimed primarily at an advanced undergraduate/masters audience with no prior background in dynamic programming. The presentation does expect a first course in probability and statistics. Some topics require an introductory course in linear programming. A major goal of the book is the clear and precise presentation of dynamic problems, which means there is an emphasis on modeling and notation.

The body of every chapter focuses on models and algorithms with a minimum of the mathematical formalism that so often makes presentations of dynamic programs inaccessible to a broader audience. Using numerous examples, each chapter emphasizes the presentation of algorithms that can be directly applied to a variety of applications. The book contains dozens of algorithms that are intended to serve as a starting point in the design of practical solutions for real problems. Material for more advanced graduate students (with measure-theoretic training and an interest in theory) is contained in sections marked with **.

The book can be used quite effectively in a graduate level course. Several chapters include “Why does it work” sections at the end that present proofs at an advanced level (these are all marked with **). This material can be easily integrated into the teaching of the material within the chapter.

Approximate dynamic programming is also a field that has emerged from several disciplines. I have tried to expose the reader to the many dialects of ADP, reflecting its origins in artificial intelligence, control theory, and operations research. In addition to the diversity of words and phrases that mean the same thing—but often with different connotations—I have had to make difficult notational choices.

I have found that different communities offer unique insights into different dimensions of the problem. In the main, the control theory community has the most thorough understanding of the meaning of a state variable. The artificial intelligence community has the most experience with deeply nested problems (which require numerous steps before earning a reward). The operations research community has evolved a set of tools that are well suited for high-dimensional resource allocation, contributing both math programming and a culture of careful modeling.

W. B. P.

Acknowledgments

The work in this book reflects the contributions of many. Perhaps most important are the problems that motivated the development of this material. This work would not have been possible without the corporate sponsors who posed these problems in the first place. I would like to give special recognition to Schneider National, the largest truckload carrier in the United States, Yellow Freight System, the largest less-than-truckload carrier, and Norfolk Southern Railroad, one of the four major railroads that serves the United States. These three companies not only posed difficult problems, they provided years of research funding that allowed me to work on the development of tools that became the foundation of this book. This work would never have progressed without the thousands of hours of my two senior professional staff members, Hugo Simão and Belgacem Bouzaiène-Ayari, who have written hundreds of thousands of lines of code to solve industrial-strength problems. It is their efforts working with our corporate sponsors that brought out the richness of real applications, and therefore the capabilities that our tools needed to possess.

While industrial sponsors provided the problems, without the participation of my graduate students, I would simply have a set of ad hoc procedures. It is the work of my graduate students that provided most of the fundamental insights and algorithms, and virtually all of the convergence proofs. In the order in which they joined by research program, the students are Linos Frantzeskakis, Raymond Cheung, Tassio Carvalho, Zhi-Long Chen, Greg Godfrey, Joel Shapiro, Mike Spivey, Huseyin Topaloglu, Katerina Papadaki, Arun Marar, Tony Wu, Abraham George, Juliana Nascimento, Peter Frazier, and Ilya Ryzhov, all of whom are my current and former students and have contributed directly to the material presented in this book. My undergraduate senior thesis advisees provided many colorful applications of dynamic programming, and they contributed their experiences with their computational work.

The presentation has benefited from numerous conversations with professionals in this community. I am particularly grateful to Erhan Çinlar, who taught me the language of stochastic processes that played a fundamental role in guiding my notation in the modeling of information. I am also grateful for many conversations with Ben van Roy, Dimitri Bertsekas, Andy Barto, Mike Fu, Dan Adelman, Lei Zhao, and Diego Klabjan. I would also like to thank Paul Werbos at NSF

for introducing me to the wonderful neural net community in IEEE, which contributed what for me was a fresh perspective on dynamic problems. Jennie Si, Don Wunsch, George Lendaris and Frank Lewis all helped educate me in the language and concepts of the control theory community.

For the second edition of the book, I would like to add special thanks to Peter Frazier and Ilya Ryzhov, who contributed the research on the knowledge gradient for optimal learning in ADP, and improvements in my presentation of Gittins indices. The research of Jun Ma on convergence theory for approximate policy iteration for continuous states and actions contributed to my understanding in a significant way. This edition also benefited from the contributions of Warren Scott, Lauren Hannah, and Emre Barut who have combined to improve my understanding of nonparametric statistics.

This research was first funded by the National Science Foundation, but the bulk of my research in this book was funded by the Air Force Office of Scientific Research, and I am particularly grateful to Dr. Neal Glassman for his support through the early years. The second edition has enjoyed continued support from AFOSR by Donald Hearn, and I appreciate Don's dedication to the AFOSR program.

Many people have assisted with the editing of this volume through numerous comments. Mary Fan, Tamas Papp, and Hugo Simão all read various drafts of the first edition cover to cover. I would like to express my appreciation to Boris Defourny for an exceptionally thorough proofreading of the second edition. Diego Klabjan and his dynamic programming classes at the University of Illinois provided numerous comments and corrections. Special thanks are due to the students in my own undergraduate and graduate dynamic programming classes who had to survive the very early versions of the text. The second edition of the book benefited from the many comments of my graduate students, and my ORF 569 graduate seminar on approximate dynamic programming. Based on their efforts, many hundreds of corrections have been made, though I am sure that new errors have been introduced. I appreciate the patience of the readers who understand that this is the price of putting in textbook form material that is evolving so quickly.

Of course, the preparation of this book required tremendous patience from my wife Shari and my children Elyse and Danny, who had to tolerate my ever-present laptop at home. Without their support, this project could never have been completed.

W.B.P.

C H A P T E R 1

The Challenges of Dynamic Programming

The optimization of problems over time arises in many settings, ranging from the control of heating systems to managing entire economies. In between are examples including landing aircraft, purchasing new equipment, managing blood inventories, scheduling fleets of vehicles, selling assets, investing money in portfolios, and just playing a game of tic-tac-toe or backgammon. These problems involve making decisions, then observing information, after which we make more decisions, and then more information, and so on. Known as *sequential decision problems*, they can be straightforward (if subtle) to formulate, but solving them is another matter.

Dynamic programming has its roots in several fields. Engineering and economics tend to focus on problems with continuous states and decisions (these communities refer to decisions as controls), which might be quantities such as location, speed, and temperature. By contrast, the fields of operations research and artificial intelligence work primarily with discrete states and decisions (or actions). Problems that are modeled with continuous states and decisions (and typically in continuous time) are often addressed under the umbrella of “control theory,” whereas problems with discrete states and decisions, modeled in discrete time, are studied at length under the umbrella of “Markov decision processes.” Both of these subfields set up recursive equations that depend on the use of a state variable to capture history in a compact way. There are many high-dimensional problems such as those involving the allocation of resources that are generally studied using the tools of mathematical programming. Most of this work focuses on deterministic problems using tools such as linear, nonlinear, or integer programming, but there is a subfield known as stochastic programming that incorporates uncertainty. Our presentation spans all of these fields.

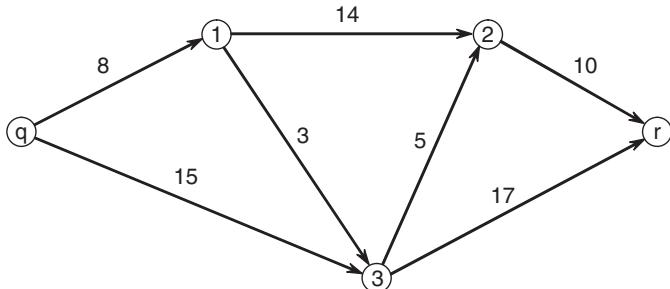


Figure 1.1 Illustration of a shortest path problem from origin q to destination r .

1.1 A DYNAMIC PROGRAMMING EXAMPLE: A SHORTEST PATH PROBLEM

Perhaps one of the best-known applications of dynamic programming is that faced by a driver choosing a path in a transportation network. For simplicity (and this is a real simplification for this application), we assume that the driver has to decide at each node (or intersection) which link to traverse next (we are not going to get into the challenges of left turns versus right turns). Let \mathcal{I} be the set of intersections. If the driver is at intersection i , he can go to a subset of intersections \mathcal{I}_i^+ at a cost c_{ij} . He starts at the origin node $q \in \mathcal{I}$ and has to find his way to the destination node $r \in \mathcal{I}$ at the least cost. An illustration is shown in Figure 1.1.

The problem can be easily solved using dynamic programming. Let

$$v_i = \text{Cost to get from intersection } i \in \mathcal{I} \text{ to the destination node } r.$$

We assume that $v_r = 0$. Initially we do not know v_i , and so we start by setting $v_i = M$, where “ M ” is known as “big M ” and represents a large number. We can solve the problem by iteratively computing

$$v_i \leftarrow \min \left\{ v_i, \min_{j \in \mathcal{I}^+} \{c_{ij} + v_j\} \right\} \quad \text{for all } i \in \mathcal{I}. \quad (1.1)$$

Equation (1.1) has to be solved iteratively, where at each iteration, we loop over all the nodes i in the network. We stop when none of the values v_i change. It should be noted that this is not a very efficient way of solving a shortest path problem. For example, in the early iterations it may well be the case that $v_j = M$ for all $j \in \mathcal{I}^+$. However, we use the method to illustrate dynamic programming.

Table 1.1 illustrates the algorithm, assuming that we always traverse the nodes in the order $(q, 1, 2, 3, r)$. Note that we handle node 2 before node 3, which is the reason why, even in the first pass, we learn that the path cost from node 3 to node r is 15 (rather than 17). We are done after iteration 3, but we require iteration 4 to verify that nothing has changed.

Shortest path problems arise in a variety of settings that have nothing to do with transportation or networks. Consider, for example, the challenge faced by a college

Table 1.1 Path cost from each node to node r after each node has been visited

Iteration	Cost from Node				
	q	1	2	3	r
1	100	100	100	100	0
2	100	100	10	15	0
3	30	18	10	15	0
4	26	18	10	15	0
	26	18	10	15	0

freshman trying to plan her schedule up to graduation. By graduation, she must take 32 courses overall, including eight departmentals, two math courses, one science course, and two language courses. We can describe the state of her academic program in terms of how many courses she has taken under each of these five categories. Let S_{tc} be the number of courses she has taken by the end of semester t in category $c = \{\text{Total courses, Departmentals, Math, Science, Language}\}$, and let $S_t = (S_{tc})_c$ be the state vector. Based on this state, she has to decide which courses to take in the next semester. To graduate, she has to reach the state $S_8 = (32, 8, 2, 1, 2)$. We assume that she has a measurable desirability for each course she takes, and that she would like to maximize the total desirability of all her courses.

The problem can be viewed as a shortest path problem from the state $S_0 = (0, 0, 0, 0, 0)$ to $S_8 = (32, 8, 2, 1, 2)$. Let S_t be her current state at the beginning of semester t , and let a_t represent the decisions she makes while determining what courses to take. We then assume we have access to a *transition function* $S^M(S_t, a_t)$, which tells us that if she is in state S_t and takes action a_t , she will land in state S_{t+1} , which we represent by simply using

$$S_{t+1} = S^M(S_t, a_t).$$

In our transportation problem, we would have $S_t = i$ if we are at intersection i , and a_t would be the decision to “go to j ,” leaving us in the state $S_{t+1} = j$.

Finally, let $C_t(S_t, a_t)$ be the contribution or reward she generates from being in state S_t and taking the action a_t . The value of being in state S_t is defined by the equation

$$V_t(S_t) = \max_{x_t} \{C_t(S_t, a_t) + V_{t+1}(S_{t+1})\} \quad \forall s_t \in \mathcal{S}_t,$$

where $S_{t+1} = S^M(S_t, a_t)$ and where \mathcal{S}_t is the set of all possible (discrete) states that she can be in at the beginning of the year.

1.2 THE THREE CURSES OF DIMENSIONALITY

All dynamic programs can be written in terms of a recursion that relates the value of being in a particular state at one point in time to the value of the states that we

are carried into at the next point in time. For deterministic problems this equation can be written

$$V_t(S_t) = \max_{a_t} (C_t(S_t, a_t) + V_{t+1}(S_{t+1})). \quad (1.2)$$

where S_{t+1} is the state we transition to if we are currently in state S_t and take action a_t . Equation (1.2) is known as Bellman's equation, or the Hamilton–Jacobi equation, or increasingly, the Hamilton–Jacobi–Bellman equation (HJB for short). Some textbooks (in control theory) refer to them as the “functional equation” of dynamic programming (or the “recurrence equation”). We primarily use the term “optimality equation” in our presentation, but often use the term “Bellman equation” because this is so widely used in the dynamic programming community.

Most of the problems that we address in this volume involve some form of uncertainty (prices, travel times, equipment failures, weather). For example, in a simple inventory problem we might have S_t DVD players in stock. We might then order a_t new DVD players, after which we satisfy a random demand \hat{D}_{t+1} that follows some probability distribution. The state variable would be described by the transition equation

$$S_{t+1} = \max\{0, S_t + a_t - \hat{D}_{t+1}\}.$$

Assume that $C_t(S_t, a_t)$ is the contribution we earn at time t , given by

$$C_t(S_t, a_t, \hat{D}_{t+1}) = p_t \min\{S_t + a_t, \hat{D}_{t+1}\} - c a_t.$$

To find the best decision, we need to maximize the contribution we receive from a_t plus the expected value of the state that we end up at (which is random). That means we need to solve

$$V_t(S_t) = \max_{a_t} \mathbb{E}\{C_t(S_t, a_t, \hat{D}_{t+1}) + V_{t+1}(S_{t+1})|S_t\}. \quad (1.3)$$

This problem is not too hard to solve. Assume that we know $V_{t+1}(S_{t+1})$ for each state S_{t+1} . We just have to compute (1.3) for each value of S_t , which then gives us $V_t(S_t)$. We can keep stepping backward in time to compute all the value functions.

For the vast majority of problems the state of the system is a vector. For example, if we have to track the inventory of N different products, where we might have 0, 1, ..., $M-1$ units of inventory of each product, then we would have M^N different states. As we can see, the size of the state space grows very quickly as the number of dimensions grows. This is the widely known “curse of dimensionality” of dynamic programming and is the most often-cited reason why dynamic programming cannot be used.

In fact, there are many applications where there are three curses of dimensionality. Consider the problem of managing blood inventories. There are eight blood types (AB+, AB-, A+, A-, B+, B-, O+, O-), which means we have eight types of blood supplies and eight types of blood demands. Let R_{ti} be the supply of blood type i at time t ($i = 1, 2, \dots, 8$), and let D_{ti} be the demand for blood type

i at time t . Our state variable is given by $S_t = (R_t, D_t)$, where $R_t = (R_{ti})_{i=1}^8$ (D_t is defined similarly).

In each time period there are two types of randomness: random blood donations and random demands. Let \hat{R}_{ti} be the random new donations of blood of type i in week t , and let \hat{D}_{ti} be the random new demands for blood of type i in week t . We are going to let $W_t = (\hat{R}_t, \hat{D}_t)$ be the vector of random information (new supplies and demands) that becomes known in week t .

Finally, let x_{tij} be the amount of blood type i used to satisfy a demand for blood of type j . We switch to “ x ” for the action because this is the standard notation used in the field of mathematical programming for solving vector-valued decision problems. There are rules that govern what blood types can substitute for different demand types, shown in Figure 1.2.

We can quickly see that S_t and W_t have 16 dimensions each. If we have up to 100 units of blood of any type, then our state space has $100^{16} = 10^{32}$ states. If we have up to 20 units of blood being donated or needed in any week, then W_t has $20^{16} = 6.55 \times 10^{20}$ outcomes. We would need to evaluate 16 nested summations to evaluate the expectation. Finally, x_t has 27 dimensions (there are 27 feasible substitutions of blood types for demand types). Needless to say, evaluating all possible values of x_t is completely intractable.

This problem illustrates what is, for many applications, the three curses of dimensionality:

1. *State space.* If the state variable $S_t = (S_{t1}, S_{t2}, \dots, S_{ti}, \dots, S_{tI})$ has I dimensions, and if S_{ti} can take on L possible values, then we might have up to L^I different states.

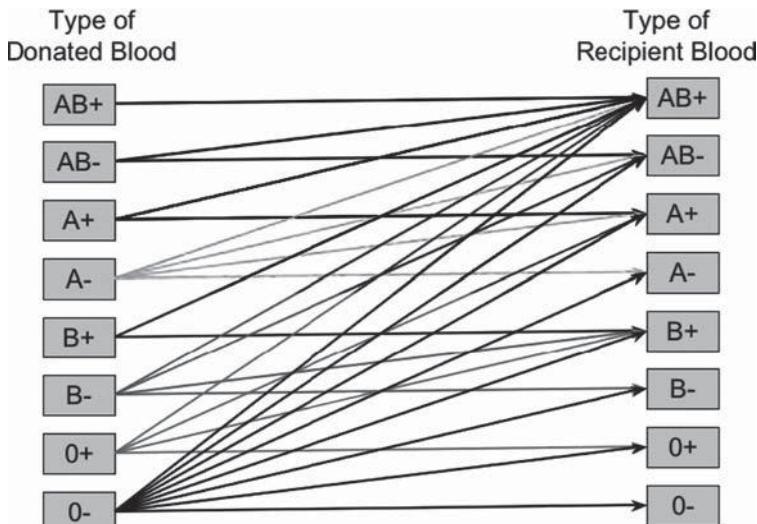


Figure 1.2 Different substitution possibilities between donated blood and patient types (from Cant, 2006).

2. *Outcome space.* The random variable $W_t = (W_{t1}, W_{t2}, \dots, W_{tj}, \dots, W_{tJ})$ might have J dimensions. If W_{tj} can take on M outcomes, then our outcome space might take on up to M^J outcomes.
3. *Action space.* The decision vector $x_t = (x_{t1}, x_{t2}, \dots, x_{tk}, \dots, x_{tK})$ might have K dimensions. If x_{tk} can take on N outcomes, then we might have up to N^K outcomes. In the language of math programming, we refer to the action space as the *feasible region*, and we may assume that the vector x_t is discrete (integer) or continuous.

By the time we get to Chapter 14, we will be able to produce high-quality, implementable solutions not just to the blood problem (see Section 14.2) but for problems that are far larger. The techniques that we are going to describe have produced production quality solutions to plan the operations of some of the largest transportation companies in the country. These problems require state variables with millions of dimensions, with very complex dynamics. We will show that these same algorithms converge to optimal solutions for special cases. For these problems we will produce solutions that are within 1 percent of optimality in a small fraction of the time required to find the optimal solution using classical techniques. However, we will also describe algorithms for problems with unknown convergence properties that produce solutions of uncertain quality and with behaviors that can range from the frustrating to the mystifying. This is a very young field.

Not all problems suffer from the three curses of dimensionality. Many problems have small sets of actions (do we buy or sell?), easily computable expectations (did a customer arrive or not?), and small state spaces (the nodes of a network). The field of dynamic programming has identified many problems, some with genuine industrial applications, that avoid the curses of dimensionality. Our goal is to provide guidance for the problems that do not satisfy some or all of these convenient properties.

1.3 SOME REAL APPLICATIONS

Our experiences using approximate dynamic programming have been driven by problems in transportation with decision vectors with thousands or tens of thousands of dimensions, and state variables with thousands to millions of dimensions. We have solved energy problems with 175,000 time periods. Our applications have spanned applications in air force, finance, and health.

As our energy environment changes, we have to plan new energy resources (such as the wind turbines in Figure 1.3). A challenging dynamic problem requires determining when to acquire or install new energy technologies (wind turbines, solar panels, energy storage using flywheels or compressed air, hydrogen cars) and how to operate them. These decisions have to be made when considering uncertainty in the demand, prices, and the underlying technologies for creating, storing, and using energy. For example, adding ethanol capacity has to include the possibility that oil prices will drop (reducing the demand for ethanol) or that government regulations may favor alternative fuels (increasing the demand).



Figure 1.3 Wind turbines are one form of alternative energy resources (from <http://www.nrel.gov/data/pix/searchpix.cgi>).

An example of a very complex resource allocation problem arises in railroads (Figure 1.4). In North America there are six major railroads (known as “Class I” railroads) that operate thousands of locomotives, many of which cost over \$1 million. Decisions have to be made now to assign locomotives to trains, taking into account how the locomotives will be used at the destination. For example, a train may be going to a location that needs additional power. Or a locomotive might have to be routed to a maintenance facility, and the destination of a train may or may not offer good opportunities for getting the locomotive to the shop. There are many types of locomotives, and different types of locomotives are suited to different types of trains (e.g., trains moving coal, grain, or merchandise). Other applications of dynamic programming include the management of freight cars, where decisions about when, where, and how many to move have to be made in the presence of numerous sources of uncertainty, including customer demands, transit times, and equipment problems.

The military faces a broad range of operational challenges that require positioning resources to anticipate future demands. The problem may be figuring out when and where to position tankers for mid-air refueling (Figure 1.5), or whether



Figure 1.4 Major railroads in the United States have to manage complex assets such as boxcars, locomotives and the people who operate them. Courtesy Norfolk Southern.

a cargo aircraft should be modified to carry passengers. The air mobility command needs to think about not only what aircraft is best to move a particular load of freight but also the value of aircraft in the future (are there repair facilities near the destination?). The military is further interested in the value of more reliable aircraft and the impact of last-minute requests. Dynamic programming provides a means to produce robust decisions, allowing the military to respond to **last-minute requests**.

Managing the electric power grid requires evaluating the reliability of equipment such as the transformers that convert high-voltage power to the voltages used by homes and businesses. Figure 1.6 shows the high-voltage backbone network managed by PJM Interconnections that provides electricity to the northeastern United States. To ensure the reliability of the grid, PJM helps utilities maintain an appropriate inventory of spare transformers. They cost five million dollars each, weigh over 200 tons, and require at least a year to deliver. We must make decisions about how many to buy, how fast they should be delivered (fast delivery costs more), and where to put them when they do arrive. If a transformer fails, the electric power grid may have to purchase power from more expensive utilities to avoid a bottleneck, possibly costing millions of dollars per month. **As a result it is not possible to wait until problems happen.** Utilities also face the problem of pricing their energy in a **dynamic market**, and purchasing commodities such as coal and natural gas in the presence of fluctuating prices.



Figure 1.5 Mid-air refueling is a major challenge for air operations, requiring that tankers be positioned in anticipation of future needs (from <http://www.amc.af.mil/photos/>).

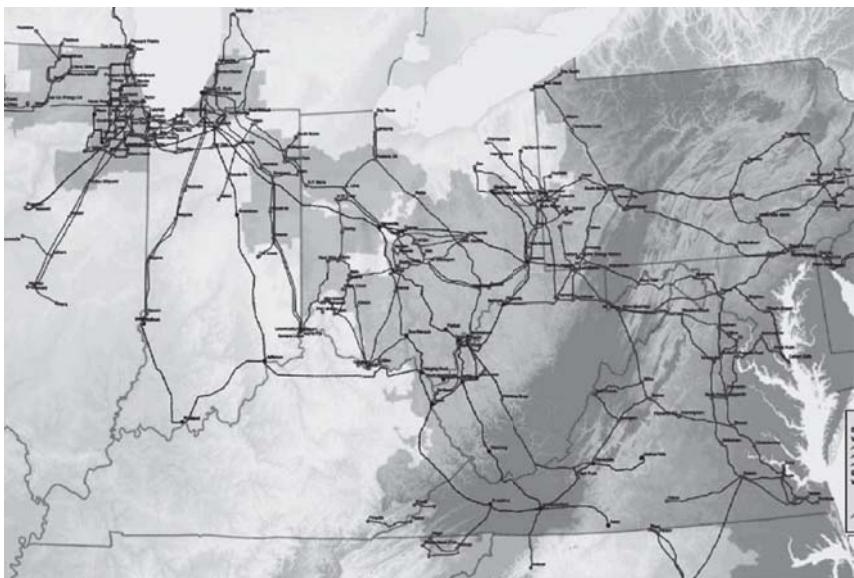


Figure 1.6 High-voltage backbone network managed by PJM Interconnections provides electricity to the northeastern United States. Courtesy PJM Interconnections.

Similar issues arise in the truckload motor carrier industry, where drivers are assigned to move loads that arise in a highly dynamic environment. Large companies manage fleets of thousands of drivers, and the challenge at any moment in time is to find the best driver (Figure 1.7 is from Schneider National, the largest truckload carrier in the United States). There is much more to the problem than



Figure 1.7 Schneider National, the largest truckload motor carrier in the United States, manages a fleet of over 15,000 drivers. Courtesy Schneider National.

simply finding the closest driver; each driver is characterized by attributes such as his or her home location and equipment type as well as his or her skill level and experience. There is a need to balance decisions that maximize profits now versus those that produce good long-run behavior. Approximate dynamic programming produced the first accurate model of a large truckload operation. Modeling this large-scale problem produces some of the advances described in this volume.

Challenging dynamic programs can be found in much simpler settings. A good example involves optimizing the amount of cash held in a mutual fund, which is a function of current market performance (should more money be invested?) and interest rates, illustrated in Figure 1.8. While this problem can be modeled with just three dimensions, the lack of structure and need to discretize at a fine level produced a very challenging optimization problem. Other applications include portfolio allocation problems and determining asset valuations that depend on portfolios of assets.

A third problem class is the acquisition of information. Consider the problem faced by the government that is interested in researching a new technology such as fuel cells or converting coal to hydrogen. There may be dozens of avenues to pursue, and the challenge is to determine the projects in which the government should invest. The state of the system is the set of estimates of how well different components of the technology work. The government funds research to collect information. The result of the research may be the anticipated improvement, or the

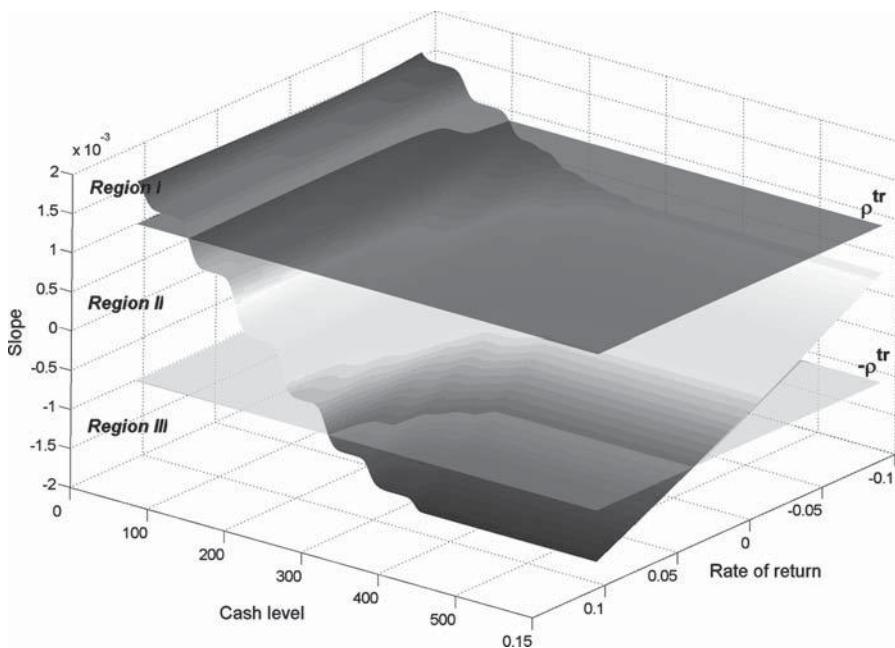


Figure 1.8 Value of holding cash in a mutual fund as a function of market performance and interest rates.

results may be disappointing. The government wants to plan a research program to maximize the likelihood that a successful technology is developed within a reasonable time frame (e.g., 20 years). Depending on time and budget constraints, the government may wish to fund competing technologies in the event that one does not work. Alternatively, it may be more effective to fund one promising technology and then switch to an alternative if the first does not work out.

1.4 PROBLEM CLASSES

Most of the problems that we use as examples in this book can be described as involving the management of physical, financial, or informational resources. Sometimes we use the term “assets,” which carries the connotation of money or valuable resources (aircraft, real estate, energy commodities). But in some settings, even these terms may seem inappropriate, for example, training computers to play a game such as tic-tac-toe, where it will be more natural to think in terms of managing an “entity.” Regardless of the term, there are a number of major problem classes we consider in our presentation:

Budgeting. Here we face the problem of allocating a fixed resource over a set of activities that incurs costs that are a function of how much we invest in the activity. For example, drug companies have to decide how much to

invest in different research projects or how much to spend on advertising for different drugs. Oil exploration companies have to decide how much to spend exploring potential sources of oil. Political candidates have to decide how much time to spend campaigning in different states.

Asset acquisition with concave costs. A company can raise capital by issuing stock or floating a bond. There are costs associated with these financial instruments independent of how much money is being raised. Similarly an oil company purchasing oil will be given quantity discounts (or it may face the fixed cost of purchasing a tanker-load of oil). Retail outlets get a discount if they purchase a truckload of an item. All of these are instances of acquiring assets with a concave (or, more generally, nonconvex) cost function, which means there is an incentive for purchasing larger quantities.

Asset acquisition with lagged information processes. We can purchase commodity futures that allow us to purchase a product in the future at a lower cost. Alternatively, we may place an order for memory chips from a factory in southeast Asia with one- to two-week delivery times. A transportation company has to provide containers for a shipper who may make requests several days in advance or at the last minute. All of these are asset acquisition problems with *lagged information processes*.

Buying/selling an asset. In this problem class the process stops when we either buy an asset when it looks sufficiently attractive or sell an asset when market conditions warrant. The game ends when the transaction is made. For these problems we tend to focus on the price (the purchase price or the sales price), and our success depends on our ability to trade off current value with future price expectations.

General resource allocation problems. This class encompasses the problem of managing reusable and substitutable resources over time (equipment, people, products, commodities). Applications abound in transportation and logistics. Railroads have to move locomotives and boxcars to serve different activities (moving trains, moving freight) over time. An airline has to move aircraft and pilots in order to move passengers. Consumer goods have to move through warehouses to retailers to satisfy customer demands.

Demand management. There are many applications where we focus on managing the demands being placed on a process. Should a hospital admit a patient? Should a trucking company accept a request by a customer to move a load of freight?

Storage problems. We face problems determining how much energy to store in a water reservoir or battery, how much cash to hold in a mutual fund, how many vaccines to order or blood to hold, and how much inventory to keep. These are all examples of “storage” problems, which may involve one, several or many types of resources, and where these decisions have to be made in the context of state-of-the-world variables such as commodity prices, interest rates, and weather.

Table 1.2 Major problem classes

Number of Entities	Attributes	
	Simple	Complex
Single	Single, simple entity	Single, complex entity
Multiple	Multiple, simple entities	Multiple, complex entities

Shortest paths. In this problem class we typically focus on managing a single, discrete entity. The entity may be someone playing a game, a truck driver we are trying to route to return him home, a driver who is trying to find the best path to his destination, or a locomotive we are trying to route to its maintenance shop. Shortest path problems, however, also represent a general mathematical structure that applies to a broad range of dynamic programs.

Dynamic assignment. Consider the problem of managing multiple entities, such as computer programmers, to perform different tasks over time (writing code or fixing bugs). Each entity and task is characterized by a set of attributes that determines the cost (or contribution) from assigning a particular resource to a particular task.

All these problems focus on the problem of managing physical or financial resources (or assets, or entities). It is useful to think of four major problem classes (depicted in Table 1.2) in terms of whether we are managing a single or multiple entities (e.g., one robot or a fleet of trucks), and whether the entities are simple (an entity may be described by its location on a network) or complex (an entity may be a truck driver described by a 10-dimensional vector of attributes).

If we are managing a single, simple entity, then this is a problem that can be solved exactly using classical algorithms described in Chapter 3. The problem of managing a single, complex entity (e.g., playing backgammon) is commonly studied in computer science under the umbrella of reinforcement learning, or in the engineering community under the umbrella of control theory (e.g., landing an aircraft). The problem of managing multiple, simple entities is widely studied in the field of operations research (managing fleets of vehicles or distribution systems), although this work most commonly focuses on deterministic models. By the end of this book, we are going to show the reader how to handle (approximately) problem classes that include multiple, complex entities, in the presence of different forms of uncertainty.

There is a wide range of problems in dynamic programming that involve controlling resources, where decisions directly involve transforming resources (purchasing inventory, moving robots, controlling the flow of water from reservoirs), but there are other important types of controls. Some examples include:

Pricing. Often the question being asked is, What price should be paid for an asset? The right price for an asset depends on how it is managed, so it should not be surprising that we often find asset prices as a by-product from determining how to best manage the asset.

Information collection. Since we are modeling sequential information and decision processes, we explicitly capture the information that is available when we make a decision, allowing us to undertake studies that change the information process. For example, the military uses unmanned aerial vehicles (UAVs) to collect information about targets in a military setting. Oil companies drill holes to collect information about underground geologic formations. Travelers try different routes to collect information about travel times. Pharmaceutical companies use test markets to experiment with different pricing and advertising strategies.

Technology switching. The last class of questions addresses the underlying technology that controls how the physical process evolves over time. For example, when should a power company upgrade a generating plant (e.g., to burn oil and natural gas)? Should an airline switch to aircraft that fly faster or more efficiently? How much should a communications company invest in a technology given the likelihood that better technology will be available in a few years?

Most of these problems entail both discrete and continuous states and actions.

Continuous models would be used for money, for physical products such as oil, grain, and coal, or for discrete products that occur in large volume (most consumer products). In other settings, it is important to retain the integrality of the resources being managed (people, aircraft, locomotives, trucks, and expensive items that come in small quantities). For example, how do we position emergency response units around the country to respond to emergencies (bioterrorism, major oil spills, failure of certain components in the electric power grid)?

What makes these problems hard? With enough assumptions, none of these problems are inherently difficult. But in real applications, a variety of issues emerge that can make all of them intractable. These include:

- *Evolving information processes.* We have to make decisions now before we know the information that will arrive later. This is the essence of stochastic models, and this property quickly turns the easiest problems into computational nightmares.
- *High-dimensional problems.* Most problems are easy if they are small enough. In real applications, there can be many types of resources, producing decision vectors of tremendous size.
- *Measurement problems.* Normally we assume that we look at the state of our system and from this determine what decision to make. In many problems we cannot measure the state of our system precisely. The problem may be delayed information (stock prices), incorrectly reported information (the truck is in the wrong location), misreporting (a manager does not properly add up his total sales), theft (retail inventory), or deception (an equipment manager underreports his equipment so it will not be taken from him).
- *Unknown models (information, system dynamics).* We can anticipate the future by being able to say something about what might happen (even if it is with

uncertainty) or the effect of a decision (which requires a model of how the system evolves over time).

- *Missing information.* There may be costs that simply cannot be computed and that are instead ignored. The result is a consistent model bias (although we do not know when it arises).
- *Comparing solutions.* Primarily as a result of uncertainty, it can be difficult comparing two solutions to determine which is better. Should we be better on average, or are we interested in the best and worst solution? Do we have enough information to draw a firm conclusion?

1.5 THE MANY DIALECTS OF DYNAMIC PROGRAMMING

Dynamic programming arises from the study of sequential decision processes. Not surprisingly, these arise in a wide range of applications. While we do not wish to take anything from Bellman’s fundamental contribution, the optimality equations are, to be quite honest, somewhat obvious. As a result they were discovered independently by the different communities in which these problems arise.

The problems arise in a variety of engineering problems, typically in continuous time with continuous control parameters. These applications gave rise to what is now referred to as **control theory**. While uncertainty is a major issue in these problems, the formulations tend to focus on deterministic problems (the uncertainty is typically in the estimation of the state or the parameters that govern the system). Economists adopted control theory for a variety of problems involving the control of activities from allocating single budgets or managing entire economies (admittedly at a very simplistic level). Operations research (through Bellman’s work) did the most to advance the theory of controlling stochastic problems, thereby producing the very rich theory of Markov decision processes. Computer scientists, especially those working in the realm of artificial intelligence, found that dynamic programming was a useful framework for approaching certain classes of machine learning problems known as reinforcement learning.

As different communities discovered the same concepts and algorithms, they invented their own vocabularies to go with them. As a result we can solve the Bellman equations, the Hamiltonian, the Jacobian, the Hamilton–Jacobian, or the all-purpose Hamilton–Jacobian–Bellman equations (typically referred to as the HJB equations). In our presentation we prefer the term “optimality equations,” but “Bellman” has become a part of the language, imbedded in algorithmic strategies such as minimizing the “Bellman error” and “Bellman residual minimization.”

There is an even richer vocabulary for the types of algorithms that are the focal point of this book. Everyone has discovered that the backward recursions required to solve the optimality equations in Section 1.1 do not work if the state variable is multidimensional. For example, instead of visiting node i in a network, we might visit state $S_t = (S_{t1}, S_{t2}, \dots, S_{tB})$, where S_{tb} is the amount of blood on hand of type b . A variety of authors have independently discovered that an alternative strategy is to step forward through time, using iterative algorithms to help estimate

the value function. This general strategy has been referred to as forward dynamic programming, incremental dynamic programming, iterative dynamic programming, adaptive dynamic programming, heuristic dynamic programming, reinforcement learning, and neuro-dynamic programming. The term that is being increasingly adopted is *approximate dynamic programming*, although perhaps it is convenient that the initials, ADP, apply equally well to “adaptive dynamic programming.”

The different communities have each evolved their own vocabularies and notational systems. The notation developed for Markov decision processes, and then adopted by the computer science community in the field of reinforcement learning, uses state S and action a . In control theory, it is state x and control u . The field of stochastic programming uses x for decisions. At first, it is tempting to view these as different words for the same quantities, but the cosmetic differences in vocabulary and notation tend to hide more fundamental differences in the nature of the problems being addressed by each community. In reinforcement learning, there is typically a small number of discrete actions. In control theory, u is usually a low-dimensional continuous vector. In operations research, it is not unusual for x to have hundreds or thousands of dimensions.

An unusual characteristic of the reinforcement learning community is their habit of naming algorithms after their notation. Algorithms such as Q -learning (named from the use of Q -factors), $TD(\lambda)$, ϵ -greedy exploration, and SARSA (which stands for state-action-reward-state-action), are some of the best examples. As a result care has to be taken when designing a notational system.

The field continues to be characterized by a plethora of terms that often mean the same thing. The transition function (which models the evolution of a system over time) is also known as the system model, transfer function, state model, and plant model. The behavior policy is the same as the sampling policy, and a stepsize is also known as the learning rate or the gain.

There is a separate community that evolved from the field of deterministic math programming that focuses on problems with high-dimensional decisions. The reinforcement learning community focuses almost exclusively on problems with finite (and fairly small) sets of discrete actions. The control theory community is primarily interested in multidimensional and continuous actions (but not very many dimensions). In operations research it is not unusual to encounter problems where decisions are vectors with thousands of dimensions.

As early as the 1950s the math programming community was trying to introduce uncertainty into mathematical programs. The resulting subcommunity is called stochastic programming and uses a vocabulary that is quite distinct from that of dynamic programming. The relationship between dynamic programming and stochastic programming has not been widely recognized, despite the fact that Markov decision processes are considered standard topics in graduate programs in operations research.

Our treatment will try to bring out the different dialects of dynamic programming, although we will tend toward a particular default vocabulary for important concepts. Students need to be prepared to read books and papers in this field

that will introduce and develop important concepts using a variety of dialects. The challenge is realizing when authors are using different words to say the same thing.

1.6 WHAT IS NEW IN THIS BOOK?

As of this writing, dynamic programming has enjoyed a relatively long history, with many superb books. Within the operations research community, the original text by Bellman (Bellman, 1957) was followed by a sequence of books focusing on the theme of Markov decision processes. Of these, the current high-water mark is *Markov Decision Processes* by Puterman, which played an influential role in the writing of Chapter 3. The first edition appeared in 1994, followed in 2005 by the second edition. The dynamic programming field offers a powerful theoretical foundation, but the algorithms are limited to problems with very low-dimensional state and action spaces.

This book focuses on a field that is coming to be known as *approximate dynamic programming*; it emphasizes modeling and computation for much harder classes of problems. The problems may be hard because they are large (e.g., large state spaces), or because we lack a model of the underlying process that the field of Markov decision processes takes for granted. Two major references preceded the first edition of this volume. *Neuro-dynamic Programming* by Bertsekas and Tsitsiklis was the first book to appear (in 1996) that integrated stochastic approximation theory with the power of statistical learning to approximate value functions, in a rigorous if demanding presentation. *Reinforcement Learning* by Sutton and Barto, published in 1998 (but building on research that began in 1980), presents the strategies of approximate dynamic programming in a very readable format, with an emphasis on the types of applications that are popular in the computer science/artificial intelligence community. Along with these books, the survey of reinforcement learning in Kaelbling et al. (1996) is a major reference.

There is a sister community that goes by the name of *simulation optimization* that has evolved out of the simulation community that needs to select the best from a set of designs. Nice reviews of this literature are given in Fu (2002) and Kim and Nelson (2006). Books on the topic include Gosavi (2003), Chang et al. (2007), and Cao (2007). Simulation optimization is part of a larger community called stochastic search, which is nicely described in the book Spall (2003). As we show later, this field is directly relevant to policy search methods in approximate dynamic programming.

This volume presents approximate dynamic programming with a much stronger emphasis on modeling, with explicit and careful notation to capture the timing of information. We draw heavily on the modeling framework of control theory with its emphasis on transition functions, which easily handle complex problems, rather than transition matrices, which are used heavily in both Bertsekas and Tsitsiklis (1996) and Sutton and Barto (1998). We start with the classical notation of Markov decision processes that is familiar to the reinforcement learning community, but

we build bridges to math programming so that by the end of the book, we are able to solve problems with very high-dimensional decision vectors. For this reason we adopt two notational styles for modeling decisions: a for discrete actions common in the models solved in reinforcement learning, and x for the continuous and sometimes high-dimensional decision vectors common in operations research and math programming. Throughout the book, we use action a as our default notation, but switch to x in the context of applications that require continuous or multidimensional decisions.

Some other important features of this book are as follows:

- We identify the three curses of dimensionality that characterize some dynamic programs, and develop a comprehensive algorithmic strategy for overcoming them.
- We cover problems with discrete action spaces, denoted using a (standard in Markov decision processes and reinforcement learning), and vector-valued decisions, denoted using x (standard in mathematical programming). The book integrates approximate dynamic programming with math programming, making it possible to solve intractably large deterministic or stochastic optimization problems.
- We cover in depth the concept of the post-decision state variable, which plays a central role in our ability to solve problems with vector-valued decisions. The post-decision state offers the potential for dramatically simplifying many ADP algorithms by avoiding the need to compute a one-step transition matrix or otherwise approximate the expectation within Bellman’s equation.
- We place considerable attention on the proper modeling of random variables and system dynamics. **We feel that it is important to properly model a problem before attempting to solve it.**
- The theoretical foundations of this material can be deep and rich, but our presentation is aimed at advanced undergraduate or masters level students with introductory courses in statistics, probability, and for Chapter 14, linear programming. For more advanced students, proofs are provided in “Why does it work” sections. The presentation is aimed primarily at students in engineering interested in taking real, complex problems, developing proper mathematical models, and producing computationally tractable algorithms.
- We identify four fundamental classes of policies (myopic, lookahead, policies based on value function approximations, and policy function approximations), with careful treatments of the last three. An entire chapter is dedicated to policy search methods, and three chapters develop the critical idea of using value function approximations.
- We carefully deal with the challenge of stepsizes, which depend critically on whether the algorithm is based on approximate value iteration (including Q -learning and TD learning) or approximate policy iteration. Optimal stepsize rules are given for each of these two major classes of algorithms.

Our presentation integrates the fields of Markov decision processes, math programming, statistics, and simulation. The use of statistics to estimate value functions dates back to Bellman and Dreyfus (1959). Although the foundations for proving convergence of special classes of these algorithms traces its origins to the seminal paper on stochastic approximation theory (Robbins and Monro, 1951), the use of this theory (in a more modern form) to prove convergence of special classes of approximate dynamic programming algorithms did not occur until 1994 (Tsitsiklis 1994; Jaakkola et al. 1994). The first book to bring these themes together is Bertsekas and Tsitsiklis (1996), which remains a seminal reference for researchers looking to do serious theoretical work.

1.7 PEDAGOGY

The book is roughly organized into four parts. Part I comprises Chapters 1 to 4, which provide a relatively easy introduction using a simple, discrete representation of states. Part II covers modeling, a description of major classes of policies and policy optimization. Part III covers policies based on value function approximations, along with efficient learning. Part IV describes specialized methods for resource allocation problems.

A number of sections are marked with an *. These can all be skipped when first reading the book without loss of continuity. Sections marked with ** are intended only for advanced graduate students with an interest in the theory behind the techniques.

Part I *Introduction to dynamic programming using simple state representations*—In the first four chapters we introduce dynamic programming, using what is known as a “flat” state representation. That is to say, we assume that we can represent states as $s = 1, 2, \dots$. We avoid many of the rich modeling and algorithmic issues that arise in more realistic problems.

Chapter 1 Here we set the tone for the book, introducing the challenge of the three “curses of dimensionality” that arise in complex systems.

Chapter 2 Dynamic programs are best taught by example. Here we describe three classes of problems: deterministic problems, stochastic problems, and information acquisition problems. Notation is kept simple but precise, and readers see a range of different applications.

Chapter 3 This is an introduction to classic Markov decision processes. While these models and algorithms are typically dismissed because of “the curse of dimensionality,” these ideas represent the foundation of the rest of the book. The proofs in the “why does it work” section are particularly elegant and help provide a deep understanding of this material.

Chapter 4 This chapter provides a fairly complete introduction to approximate dynamic programming, but focusing purely on estimating value functions using lookup tables. The material is particularly familiar to the reinforcement learning community. The presentation steps through classic algorithms, starting with Q -learning and SARSA, and then, progressing through real-time

dynamic programming (which assumes you can compute the one-step transition matrix), approximate value iteration using a pre-decision state variable, and finally, approximate value iteration using a post-decision state variable. Along the way the chapter provides a thorough introduction to the concept of the post-decision state variable, and introduces the issue of exploration and exploitation, as well as on-policy and off-policy learning.

Part II Approximate dynamic programming with policy optimization—This block introduces modeling, the design of policies, and policy optimization. Policy optimization is the simplest method for making good decisions, but it is generally restricted to relatively simple problems. As such, it makes for a good introduction to ADP before getting into the complexities of designing policies based on value function approximations.

Chapter 5 This chapter on modeling hints at the richness of dynamic problems.

To help with assimilating this chapter, we encourage readers to skip sections marked with an * the first time they go through the chapter. It is also useful to reread this chapter from time to time as you are exposed to the rich set of modeling issues that arise in real applications.

Chapter 6 This chapter introduces four fundamental classes of policies: myopic policies, lookahead policies, policies based on value function approximations, and policy function approximations. We note that there are three classes of approximation strategies: lookup table, and parametric and nonparametric models. These fundamental categories appear to cover all the variations of policies that have been suggested.

Chapter 7 There are many problems where the structure of a policy is fairly apparent, but it depends on tunable parameters. Here we introduce the reader to communities that seek to optimize functions of deterministic parameters (which determines the policy) where we depend on noisy evaluations to estimate the performance of the policy. We cover classical stochastic search, add algorithms from the field of simulation optimization, and introduce the idea of the knowledge gradient, which has proved to be a useful general-purpose algorithmic strategy. In the process, the chapter provides an initial introduction to the exploration-exploitation problem for (offline) ranking and selection problems.

Part III Approximate dynamic programming using value function approximations—This is the best-known strategy for solving dynamic programs (approximately), and also the most difficult to master. We break this process into three steps, organized into the three chapters below:

Chapter 8 This chapter covers the basics of approximating functions using lookup tables (very briefly), parametric models (primarily linear regression) and a peek into nonparametric methods.

Chapter 9 Let $\bar{V}^\pi(s)$ be an approximation of the value of being in state s while following a fixed policy π . We fit this approximation using sample observations \hat{v}^n . This chapter focuses on different ways of computing \hat{v}^n for finite and infinite horizon problems, which can then be used in conjunction with the methods in Chapter 8 to find $\bar{V}^\pi(s)$.

Chapter 10 The real challenge is estimating the value of a policy while simultaneously searching for better policies. This chapter introduces algorithms based on approximate value iteration (including Q -learning and TD learning) and approximate policy iteration. The discussion covers issues of convergence that arise while one simultaneously tries to estimate and optimize.

Chapter 11 Stepsizes are an often overlooked dimension of approximate dynamic programming. This chapter reviews four classes of stepsizes: deterministic formulas, heuristic stochastic formulas, optimal stepsize rules based on signal processing (ideally suited for policy iteration), and a new optimal stepsize designed purely for approximate value iteration.

Chapter 12 It is well known in the ADP/RL communities that it is sometimes necessary to visit a state in order to learn about the value of being in a state. Chapter 4 introduces this issue, and Chapter 7 returns to the issue again in the context of policy search. Here we address the problem in its full glory, making the transition from pure learning (no physical state) for both online and offline problems, but also from learning in the presence of a physical state.

Part IV Resource allocation and implementation challenges—We close with methods that are specifically designed for problems that arise in the context of resource allocation:

Chapter 13 Resource allocation problems have special structure such as concavity (or convexity for minimization problems). This chapter describes a series of approximation techniques that are directly applicable for these problems.

Chapter 14 There are many problems that can be described under the umbrella of “resource allocation” that offer special structure that we can exploit. These problems tend to be high-dimensional, with state variables that can easily have thousands or even millions of dimensions. However, when we combine concavity with the post-decision state variable, we produce algorithms that can handle industrial-strength applications.

Chapter 15 We close with a discussion of a number of more practical issues that arise in the development and testing of ADP algorithms.

This material is best covered in order. Depending on the length of the course and the nature of the class, an instructor may want to skip some sections, or to weave in some of the theoretical material in the “why does it work” sections. Additional material (exercises, solutions, datasets, errata) will be made available over time at the website <http://www.castlelab.princeton.edu/adp.htm> (this can also be accessed from the CASTLE Lab website at <http://www.castlelab.princeton.edu/>).

There are two faces of approximate dynamic programming, and we try to present both of them. The first emphasizes models and algorithms, with an emphasis on applications and computation. It is virtually impossible to learn this material without writing software to test and compare algorithms. The other face is a deeply theoretical one that focuses on proofs of convergence and rate of convergence. This material is advanced and accessible primarily to students with training in probability and stochastic processes at an advanced level.

1.8 BIBLIOGRAPHIC NOTES

There have been three major lines of investigation that have contributed to approximate dynamic programming. The first started in operations research with Bellman's seminal text (Bellman, 1957). Numerous books followed using the framework established by Bellman, each making important contributions to the evolution of the field. Selected highlights include Howard (1960), Derman (1970), Ross (1983), and Heyman and Sobel (1984). As of this writing, the best overall treatment of what has become known as the field of Markov decision processes is given in Puterman (2005). However, this work has focused largely on theory, since the field of discrete Markov decision processes has not proved easy to apply, as discrete representations of state spaces suffer from the well-known curse of dimensionality, which restricts this theory to extremely small problems. The use of statistical methods to approximate value functions originated with Bellman, in Bellman and Dreyfus (1959), but little subsequent progress was made within operations research.

The second originated with efforts by computer scientists to get computers to solve problems, starting with the work of Samuel (1959) to train a computer to play checkers, helping to launch the field that would become known in artificial intelligence as reinforcement learning. The real origins of the field lay in the seminal work in psychology initiated by Andy Barto and Richard Sutton (Sutton and Barto, 1981; Barto et al., 1981; Barto and Sutton, 1981). This team made many contributions over the next two decades, leading up to their landmark volume *Reinforcement Learning* (Sutton and Barto, 1998) which has effectively defined this field. Reinforcement learning evolved originally as an intuitive framework for describing human (and animal) behavior, and only later was the connection made with dynamic programming, when computer scientists adopted the notation developed within operations research. For this reason reinforcement learning as practiced by computer scientists and Markov decision processes as practiced by operations research share a common notation, but a very different culture.

The third line of investigation started completely independently under the umbrella of control theory. Instead of Bellman's optimality equation, it was the Hamiltonian or Jacobi equations, which evolved to the Hamilton–Jacobi equations. Aside from different notation, control theorists were motivated by problems of operating physical processes, and as a result focused much more on problems in continuous time, with continuous states and actions. While analytical solutions could be obtained for special cases, it is perhaps not surprising that control theorists quickly developed their own style of approximate dynamic programming, initially called heuristic dynamic programming (Werbos, 1974, 1989, 1992b). It was in this community that the first connection was made between the adaptive learning algorithms of approximate dynamic programming and reinforcement learning, and the field of stochastic approximation theory. The seminal papers that made this connection were Tsitsiklis (1994), Tsitsiklis and van Roy (1997), and the seminal book *Neuro-dynamic Programming* written by Bertsekas and Tsitsiklis (1996). A major breakthrough in control theory was

the recognition that the powerful technology of neural networks (Haykin, 1999) could be a general-purpose tool for approximating both value functions as well as policies. Major contributions were also made within the field of economics, including Rust (1997) and Judd (1998).

While these books have received the greatest visibility, special recognition is due to a series of workshops funded by the National Science Foundation under the leadership of Paul Werbos, some of which have been documented in several edited volumes (Werbos et al., 1990; White and Sofge, 1992; Si et al., 2004). These workshops have played a significant role in bringing different communities together, the effect of which can be found throughout this volume.

Our presentation is primarily written from the perspective of approximate dynamic programming and reinforcement learning as it is practiced in operations research and computer science, but there are substantial contributions to this field that have come from the engineering controls community. The edited volume by White and Sofge, (1992) provides the first comprehensive coverage of ADP/RL, primarily from the perspective of the controls community (there is a single chapter by Andy Barto on reinforcement learning). Bertsekas and Tsitsiklis (1996) is also written largely from a control perspective, although the influence of problems from operations research and artificial intelligence are apparent. A separate and important line of research grew out of the artificial intelligence community, which is nicely summarized in the review by Kaelbling et al. (1996) and the introductory textbook by Sutton and Barto (1998). More recently Si et al. (2004) brought together papers from the engineering controls community, artificial intelligence, and operations research.

Since the first edition of this book appeared in 2007, a number of other important references have appeared. The third edition of *Dynamic Programming and Optimal Control*, volume 2, by Bertsekas contains a lengthy chapter entitled *Approximate Dynamic Programming*, which he updates continuously and which can be downloaded directly from <http://web.mit.edu/dimitrib/www/dpchapter.html>. Sutton and Barto released a version of their 1998 book to the Internet at <http://www.cs.wmich.edu/~trenary/files/cs5300/RLBook/the-book.html>, which makes their classic book easily accessible. Two recent additions to the literature include Busoniu et al. (2010) and Szepesvari (2010), the latter of which is also available as a free download at <http://www.morganclaypool.com/doi/abs/10.2200/S00268ED1V01Y201005AIM009>. These are more advanced monographs, but both contain recent theoretical and algorithmic developments.

Energy applications represent a growing area of research in approximate dynamic programming. Applications include Löhndorf and Minner (2010) and Powell et al. (2011). An application of ADP to truckload trucking at Schneider National is described in Simao et al. (2009, 2010). The work on Air Force operations is described in Wu et al. (2009). The application to the mutual fund cash balance problem is given in Nascimento and Powell (2010b).

C H A P T E R 2

Some Illustrative Models

Dynamic programming is one of those incredibly rich fields that has filled the careers of many. But it is also a deceptively easy idea to illustrate and use. This chapter presents a series of applications that illustrate the modeling of dynamic programs. The goal of the presentation is to teach dynamic programming by example. The applications range from problems that can be solved analytically, to those that can be solved using fairly simple numerical algorithms, to very large scale problems that will require carefully designed applications. The examples in this chapter effectively communicate the range of applications that the techniques in this book are designed to solve.

It is possible, after reading this chapter, to conclude that “dynamic programming is easy” and to wonder “why do I need the rest of this book?” The answer is: sometimes dynamic programming *is* easy and requires little more than the understanding gleaned from these simple problems. But there is a vast array of problems that are quite difficult to model, and where standard solution approaches are computationally intractable.

We present three classes of examples: (1) deterministic problems, where everything is known; (2) stochastic problems, where some information is unknown, that are described by a known probability distribution; and (3) information acquisition problems, where we have uncertainty described by an unknown distribution. In the last problem class the focus is on collecting information so that we can better estimate the distribution.

These illustrations are designed to teach by example. The careful reader will pick up subtle modeling decisions, in particular the indexing with respect to time. We defer to Chapter 5 a more complete explanation of our choices, where we provide an in-depth treatment of how to model a dynamic program.

2.1 DETERMINISTIC PROBLEMS

Dynamic programming is widely used in many deterministic problems as a technique for breaking down what might be a very large problem into a sequence of much smaller problems. The focus of this book is on stochastic problems, but the value of dynamic programming for solving some large, deterministic problems should never be lost.

2.1.1 The Shortest Path Problem

Perhaps one of the most popular dynamic programming problems is known as the shortest path problem. Although it has many applications, it is easiest to describe in terms of the problem faced by every driver when finding a path from one location to the next over a road network. Let

\mathcal{I} = the set of nodes (intersections) in the network,

\mathcal{L} = the set of links (i, j) in the network,

c_{ij} = the cost (typically the time) to drive from node i to node j , $i, j \in \mathcal{I}$,
 $(i, j) \in \mathcal{L}$,

\mathcal{I}_i^+ = the set of nodes j for which there is a link $(i, j) \in \mathcal{L}$,

\mathcal{I}_j^- = the set of nodes i for which there is a link $(i, j) \in \mathcal{L}$.

We assume that a traveler at node i can choose to traverse any link (i, j) , where $j \in \mathcal{I}_i^+$. Suppose that our traveler is starting at some node q and needs to get to a destination node r at the least cost. Let

v_j = the minimum cost required to get from node j to node r .

We do not know v_j , but we do know that $v_r = 0$. Let v_j^n be our estimate, at iteration n , of the cost to get from j to r . We can find the optimal costs, v_j , by initially setting v_j^0 to a large number for $j \neq r$ and then iteratively looping over all the nodes, finding the best link to traverse out of an intersection i by minimizing the sum of the outbound link cost c_{ij} plus our current estimate of the downstream value v_j^{n-1} . The complete algorithm is summarized in Figure 2.1. This algorithm has been proved to converge to the optimal set of node values.

There is a substantial literature on solving shortest path problems. Because they arise in so many applications, there is tremendous value in solving these problems very quickly. Our basic algorithm is not very efficient because we are often solving equation (2.1) for an intersection i where $v_i^{n-1} = M$, and where $v_j^{n-1} = M$ for all $j \in \mathcal{I}_i^+$. A more standard strategy is to maintain a candidate list of nodes \mathcal{C} that consists of an ordered list i_1, i_2, \dots . Initially the list will consist only of the destination node r (since we are solving the problem of finding paths into the destination node r). As we reach over links into node i in the candidate list, we

Step 0. Let

$$v_j^0 = \begin{cases} M, & j \neq r, \\ 0, & j = r, \end{cases}$$

where “ M ” is known as “big- M ” and represents a large number. Let $n = 1$.

Step 1. Solve for all $i \in \mathcal{I}$,

$$v_i^n = \min_{j \in \mathcal{I}_i^+} (c_{ij} + v_j^{n-1}). \quad (2.1)$$

Step 2. If $v_i^n < v_i^{n-1}$ for any i , let $n = n+1$ and return to step 1. Else stop.

Figure 2.1 Basic shortest path algorithm.

Step 0. Let

$$v_j = \begin{cases} M, & j \neq r, \\ 0, & j = r, \end{cases}$$

Let $n = 1$. Set the candidate list $\mathcal{C} = \{q\}$.

Step 1. Choose node $j \in \mathcal{C}$ from the top of the candidate list.

Step 2. For all nodes $i \in \mathcal{I}_j^-$ do:

Step 2a.

$$\hat{v}_i = c_{ij} + v_j. \quad (2.2)$$

Step 2b. If $\hat{v}_i < v_i$, then set $v_i = \hat{v}_i$. If $i \notin \mathcal{C}$, add i to the candidate list: $\mathcal{C} = \mathcal{C} \cup \{i\}$ (i is assumed to be put at the bottom of the list).

Step 3. Drop node j from the candidate list. If the candidate list \mathcal{C} is not empty, return to step 1.

Figure 2.2 More efficient shortest path algorithm.

may find a betterw path from some node j , which is then added to the candidate list (if it is not already there). The algorithm is illustrated in Figure 2.2.

Almost any (deterministic) discrete dynamic program can be viewed as a shortest path problem. We can view each node i as representing a particular discrete state of the system. The origin node q is our starting state, and the ending state r might be any state at an ending time T . We can also have shortest path problems defined over infinite horizons, although we would typically include a discount factor.

2.1.2 The Discrete Budgeting Problem

Assume that we have to allocate a budget of size R to a series of tasks \mathcal{T} . Let a_t be a discrete action representing the amount of money allocated to task t , and let $C_t(a_t)$ be the contribution (or reward) that we receive from this allocation. We

would like to maximize our total contribution

$$\max_a \sum_{t \in T} C_t(a_t) \quad (2.3)$$

subject to the constraint on our available resources

$$\sum_{t \in T} a_t = R. \quad (2.4)$$

In addition we cannot allocate negative resources to any task, so we include

$$a_t \geq 0. \quad (2.5)$$

We refer to (2.3) through (2.5) as the *budgeting problem* (other authors refer to it as the “resource allocation problem,” a term we find too general for such a simple problem). In this example all data are deterministic. There are a number of algorithmic strategies for solving this problem that depend on the structure of the contribution function, but we are going to show how it can be solved without any assumptions.

We will approach this problem by first deciding how much to allocate to task 1, then to task 2, and so on, until the last task, T . In the end, however, we want a solution that optimizes over all tasks. Let

$$V_t(R_t) = \text{the value of having } R_t \text{ resources remaining to} \\ \text{allocate to task } t \text{ and later tasks.}$$

Implicit in our definition of $V_t(R_t)$ is that we are going to solve the problem of allocating R_t over tasks $t, t + 1, \dots, T$ in an optimal way. Imagine that we somehow know the function $V_{t+1}(R_{t+1})$. The relationship between R_{t+1} and R_t is given by

$$R_{t+1} = R_t - a_t. \quad (2.6)$$

In the language of dynamic programming, R_t is known as the *state variable*, which captures all the information we need to model the system forward in time (we provide a more careful definition in Chapter 5). Equation (2.6) is the *transition function* that relates the state at time t to the state at time $t + 1$. Sometimes we need to explicitly refer to the transition function (rather than just the state at time $t + 1$), in which case we use

$$R^M(R_t, a_t) = R_t - a_t. \quad (2.7)$$

Equation (2.7) is referred to in some communities as the *system model*, since it models the physics of the system over time (hence our use of the superscript M).

The relationship between $V_t(R_t)$ and $V_{t+1}(R_{t+1})$ is given by

$$V_t(R_t) = \max_{0 \leq a_t \leq R_t} (C_t(a_t) + V_{t+1}(R^M(R_t, a_t))). \quad (2.8)$$

Equation (2.8) is the optimality equation and represents the foundational equation for dynamic programming. It says that the value of having R_t resources for task t is the value of optimizing the contribution from task t plus the value of then having $R^M(R_t, a_t) = R_{t+1} = R_t - a_t$ resources for task $t + 1$ (and beyond). It forces us to balance the contribution from task t against the value that we would receive from all future tasks (which is captured in $V_{t+1}(R_t - a_t)$). One way to solve (2.8) is to assume that a_t is discrete. For example, if our budget is $R = \$10$ million, we might require a_t to be in units of \$100,000 dollars. In this case we would solve (2.8) simply by searching over all possible values of a_t (since it is a scalar, this is not too hard). The problem is that we do not know what $V_{t+1}(R_{t+1})$ is.

The simplest strategy for solving our dynamic program in (2.8) is to start by using $V_{T+1}(R) = 0$ (for any value of R). Then we would solve

$$V_T(R_T) = \max_{0 \leq a_T \leq R_T} C_T(a_T) \quad (2.9)$$

for $0 \leq R_T \leq R$. Now we know $V_T(R_T)$ for any value of R_T that might actually happen. Next we can solve

$$V_{T-1}(R_{T-1}) = \max_{0 \leq a_{T-1} \leq R_{T-1}} (C_{T-1}(a_{T-1}) + V_T(R_{T-1} - a_{T-1})). \quad (2.10)$$

Clearly, we can play this game recursively, solving (2.8) for $t = T-1, T-2, \dots, 1$. Once we have computed V_t for $t = (1, 2, \dots, T)$, we can then start at $t = 1$ and step forward in time to determine our optimal allocations.

This strategy is simple, easy, and optimal. It has the nice property that we do not need to make any assumptions about the shape of $C_t(a_t)$, other than finiteness. We do not need concavity or even continuity; we just need the function to be defined for the discrete values of a_t that we are examining.

2.1.3 The Continuous Budgeting Problem

It is usually the case that dynamic programs have to be solved numerically. In this section we introduce a form of the budgeting problem that can be solved analytically. Assume that the resources we are allocating are continuous (e.g., how much money to assign to various activities), which means that R_t is continuous, as is the decision of how much to budget. Throughout this book we use action a whenever we have a finite set of discrete actions. Often we face the problem of solving problems with continuous, and possibly vector-valued, decisions. For these problems we use the notation x that is standard in the field of mathematical programming. We are then going to assume that the contribution from allocating x_t dollars to task t is given by

$$C_t(x_t) = \sqrt{x_t}.$$

This function assumes that there are diminishing returns from allocating additional resources to a task, as is common in many applications. We can solve this problem

exactly using dynamic programming. We first note that if we have R_T dollars left for the last task, the value of being in this state is

$$V_T(R_T) = \max_{x_T \leq R_T} \sqrt{x_T}.$$

Since the contribution increases monotonically with x_T , the optimal solution is $x_T = R_T$, which means that $V_T(R_T) = \sqrt{R_T}$. Now consider the problem at time $t = T-1$. The value of being in state R_{T-1} would be

$$V_{T-1}(R_{T-1}) = \max_{x_{T-1} \leq R_{T-1}} (\sqrt{x_{T-1}} + V_T(R_T(x_{T-1}))), \quad (2.11)$$

where $R_T(x_{T-1}) = R_{T-1} - x_{T-1}$ is the money left over from time period $T-1$. Since we know $V_T(R_T)$, we can rewrite (2.11) as

$$V_{T-1}(R_{T-1}) = \max_{x_{T-1} \leq R_{T-1}} (\sqrt{x_{T-1}} + \sqrt{R_{T-1} - x_{T-1}}). \quad (2.12)$$

We solve (2.12) by differentiating with respect to x_{T-1} and setting the derivative equal to zero (we are taking advantage of the fact that we are maximizing a continuously differentiable, concave function). Let

$$F_{T-1}(R_{T-1}, x_{T-1}) = \sqrt{x_{T-1}} + \sqrt{R_{T-1} - x_{T-1}}.$$

Differentiating $F_{T-1}(R_{T-1}, x_{T-1})$ and setting this equal to zero gives

$$\begin{aligned} \frac{\partial F_{T-1}(R_{T-1}, x_{T-1})}{\partial x_{T-1}} &= \frac{1}{2}(x_{T-1})^{-1/2} - \frac{1}{2}(R_{T-1} - x_{T-1})^{-1/2} \\ &= 0. \end{aligned}$$

This implies that

$$x_{T-1} = R_{T-1} - x_{T-1},$$

which gives

$$x_{T-1}^* = \frac{1}{2}R_{T-1}.$$

We now have to find V_{T-1} . Substituting x_{T-1}^* back into (2.12) gives

$$\begin{aligned} V_{T-1}(R_{T-1}) &= \sqrt{\frac{R_{T-1}}{2}} + \sqrt{\frac{R_{T-1}}{2}} \\ &= 2\sqrt{\frac{R_{T-1}}{2}}. \end{aligned}$$

We can continue this exercise, but there seems to be a bit of a pattern forming (this is a common trick when trying to solve dynamic programs analytically). It seems that a general formula might be

$$V_{T-t+1}(R_{T-t+1}) = t \sqrt{\frac{R_{T-t+1}}{t}} \quad (2.13)$$

or, equivalently,

$$V_t(R_t) = (T - t + 1) \sqrt{\frac{R_t}{T - t + 1}}. \quad (2.14)$$

How do we determine if this guess is correct? We use a technique known as proof by induction. We assume that (2.13) is true for $V_{T-t+1}(R_{T-t+1})$ and then show that we get the same structure for $V_{T-t}(R_{T-t})$. Since we have already shown that it is true for V_T and V_{T-1} , this result would allow us to show that it is true for all t .

Finally, we can determine the optimal solution using the value function in equation (2.14). The optimal value of x_t is found by solving

$$\max_{x_t} \left(\sqrt{x_t} + (T - t) \sqrt{\frac{R_t - x_t}{T - t}} \right). \quad (2.15)$$

Differentiating and setting the result equal to zero gives

$$\frac{1}{2}(x_t)^{-1/2} - \frac{1}{2} \left(\frac{R_t - x_t}{T - t} \right)^{-1/2} = 0.$$

This implies that

$$x_t = \frac{R_t - x_t}{T - t}.$$

Solving for x_t gives

$$x_t^* = \frac{R_t}{T - t + 1}.$$

This gives us the very intuitive result that we want to evenly divide the available budget among all remaining tasks. This is what we would expect since all the tasks produce the same contribution.

2.2 STOCHASTIC PROBLEMS

Dynamic programming can be a useful algorithmic strategy for deterministic problems, but it is often an essential strategy for stochastic problems. In this section we illustrate a number of stochastic problems, with the goal of illustrating the challenge of modeling the flow of information (essential to any stochastic problem). We also make the transition from problems with fairly simple state variables to problems with extremely large state spaces.

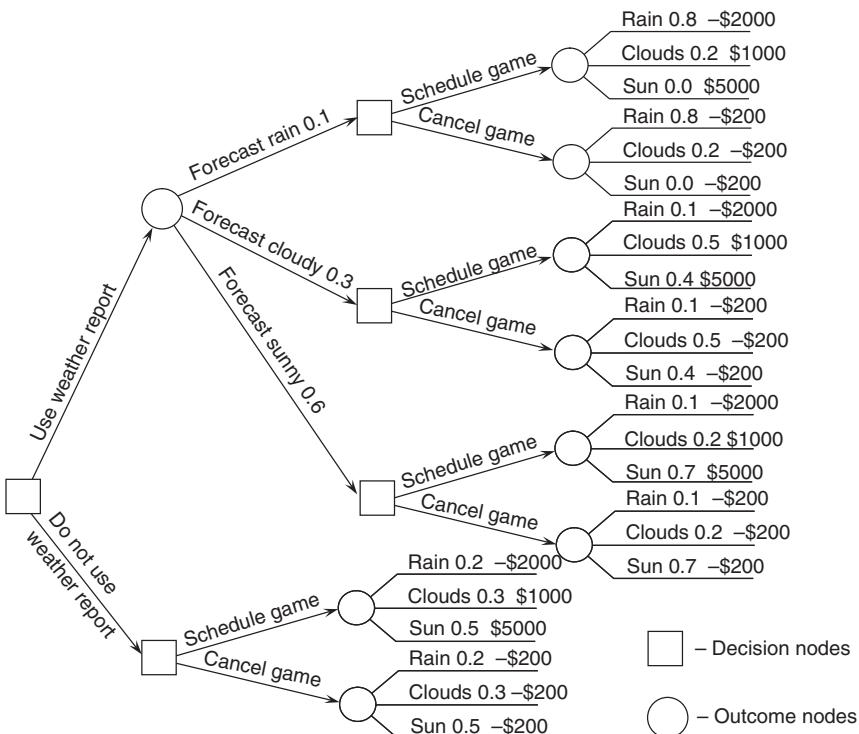


Figure 2.3 Decision tree showing decision nodes and outcome nodes.

2.2.1 Decision Trees

One of the most effective ways of communicating the process of making decisions under uncertainty is to use decision trees. Figure 2.3 illustrates a problem facing a Little League baseball coach trying to schedule a playoff game under the threat of bad weather. The coach first has to decide if he should check the weather report. Then he has to decide if he should schedule the game. Bad weather brings a poor turnout that reduces revenues from tickets and the concession stand. There are costs if the game is scheduled (umpires, food, people to handle parking and the concession stand) that need to be covered by the revenue the game might generate.

In Figure 2.3, squares denote decision nodes where we have to choose an action (Does he check the weather report? Does he schedule the game?), while circles represent outcome nodes where new (and random) information arrives (What will the weather report say? What will the weather be?). We can “solve” the decision tree (i.e., find the best decision given the information available) by rolling backward through the tree. In Figure 2.4a we find the expected value of being at each of the end outcome nodes. For example, if we check the weather report and see a forecast of rain, the probability that it will actually rain is 0.80, producing a loss of \$2000; the probability that it will be cloudy is 0.20, producing a profit of \$1000; the probability it will be sunny is zero (if it were sunny, we would make a profit

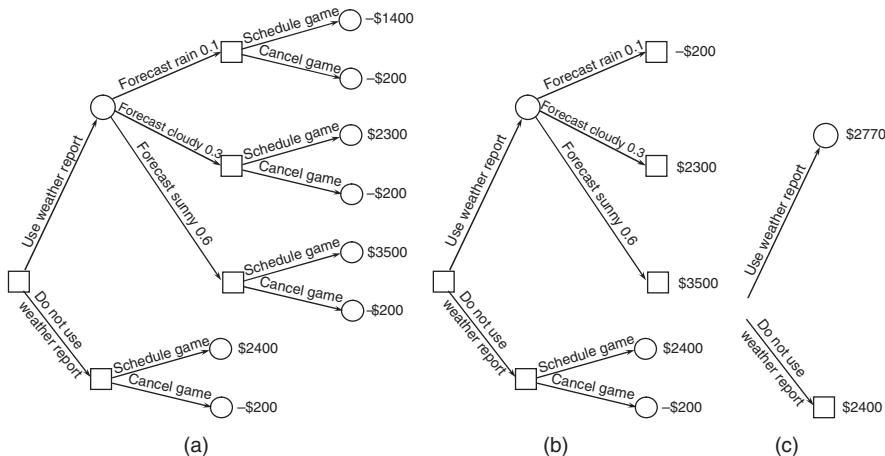


Figure 2.4 Evaluating a decision tree. (a) Evaluating the final outcome nodes. (b) Evaluating the final decision nodes. (c) Evaluating the first outcome nodes.

of \$5000). The expected value of scheduling the game, when the weather forecast is rain, is $(0.8)(-\$2000) + (0.2)(\$1000) + (0)(\$5000) = -\1400 . Repeating this calculation for each of the ending outcome nodes produces the results given in Figure 2.4a.

At a decision node we get to choose an action, and of course we choose the action with the highest expected profit. The results of this calculation are given in Figure 2.4b. Finally, we have to determine the expected value of checking the weather report by again multiplying the probability of each possible weather forecast (rainy, cloudy, sunny) times the expected value of each outcome. Thus the expected value of checking the weather report is $(0.1)(-\$200) + (0.3)(\$2300) + (0.6)(\$3500) = \2770 , shown in Figure 2.4c. The expected value of making decisions without the weather report is \$2400, so the analysis shows that we should check the weather report. Alternatively, we can interpret the result as telling us that we would be willing to pay up to \$300 for the weather report.

Almost any dynamic program with discrete states and actions can be modeled as a decision tree. The problem is that decision trees are not practical when there are a large number of states and actions.

2.2.2 A Stochastic Shortest Path Problem

We are often interested in shortest path problems where there is uncertainty in the cost of traversing a link. For our transportation example it is natural to view the travel time on a link as random, reflecting the variability in traffic conditions on each link. There are two ways we can handle the uncertainty. The simplest is to assume that our driver has to make a decision before seeing the travel time over

the link. In this case our updating equation would look like

$$v_i^n = \min_{j \in \mathcal{I}_i^+} \mathbb{E}\{c_{ij}(W) + v_j^{n-1}\},$$

where W is some random variable that contains information about the network (e.g., travel times). This problem is identical to our original problem; all we have to do is to let $c_{ij} = \mathbb{E}\{c_{ij}(W)\}$ be the expected cost on an arc.

An alternative model is to assume that we know the travel time on a link from i to j as soon as we arrive at node i . In this case we would have to solve

$$v_i^n = \mathbb{E} \left\{ \min_{j \in \mathcal{I}_i^+} (c_{ij}(W) + v_j^{n-1}) \right\}.$$

Here the expectation is outside of the min operator that chooses the best decision, capturing the fact that now the decision itself is random.

Note that our notation is ambiguous, in that with the same notation we have two very different models. In Chapter 5 we are going to refine our notation so that it will be immediately apparent when a decision “sees” the random information and when the decision has to be made before the information becomes available.

2.2.3 The Gambling Problem

A gambler has to determine how much of his capital he should bet on each round of a game, where he will play a total of N rounds. He will win a bet with probability p and lose with probability $q = 1-p$ (assume $q < p$). Let s^n be his total capital after n plays, $n = 1, 2, \dots, N$, with s^0 being his initial capital. For this problem we refer to s^n as the state of the system. Let a^n be the (discrete) amount he bets in round n , where we require that $a^n \leq s^{n-1}$. Our gambler wants to maximize $\ln s^N$ (this provides a strong penalty for ending up with a small amount of money at the end and a declining marginal value for higher amounts).

Let

$$W^n = \begin{cases} 1 & \text{if the gambler wins the } n\text{th game,} \\ 0 & \text{otherwise.} \end{cases}$$

The system evolves according to

$$S^n = S^{n-1} + a^n W^n - a^n(1 - W^n).$$

Let $V^n(S^n)$ be the value of having S^n dollars at the end of the n th game. The value of being in state S^n at the end of the n th round can be written

$$\begin{aligned} V^n(S^n) &= \max_{0 \leq a^{n+1} \leq S^n} \mathbb{E}\{V^{n+1}(S^{n+1})|S^n\} \\ &= \max_{0 \leq a^{n+1} \leq S^n} \mathbb{E}\{V^{n+1}(S^n + a^{n+1} W^{n+1} - a^{n+1}(1 - W^{n+1}))|S^n\}. \end{aligned}$$

Here we claim that the value of being in state S^n is found by choosing the decision that maximizes the expected value of being in state S^{n+1} given what we know at the end of the n th round.

We solve this by starting at the end of the N th trial, and assuming that we have finished with S^N dollars. The value of this is

$$V^N(S^N) = \ln S^N.$$

Now step back to $n = N-1$, where we may write

$$\begin{aligned} V^{N-1}(S^{N-1}) &= \max_{0 \leq a^N \leq S^{N-1}} \mathbb{E}\{V^N(S^{N-1} + a^N W^N - a^N(1 - W^N)) | S^{N-1}\} \\ &= \max_{0 \leq a^N \leq S^{N-1}} [p \ln(S^{N-1} + a^N) + (1-p) \ln(S^{N-1} - a^N)]. \end{aligned} \quad (2.16)$$

Let $V^{N-1}(S^{N-1}, a^N)$ be the value within the max operator. We can find a^N by differentiating $V^{N-1}(S^{N-1}, a^N)$ with respect to a^N , giving

$$\begin{aligned} \frac{\partial V^{N-1}(S^{N-1}, a^N)}{\partial a^N} &= \frac{p}{S^{N-1} + a^N} - \frac{1-p}{S^{N-1} - a^N} \\ &= \frac{2S^{N-1}p - S^{N-1} - a^N}{(S^{N-1})^2 - (a^N)^2}. \end{aligned}$$

Setting this equal to zero and solving for a^N gives

$$a^N = (2p - 1)S^{N-1}.$$

The next step is to plug this back into (2.16) to find $V^{N-1}(S^{N-1})$ using

$$\begin{aligned} V^{N-1}(S^{N-1}) &= p \ln(S^{N-1} + S^{N-1}(2p - 1)) + (1-p) \ln(S^{N-1} - S^{N-1}(2p - 1)) \\ &= p \ln(S^{N-1}2p) + (1-p) \ln(S^{N-1}2(1-p)) \\ &= p \ln S^{N-1} + (1-p) \ln S^{N-1} + \underbrace{p \ln(2p) + (1-p) \ln(2(1-p))}_K \\ &= \ln S^{N-1} + K, \end{aligned}$$

where K is a constant with respect to S^{N-1} . Since the additive constant does not change our decision, we may ignore it and use $V^{N-1}(S^{N-1}) = \ln S^{N-1}$ as our value function for $N-1$, which is the same as our value function for N . Not surprisingly, we can keep applying this same logic backward in time and obtain

$$V^n(S^n) = \ln S^n (+K^n)$$

for all n , where again, K^n is some constant that can be ignored. This means that for all n , our optimal solution is

$$a^n = (2p - 1)S^{n-1}.$$

The optimal strategy at each iteration is to bet a fraction $\beta = (2p - 1)$ of our current money on hand. Of course, this requires that $p > 0.5$.

2.2.4 Asset Valuation

Imagine that you are holding an asset that you can sell at a price that fluctuates randomly. In this problem we want to determine the best time to sell the asset and, from this, to infer the value of the asset. For this reason this type of problem arises frequently in the context of asset valuation and pricing.

Let \hat{p}_t be the price that is revealed in period t , at which point you have to make a decision:

$$a_t = \begin{cases} 1 & \text{sell.} \\ 0 & \text{hold.} \end{cases}$$

For our simple model we assume that \hat{p}_t is independent of prior prices (a more typical model would assume that the *change* in price is independent of prior history). With this assumption, our system has two states:

$$S_t = \begin{cases} 1 & \text{we are holding the asset,} \\ 0 & \text{we have sold the asset.} \end{cases}$$

Assume that we measure the state immediately after the price \hat{p}_t has been revealed but before we have made a decision. If we have sold the asset, then there is nothing we can do. We want to maximize the price we receive when we sell our asset. Let the scalar V_t be the value of holding the asset at time t . This can be written

$$V_t = \max_{a_t \in \{0, 1\}} (a_t \hat{p}_t + (1 - a_t) \gamma \mathbb{E} V_{t+1}).$$

So either we get the price \hat{p}_t if we sell, or we get the discounted future value of the asset. Assuming the discount factor $\gamma < 1$, we do not want to hold too long simply because the value in the future is worth less than the value now. In practice, we eventually will see a price \hat{p}_t that is greater than the future expected value, at which point we will stop the process and sell our asset.

The time at which we sell our asset is known as a *stopping time*. By definition, $a_\tau = 1$. It is common to think of τ as the decision variable, where we wish to solve

$$\max_{\tau} \mathbb{E} \hat{p}_{\tau}. \quad (2.17)$$

Equation (2.17) is a little tricky to interpret. Clearly, the choice of when to stop is a random variable since it depends on the price \hat{p}_t . We cannot optimally choose a random variable, so what is meant by (2.17) is that we wish to choose a *function* (or *policy*) that determines when we are going to sell. For example, we would expect that we might use a rule that says

$$A_t(S_t, \bar{p}) = \begin{cases} 1 & \text{if } \hat{p}_t \geq \bar{p} \text{ and } S_t = 1, \\ 0 & \text{otherwise.} \end{cases} \quad (2.18)$$

In this case we have a function parameterized by \bar{p} that allows us to write our problem in the form

$$\max_{\bar{p}} \mathbb{E} \sum_{t=1}^{\infty} \gamma^t A_t(S_t, \bar{p}).$$

This formulation raises two questions. First, while it seems very intuitive that our policy would take the form given in equation (2.18), there is the theoretical question of whether this in fact is the structure of an optimal policy. The second question is how to find the best policy within this class. For this problem, that means finding the parameter \bar{p} . For problems where the probability distribution of the random process driving prices is (assumed) known, this is a rich and deep theoretical challenge. Alternatively, there is a class of algorithms from stochastic optimization that allows us to find “good” values of the parameter in a fairly simple way.

2.2.5 The Asset Acquisition Problem I

A basic asset acquisition problem arises in applications where we purchase product at time t to be used during time interval $t + 1$. We are going to encounter this problem again, sometimes as discrete problems (where we would use action a), but often as continuous problems, and sometimes as vector-valued problems (when we have to acquire different types of assets). For this reason we use x as our decision variable.

We can model the problem using

R_t = assets on hand at time t before we make a new ordering decision, and before we have satisfied any demands arising in time interval t ;

x_t = amount of product purchased at time t to be used during time interval $t + 1$;

\hat{D}_t = random demands that arise between $t - 1$ and t .

We have chosen to model R_t as the resources on hand in period t before demands have been satisfied. Our definition here makes it easier to introduce (in the next section) the decision of how much demand we should satisfy.

We could purchase new assets at a fixed price p^p and sell them at a fixed price p^s . The amount we earn between $t - 1$ and t , including the decision we make at time t , is then given by

$$C_t(x_t) = p^s \min\{R_t, \hat{D}_t\} - p^p x_t.$$

Our inventory R_t is described using the equation

$$R_{t+1} = R_t - \min\{R_t, \hat{D}_t\} + x_t.$$

We assume that any unsatisfied demands are lost to the system.

This problem can be solved using Bellman's equation. For this problem, R_t is our state variable. Let $V_t(R_t)$ be the value of being in state R_t . Then Bellman's equation tells us that

$$V_t(R_t) = \max_{x_t} (C_t(x_t) + \gamma \mathbb{E} V_{t+1}(R_{t+1})).$$

where the expectation is over all the possible realizations of the demands \hat{D}_{t+1} .

2.2.6 The Asset Acquisition Problem II

Many asset acquisition problems introduce additional sources of uncertainty. The assets we are acquiring could be stocks, planes, energy commodities such as oil, consumer goods, and blood. In addition to the need to satisfy random demands (the only source of uncertainty we considered in our basic asset acquisition problem), we might have randomness in the prices at which we buy and sell assets. We may also include exogenous changes to the assets on hand due to additions (cash deposits, blood donations, energy discoveries) and subtractions (cash withdrawals, equipment failures, theft of product).

We can model the problem using

x_t^p = assets purchased (acquired) at time t to be used during time interval $t + 1$,

x_t^s = amount of assets sold to satisfy demands during time interval t ,

$x_t = (x_t^p, x_t^s)$,

R_t = resource level at time t before any decisions are made,

D_t = demands waiting to be served at time t .

Of course, we are going to require that $x_t^s \leq \min\{R_t, D_t\}$ (we cannot sell what we do not have, and we cannot sell more than the market demand). We are also going to assume that we buy and sell our assets at market prices that fluctuate over time. These are described using

p_t^p = market price for purchasing assets at time t ,

p_t^s = market price for selling assets at time t ,

$p_t = (p_t^s, p_t^p)$.

Our system evolves according to several types of exogenous information processes that include random changes to the supplies (assets on hand), demands, and prices. We model these using

\hat{R}_t = exogenous changes to the assets on hand that occur during time interval t ,

\hat{D}_t = demand for the resources during time interval t ,

\hat{p}_t^p = change in the purchase price that occurs between $t - 1$ and t ,

\hat{p}_t^s = change in the selling price that occurs between $t - 1$ and t ,

$\hat{p}_t = (\hat{p}_t^p, \hat{p}_t^s)$.

We assume that the exogenous changes to assets, \hat{R}_t , occurs before we satisfy demands.

For more complex problems such as this, it is convenient to have a generic variable for exogenous information. We use the notation W_t to represent all the information that first arrives between $t-1$ and t , where for this problem, we would have

$$W_t = (\hat{R}_t, \hat{D}_t, \hat{p}_t).$$

The state of our system is described by

$$S_t = (R_t, D_t, p_t).$$

We represent the evolution of our state variable generically using

$$S_{t+1} = S^M(S_t, x_t, W_{t+1}).$$

Some communities refer to this as the “system model,” hence our notation. We refer to this as the transition function. More specifically, the equations that make up our transition function would be

$$\begin{aligned} R_{t+1} &= R_t - x_t^s + x_t^p + \hat{R}_{t+1}, \\ D_{t+1} &= D_t - x_t^s + \hat{D}_{t+1}, \\ p_{t+1}^p &= p_t^p + \hat{p}_{t+1}^p, \\ p_{t+1}^s &= p_t^s + \hat{p}_{t+1}^s. \end{aligned}$$

The one-period contribution function is

$$C_t(S_t, x_t) = p_t^s x_t^s - p_t^p x_t.$$

We can find optimal decisions by solving Bellman’s equation

$$V_t(S_t) = \max \left(C_t(S_t, x_t) + \gamma \mathbb{E} V_{t+1}(S_{t+1}^M(S_t, x_t, W_{t+1})) | S_t \right). \quad (2.19)$$

This problem allows us to capture a number of dimensions of the modeling of stochastic problems. This is a fairly classical problem, but we have stated it in a more general way by allowing for unsatisfied demands to be held for the future, and by allowing for random purchasing and selling prices.

2.2.7 The Lagged Asset Acquisition Problem

A variation of the basic asset acquisition problem we introduced in Section 2.2.5 arises when we can purchase assets now to be used in the future. For example, a hotel might book rooms at time t for a date t' in the future. A travel agent might purchase space on a flight or a cruise line at various points in time before the trip actually happens. An airline might purchase contracts to buy fuel in the future. In

all these cases the assets purchased farther in advance will generally be cheaper, although prices may fluctuate. For this problem we are going to assume that selling prices are

$x_{tt'} = \text{assets purchased at time } t \text{ to be used to satisfy demands that become known during time interval between } t' - 1 \text{ and } t'$,

$$x_t = (x_{t,t+1}, x_{t,t+2}, \dots),$$

$$= (x_{tt'})_{t' > t},$$

$\hat{D}_t = \text{demand for the resources that become known during time interval } t,$

$R_{tt'} = \text{total assets acquired on or before time } t \text{ that may be used to satisfy demands that become known between } t' - 1 \text{ and } t',$

$$R_t = (R_{tt'})_{t' \geq t}.$$

Now, R_{tt} is the resources on hand in period t that can be used to satisfy demands \hat{D}_t that become known during time interval t . In this formulation we do not allow x_{tt} , which would represent purchases on the spot market. If this were allowed, purchases at time t could be used to satisfy unsatisfied demands arising during time interval between $t-1$ and t .

The transition function is given by

$$R_{t+1,t'} = \begin{cases} \left(R_{t,t} - \min(R_{t,t}, \hat{D}_t) \right) + x_{t,t+1} + R_{t,t+1}, & t' = t + 1, \\ R_{tt'} + x_{tt'}, & t' > t + 1. \end{cases}$$

The one-period contribution function (measuring forward in time) is

$$C_t(R_t, \hat{D}_t) = p^s \min(R_{t,t}, \hat{D}_t) - \sum_{t' > t} p^p x_{tt'}.$$

We can again formulate Bellman's equation as in (2.19) to determine an optimal set of decisions. From a computational perspective, however, there is a critical difference. Now x_t and R_t are vectors with elements $x_{tt'}$ and $R_{tt'}$, which makes it computationally impossible to enumerate all possible states (or actions).

2.2.8 The Batch Replenishment Problem

One of the classical problems in operations research is one that we refer to here as the batch replenishment problem. To illustrate the basic problem, assume that we have a single type of resource that is consumed over time. As the reserves of the resource run low, it is necessary to replenish the resources. In many problems there are economies of scale in this process. It is cheaper (on an average cost basis) to increase the level of resources in one jump (see examples).

■ EXAMPLE 2.1

A startup company has to maintain adequate reserves of operating capital to fund product development and marketing. As the cash is depleted, the finance

officer has to go to the markets to raise additional capital. There are fixed costs of raising capital, so this tends to be done in batches. ■

■ EXAMPLE 2.2

An oil company maintains an aggregate level of oil reserves. As these are depleted, it will undertake exploration expeditions to identify new oil fields, which will produce jumps in the total reserves under the company's control. ■

To introduce the core elements, let

\hat{D}_t = demand for the resources during time interval t ,

R_t = resource level at time t ,

x_t = additional resources acquired at time t to be used during time interval $t + 1$.

The transition function is given by

$$R_{t+1}^M(R_t, x_t, \hat{D}_{t+1}) = \max\{0, (R_t + x_t - \hat{D}_{t+1})\}.$$

Our one period cost function (which we wish to minimize) is given by

$$\begin{aligned} \hat{C}_{t+1}(R_t, x_t, \hat{D}_{t+1}) &= \text{total cost of acquiring } x_t \text{ units of the resource} \\ &= c^f I_{\{x_t > 0\}} + c^p x_t + c^h R_{t+1}^M(R_t, x_t, \hat{D}_{t+1}), \end{aligned}$$

where

c^f = fixed cost of placing an order,

c^p = unit purchase cost,

c^h = unit holding cost.

For our purposes, $\hat{C}_{t+1}(R_t, x_t, \hat{D}_{t+1})$ could be any nonconvex function; this is a simple example of one. Since the cost function is nonconvex, it helps to order larger quantities at the same time.

Suppose that we have a family of decision functions $X^\pi(R_t)$, $\pi \in \Pi$, for determining x_t . For example, we might use a decision rule such as

$$X^\pi(R_t) = \begin{cases} 0 & \text{if } R_t \geq s, \\ Q - R_t & \text{if } R_t < q. \end{cases}$$

where Q and q are specified parameters. In the language of dynamic programming, a decision rule such as $X^\pi(R_t)$ is known as a *policy* (literally, a rule for making decisions). We index policies by π , and denote the set of policies by Π . In this example a combination (S, s) represents a policy, and Π would represent all the possible values of Q and q .

Our goal is to solve

$$\min_{\pi \in \Pi} \mathbb{E} \left\{ \sum_{t=0}^T \gamma^t \hat{C}_{t+1}(R_t, X^\pi(R_t), \hat{D}_{t+1}) \right\}.$$

This means that we want to search over all possible values of Q and q to find the best performance (on average).

The basic batch replenishment problem, where R_t and x_t are scalars, is quite easy (if we know things like the distribution of demand). But there are many real problems where these are vectors because there are different types of resources. The vectors may be small (different types of fuel, raising different types of funds) or extremely large (hiring different types of people for a consulting firm or the military; maintaining spare parts inventories). Even a small number of dimensions would produce a very large problem using a discrete representation.

2.2.9 The Transformer Replacement Problem

The electric power industry uses equipment known as transformers to convert the high-voltage electricity that comes out of power-generating plants into currents with successively lower voltage, finally delivering the current we can use in our homes and businesses. The largest of these transformers can weigh 200 tons, might cost millions of dollars to replace, and may require a year or more to build and deliver. Failure rates are difficult to estimate (the most powerful transformers were first installed in the 1960s and have yet to reach the end of their natural lifetime). Actual failures can be very difficult to predict, as they often depend on heat, power surges, and the level of use.

We are going to build an aggregate replacement model where we only capture the age of the transformers. Let

$$r = \text{age of a transformer (in units of time periods) at time } t,$$

$$R_{tr} = \text{number of active transformers of age } r \text{ at time } t.$$

Here and elsewhere, we need to model the attributes of a resource (in this case, the age). While “ a ” might be the obvious notation, this conflicts with our notation for actions. Instead, we use “ r ” for the attributes of a resource, which can be a scalar or, in other applications, a vector.

For our model we assume that age is the best predictor of the probability that a transformer will fail. Let

$$\hat{R}_{tr} = \text{number of transformers of age } r \text{ that fail between } t - 1 \text{ and } t,$$

$$p_r = \text{probability a transformer of age } r \text{ will fail between } t - 1 \text{ and } t.$$

Of course, \hat{R}_{tr} depends on R_{tr} since transformers can only fail if we own them.

It can take a year or two to acquire a new transformer. Assume that we are measuring time, and therefore age, in fractions of a year (e.g., three months). Normally it can take about six time periods from the time of purchase before a transformer is installed in the network. However, we may pay extra and get a new transformer in as little as three quarters. If we purchase a transformer that arrives in six time periods, then we might say that we have acquired a transformer that is $r = -6$ time periods old. Paying extra gets us a transformer that is $r = -3$ time

periods old. Of course, the transformer is not productive until it is at least $r = 0$ time periods old. Let

$$\begin{aligned}x_{tr} &= \text{number of transformers of age } r \text{ that we purchase at time } t, \\c_r &= \text{cost of purchasing a transformer of age } r.\end{aligned}$$

If we have too few transformers, then we incur what are known as “congestion costs,” which represent the cost of purchasing power from more expensive utilities because of bottlenecks in the network. To capture this, let

$$\begin{aligned}\bar{R} &= \text{target number of transformers that we should have available,} \\R_t^A &= \text{actual number of transformers that are available at time } t, \\&= \sum_{r \geq 0} R_{tr}, \\C_t(R_t^A, \bar{R}) &= \text{expected congestion costs if } R_t^A \text{ transformers are available,} \\&= c_0 (\bar{R}/R_t^A)^\beta.\end{aligned}$$

The function $C_t(R_t^A, \bar{R})$ captures the behavior that as R_t^A falls below \bar{R} , the congestion costs rise quickly.

Assume that x_{tr} is determined immediately after R_{tr} is measured. The transition function is given by

$$R_{t+1,r} = R_{t,r-1} + x_{t,r-1} - \hat{R}_{t+1,r}.$$

Let R_t , \hat{R}_t , and x_t be vectors with components R_{tr} , \hat{R}_{tr} , and x_{tr} , respectively. We can write our system dynamics more generally as

$$R_{t+1} = R^M(R_t, x_t, \hat{R}_{t+1}).$$

If we let $V_t(R_t)$ be the value of having a set of transformers with an age distribution described by R_t , then, as previously, we can write this value using Bellman’s equation

$$V_t(R_t) = \min_{x_t} (cx_t + \mathbb{E} V_{t+1}(R^M(R_t, x_t, \hat{R}_{t+1}))).$$

For this application, our state variable R_t might have as many as 100 dimensions. If we have, say, 200 transformers, each of which might be as many as 100 years old, then the number of possible values of R_t could be 100^{200} . Fortunately we can develop continuous approximations that allow us to approximate problems such as this relatively easily.

2.2.10 The Dynamic Assignment Problem

Consider the challenge of managing a group of technicians that help with the installation of expensive medical devices (e.g., medical imaging equipment). As hospitals install this equipment, they need technical assistance with getting the

machines running and training hospital personnel. The technicians may have different skills (some may be trained to handle specific types of equipment), and as they travel around the country, we may want to keep track not only of their current location but also how long they have been on the road. We describe the attributes of a technician using

$$r_t = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \end{pmatrix} = \begin{pmatrix} \text{Location of the technician} \\ \text{Type of equipment the technician is trained to handle} \\ \text{Number of days the technician has been on the road} \end{pmatrix}.$$

Since we have more than one technician, we can model the set of all technicians using

$$R_{tr} = \text{number of technicians with attribute } r,$$

$$\mathcal{R} = \text{set of all possible values of the attribute vector } r,$$

$$R_t = (R_{tr})_{r \in \mathcal{R}}.$$

Over time, demands arise for technical services as the equipment is installed. Let

$$b = \text{characteristics of a piece of equipment (location, type of equipment)},$$

$$\mathcal{B} = \text{set of all possible values of the vector } b,$$

$$\hat{D}_{tb} = \text{number of new pieces of equipment of type } b \text{ that were installed between } t - 1 \text{ and } t \text{ (and now need service)},$$

$$\hat{D}_t = (\hat{D}_{tb})_{b \in \mathcal{B}},$$

$$D_{tb} = \text{total number of pieces of equipment of type } b \text{ that still need to be installed at time } t,$$

$$D_t = (D_{tb})_{b \in \mathcal{B}}.$$

We next have to model the decisions that we have to make. Assume that at any point in time, we can either assign a technician to handle a new installation or send the technician home. Let

$$\mathcal{D}^H = \text{set of decisions to send a technician home, where } d \in \mathcal{D}^H \text{ represents a particular location};$$

$$\mathcal{D}^D = \text{set of decisions to have a technician serve a demand, where } d \in \mathcal{D} \text{ represents a decision to serve a demand of type } b_d;$$

$$d^\phi = \text{decision to "do nothing" with a technician};$$

$$\mathcal{D} = \mathcal{D}^H \cup \mathcal{D}^D \cup d^\phi.$$

A decision has the effect of changing the attributes of a technician, as well as possibly satisfying a demand. The impact on the resource attribute vector of a

technician is captured using the resource attribute transition function, represented using

$$r_{t+1} = r^M(a_t, d).$$

For algebraic purposes, it is useful to define the indicator function

$$\delta_{r'}(r_t, d) = \begin{cases} 1 & \text{for } r^M(r_t, d) = r', \\ 0 & \text{otherwise.} \end{cases}$$

A decision $d \in \mathcal{D}^D$ means that we are serving a piece of equipment described by an attribute vector b_d . This is only possible, of course, if $D_{tb} > 0$. Typically D_{tb} will be 0 or 1, although our model will allow multiple pieces of equipment with the same attributes. We indicate which decisions we have made using

x_{trd} = number of times we apply a decision of type d to a technician with attribute r ,

$$x_t = (x_{trd})_{r \in \mathcal{R}, d \in \mathcal{D}}.$$

Similarly we define the cost of a decision to be

c_{trd} = cost of applying a decision of type d to a technician with attribute r ,

$$c_t = (c_{trd})_{r \in \mathcal{R}, d \in \mathcal{D}}.$$

We could solve this problem myopically by making what appears to be the best decisions now, ignoring their impact on the future. We would do this by solving

$$\min_{x_t} \sum_{r \in \mathcal{R}} \sum_{d \in \mathcal{D}} c_{trd} x_{trd} \quad (2.20)$$

subject to

$$\sum_{d \in \mathcal{D}} x_{trd} = R_{tr}, \quad (2.21)$$

$$\sum_{r \in \mathcal{R}} x_{trd} \leq D_{tb_d}, \quad d \in \mathcal{D}^D, \quad (2.22)$$

$$x_{trd} \geq 0. \quad (2.23)$$

Equation (2.21) says that we have to either send a technician home or assign him to a job. Equation (2.22) says that we can only assign a technician to a job of type b_d if there is in fact a job of type b_d . Said differently, we cannot assign more than one technician to a job. But we do not have to assign a technician to every job (we may not have enough technicians).

The problem posed by equations (2.20) to (2.23) is a linear program. Real problems may involve managing hundreds or even thousands of individual entities. The decision vector $x_t = (x_{trd})_{r \in \mathcal{R}, d \in \mathcal{D}}$ may have over ten thousand dimensions.

But commercial linear programming packages handle problems of this size quite easily.

If we make decisions by solving (2.20) to (2.23), we say that we are using a *myopic policy* since we are using only what we know now, and we are ignoring the impact of decisions now on the future. For example, we may decide to send a technician home rather than have him sit in a hotel room waiting for a job. But this ignores the likelihood that another job may suddenly arise close to the technician's current location. Alternatively, we may have two different technicians with two different skill sets. If we only have one job, we might assign what appears to be the closest technician, ignoring the fact that this technician has specialized skills that are best reserved for difficult jobs.

Given a decision vector, the dynamics of our system can be described using

$$R_{t+1,r} = \sum_{r' \in \mathcal{R}} \sum_{d \in \mathcal{D}} x_{tr'd} \delta_r(r', d), \quad (2.24)$$

$$D_{t+1,b_d} = D_{t,b_d} - \sum_{r \in \mathcal{R}} x_{trd} + \hat{D}_{t+1,b_d}, \quad d \in \mathcal{D}^D. \quad (2.25)$$

Equation (2.24) captures the effect of all decisions (including serving demands) on the attributes of a technician. This is easiest to visualize if we assume that all tasks are completed within one time period. If this is not the case, then we simply have to augment the state vector to capture the attribute that we have partially completed a task. Equation (2.25) subtracts from the list of available demands any of type b_d that are served by a decision $d \in \mathcal{D}^D$ (recall that each element of \mathcal{D}^D corresponds to a type of task, which we denote b_d).

The state of our system is given by

$$S_t = (R_t, D_t).$$

The evolution of our state variable over time is determined by equations (2.24) and (2.25). We can now set up an optimality recursion to determine the decisions that minimize costs over time using

$$V_t = \min_{x_t \in \mathcal{X}_t} (C_t(S_t, x_t) + \gamma \mathbb{E} V_{t+1}(S_{t+1})),$$

where S_{t+1} is the state at time $t + 1$ given that we are in state S_t and action x_t . S_{t+1} is random because at time t , we do not know \hat{D}_{t+1} . The feasible region \mathcal{X}_t is defined by equations (2.21) to (2.23).

Needless to say, the state variable for this problem is quite large. The dimensionality of R_t is determined by the number of attributes of our technician, while the dimensionality of D_t is determined by the relevant attributes of a demand. In real applications these attributes can become fairly detailed. Fortunately the methods of approximate dynamic programming can handle these complex problems.

2.3 INFORMATION ACQUISITION PROBLEMS

Information acquisition is an important problem in many applications where we face uncertainty about the value of an action, but the only way to obtain better estimates of the value is to take the action. For example, a baseball manager may not know how well a particular player will perform at the plate. The only way to find out is to put him in the lineup and let him hit. The only way a mutual fund can learn how well a manager will perform may be to let her manage a portion of the portfolio. A pharmaceutical company does not know how the market will respond to a particular pricing strategy. The only way to learn is to offer the drug at different prices in test markets.

Information acquisition plays a particularly important role in approximate dynamic programming. Assume that a system is in state i and that a particular action might bring the system to state j . We may know the contribution of this decision, but we do not know the value of being in state j (although we may have an estimate). The only way to learn is to try making the decision and then obtain a better estimate of being in state j by actually visiting the state. This process of approximating a value function, which we introduce in Chapter 4, is fundamental to approximate dynamic programming. For this reason the information acquisition problem is of special importance, even in the context of solving classical dynamic programs.

The information acquisition problem introduces a new dimension to our thinking about dynamic programs. In all the examples we have considered in this chapter, the state of our system is the state of the resources we are managing. In the information acquisition problem, our state variable has to also include our estimates of unknown parameters. We illustrate this idea in the examples that follow.

2.3.1 The Bandit Problem

The classic information acquisition problem is known as the *bandit problem*. Consider the situation faced by a gambler trying to choose which of K slot machines to play. Now assume that the probability of winning may be different for each machine, but the gambler does not know what these probabilities are. The only way to obtain information is to actually play a slot machine. To formulate this problem, let

$$x_k^n = \begin{cases} 1 & \text{if we choose to play the } k\text{th slot machine in the } n\text{th trial,} \\ 0 & \text{otherwise.} \end{cases}$$

W_k^n = winnings from playing the k th slot machine during the n th trial.

\bar{w}_k^n = our estimate of the expected winnings from playing the k th slot machine after the n th trial.

$(\tilde{s}_k^2)^n$ = our estimate of the variance of the winnings from playing the k th slot machine after the n th trial.

N_k^n = number of times we have played the k th slot machine after n trials.

If $x_k^n = 1$, then we observe W_k^n and can update \bar{w}_k^n and $(\tilde{s}_k^2)^n$ using

$$\bar{w}_k^n = \left(1 - \frac{1}{N_k^n}\right) \bar{w}_k^{n-1} + \frac{1}{N_k^n} W_k^n, \quad (2.26)$$

$$(\tilde{s}_k^2)^n = \frac{N_k^n - 2}{N_k^n - 1} (\tilde{s}_k^2)^{n-1} + \frac{1}{N_k^n} (W_k^n - \bar{w}_k^{n-1})^2. \quad (2.27)$$

Equations (2.26) and (2.27) are equivalent to computing averages and variances from sample observations. Let \mathcal{N}_k^n be the iterations where $x_k^n = 1$. Then we can write the mean and variance using

$$\begin{aligned} \bar{w}_k^n &= \frac{1}{N_k^n} \sum_{m \in \mathcal{N}_k^n} W_k^m, \\ (\tilde{s}_k^2)^n &= \frac{1}{N_k^n - 1} \sum_{m \in \mathcal{N}_k^n} (W_k^m - \bar{w}_k^n)^2. \end{aligned}$$

The recursive equations (2.26) and (2.27), make the transition from state $(\bar{w}_k^{n-1}, (\tilde{s}_k^2)^{n-1})$ to $(\bar{w}_k^n, (\tilde{s}_k^2)^n)$ more transparent.

All other estimates remain unchanged. For this system the “state” of our system (measured after the n th trial) is

$$S^n = (\bar{w}_k^n, (\tilde{s}_k^2)^n, N_k^n)_{k=1}^K.$$

We have to keep track of N_k^n for each bandit k since these are needed in the updating equations.

It is useful to think of S^n as our “state of knowledge” since it literally captures what we know about the system (given by \bar{w}_k^n) and how well we know it (given by $(\tilde{s}_k^2)^n$). Some authors call this the information state, or the “hyperstate.” Equations (2.26) and (2.27) represent the transition equations that govern how the state evolves over time. Note that the state variable has $2K$ elements, which are also continuous (if the winnings are integer, S^n takes on discrete outcomes, but this is of little practical value).

This is a pure information acquisition problem. Normally we would choose to play the machine with the highest expected winnings. To express this we would write

$$x^n = \arg \max_k \bar{w}_k^{n-1}.$$

Here “ $\arg \max$ ” means the value of the decision variable (in this case k) that maximizes the problem. We set x^n equal to the value of k that corresponds to the largest value of \bar{w}_k^{n-1} (ties are broken arbitrarily). This rule ignores the possibility that our *estimate* of the expected winnings for a particular machine might be wrong.

Since we use these estimates to help us make better decisions, we might want to try a machine where the current estimated winnings are lower, because we might obtain information that indicates that the true mean might actually be higher.

The problem can be solved, in theory, using Bellman's equation

$$V^n(S^n) = \max_x \mathbb{E} \left(\sum_k W_k^{n+1} x_k + V^{n+1}(S^{n+1}) | S^n \right) \quad (2.28)$$

subject to $\sum_k x_k = 1$. Equation (2.28) is hard to compute since the state variable is continuous. Also we do not know the distribution of the random variables W^{n+1} (if we knew the distributions, then we would know the expected reward for each bandit). However, the estimates of the mean and variance, specified by the state S^n , allow us to infer the distribution of possible means (and variances) of the true distribution.

There are numerous examples of information acquisition problems. The examples provide some additional illustrations of bandit problems. These problems can be solved optimally using something known as an *index policy*. In the case of the bandit problem, it is possible to compute a single index for each bandit. The best decision is to then choose the bandit with the highest index. The value of this result is that we can solve a K -dimensional problem as K one-dimensional problems. See Chapter 12 for more on this topic.

■ EXAMPLE 2.3

Consider someone who has just moved to a new city and who now has to find the best path to work. Let T_p be a random variable giving the time he will experience if he chooses path p from a predefined set of paths \mathcal{P} . The only way he can obtain observations of the travel time is to actually travel the path. Of course, he would like to choose the path with the shortest average time, but it may be necessary to try a longer path because it may be that he simply has a poor estimate. The problem is identical to our bandit problem if we assume that driving one path does not teach us anything about a different path (this is a richer form of bandit problem). ■

■ EXAMPLE 2.4

A baseball manager is trying to decide which of four players makes the best designated hitter. The only way to estimate how well they hit is to put them in the batting order as the designated hitter. ■

■ EXAMPLE 2.5

A couple that has acquired some assets over time is looking to find a good money manager who will give them a good return without too much risk. The only way to determine how well a money manager performs is to have him actually manage the money for a period of time. The challenge then is

determining whether to stay with the money manager or to switch to a new money manager. ■

■ EXAMPLE 2.6

A doctor is trying to determine the best blood pressure medication for a patient. Each patient responds differently to each medication, so it is necessary to try a particular medication for a while, and then switch if the doctor feels that better results can be achieved with a different medication. ■

2.3.2 An Information-Collecting Shortest Path Problem

Now suppose that we have to choose a path through a network, just as we did in Sections 2.1.1 and 2.2.2, but this time we face the problem that we not only do not know the actual travel time on any of the links of the network, we do not even know the mean or variance (we might be willing to assume that the probability distribution is normal). As with the two previous examples, we solve the problem repeatedly, and sometimes we want to try new paths just to collect more information.

There are two significant differences between this simple problem and the two previous problems. First, imagine that you are at a node i and you are trying to decide whether to follow the link from i to j_1 or from i to j_2 . We have an estimate of the time to get from j_1 and j_2 to the final destination. These estimates may be correlated because they may share common links to the destination. Following the path from j_1 to the destination may teach us something about the time to get from j_2 to the destination (if the two paths share common links). The second difference is that making the decision to go from node i to node j changes the set of options that we face. In the bandit problem we always faced the same set of slot machines.

Information-collecting shortest path problems arise in any information collection problem where the decision now affects not only the information you collect but also the decisions you can make in the future. While we can solve basic bandit problems optimally, this broader problem class remains unsolved.

2.4 A SIMPLE MODELING FRAMEWORK FOR DYNAMIC PROGRAMS

Now that we have covered a number of simple examples, it is useful to briefly review the elements of a dynamic program. We are going to revisit this topic in considerably greater depth in Chapter 5, but this discussion provides a brief introduction. Our presentation focuses on *stochastic* dynamic programs that exhibit a flow of uncertain information. These problems, at a minimum, consist of the following elements:

State variable. This captures all the information we need to make a decision, as well as the information that we need to describe how the system evolves over time.

Decision variable. Decisions/actions represent how we control the process.

Exogenous information. This is data that first become known each time period (e.g., the demand for product, or the price at which it can be purchased or sold). In addition we have to be told the initial state of our system.

Transition function. This function determines how the system evolves from the state S_t to the state S_{t+1} given the decision that was made at time t and the new information that arrived between t and $t + 1$.

Objective function. This function specifies the costs being minimized, or the contributions/rewards being maximized, over a time horizon.

We can illustrate these elements using the simple asset acquisition problem from Section 2.2.6.

The state variable is the information we need to make a decision and compute functions that determine how the system evolves into the future. In our asset acquisition problem, we need three pieces of information. The first is R_t , the resources on hand before we make any decisions (including how much of the demand to satisfy). The second is the demand itself, denoted D_t , and the third is the price p_t . We would write our state variable as $S_t = (R_t, D_t, p_t)$.

We have two decisions to make. The first, denoted x_t^D , is how much of the demand D_t during time interval t that should be satisfied using available assets, which means that we require $x_t^D \leq R_t$. The second, denoted x_t^O , is how many new assets should be acquired at time t , which can be used to satisfy demands during time interval $t + 1$.

The exogenous information process consists of three types of information. The first is the new demands that arise during time interval t , denoted \hat{D}_t . The second is the change in the price at which we can sell our assets, denoted \hat{p}_t . Finally, we are going to assume that there may be exogenous changes to our available resources. These might be blood donations or cash deposits (producing positive changes), or equipment failures and cash withdrawals (producing negative changes). We denote these changes by \hat{R}_t . We often use a generic variable W_t to represent all the new information that is first learned during time interval t , which for our problem would be written $W_t = (\hat{R}_t, \hat{D}_t, \hat{p}_t)$. In addition to specifying the types of exogenous information, for stochastic models we also have to specify the likelihood of a particular outcome. This might come in the form of an assumed probability distribution for \hat{R}_t , \hat{D}_t , and \hat{p}_t , or we may depend on an exogenous source for sample realizations (the actual price of the stock or the actual travel time on a path).

Once we have determined what action we are going to take from our decision rule, we compute our contribution $C_t(S_t, a_t)$, which might depend on our current state and the action a_t (or x_t if we have continuous or vector-valued decisions) that we take at time t . For our asset acquisition problem (where the state variable is R_t) the contribution function is

$$C_t(S_t, a_t) = p_t a_t^D - c_t a_t^O.$$

In this particular model $C_t(S_t, a_t)$ is a deterministic function of the state and action. In other applications the contribution from action a_t depends on what happens during time $t + 1$.

Next we have to specify how the state variable changes over time. This is done using a *transition function*, which we might represent in a generic way as

$$S_{t+1} = S^M(S_t, a_t, W_{t+1}),$$

where S_t is the state at time t , a_t is the decision we made at time t , and W_{t+1} is our generic notation for the information that arrives between t and $t + 1$. We use the notation $S^M(\cdot)$ to denote the transition function, where the superscript M stands for “model” (or “system model” in recognition of vocabulary that has been in place for many years in the engineering community). The transition function for our asset acquisition problem is given by

$$\begin{aligned} R_{t+1} &= R_t - a_t^D + a_t^O + \hat{R}_{t+1}, \\ D_{t+1} &= D_t - a_t^D + \hat{D}_{t+1}, \\ p_{t+1} &= p_t + \hat{p}_{t+1}. \end{aligned}$$

This model assumes that unsatisfied demands are held until the next time period.

Our final step in formulating a dynamic program is to specify the objective function. Suppose that we are trying to maximize the total contribution received over a finite horizon $t = (0, 1, \dots, T)$. If we were solving a deterministic problem, we might formulate the objective function as

$$\max_{(a_t)_{t=0}^T} \sum_{t=0}^T C_t(S_t, a_t). \quad (2.29)$$

We would have to optimize (2.29) subject to a variety of constraints on the actions (a_0, a_1, \dots, a_T) .

If we have a stochastic problem, which is to say that there are a number of possible realizations of the exogenous information process $(W_t)_{t=0}^T$, then we have to formulate the objective function in a different way. If the exogenous information process is uncertain, we do not know which state we will be in at time t . Since the state S_t is a random variable, then the choice of decision (which depends on the state) is also a random variable.

We get around this problem by formulating the objective in terms of finding the best *policy* (or decision rule) for choosing decisions. A policy tells us what to do for all possible states, so regardless of which state we find ourselves in at some time t , the policy will tell us what decision to make. This policy must be chosen to produce the best *expected* contribution over all outcomes. If we let $A^\pi(S_t)$ be a particular decision rule indexed by π , and let Π be a set of decision rules, then the problem of finding the best policy would be written

$$\max_{\pi \in \Pi} \mathbb{E} \sum_{t=0}^T C_t(S_t, A^\pi(S_t)). \quad (2.30)$$

Exactly what is meant by finding the best policy out of a set of policies is very problem specific. Our decision rule might be to order $A^\pi(R_t) = S - R_t$ if $R_t < s$

and order $A^\pi(R_t) = 0$ if $R_t \geq s$. The family of policies is the set of all values of the parameters (s, S) for $s < S$ (here s and S are parameters to be determined, not state variables). If we are selling an asset, we might adopt a policy of selling if the price of the asset p_t falls below some value \bar{p} . The set of all policies is the set of all values of \bar{p} . However, policies of this sort tend to work only for very special problems.

Equation (2.30) states our problem as one of finding the best policy (or decision rule, or function X^π) to maximize the expected value of the total contribution over our horizon. There are a number of variations of this objective function. For applications where the horizon is long enough to affect the time value of money, we might introduce a discount factor γ and solve

$$\max_{\pi \in \Pi} \mathbb{E} \sum_{t=0}^T \gamma^t C_t(S_t, A^\pi(S_t)). \quad (2.31)$$

There is also considerable interest in infinite horizon problems of the form

$$\max_{\pi \in \Pi} \mathbb{E} \sum_{t=0}^{\infty} \gamma^t C_t(S_t, A^\pi(S_t)). \quad (2.32)$$

Equation (2.32) is often used when we want to study the behavior of a system in steady state.

Equations such as (2.30), (2.31), and (2.32) are all easy to write on a sheet of paper. Solving them computationally is a different matter. That challenge is the focus of this book.

A complete specification of a dynamic program requires that we specify both data and functions, as follows:

Data.

- The initial state S_0 .
- The exogenous information process W_t . We need to know what information is arriving from outside the system, and how it is being generated. For example, we might be given the probability of an outcome, or given a process that creates the data for us.

Functions.

- The contribution function $C(S_t, a_t)$. This may be specified in the form $C(S_t, a_t, W_{t+1})$. We may need to compute the expectation, or we may work directly with this form of the contribution function.
- The transition function $S^M(S_t, a_t, W_{t+1})$.
- The family of decision functions $(A^\pi(S))_{\pi \in \Pi}$.

This description provides only a taste of the richness of sequential decision processes. Chapter 5 describes the different elements of a dynamic program in far greater detail.

2.5 BIBLIOGRAPHIC NOTES

Most of the problems in this chapter are fairly classic, in particular the deterministic and stochastic shortest path problems (see Bertsekas et al. (1991)), asset acquisition problem (e.g., see Porteus, 1990) and the batch replenishment problem (e.g., see Puterman 2005).

Section 2.1.1 The shortest path problem is one of the most widely studied problems in optimization. One of the early treatments of shortest paths is given in the seminal book on network flows by Ford and Fulkerson (1962). It has long been recognized that shortest paths could be solved directly (if inefficiently) using Bellman's equation.

Section 2.2.2 Many problems in discrete stochastic dynamic programming can at least conceptually be formulated as some form of stochastic shortest path problem. There is an extensive literature on stochastic shortest paths (e.g., see Frank 1969; Sigal et al., 1980; Frieze and Grimmet, 1985; Andreatta and Romeo, 1988; Psaraftis and Tsitsiklis, 1993; Bertsekas et al., 1991).

Section 2.2.10 The dynamic assignment problem is based on Spivey and Powell, (2004).

Section 2.3.1 Bandit problems have long been studied as classic exercises in information collection. For good introductions to this material, see Ross (1983) and Whittle (1982). A more detailed discussion of bandit problems is given in Chapter 12.

PROBLEMS

- 2.1** Give an example of a sequential decision process from your own experience. Describe the elements of your problem following the framework provided in Section 2.4. Then describe the types of rules you might use to make a decision.
- 2.2** What is the state variable at each node in the decision tree in Figure 2.3?
- 2.3** Describe the gambling problem in Section 2.2.3 as a decision tree, assuming that we can gamble only 0, 1, or 2 dollars in each round (this is just to keep the decision tree from growing too large).
- 2.4** Repeat the gambling problem assuming that the value of ending up with S^N dollars is $\sqrt{S^N}$.
- 2.5** Write out the steps of a shortest path algorithm, similar to that shown in Figure 2.2, that starts at the destination and works backward to the origin.
- 2.6** Carry out the proof by induction described at the end of Section 2.1.3.
- 2.7** Repeat the derivation in Section 2.1.3 assuming that the reward for task t is $c_t \sqrt{x_t}$.
- 2.8** Repeat the derivation in Section 2.1.3 assuming that the reward for task t is given by $\ln(x)$.

- 2.9** Repeat the derivation in Section 2.1.3 one more time, but now assume that all you know is that the reward is continuously differentiable, monotonically increasing and concave.
- 2.10** What happens to the answer to the budget allocation problem in Section 2.1.3 if the contribution is convex instead of concave (e.g., $C_t(x_t) = x_t^2$)?
- 2.11** Consider three variations of a shortest path problem:

Case I. All costs are known in advance. Here we assume that we have a real-time network tracking system that allows us to see the cost on each link of the network before we start our trip. We also assume that the costs do not change during the time when we start the trip to when we arrive at the link.

Case II. Costs are learned as the trip progresses. In this case we assume that we see the actual link costs for links out of node i when we arrive at node i .

Case III. Costs are learned after the fact. In this setting we only learn the cost on each link after the trip is finished.

Let v_i^I be the expected cost to get from node i to the destination for case I. Similarly let v_i^{II} and v_i^{III} be the expected costs for cases II and III. Show that $v_i^I \leq v_i^{II} \leq v_i^{III}$.

- 2.12** We are now going to do a budgeting problem where the reward function does not have any particular properties. It may have jumps, as well as being a mixture of convex and concave functions. But this time we will assume that $R = \$30$ dollars and that the allocations x_t must be in integers between 0 and 30. Assume that we have $T = 5$ products, with a contribution function $C_t(x_t) = cf(x_t)$ where $c = (c_1, \dots, c_5) = (3, 1, 4, 2, 5)$ and where $f(x)$ is given by

$$f(x) = \begin{cases} 0, & x \leq 5, \\ 5, & x = 6, \\ 7, & x = 7, \\ 10, & x = 8, \\ 12, & x \geq 9. \end{cases}$$

Find the optimal allocation of resources over the five products.

- 2.13** You suddenly realize toward the end of the semester that you have three courses that have assigned a term project instead of a final exam. You quickly estimate how much each one will take to get 100 points (equivalent to an A+) on the project. You then guess that if you invest t hours in a project, which you estimated would need T hours to get 100 points, then for $t < T$ your score will be

$$R = 100\sqrt{t/T}.$$

That is, there are declining marginal returns to putting more work into a project. So, if a project is projected to take 40 hours and you only invest 10,

you estimate that your score will be 50 points (100 times the square root of 10 over 40). You decide that you cannot spend more than a total of 30 hours on the projects, and you want to choose a value of t for each project that is a multiple of 5 hours. You also feel that you need to spend at least 5 hours on each project (that is, you cannot completely ignore a project). The time you estimate to get full score on each of the four projects is given by

Project	Completion time T
1	20
2	15
3	10

You decide to solve the problem as a dynamic program.

- (a) What is the state variable and decision epoch for this problem?
 - (b) What is your reward function?
 - (c) Write out the problem as an optimization problem.
 - (d) Set up the optimality equations.
 - (e) Solve the optimality equations to find the right time investment strategy.
- 2.14** Rewrite the transition function for the asset acquisition problem II (Section 2.2.6), assuming that R_t is the resources on hand after we satisfy the demands.
- 2.15** Write out the transition equations for the lagged asset acquisition problem in Section 2.2.7 when we allow spot purchases, which means that we may have $x_{tt} > 0$. x_{tt} refers to purchases that are made at time t which can be used to serve unsatisfied demands D_t that occur during time interval t .
- 2.16** You have to send a set of questionnaires to each of N population segments. The size of each population segment is given by w_i . You have a budget of B questionnaires to allocate among the population segments. If you send x_i questionnaires to segment i , you will have a sampling error proportional to

$$f(x_i) = \frac{1}{\sqrt{x_i}}.$$

You want to minimize the weighted sum of sampling errors, given by

$$F(x) = \sum_{i=1}^N w_i f(x_i)$$

You wish to find the allocation x that minimizes $F(x)$ subject to the budget constraint $\sum_{i=1}^N x_i \leq B$. Set up the optimality equations to solve this problem as a dynamic program (needless to say, we are only interested in integer solutions).

- 2.17** Identify three examples of problems where you have to try an action to learn about the reward for an action.

C H A P T E R 3

Introduction to Markov Decision Processes

There is a very elegant theory for solving stochastic, dynamic programs if we are willing to live within some fairly limiting assumptions. Assume that we have a discrete state space $\mathcal{S} = \{1, 2, \dots, |\mathcal{S}|\}$, where \mathcal{S} is small enough to enumerate. Next assume that there is a relatively small set of decisions or actions, which we denote by $a \in \mathcal{A}$, and that we can compute a cost (if minimizing) or contribution (if maximizing) given by $C(s, a)$. Finally, assume that we are given a transition matrix $p_t(S_{t+1}|S_t, a_t)$ that gives the probability that if we are in state S_t (at time t) and take action a_t , then we will next be in state S_{t+1} .

From time to time we are going to switch gears to consider problems where the decision is a vector. When this happens, we will use x as our decision variable. However, there are many applications where the number of actions is discrete and small, and there are many algorithms that are specifically designed for small action spaces. In particular, the material in this chapter is designed for small action spaces, and as a result we use a for action throughout.

There are many problems where states are continuous, or the state variable is a vector producing a state space that is far too large to enumerate. As a result, the one-step transition matrix $p_t(S_{t+1}|S_t, a_t)$ can be difficult or impossible to compute. So why cover material that is widely acknowledged to work only on small or highly specialized problems? First, some problems have small state and action spaces and can be solved with these techniques. Second, the theory of Markov decision processes can be used to identify structural properties that can dramatically simplify computational algorithms. But far more important, this material provides the intellectual foundation for the types of algorithms that we present in later chapters. Using the framework in this chapter, we can prove very powerful results that will provide a guiding hand as we step into richer and more complex problems

Approximate Dynamic Programming: Solving the Curses of Dimensionality, Second Edition.

Warren B. Powell.

© 2011 John Wiley & Sons, Inc. Published 2011 by John Wiley & Sons, Inc.

in many real-world settings. Furthermore the behavior of these algorithms provide critical insights on the behavior of algorithms for more general problems.

There is a rich and elegant theory behind Markov decision processes. Even if the algorithms have limited application, the ideas behind these algorithms, which enjoy a rich history, represent the fundamental underpinnings of most of the algorithms in the remainder of this book. As with most of the chapters in the book, the body of this chapter focuses on the algorithms, and the convergence proofs have been deferred to the “Why does it work” section (Section 3.10). The intent is to allow the presentation of results to flow more naturally, but serious students of dynamic programming are encouraged to delve into these proofs, which are quite elegant. This is partly to develop a deeper appreciation of the properties of the problem as well as to develop an understanding of the proof techniques that are used in this field.

3.1 THE OPTIMALITY EQUATIONS

In the last chapter we illustrated a number of stochastic applications that involve solving the following objective function:

$$\max_{\pi} \mathbb{E}^{\pi} \left\{ \sum_{t=0}^T \gamma^t C_t^{\pi}(S_t, A_t^{\pi}(S_t)) \right\}. \quad (3.1)$$

We sometimes wrote our objective function using $\mathbb{E} \dots$, and in other cases we wrote it as $\mathbb{E}^{\pi} \dots$. We defer until Chapter 5 a more complete discussion of this choice, but in a nutshell it has to do with whether the decisions we make can influence the information we observe. If this is the case, the notation \mathbb{E}^{π} is more general.

For most problems, solving equation (3.1) is computationally intractable, but it provides the basis for identifying the properties of optimal solutions and finding and comparing “good” solutions.

3.1.1 Bellman’s Equations

With a little thought, we realize that we do not have to solve this entire problem at once. Suppose that we are solving a deterministic shortest path problem where S_t is the index of the node in the network where we have to make a decision. If we are in state $S_t = i$ (i.e., we are at node i in our network) and take action $a_t = j$ (i.e., we wish to traverse the link from i to j), our transition function will tell us that we are going to land in some state $S_{t+1} = S^M(S_t, a_t)$ (in this case, node j). What if we had a function $V_{t+1}(S_{t+1})$ that told us the value of being in state S_{t+1} (giving us the value of the path from node j to the destination)? We could evaluate each possible action a_t and simply choose the action a_t that has the largest one-period contribution, $C_t(S_t, a_t)$, plus the value of landing in state $S_{t+1} = S^M(S_t, a_t)$, which we represent using $V_{t+1}(S_{t+1})$. Since this value represents the money we receive

one time period in the future, we might discount this by a factor γ . In other words, we have to solve

$$a_t^*(S_t) = \arg \max_{a_t \in \mathcal{A}_t} (C_t(S_t, a_t) + \gamma V_{t+1}(S_{t+1})),$$

where “ $\arg \max$ ” means that we want to choose the action a_t that maximizes the expression in parentheses. We also note that S_{t+1} is a function of S_t and a_t , meaning that we could write it as $S_{t+1}(S_t, a_t)$. Both forms are fine. It is common to write S_{t+1} alone, but the dependence on S_t and a_t needs to be understood.

The value of being in state S_t is the value of using the optimal decision $a_t^*(S_t)$. That is,

$$\begin{aligned} V_t(S_t) &= \max_{a_t \in \mathcal{A}_t} (C_t(S_t, a_t) + \gamma V_{t+1}(S_{t+1}(S_t, a_t))) \\ &= C_t(S_t, a_t^*(S_t)) + \gamma V_{t+1}(S_{t+1}(S_t, a_t^*(S_t))). \end{aligned} \quad (3.2)$$

Equation (3.2) is the optimality equation for deterministic problems.

When we are solving stochastic problems, we have to model the fact that new information becomes available after we make the decision a_t . The result can be uncertainty both in the contribution earned and in the determination of the next state we visit, S_{t+1} . For example, consider the problem of managing oil inventories for a refinery. Let the state S_t be the inventory in thousands of barrels of oil at time t (we require S_t to be integer). Let a_t be the amount of oil ordered at time t that will be available for use between t and $t+1$, and let \hat{D}_{t+1} be the demand for oil between t and $t+1$. The state variable is governed by the simple inventory equation

$$S_{t+1}(S_t, a_t, \hat{D}_{t+1}) = \max\{0, S_t + a_t - \hat{D}_{t+1}\}.$$

We have written the state S_{t+1} using $S_{t+1}(S_t, a_t)$ to express the dependence on S_t and a_t , but it is common to simply write S_{t+1} and let the dependence on S_t and a_t be implicit. Since \hat{D}_{t+1} is random at time t when we have to choose a_t , we do not know S_{t+1} . But if we know the probability distribution of the demand \hat{D} , we can work out the probability that S_{t+1} will take on a particular value. If $\mathbb{P}^D(d) = \mathbb{P}[\hat{D} = d]$ is our probability distribution, then we can find the probability distribution for S_{t+1} using

$$\text{Prob}(S_{t+1} = s') = \begin{cases} 0 & \text{if } s' > S_t + a_t, \\ \mathbb{P}^D(S_t + a_t - s') & \text{if } 0 < s' \leq S_t + a_t, \\ \sum_{d=S_t+a_t}^{\infty} \mathbb{P}^D(d) & \text{if } s' = 0. \end{cases}$$

These probabilities depend on S_t and a_t , so we write the probability distribution as

$$\mathbb{P}(S_{t+1}|S_t, a_t) = \text{probability of } S_{t+1} \text{ given } S_t \text{ and } a_t.$$

We can then modify the deterministic optimality equation in (3.2) by simply adding an expectation, giving us

$$V_t(S_t) = \max_{a_t \in \mathcal{A}_t} \left(C_t(S_t, a_t) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(S_{t+1} = s' | S_t, a_t) V_{t+1}(s') \right). \quad (3.3)$$

We refer to this as the *standard form* of Bellman's equations, since this is the version that is used by virtually every textbook on stochastic, dynamic programming. An equivalent form that is more natural for approximate dynamic programming is to write

$$V_t(S_t) = \max_{a_t \in \mathcal{A}_t} \left(C_t(S_t, a_t) + \gamma \mathbb{E}\{V_{t+1}(S_{t+1}) | S_t\} \right), \quad (3.4)$$

where we simply use an expectation instead of summing over probabilities. We refer to this equation as the *expectation form* of Bellman's equation. This version forms the basis for our algorithmic work in later chapters.

Remark Equation (3.4) is often written in the slightly more compact form

$$V_t(S_t) = \max_{a_t \in \mathcal{A}_t} \left(C_t(S_t, a_t) + \gamma \mathbb{E}\{V_{t+1}(S_{t+1}) | S_t\} \right), \quad (3.5)$$

where the functional relationship $S_{t+1} = S^M(S_t, a_t, W_{t+1})$ is implicit. At this point, however, we have to deal with some subtleties of mathematical notation. In equation (3.4) we have captured the *functional dependence* of S_{t+1} on S_t and a_t , while capturing the *conditional dependence* of S_{t+1} (more specifically W_{t+1}) on the state S_t ,

$$V_t(S_t) = \max_{a_t \in \mathcal{A}_t} \left(C_t(S_t, a_t) + \gamma \mathbb{E}\{V_{t+1}(S_{t+1}) | S_t, a_t\} \right) \quad (3.6)$$

to capture the fact that S_{t+1} may depend on a_t . However, it is important to understand whether S_{t+1} is *functionally dependent on* S_t and a_t , or whether the distribution of S_{t+1} is *probabilistically dependent on* S_t and a_t . To see the difference, imagine that we have a problem where W_{t+1} is the wind or some exogenous process whose outcomes are independent of S_t or a_t . Then it is perfectly valid to write

$$V_t(S_t) = \max_{a_t \in \mathcal{A}_t} \left(C_t(S_t, a_t) + \gamma \mathbb{E}V_{t+1}(S_{t+1} = S^M(S_t, a_t, W_{t+1})) \right),$$

where we are explicitly capturing the functional dependence of S_{t+1} on S_t and a_t , but where the expectation is not conditioned on anything, because the distribution of W_{t+1} does not depend on S_t or a_t . However, there are problems where W_{t+1} depends on the state, such as the random perturbations of a robot that depend on how close the robot is to a boundary. In this case S_{t+1} depends functionally on S_t , a_t and W_{t+1} , but the *distribution* of W_{t+1} also depends on S_t , in which case the expectation needs to be a conditional expectation. Then there are problems where the distribution of W_{t+1} depends on both S_t and a_t , such as the random changes

in a stock that might be depressed if a mutual fund is holding large quantities (S_t) and begins selling in large amounts (a_t). In this case the proper interpretation of equation (3.6) is that we are computing the conditional expectation over W_{t+1} that now depends on both the state and action.

The standard form of Bellman's equation (3.3) has been popular in the research community since it lends itself to elegant algebraic manipulation when we assume we know the transition matrix. It is common to write it in a more compact form. Recall that a policy π is a rule that specifies the action a_t given the state S_t . In this chapter it is easiest if we always think of a policy in terms of a rule "when we are in state s we take action a ." This is a form of "lookup table" representation of a policy that is very clumsy for most real problems, but it will serve our purposes here. The probability that we transition from state $S_t = s$ to $S_{t+1} = s'$ can be written as

$$p_{ss'}(a) = \mathbb{P}(S_{t+1} = s' | S_t = s, a_t = a).$$

We would say that " $p_{ss'}(a)$ is the probability that we end up in state s' if we start in state s at time t when we are taking action a ." Now say that we have a function $A_t^\pi(s)$ that determines the action a we should take when in state s . It is common to write the transition probability $p_{ss'}(a)$ in the form

$$p_{ss'}^\pi = \mathbb{P}(S_{t+1} = s' | S_t = s, A_t^\pi(s) = a).$$

We can write this in matrix form

$$P_t^\pi = \text{one-step transition matrix under policy } \pi,$$

where $p_{ss'}^\pi$ is the element in row s and column s' . There is a different matrix P^π for each policy (decision rule) π .

Now let c_t^π be a column vector with element $c_t^\pi(s) = C_t(s, A_t^\pi(s))$, and let v_{t+1} be a column vector with element $V_{t+1}(s)$. Then (3.3) is equivalent to

$$\begin{bmatrix} \vdots \\ v_t(s) \\ \vdots \end{bmatrix} = \max_\pi \left(\begin{bmatrix} \vdots \\ c_t^\pi(s) \\ \vdots \end{bmatrix} + \gamma \begin{bmatrix} \ddots & & \\ & p_{ss'}^\pi & \\ & & \ddots \end{bmatrix} \begin{bmatrix} \vdots \\ v_{t+1}(s') \\ \vdots \end{bmatrix} \right), \quad (3.7)$$

where the maximization is performed for each element (state) in the vector. In matrix/vector form, equation (3.7) can be written

$$v_t = \max_\pi (c_t^\pi + \gamma P_t^\pi v_{t+1}). \quad (3.8)$$

Here we maximize over policies because we want to find the best action for *each* state. The vector v_t is known widely as the *value function* (the value of being in each state). In control theory it is known as the *cost-to-go function*, where it is typically denoted as J .

Equation (3.8) can be solved by finding a_t for each state s . The result is a decision vector $a_t^* = (a_t^*(s))_{s \in \mathcal{S}}$, which is equivalent to determining the best policy. This is easiest to envision when a_t is a scalar (how much to buy, whether to sell), but in many applications $a_t(s)$ is itself a vector. For example, suppose that our problem is to assign individual programmers to different programming tasks, where our state S_t captures the availability of programmers and the different tasks that need to be completed. Of course, computing a vector a_t for each state S_t , which is itself a vector, is much easier to write than to implement.

It is very easy to lose sight of the relationship between Bellman's equation and the original objective function that we stated in equation (3.1). To bring this out, we begin by writing the expected profits using policy π from time t onward:

$$F_t^\pi(S_t) = \mathbb{E} \left\{ \sum_{t'=t}^{T-1} C_{t'}(S_{t'}, A_{t'}^\pi(S_{t'})) + C_T(S_T) | S_t \right\}.$$

$F_t^\pi(S_t)$ is the expected total contribution if we are in state S_t in time t , and follow policy π from time t onward. If $F_t^\pi(S_t)$ were easy to calculate, we would probably not need dynamic programming. Instead, it seems much more natural to calculate V_t^π recursively using

$$V_t^\pi(S_t) = C_t(S_t, A_t^\pi(S_t)) + \mathbb{E} \{ V_{t+1}^\pi(S_{t+1}) | S_t \}.$$

It is not hard to show (by stepping backward in time) that

$$F_t^\pi(S_t) = V_t^\pi(S_t).$$

The proof, given in Section 3.10.1, uses a proof by induction: assume that it is true for V_{t+1}^π , and then show that it is true for V_t^π (not surprisingly, inductive proofs are very popular in dynamic programming).

With this result in hand we can then establish the following key result. Let $V_t(S_t)$ be a solution to equation (3.4) (or (3.3)). Then

$$\begin{aligned} F_t^* &= \max_{\pi \in \Pi} F_t^\pi(S_t) \\ &= V_t(S_t). \end{aligned} \tag{3.9}$$

Equation (3.9) establishes the equivalence between (1) the value of being in state S_t and following the optimal policy and (2) the optimal value function at state S_t . While these are indeed equivalent, the equivalence is the result of a theorem (established in Section 3.10.1). However, it is not unusual to find people who lose sight of the original objective function. Later we have to solve these equations approximately, and we will need to use the original objective function to evaluate the quality of a solution.

3.1.2 Computing the Transition Matrix

It is very common in stochastic, dynamic programming (more precisely, Markov decision processes) to assume that the one-step transition matrix P^π is given as data (remember that there is a different matrix for each policy π). In practice, we generally can assume that we know the transition function $S^M(S_t, a_t, W_{t+1})$ from which we have to derive the one-step transition matrix.

Assume that the random information W_{t+1} that arrives between t and $t+1$ is independent of all prior information. Let Ω_{t+1} be the set of possible outcomes of W_{t+1} (for simplicity, we assume that Ω_{t+1} is discrete), where $\mathbb{P}(W_{t+1} = \omega_{t+1})$ is the probability of outcome $\omega_{t+1} \in \Omega_{t+1}$. Also define the indicator function

$$1_{\{X\}} = \begin{cases} 1 & \text{if the statement "X" is true,} \\ 0 & \text{otherwise.} \end{cases}$$

Here “X” represents a logical condition (e.g., “is $S_t = 6$?”). We now observe that the one-step transition probability $\mathbb{P}_t(S_{t+1}|S_t, a_t)$ can be written

$$\begin{aligned} \mathbb{P}_t(S_{t+1}|S_t, a_t) &= \mathbb{E}1_{\{s' = S^M(S_t, a_t, W_{t+1})\}} \\ &= \sum_{\omega_{t+1} \in \Omega_{t+1}} \mathbb{P}(\omega_{t+1})1_{\{s' = S^M(S_t, a_t, \omega_{t+1})\}} \end{aligned}$$

So finding the one-step transition matrix means that all we have to do is to sum over all possible outcomes of the information W_{t+1} and add up the probabilities that take us from a particular state-action pair (S_t, a_t) to a particular state $S_{t+1} = s'$. Sounds easy.

In some cases this calculation is straightforward (consider our oil inventory example earlier in the section). But in other cases this calculation is impossible. For example, W_{t+1} might be a vector of prices or demands. Such a set of outcomes Ω_{t+1} can be much too large to enumerate. We can estimate the transition matrix statistically, but in later chapters (starting in Chapter 4) we are going to avoid the need to compute the one-step transition matrix entirely. For the remainder of this chapter, we assume that the one-step transition matrix is available.

3.1.3 Random Contributions

In many applications the one-period contribution function is a deterministic function of S_t and a_t , and hence we routinely write the contribution as the deterministic function $C_t(S_t, a_t)$. However, this is not always the case. For example, a car traveling over a stochastic network may choose to traverse the link from node i to node j , and only learn the cost of the movement after making the decision. For such cases the contribution function is random, and we might write it as

$\hat{C}_{t+1}(S_t, a_t, W_{t+1}) = \text{contribution received in period } t+1 \text{ given the state } S_t \text{ and decision } a_t, \text{ as well as the new information } W_{t+1} \text{ that arrives in period } t+1.$

Then we simply bring the expectation in front, giving us

$$V_t(S_t) = \max_{a_t} \mathbb{E} \left\{ \hat{C}_{t+1}(S_t, a_t, W_{t+1}) + \gamma V_{t+1}(S_{t+1}) | S_t \right\}. \quad (3.10)$$

Now let

$$C_t(S_t, a_t) = \mathbb{E}\{\hat{C}_{t+1}(S_t, a_t, W_{t+1}) | S_t\}.$$

Thus we may view $C_t(S_t, a_t)$ as the expected contribution given that we are in state S_t and take action a_t .

3.1.4 Bellman's Equation Using Operator Notation*

The vector form of Bellman's equation in (3.8) can be written even more compactly using operator notation. Let \mathcal{M} be the “max” (or “min”) operator in (3.8) that can be viewed as acting on the vector v_{t+1} to produce the vector v_t . If we have a given policy π , we can write

$$\mathcal{M}^\pi v(s) = C_t(s, A^\pi(s)) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}_t(s'|s, A^\pi(s)) v_{t+1}(s').$$

Alternatively, we can find the best action, which we represent using

$$\mathcal{M}v(s) = \max_a (C_t(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}_t(s'|s, a) v_{t+1}(s')).$$

Here $\mathcal{M}v$ produces a vector, and $\mathcal{M}v(s)$ refers to element s of this vector. In vector form we would write

$$\mathcal{M}v = \max_\pi (c_t^\pi + \gamma P_t^\pi v_{t+1}).$$

Now let \mathcal{V} be the space of value functions. Then \mathcal{M} is a mapping

$$\mathcal{M} : \mathcal{V} \rightarrow \mathcal{V}.$$

We may also define the operator \mathcal{M}^π for a particular policy π using

$$\mathcal{M}^\pi(v) = c_t^\pi + \gamma P^\pi v \quad (3.11)$$

for some vector $v \in \mathcal{V}$. \mathcal{M}^π is known as a *linear operator* since the operations that it performs on v are additive and multiplicative. In mathematics the function $c_t^\pi + \gamma P^\pi v$ is known as an *affine function*. This notation is particularly useful in mathematical proofs (see in particular some of the proofs in Section 3.10), but we will not use this notation when we describe models and algorithms.

We see later in the chapter that we can exploit the properties of this operator to derive some very elegant results for Markov decision processes. These proofs provide insights into the behavior of these systems, which can guide the design of algorithms. For this reason it is relatively immaterial that the actual computation of these equations may be intractable for many problems; the insights still apply.

3.2 FINITE HORIZON PROBLEMS

Finite horizon problems tend to arise in two settings. First, some problems have a very specific horizon. For example, we might be interested in the value of an American option where we are allowed to sell an asset at any time $t \leq T$, where T is the exercise date. Another problem is to determine how many seats to sell at different prices for a particular flight departing at some point in the future. In the same class are problems that require reaching some goal (but not at a particular point in time). Examples include driving to a destination, selling a house, or winning a game.

A second class of problems is actually infinite horizon, but where the goal is to determine what to do right now given a particular state of the system. For example, a transportation company might want to know what drivers should be assigned to a particular set of loads right now. Of course, these decisions need to consider the downstream impact, so models have to extend into the future. For this reason we might model the problem over a horizon T that, when solved, yields a decision of what to do right now.

When we encounter a finite horizon problem, we assume that we are given the function $V_T(S_T)$ as data. Often we simply use $V_T(S_T) = 0$ because we are primarily interested in what to do now, given by a_0 , or in projected activities over some horizon $t = 0, 1, \dots, T^{ph}$, where T^{ph} is the length of a planning horizon. If we set T sufficiently larger than T^{ph} , then we may be able to assume that the decisions $a_0, a_1, \dots, a_{T^{ph}}$ are of sufficiently high quality to be useful.

Solving a finite horizon problem, in principle, is straightforward. As outlined in Figure 3.1, we simply have to start at the last time period, compute the value function for each possible state $s \in \mathcal{S}$, and then step back another time period. This way at time period t we have already computed $V_{t+1}(S)$. Not surprisingly, this method is often referred to as “backward dynamic programming.” The critical element that attracts so much attention is the requirement that we compute the value function $V_t(S_t)$ for all states $S_t \in \mathcal{S}$.

Step 0. Initialization:

Initialize the terminal contribution $V_T(S_T)$.

Set $t = T - 1$.

Step 1. Calculate:

$$V_t(S_t) = \max_{a_t} \left\{ C_t(S_t, a_t) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|S_t, a_t) V_{t+1}(s') \right\}$$

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

Step 2. If $t > 0$, decrement t and return to step 1. Else stop.

Figure 3.1 Backward dynamic programming algorithm.

We first saw backward dynamic programming in Section 2.2.1 when we described a simple decision tree problem. The only difference between the backward dynamic programming algorithm in Figure 3.1 and our solution of the decision tree problem is primarily notational. Decision trees are visual and tend to be easier to understand, whereas in this section the methods are described using notation. However, decision tree problems tend to be always presented in the context of problems with relatively small numbers of states and actions (What job should I take? Should the United States put a blockade around Cuba? Should the shuttle launch have been canceled due to cold weather?).

Another popular illustration of dynamic programming is the discrete asset acquisition problem. Say that you order a quantity a_t at each time period to be used in the next time period to satisfy a demand \hat{D}_{t+1} . Any unused product is held over to the following time period. For this, our state variable S_t is the quantity of inventory left over at the end of the period after demands are satisfied. The transition equation is given by $S_{t+1} = [S_t + a_t - \hat{D}_{t+1}]^+$ where $[x]^+ = \max(x, 0)$. The cost function (which we seek to minimize) is given by $\hat{C}_{t+1}(S_t, a_t) = c^h S_t + c^o I_{\{a_t > 0\}}$, where $I_{\{X\}} = 1$ if X is true and 0 otherwise. Note that the cost function is nonconvex. This does not create problems if we solve our minimization problem by searching over different (discrete) values of a_t . Since all of our quantities are scalar, there is no difficulty finding $C_t(S_t, a_t)$.

To compute the one-step transition matrix, let Ω be the set of possible outcomes of \hat{D}_t , and let $\mathbb{P}(\hat{D}_t = \omega)$ be the probability that $\hat{D}_t = \omega$ (if this use of ω seems weird, get used to it—we are going to use it a lot).

The one-step transition matrix is computed using

$$\mathbb{P}(s'|s, a) = \sum_{\omega \in \Omega} \mathbb{P}(\hat{D}_{t+1} = \omega) 1_{\{s' = [s+a-\omega]^+\}},$$

where Ω is the set of (discrete) outcomes of the demand \hat{D}_{t+1} .

Another example is the shortest path problem with random arc costs. Suppose that you are trying to get from origin node q to destination node r in the shortest time possible. As you reach each intermediate node i , you are able to observe the time required to traverse each arc out of node i . Let V_j be the expected shortest path time from j to the destination node r . At node i , you see the link time $\hat{\tau}_{ij}$, which represents a random observation of the travel time. Now we choose to traverse arc (i, j^*) , where j^* solves $\min_j (\hat{\tau}_{ij} + V_j)$ (j^* is random since the travel time is random). We would then compute the value of being at node i using $V_i = \mathbb{E}\{\min_j (\hat{\tau}_{ij} + V_j)\}$.

3.3 INFINITE HORIZON PROBLEMS

We typically use infinite horizon formulations whenever we wish to study a problem where the parameters of the contribution function, transition function, and the process governing the exogenous information process do not vary over time, although they may vary in cycles (e.g., an infinite horizon model of energy storage

from a solar panel may depend on time of day). Often we wish to study such problems in steady state. More importantly, infinite horizon problems provide a number of insights into the properties of problems and algorithms, drawing off an elegant theory that has evolved around this problem class. Even students who wish to solve complex, nonstationary problems will benefit from an understanding of this problem class.

We begin with the optimality equations

$$V_t(S_t) = \max_{a_t \in \mathcal{A}} \mathbb{E} \{ C_t(S_t, a_t) + \gamma V_{t+1}(S_{t+1}) | S_t \}.$$

We can think of a steady-state problem as one without the time dimension. Letting $V(s) = \lim_{t \rightarrow \infty} V_t(S_t)$ (and assuming the limit exists), we obtain the steady-state optimality equations

$$V(s) = \max_{a \in \mathcal{A}} \left\{ C(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a) V(s') \right\}. \quad (3.12)$$

The functions $V(s)$ can be shown (as we do later) to be equivalent to solving the infinite horizon problem

$$\max_{\pi \in \Pi} \mathbb{E} \left\{ \sum_{t=0}^{\infty} \gamma^t C_t(S_t, A_t^\pi(S_t)) \right\}. \quad (3.13)$$

Now define

$$\begin{aligned} P^{\pi, t} &= t\text{-step transition matrix, over periods } 0, 1, \dots, t-1, \text{ given policy } \pi \\ &= \Pi_{t'=0}^{t-1} P_{t'}^\pi. \end{aligned} \quad (3.14)$$

We further define $P^{\pi, 0}$ to be the identity matrix. As before, let c_t^π be the column vector of the expected cost of being in each state given that we choose the action a_t described by policy π , where the element for state s is $c_t^\pi(s) = C_t(s, A^\pi(s))$. The infinite horizon, discounted value of a policy π starting at time t is given by

$$v_t^\pi = \sum_{t'=t}^{\infty} \gamma^{t'-t} P^{\pi, t'-t} c_{t'}^\pi. \quad (3.15)$$

Assume that after following policy π_0 we follow policy $\pi_1 = \pi_2 = \dots = \pi$. In this case equation (3.15) can now be written as (starting at $t = 0$)

$$v^{\pi_0} = c^{\pi_0} + \sum_{t'=1}^{\infty} \gamma^{t'} P^{\pi, t'} c_{t'}^\pi \quad (3.16)$$

$$= c^{\pi_0} + \sum_{t'=1}^{\infty} \gamma^{t'} \left(\prod_{t''=0}^{t'-1} P_{t''}^\pi \right) c_{t'}^\pi \quad (3.17)$$

$$= c^{\pi_0} + \gamma P^{\pi_0} \sum_{t'=1}^{\infty} \gamma^{t'-1} \left(\prod_{t''=1}^{t'-1} P_{t''}^{\pi} \right) c_{t'}^{\pi} \quad (3.18)$$

$$= c^{\pi_0} + \gamma P^{\pi_0} v^{\pi}. \quad (3.19)$$

Equation (3.19) shows us that the value of a policy is the single period reward plus a discounted terminal reward that is the same as the value of a policy starting at time 1. If our decision rule is stationary, then $\pi_0 = \pi_1 = \dots = \pi_t = \pi$, which allows us to rewrite (3.19) as

$$v^{\pi} = c^{\pi} + \gamma P^{\pi} v^{\pi}. \quad (3.20)$$

This allows us to solve for the stationary reward explicitly (as long as $0 \leq \gamma < 1$), giving us

$$v^{\pi} = (I - \gamma P^{\pi})^{-1} c^{\pi}.$$

We can also write an infinite horizon version of the optimality equations using our operator notation. Letting \mathcal{M} be the “max” (or “min”) operator (also known as the Bellman operator), the infinite horizon version of equation (3.11) would be written

$$\mathcal{M}^{\pi}(v) = c^{\pi} + \gamma P^{\pi} v. \quad (3.21)$$

There are several algorithmic strategies for solving infinite horizon problems. The first, value iteration, is the most widely used method. It involves iteratively estimating the value function. At each iteration the estimate of the value function determines which decisions we will make and as a result defines a policy. The second strategy is *policy iteration*. At every iteration we define a policy (literally, the rule for determining decisions) and then determine the value function for that policy. Careful examination of value and policy iteration reveals that these are closely related strategies that can be viewed as special cases of a general strategy that uses value and policy iteration. Finally, the third major algorithmic strategy exploits the observation that the value function can be viewed as the solution to a specially structured linear programming problem.

3.4 VALUE ITERATION

Value iteration is perhaps the most widely used algorithm in dynamic programming because it is the simplest to implement and, as a result, often tends to be the most natural way of solving many problems. It is virtually identical to backward dynamic programming for finite horizon problems. In addition most of our work in approximate dynamic programming is based on value iteration.

Value iteration comes in several flavors. The basic version of the value iteration algorithm is given in Figure 3.2. The proof of convergence (see Section 3.10.2) is

Step 0. Initialization:

Set $v^0(s) = 0 \forall s \in \mathcal{S}$.

Fix a tolerance parameter $\epsilon > 0$.

Set $n = 1$.

Step 1. For each $s \in \mathcal{S}$ compute:

$$v^n(s) = \max_{a \in \mathcal{A}} \left(C(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a) v^{n-1}(s') \right). \quad (3.22)$$

Step 2. If $\|v^n - v^{n-1}\| < \epsilon(1 - \gamma)/2\gamma$, let π^ϵ be the resulting policy that solves (3.22), and let $v^\epsilon = v^n$ and stop; else set $n = n+1$ and go to step 1.

Figure 3.2 Value iteration algorithm for infinite horizon optimization.

quite elegant for students who enjoy mathematics. The algorithm also has several nice properties that we explore below.

It is easy to see that the value iteration algorithm is similar to the backward dynamic programming algorithm. Rather than using a subscript t , which we decrement from T back to 0, we use an iteration counter n that starts at 0 and increases until we satisfy a convergence criterion. We stop the algorithm when

$$\|v^n - v^{n-1}\| < \epsilon(1 - \gamma)/2\gamma,$$

where $\|v\|$ is the max-norm defined by

$$\|v\| = \max_s |v(s)|.$$

Thus $\|v\|$ is the largest absolute value of a vector of elements. We stop if the largest change in the value of being in any state is less than $\epsilon(1 - \gamma)/2\gamma$, where ϵ is a specified error tolerance.

Below we describe a Gauss–Seidel variant that is a useful method for accelerating value iteration, and a version known as relative value iteration.

3.4.1 A Gauss–Seidel Variation

A slight variant of the value iteration algorithm provides a faster rate of convergence. In this version (typically called the Gauss–Seidel variant), we take advantage of the fact that when we are computing the expectation of the value of the future, we have to loop over all the states s' to compute $\sum_{s'} \mathbb{P}(s'|s, a) v^n(s')$. For a particular state s , we would have already computed $v^{n+1}(\hat{s})$ for $\hat{s} = 1, 2, \dots, s-1$. By simply replacing $v^n(\hat{s})$ with $v^{n+1}(\hat{s})$ for the states we have already visited, we obtain an algorithm that typically exhibits a noticeably faster rate of convergence. The algorithm requires a change to step 1 of the value iteration, as shown in Figure 3.3.

Replace step 1 with

Step 1'. For each $s \in \mathcal{S}$ compute

$$v^n(s) = \max_{a \in \mathcal{A}} \left\{ C(s, a) + \gamma \left(\sum_{s' < s} \mathbb{P}(s'|s, a)v^n(s') + \sum_{s' \geq s} \mathbb{P}(s'|s, a)v^{n-1}(s') \right) \right\}$$

Figure 3.3 Gauss–Seidel variation of value iteration.

3.4.2 Relative Value Iteration

Another version of value iteration is called *relative value iteration*; it is useful in problems that do not have a discount factor or where the optimal policy converges much more quickly than the value function, which may grow steadily for many iterations. The relative value iteration algorithm is shown in Figure 3.4.

In relative value iteration we focus on the fact that we may be more interested in the convergence of the difference $\|v(s) - v(s')\|$ than we are in the values of $v(s)$ and $v(s')$. This would be the case if we are interested in the best policy rather than the value function (this is not always the case). What often happens is that, especially toward the limit, all the values $v(s)$ start increasing by the same rate. For this reason we can pick any state (denoted s^* in the algorithm) and subtract its value from all the other states.

To provide a bit of formalism for our algorithm, we define the *span* of a vector v as follows:

$$sp(v) = \max_{s \in \mathcal{S}} v(s) - \min_{s \in \mathcal{S}} v(s).$$

Note that our use of “span” is different than the way it is normally used in linear algebra. Here and throughout this section, we define the norm of a vector as

$$\|v\| = \max_{s \in \mathcal{S}} v(s).$$

Step 0. Initialization:

- Choose some $v^0 \in \mathcal{V}$.
- Choose a base state s^* and a tolerance ϵ .
- Let $w^0 = v^0 - v^0(s^*)e$ where e is a vector of ones.
- Set $n = 1$.

Step 1. Set

$$v^n = \mathcal{M}w^{n-1},$$

$$w^n = v^n - v^n(s^*)e.$$

Step 2. If $sp(v^n - v^{n-1}) < (1 - \gamma)\epsilon/\gamma$, go to step 3; otherwise, go to step 1.

Step 3. Set $a^\epsilon = \arg \max_{a \in \mathcal{A}} (C(a) + \gamma P^\pi v^n)$.

Figure 3.4 Relative value iteration.

Note that the span has the following six properties:

1. $sp(v) \geq 0$.
2. $sp(u + v) \leq sp(u) + sp(v)$.
3. $sp(kv) = \|k\|sp(v)$.
4. $sp(v + ke) = sp(v)$.
5. $sp(v) = sp(-v)$.
6. $sp(v) \leq 2\|v\|$.

Property 4 implies that $sp(v) = 0$ does not mean that $v = 0$, and therefore it does not satisfy the properties of a norm. For this reason it is called a *semi-norm*.

The relative value iteration algorithm is simply subtracting a constant from the value vector at each iteration. Obviously this does not change the optimal decision, but it does change the value. If we are only interested in the optimal policy, relative value iteration often offers much faster convergence, but it may not yield accurate estimates of the value of being in each state.

3.4.3 Bounds and Rates of Convergence

One important property of value iteration algorithms is that if our initial estimate is too low, the algorithm will rise to the correct value from below. Similarly, if our initial estimate is too high, the algorithm will approach the correct value from above. This property is formalized in the following theorem:

Theorem 3.4.1 *For a vector $v \in \mathcal{V}$:*

- a. *if v satisfies $v \geq \mathcal{M}v$, then $v \geq v^*$.*
- b. *if v satisfies $v \leq \mathcal{M}v$, then $v \leq v^*$.*
- c. *if v satisfies $v = \mathcal{M}v$, then v is the unique solution to this system of equations and $v = v^*$.*

The proof is given in Section 3.10.3. It is a nice property because it provides some valuable information on the nature of the convergence path. In practice, we generally do not know the true value function, which makes it hard to know if we are starting from above or below, although some problems have natural bounds (e.g., nonnegativity).

The proof of the monotonicity property above also provides us with a nice corollary. If $V(s) = \mathcal{M}V(s)$ for all s , then $V(s)$ is the unique solution to this system of equations, which must also be the optimal solution.

This result raises the question: What if some of our estimates of the value of being in some states are too high, while others are too low? This means that the values may cycle above and below the optimal solution, although at some point we may find that all the values have increased (decreased) from one iteration to

the next. If this happens, then it means that the values are all equal to or below (above) the limiting value.

Value iteration also provides a nice bound on the quality of the solution. Recall that when we use the value iteration algorithm, we stop when

$$\|v^{n+1} - v^n\| < \frac{\epsilon(1-\gamma)}{2\gamma}, \quad (3.23)$$

where γ is our discount factor and ϵ is a specified error tolerance. It is possible that we have found the optimal policy when we stop, but it is very unlikely that we have found the optimal value functions. We can, however, provide a bound on the gap between the solution v^n and the optimal values v^* by using the following theorem:

Theorem 3.4.2 *If we apply the value iteration algorithm with stopping parameter ϵ and the algorithm terminates at iteration n with value function v^{n+1} , then*

$$\|v^{n+1} - v^*\| \leq \frac{\epsilon}{2}. \quad (3.24)$$

Let π^ϵ be the policy that we terminate with, and let v^{π^ϵ} be the value of this policy. Then

$$\|v^{\pi^\epsilon} - v^*\| \leq \epsilon.$$

The proof is given in Section 3.10.4. While it is nice that we can bound the error, the bad news is that the bound can be quite poor. More important is what the bound teaches us about the role of the discount factor.

We can provide some additional insights into the bound, as well as the rate of convergence, by considering a trivial dynamic program. In this problem we receive a constant reward c at every iteration. There are no decisions, and there is no randomness. The value of this “game” is quickly seen to be

$$\begin{aligned} v^* &= \sum_{n=0}^{\infty} \gamma^n c \\ &= \frac{1}{1-\gamma} c. \end{aligned} \quad (3.25)$$

Consider what happens when we solve this problem using value iteration. Starting with $v^0 = 0$, we would use the iteration

$$v^n = c + \gamma v^{n-1}.$$

After we have repeated this n times, we have

$$\begin{aligned} v^n &= \sum_{m=0}^{n-1} \gamma^m c \\ &= \frac{1 - \gamma^n}{1 - \gamma} c. \end{aligned} \quad (3.26)$$

Comparing equations (3.25) and (3.26), we see that

$$v^n - v^* = -\frac{\gamma^n}{1-\gamma}c. \quad (3.27)$$

Similarly the change in the value from one iteration to the next is given by

$$\begin{aligned} \|v^{n+1} - v^n\| &= \left| \frac{\gamma^{n+1}}{1-\gamma} - \frac{\gamma^n}{1-\gamma} \right| c \\ &= \gamma^n \left| \frac{\gamma}{1-\gamma} - \frac{1}{1-\gamma} \right| c \\ &= \gamma^n \left| \frac{\gamma-1}{1-\gamma} \right| c \\ &= \gamma^n c. \end{aligned}$$

If we stop at iteration $n+1$, then it means that

$$\gamma^n c \leq \epsilon/2 \left(\frac{1-\gamma}{\gamma} \right). \quad (3.28)$$

If we choose ϵ so that (3.28) holds with equality, then our error bound (from 3.24) is

$$\begin{aligned} \|v^{n+1} - v^*\| &\leq \frac{\epsilon}{2} \\ &= \frac{\gamma^{n+1}}{1-\gamma} c. \end{aligned}$$

From (3.27) we know that the distance to the optimal solution is

$$|v^{n+1} - v^*| = \frac{\gamma^{n+1}}{1-\gamma} c,$$

which matches our bound.

This little exercise confirms that our bound on the error may be tight. It also shows that the error decreases geometrically at a rate determined by the discount factor. For this problem the error arises because we are approximating an infinite sum with a finite one. For more realistic dynamic programs, we also have the effect of trying to find the optimal policy. When the values are close enough that we have in fact found the optimal policy, then we have only a Markov reward process (a Markov chain where we earn rewards for each transition). Once our Markov reward process has reached steady state, it will behave just like the simple problem we have just solved, where c is the expected reward from each transition.

3.5 POLICY ITERATION

In policy iteration we choose a policy and then find the infinite horizon, discounted value of the policy. This value is then used to choose a new policy. The general algorithm is described in Figure 3.5. Policy iteration is popular for infinite horizon problems because of the ease with which we can find the value of a policy. As we showed in Section 3.3, the value of following policy π is given by

$$v^\pi = (I - \gamma P^\pi)^{-1} c^\pi. \quad (3.29)$$

While computing the inverse can be problematic as the state space grows, it is, at a minimum, a very convenient formula.

It is useful to illustrate the policy iteration algorithm in different settings. In the first, consider a batch replenishment problem where we have to replenish resources (raising capital, exploring for oil to expand known reserves, hiring people) where there are economies from ordering larger quantities. We might use a simple policy where if our level of resources $R_t < q$ for some lower limit q , then we order a quantity $a_t = Q - R_t$. This policy is parameterized by (q, Q) and is written

$$A^\pi(R_t) = \begin{cases} 0, & R_t \geq q, \\ Q - R_t, & R_t < q. \end{cases} \quad (3.30)$$

For a given set of parameters $\pi = (q, Q)$, we can compute a one-step transition matrix P^π and a contribution vector c^π .

Step 0. Initialization.

Step 0a. Select a policy π^0 .

Step 0b. Set $n = 1$.

Step 1. Given a policy π^{n-1} :

Step 1a. Compute the one-step transition matrix $P^{\pi^{n-1}}$.

Step 1b. Compute the contribution vector $c^{\pi^{n-1}}$ where the element for state s is given by $c^{\pi^{n-1}}(s) = C(s, A^{\pi^{n-1}})$.

Step 2. Let $v^{\pi,n}$ be the solution to

$$(I - \gamma P^{\pi^{n-1}})v = c^{\pi^{n-1}}.$$

Step 3. Find a policy π^n defined by

$$a^n(s) = \arg \max_{a \in \mathcal{A}} (C(a) + \gamma P^\pi v^n).$$

This requires that we compute an action for each state s .

Step 4. If $a^n(s) = a^{n-1}(s)$ for all states s , then set $a^* = a^n$; otherwise, set $n = n + 1$ and go to step 1.

Figure 3.5 Policy iteration.

Policies come in many forms. For the moment, we simply view a policy as a rule that tells us what decision to make when we are in a particular state. In Chapter 6, we describe four fundamental classes of policies, which can be used as the foundation for creating an even wider range of decision rules.

Given a transition matrix P^π and contribution vector c^π , we can use equation (3.29) to find v^π , where $v^\pi(s)$ is the discounted value of started in state s and following policy π . From this vector, we can infer a new policy by solving

$$a^n(s) = \arg \max_{a \in \mathcal{A}} (C(a) + \gamma P^\pi v^n) \quad (3.31)$$

for each state s . For our batch replenishment example, it turns out that we can show that $a^n(s)$ will have the same structure as that shown in (3.30). So we can either store $a^n(s)$ for each s or simply determine the parameters (q, Q) that correspond to the decisions produced by (3.31). The complete policy iteration algorithm is described in Figure 3.5.

The policy iteration algorithm is simple to implement and has fast convergence when measured in terms of the number of iterations. However, solving equation (3.29) is quite hard if the number of states is large. If the state space is small, we can use $v^\pi = (I - \gamma P^\pi)^{-1} c^\pi$, but the matrix inversion can be computationally expensive. For this reason, we may use a hybrid algorithm that combines the features of policy iteration and value iteration.

3.6 HYBRID VALUE-POLICY ITERATION

Value iteration is basically an algorithm that updates the value at each iteration and then determines a new policy given the new estimate of the value function. At any iteration, the value function is not the true, steady-state value of the policy. By contrast, policy iteration picks a policy and then determines the true, steady-state value of being in each state given the policy. Given this value, a new policy is chosen.

It is perhaps not surprising that policy iteration converges faster in terms of the number of iterations because it is doing a lot more work in each iteration (determining the true steady-state value of being in each state under a policy). Value iteration is much faster per iteration, but it is determining a policy given an approximation of a value function and then performing a very simple updating of the value function, which may be far from the true value function.

A hybrid strategy that combines features of both methods is to perform a somewhat more complete update of the value function before performing an update of the policy. Figure 3.6 outlines the procedure where the steady-state evaluation of the value function in equation (3.29) is replaced with a much easier iterative procedure (step 2 in Figure 3.6). This step is run for M iterations, where M is a user-controlled parameter that allows the exploration of the value of a better estimate of the value function. Not surprisingly, it will generally be the case that M should decline with the number of iterations as the overall process converges.

Step 0. Initialization:

- Set $n = 1$.
- Select a tolerance parameter ϵ and inner iteration limit M .
- Select some $v^0 \in \mathcal{V}$.

Step 1. Find a decision $a^n(s)$ for each s that satisfies

$$a^n(s) = \arg \max_{a \in \mathcal{A}} \left\{ C(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a) v^{n-1}(s') \right\},$$

which we represent as policy π^n .

Step 2. Partial policy evaluation:

- a. Set $m = 0$ and let $u^n(0) = c^\pi + \gamma P^\pi v^{n-1}$.
- b. If $\|u^n(0) - v^{n-1}\| < \epsilon(1 - \gamma)/2\gamma$, go to step 3.
- c. Else, while $m < M$ do the following:
 - i. $u^l(m+1) = c^{\pi^n} + \gamma P^{\pi^n} u^l(m) = \mathcal{M}^\pi u^l(m)$.
 - ii. Set $m = m + 1$ and repeat step i.
- d. Set $v^n = u^n(M)$, $n = n + 1$ and return to step 1.

Step 3. Set $a^\epsilon = a^{n+1}$ and stop.

Figure 3.6 Hybrid value/policy iteration.

3.7 AVERAGE REWARD DYNAMIC PROGRAMMING

There are settings where the natural objective function is to maximize the *average* contribution per unit time. Suppose that we start in state s . Then the average reward from starting in state s and following policy π is given by

$$\max_\pi F^\pi(s) = \lim_{T \rightarrow \infty} \frac{1}{T} \mathbb{E} \sum_{t=0}^T C(S_t, A^\pi(S_t)). \quad (3.32)$$

Here $F^\pi(s)$ is the expected reward *per time period*. In matrix form the total value of following a policy π over a horizon T can be written as

$$V_T^\pi = \sum_{t=0}^T (P^\pi)^t c^\pi,$$

where V_T^π is a column vector with element $V_T^\pi(s)$ giving the expected contribution over T time periods when starting in state s . We can get a sense of how $V_T^\pi(s)$ behaves by watching what happens as T becomes large. Assuming that our underlying Markov chain is ergodic (all the states communicate with each other with positive probability), we know that $(P^\pi)^T \rightarrow P^*$ where the rows of P^* are all the same.

Now define a column vector g given by

$$g^\pi = P^* c^\pi.$$

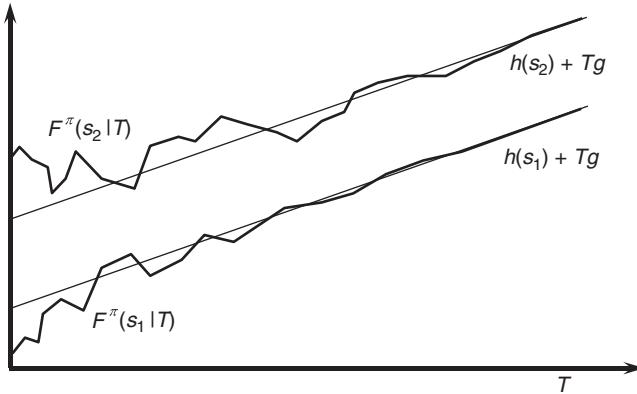


Figure 3.7 Cumulative contribution over a horizon T when starting in states s_1 and s_2 , showing growth approaching a rate that is independent of the starting state.

Since the rows of P^* are all the same, all the elements of g^π are the same, and each element gives the average contribution per time period using the steady state probability of being in each state. For finite T each element of the column vector V_T^π is not the same, since the contributions we earn in the first few time periods depends on our starting state. But it is not hard to see that as T grows large, we can write

$$V_T^\pi \rightarrow h^\pi + Tg^\pi,$$

where h^π captures the state-dependent differences in the total contribution, while g^π is the state-independent average contribution in the limit. Figure 3.7 illustrates the growth in V_T^π toward a linear function.

If we wish to find the policy that performs the best as $T \rightarrow \infty$, then clearly the contribution of h^π vanishes, and we want to focus on maximizing g^π , which we can now treat as a scalar.

3.8 THE LINEAR PROGRAMMING METHOD FOR DYNAMIC PROGRAMS

Theorem 3.4.1 showed us that if

$$v \geq \max_a \left(C(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a)v(s') \right),$$

then v is an upper bound (actually, a vector of upper bounds) on the value of being in each state. This means that the optimal solution, which satisfies $v^* = c + \gamma P v^*$, is the smallest value of v that satisfies this inequality. We can use this insight to formulate the problem of finding the optimal values as a linear program. Let β be

a vector with elements $\beta_s > 0, \forall s \in \mathcal{S}$. The optimal value function can be found by solving the following linear program:

$$\min_v \sum_{s \in \mathcal{S}} \beta_s v(s) \quad (3.33)$$

subject to

$$v(s) \geq C(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a)v(s') \quad \text{for all } s \text{ and } a. \quad (3.34)$$

The linear program has a $|\mathcal{S}|$ -dimensional decision vector (the value of being in each state), with $|\mathcal{S}| \times |\mathcal{A}|$ inequality constraints (equation (3.34)).

This formulation was viewed as primarily a theoretical result for many years, since it requires formulating a linear program where the number of constraints is equal to the number of states and actions. While even today this limits the size of problems it can solve, modern linear programming solvers can handle problems with tens of thousands of constraints without difficulty. This size is greatly expanded with the use of specialized algorithmic strategies, which are an active area of research as of this writing. The advantage of the LP method over value iteration is that it avoids the need for iterative learning with the geometric convergence exhibited by value iteration. Given the dramatic strides in the speed of linear programming solvers over the last decade, the relative performance of value iteration over the linear programming method is an unresolved question. However, this question only arises for problems with relatively small state and action spaces. While a linear program with 50,000 constraints is considered large, dynamic programs with 50,000 states and actions often arises with relatively small problems.

3.9 MONOTONE POLICIES*

One of the most dramatic success stories from the study of Markov decision processes has been the identification of the structure of optimal policies. A common example of structured policies is what are known as *monotone policies*. Simply stated, a monotone policy is one where the decision gets bigger as the state gets bigger, or the decision gets smaller as the state gets bigger (see examples).

■ EXAMPLE 3.1

A software company must decide when to ship the next release of its operating system. Let S_t be the total investment in the current version of the software. Let $a_t = 1$ denote the decision to ship the release in time period t while $a_t = 0$ means to keep investing in the system. The company adopts the rule that $a_t = 1$ if $S_t \geq \bar{S}$. Thus, as S_t gets bigger, a_t gets bigger (this is true even though a_t is equal to zero or one). ■

■ EXAMPLE 3.2

An oil company maintains stocks of oil reserves to supply its refineries for making gasoline. A supertanker comes from the Middle East each month, and the company can purchase different quantities from this shipment. Let R_t be the current inventory. The policy of the company is to order $a_t = Q - S_t$ if $S_t < R_t$. R is the reorder point, and Q is the “order up to” limit. The bigger S_t is, the less the company orders. ■

■ EXAMPLE 3.3

A mutual fund has to decide when to sell its holding in a company. Its policy is to sell the stock when the price \hat{p}_t is greater than a particular limit \bar{p} . ■

In each example the decision of what to do in each state is replaced by a function that determines the decision (otherwise known as a policy). The function typically depends on the choice of a few parameters. So, instead of determining the right action for each possible state, we only have to determine the parameters that characterize the function. Interestingly we do not need dynamic programming for this. Instead, we use dynamic programming to determine the structure of the optimal policy. This is a purely theoretical question, so the computational limitations of (discrete) dynamic programming are not relevant.

The study of monotone policies is included partly because it is an important part of the field of dynamic programming. It is also useful in the study of approximate dynamic programming because it yields properties of the value function. For example, in the process of showing that a policy is monotone, we also need to show that the value function is monotone (i.e., it increases or decreases with the state variable). Such properties can be exploited in the estimation of a value function approximation.

To demonstrate the analysis of a monotone policy, we consider a classic batch replenishment policy that arises when there is a random accumulation that is then released in batches. Examples include dispatching elevators or trucks, moving oil inventories away from producing fields in tankers, and moving trainloads of grain from grain elevators.

3.9.1 The Model

For our batch model we assume that resources accumulate and are then reduced using a batch process. For example, oil might accumulate in tanks before a tanker removes it. Money might accumulate in a cash account before it is swept into an investment.

Our model uses the following parameters:

c^r = fixed cost incurred each time we dispatch a new batch,

c^h = penalty per time period for holding a unit of the resource,

K = maximum size of a batch.

Our exogenous information process consists of

$$\begin{aligned} Q_t &= \text{quantity of new arrivals during time interval } t, \\ \mathbb{P}^Q(i) &= \text{Prob}(Q_t = i). \end{aligned}$$

Our state variable is

R_t = resources remaining at time t before we have made a decision to send a batch.

There are two decisions we have to make. The first is whether to dispatch a batch, and the second is how many resources to put in the batch. For this problem, once we make the decision to send a batch, we are going to make the batch as large as possible, so the “decision” of how large the batch should be seems unnecessary. It becomes more important when we later consider multiple resource types. For consistency with the more general problem with multiple resource types, we define

$$a_t = \begin{cases} 1 & \text{if a batch is sent at time } t, \\ 0 & \text{otherwise,} \end{cases}$$

b_t = number of resources to put in the batch.

In theory, we might be able to put a large number of resources in the batch, but we may face a nonlinear cost that makes this suboptimal. For the moment, we are going to assume that we always want to put as many as we can, so we set

$$\begin{aligned} b_t &= a_t \min\{K, R_t\}, \\ A^\pi(R_t) &= \text{decision function that returns } a_t \text{ and } b_t \text{ given } R_t. \end{aligned}$$

The transition function is described using

$$R_{t+1} = R_t - b_t + Q_{t+1}. \quad (3.35)$$

The objective function is modeled using

$$\begin{aligned} C_t(R_t, a_t, b_t) &= \text{cost incurred in period } t, \text{ given state } R_t \text{ and dispatch decision } a_t \\ &= c^r a_t + c^h(R_t - b_t). \end{aligned} \quad (3.36)$$

Our problem is to find the policy $A_t^\pi(R_t)$ that solves

$$\min_{\pi \in \Pi} \mathbb{E} \left\{ \sum_{t=0}^T C_t(R_t, A_t^\pi(R_t)) \right\}. \quad (3.37)$$

where Π is the set of policies. If we are managing a single asset class, then R_t and b_t are scalars, and the problem can be solved using standard backward dynamic programming techniques of the sort that were presented in the chapter (provided

that we have a probability model for the demand). In practice, many problems involve multiple asset classes, which makes standard techniques impractical. But we can use this simple problem to study the structure of the problem.

If R_t is a scalar, and if we know the probability distribution for Q_t , then we can solve the simple asset problem using backward dynamic programming. Indeed, this is one of the classic dynamic programming problems in operations research. However, the solution to this problem seems obvious. We should dispatch a batch whenever the level of resources R_t is greater than some number \bar{r}_t , which means we only have to find \bar{r}_t (if we have a steady-state infinite horizon problem, then we would have to find a single parameter \bar{r}). The remainder of this section helps establish the theoretical foundation for making this argument. While not difficult, the mathematical level of this presentation is somewhat higher than our usual presentation.

3.9.2 Submodularity and Other Stories

In the realm of optimization problems over a continuous set, it is important to know a variety of properties about the objective function (e.g., convexity/concavity, continuity, and boundedness). Similarly discrete problems require an understanding of the nature of the functions we are maximizing, but there is a different set of conditions that we need to establish.

One of the most important properties that we will need is supermodularity (submodularity if we are minimizing). We are interested in studying a function $g(u)$, $u \in \mathcal{U}$, where $\mathcal{U} \subseteq \Re^n$ is an n -dimensional space. Consider two vectors $u_1, u_2 \in \mathcal{U}$ where there is no particular relationship between u_1 and u_2 . Now define

$$u_1 \wedge u_2 = \min\{u_1, u_2\},$$

$$u_1 \vee u_2 = \max\{u_1, u_2\},$$

where the min and max are defined elementwise. Let $u^+ = u_1 \wedge u_2$ and $u^- = u_1 \vee u_2$. We first have to ask the question of whether $u^+, u^- \in \mathcal{U}$, since this is not guaranteed. For this purpose, we define the following:

Definition 3.9.1 *The space \mathcal{U} is a **lattice** if for each $u_1, u_2 \in \mathcal{U}$, then $u^+ = u_1 \wedge u_2 \in \mathcal{U}$ and $u^- = u_1 \vee u_2 \in \mathcal{U}$.*

The term “lattice” for these sets arises if we think of u_1 and u_2 as the northwest and southeast corners of a rectangle. In that configuration these corners are u^+ and u^- . If all four corners fall in the set (for any pair (u_1, u_2)), then the set can be viewed as containing many “squares,” similar to a lattice.

For our purposes we assume that \mathcal{U} is a lattice (if it is not, then we have to use a more general definition of the operators “ \vee ” and “ \wedge ”). If \mathcal{U} is a lattice, then a general definition of supermodularity is given by the following:

Definition 3.9.2 A function $g(u)$, $u \in \mathcal{U}$ is **supermodular** if it satisfies

$$g(u_1 \wedge u_2) + g(u_1 \vee u_2) \geq g(u_1) + g(u_2). \quad (3.38)$$

Supermodularity is the discrete analogue of a convex function. A function is **submodular** if the inequality in equation (3.38) is reversed. There is an alternative definition of supermodular when the function is defined on sets. Let \mathcal{U}_1 and \mathcal{U}_2 be two sets of elements, and let g be a function defined on these sets. Then we have

Definition 3.9.3 A function $g : \mathcal{U} \mapsto \mathbb{R}^1$ is **supermodular** if it satisfies

$$g(\mathcal{U}_1 \cup \mathcal{U}_2) + g(\mathcal{U}_1 \cap \mathcal{U}_2) \geq g(\mathcal{U}_1) + g(\mathcal{U}_2). \quad (3.39)$$

We may refer to definition 3.9.2 as the vector definition of supermodularity, while definition 3.9.3 as the set definition. We give both definitions for completeness, but our work uses only the vector definition.

In dynamic programming we are interested in functions of two variables, as in $f(s, a)$, where s is a state variable and a is a decision variable. We want to characterize the behavior of $f(s, a)$ as we change s and a . If we let $u = (s, a)$, then we can put this in the context of our definition above. Suppose that we have two states $s^+ \geq s^-$ (again, the inequality is applied elementwise) and two decisions $a^+ \geq a^-$. We form two vectors $u_1 = (s^+, a^-)$ and $u_2 = (s^-, a^+)$. With this definition, we find that $u_1 \vee u_2 = (s^+, a^+)$ and $u_1 \wedge u_2 = (s^-, a^-)$. This gives us the following:

Proposition 3.9.1 A function $g(s, a)$ is **supermodular** if for $s^+ \geq s^-$ and $a^+ \geq a^-$, then

$$g(s^+, a^+) + g(s^-, a^-) \geq g(s^+, a^-) + g(s^-, a^+). \quad (3.40)$$

For our purposes equation (3.40) will be the version we will use.

A common variation on the statement of a supermodular function is the equivalent condition

$$g(s^+, a^+) - g(s^-, a^+) \geq g(s^+, a^-) - g(s^-, a^-) \quad (3.41)$$

In this expression we are saying that the incremental change in s for larger values of a is greater than for smaller values of a . Similarly we may write the condition as

$$g(s^+, a^+) - g(s^+, a^-) \geq g(s^-, a^+) - g(s^-, a^-), \quad (3.42)$$

which states that an incremental change in a increases with s .

Some examples of supermodular functions include

1. If $g(s, a) = g_1(s) + g_2(a)$, meaning that it is separable, then (3.40) holds with equality.

2. $g(s, a) = h(s+a)$ where $h(\cdot)$ is convex and increasing.
3. $g(s, a) = sa, s, a \in \mathbb{N}^1$.

A concept that is related to supermodularity is *superadditivity*, defined by the following:

Definition 3.9.4 A superadditive function $f : \mathbb{N}^n \rightarrow \mathbb{N}^1$ satisfies

$$f(x) + f(y) \leq f(x+y). \quad (3.43)$$

Some authors use superadditivity and supermodularity interchangeably, but the concepts are not really equivalent, and we need to use both of them.

3.9.3 From Submodularity to Monotonicity

It seems intuitively obvious that we should dispatch a batch if the state R_t (the resources waiting to be served in a batch) is greater than some number (e.g., \bar{r}_t). The dispatch rule that says we should dispatch if $R_t \geq \bar{r}_t$ is known as a *control limit structure*. Similarly we might be holding an asset and we feel that we should sell it if the price p_t (which is the state of our asset) is above (or perhaps below) some number \bar{p}_t . A question arises: When is an optimal policy monotone? The following theorem establishes sufficient conditions for an optimal policy to be monotone.

Theorem 3.9.1 We want to maximize a total discounted contribution given that

- a. $C_t(R, a)$ is supermodular on $\mathcal{R} \times \mathcal{A}$.
- b. $\sum_{R' \in \mathcal{R}} \mathbb{P}(R'|R, a)v_{t+1}(R')$ is supermodular on $\mathcal{R} \times \mathcal{A}$.

Then there exists a decision rule $A^\pi(R)$ that is nondecreasing on \mathcal{R} .

The proof of this theorem is provided in Section 3.10.6.

In the presentation that follows, we need to show submodularity (instead of supermodularity) because we are minimizing costs rather than maximizing rewards.

It is obvious that $C_t(R, a)$ is nondecreasing in R . So it remains to show that $C_t(R, a)$ satisfies

$$C_t(R^+, 1) - C_t(R^-, 1) \leq C_t(R^+, 0) - C_t(R^-, 0). \quad (3.44)$$

Substituting equation (3.36) into (3.44), we must show that

$$c^r + c^h(R^+ - K)^+ - c^r - c^h(R^- - K)^+ \leq c^h R^+ - c^h R^-.$$

This simplifies to

$$(R^+ - K)^+ - (R^- - K)^+ \leq R^+ - R^-. \quad (3.45)$$

Since $R^+ \geq R^-$, $(R^+ - K)^+ = 0 \Rightarrow (R^- - K)^+ = 0$. This implies there are three possible cases for equation (3.45):

Case 1. $(R^+ - K)^+ > 0$ and $(R^- - K)^+ > 0$. Then (3.45) reduces to $R^+ - R^- = R^+ - R^-$.

Case 2. $(R^+ - K)^+ > 0$ and $(R^- - K)^+ = 0$. Then, (3.45) reduces to $R^- \leq K$, which follows since $(R^- - K)^+ = 0$ implies that $R^- \leq K$.

Case 3. $(R^+ - K)^+ = 0$ and $(R^- - K)^+ = 0$. Then (3.45) reduces to $R^- \leq R^+$, which is true by construction.

Now we have to show submodularity of $\sum_{R'=0}^{\infty} \mathbb{P}(R'|R, a) V(R')$. We will do this for the special case where the batch capacity is so large that it can be never exceeded. A proof is available for the finite capacity case, but it is much more difficult.

Submodularity requires that for $R^- \leq R^+$ we have

$$\begin{aligned} \sum_{R'=0}^{\infty} \mathbb{P}(R'|R^+, 1) V(R') - \sum_{R'=0}^{\infty} \mathbb{P}(R'|R^+, 0) V(R') &\leq \sum_{R'=0}^{\infty} \mathbb{P}(R'|R^-, 1) V(R') \\ &\quad - \sum_{R'=0}^{\infty} \mathbb{P}(R'|R^-, 0) V(R'). \end{aligned}$$

For the case that $R^-, R^+ \leq K$ we have

$$\begin{aligned} \sum_{R'=0}^{\infty} \mathbb{P}^A(R') V(R') - \sum_{R'=R^+}^{\infty} \mathbb{P}^A(R' - R^+) V(R') &\leq \sum_{R'=0}^{\infty} \mathbb{P}^A(R') V(R') \\ &\quad - \sum_{R'=R^-}^{\infty} \mathbb{P}^A(R' - R^-) V(R'), \end{aligned}$$

which simplifies to

$$\begin{aligned} \sum_{R'=0}^{\infty} \mathbb{P}^A(R') V(R') - \sum_{R'=0}^{\infty} \mathbb{P}^A(R') V(R' + R^+) &\leq \sum_{R'=0}^{\infty} \mathbb{P}^A(R') V(R') \\ &\quad - \sum_{R'=0}^{\infty} \mathbb{P}^A(R') V(R' + R^-). \end{aligned}$$

Since V is nondecreasing, we have $V(R' + R^+) \geq V(R' + R^-)$, proving the result.

3.10 WHY DOES IT WORK?**

The theory of Markov decision processes is elegant but not needed for computational work. An understanding of why Markov decision processes work will provide a deeper appreciation of the properties of these optimization problems.

Section 3.10.1 provides a proof that the optimal value function satisfies the optimality equations. Section 3.10.2 proves convergence of the value iteration algorithm. Section 3.10.3 then proves conditions under which value iteration increases or decreases monotonically to the optimal solution. Then Section 3.10.4 proves the bound on the error when value iteration satisfies the termination criterion given in Section 3.4.3. Section 3.10.5 closes with a discussion of deterministic and randomized policies, along with a proof that deterministic policies are always at least as good as a randomized policy.

3.10.1 The Optimality Equations

Until now we have been presenting the optimality equations as though they were a fundamental law of some sort. To be sure, they can easily look as though they were intuitively obvious, but it is still important to establish the relationship between the original optimization problem and the optimality equations. Because these equations are the foundation of dynamic programming, we will work through the steps of proving that they are actually true.

We start by remembering the original optimization problem:

$$F_t^\pi(S_t) = \mathbb{E} \left\{ \sum_{t'=t}^{T-1} C_{t'}(S_{t'}, A_{t'}^\pi(S_{t'})) + C_T(S_T) | S_t \right\}. \quad (3.46)$$

Since (3.46) is, in general, exceptionally difficult to solve, we resort to the optimality equations

$$V_t^\pi(S_t) = C_t(S_t, A_t^\pi(S_t)) + \mathbb{E} \{ V_{t+1}^\pi(S_{t+1}) | S_t \}. \quad (3.47)$$

Our challenge is to show that these are the same. In order to establish this result, it is going to help if we first prove the following:

Lemma 3.10.1 *Let S_t be a state variable that captures the relevant history up to time t , and let $F_{t'}(S_{t+1})$ be some function measured at time $t' \geq t + 1$ conditioned on the random variable S_{t+1} . Then*

$$\mathbb{E} [\mathbb{E}\{F_{t'}|S_{t+1}\}|S_t] = \mathbb{E} [F_{t'}|S_t]. \quad (3.48)$$

Proof This lemma is variously known as the law of iterated expectations or the tower property. Assume, for simplicity, that $F_{t'}$ is a discrete finite random variable that takes outcomes in \mathcal{F} . We start by writing

$$\mathbb{E}\{F_{t'}|S_{t+1}\} = \sum_{f \in \mathcal{F}} f \mathbb{P}(F_{t'} = f | S_{t+1}). \quad (3.49)$$

Recognizing that S_{t+1} is a random variable, we may take the expectation of both sides of (3.49), conditioned on S_t as follows:

$$\mathbb{E} [\mathbb{E}\{F_{t'}|S_{t+1}\}|S_t] = \sum_{S_{t+1} \in \mathcal{S}} \sum_{f \in \mathcal{F}} f \mathbb{P}(F_{t'} = f | S_{t+1}, S_t) \mathbb{P}(S_{t+1} = S_{t+1} | S_t). \quad (3.50)$$

First, we observe that we can write $\mathbb{P}(F_{t'} = f | S_{t+1}, S_t) = \mathbb{P}(F_{t'} = f | S_{t+1})$ because conditioning on S_{t+1} makes all prior history irrelevant. Next we can reverse the summations on the right-hand side of (3.50) (some technical conditions have to be satisfied to do this, but these are satisfied if the random variables are discrete and finite). This means that

$$\begin{aligned} \mathbb{E} [\mathbb{E}\{F_{t'} | S_{t+1} = S_{t+1}\} | S_t] &= \sum_{f \in \mathcal{F}} \sum_{S_{t+1} \in \mathcal{S}} f \mathbb{P}(F_{t'} = f | S_{t+1}, S_t) \mathbb{P}(S_{t+1} = S_{t+1} | S_t) \\ &= \sum_{f \in \mathcal{F}} f \sum_{S_{t+1} \in \mathcal{S}} \mathbb{P}(F_{t'} = f, S_{t+1} | S_t) \\ &= \sum_{f \in \mathcal{F}} f \mathbb{P}(F_{t'} = f | S_t) \\ &= \mathbb{E}[F_{t'} | S_t], \end{aligned}$$

which proves our result. Note that the essential step in the proof occurs in the first step when we add S_t to the conditioning. \square

We are now ready to show the following:

Proposition 3.10.1 $F_t^\pi(S_t) = V_t^\pi(S_t)$.

Proof To prove that (3.46) and (3.47) are equal, we use a standard trick in dynamic programming: proof by induction. Clearly, $F_T^\pi(S_T) = V_T^\pi(S_T) = C_T(S_T)$. So we assume that it holds for $t + 1, t + 2, \dots, T$. We want to show that the same is true for t . This means that we can write

$$\begin{aligned} V_t^\pi(S_t) &= C_t(S_t, A_t^\pi(S_t)) \\ &\quad + \mathbb{E} \left[\underbrace{\mathbb{E} \left\{ \sum_{t'=t+1}^{T-1} C_{t'}(S_{t'}, A_{t'}^\pi(S_{t'})) + C_t(S_T(\omega)) \middle| S_{t+1} \right\}}_{F_{t+1}^\pi(S_{t+1})} \middle| S_t \right]. \end{aligned}$$

We then use lemma 3.10.1 to write $\mathbb{E}[\mathbb{E}\{\dots | S_{t+1}\} | S_t] = \mathbb{E}[\dots | S_t]$. Hence

$$V_t^\pi(S_t) = C_t(S_t, A_t^\pi(S_t)) + \mathbb{E} \left[\sum_{t'=t+1}^{T-1} C_{t'}(S_{t'}, A_{t'}^\pi(S_{t'})) + C_t(S_T) | S_t \right].$$

When we condition on $S_t, A_t^\pi(S_t)$ (and therefore $C_t(S_t, A_t^\pi(S_t))$) is deterministic. So we can pull the expectation out to the front, giving

$$\begin{aligned} V_t^\pi(S_t) &= \mathbb{E} \left[\sum_{t'=t}^{T-1} C_{t'}(S_{t'}, y_{t'}(S_{t'})) + C_t(S_T) | S_t \right] \\ &= F_t^\pi(S_t), \end{aligned}$$

which proves our result. \square

Using equation (3.47), we have a backward recursion for calculating $V_t^\pi(S_t)$ for a given policy π . Now that we can find the expected reward for a given π , we would like to find the best π . That is, we want to find that

$$F_t^*(S_t) = \max_{\pi \in \Pi} F_t^\pi(S_t).$$

If the set Π is infinite, we replace the “max” with “sup.” We solve this problem by solving the optimality equations. These are

$$V_t(S_t) = \max_{a \in \mathcal{A}} \left(C_t(S_t, a) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, a) V_{t+1}(s') \right). \quad (3.51)$$

We are claiming that if we find the set of V 's that solves (3.51), then we have found the policy that optimizes F_t^π . We state this claim formally as:

Theorem 3.10.1 *Let $V_t(S_t)$ be a solution to equation (3.51). Then*

$$\begin{aligned} F_t^* &= V_t(S_t) \\ &= \max_{\pi \in \Pi} F_t^\pi(S_t). \end{aligned}$$

Proof The proof is in two parts. First we show by induction that $V_t(S_t) \geq F_t^*(S_t)$ for all $S_t \in \mathcal{S}$ and $t = 0, 1, \dots, T-1$. Then we show that the reverse inequality is true, which gives us the result.

Part 1

We resort again to our proof by induction. Since $V_T(S_T) = C_t(S_T) = F_T^\pi(S_T)$ for all S_T and all $\pi \in \Pi$, we get that $V_T(S_T) = F_T^*(S_T)$.

Assume that $V_{t'}(S_{t'}) \geq F_{t'}^*(S_{t'})$ for $t' = t+1, t+2, \dots, T$, and let π be an arbitrary policy. For $t' = t$, the optimality equation tells us that

$$V_t(S_t) = \max_{a \in \mathcal{A}} \left(C_t(S_t, a) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, a) V_{t+1}(s') \right).$$

By the induction hypothesis, $F_{t+1}^*(s) \leq V_{t+1}(s)$. So we get

$$V_t(S_t) \geq \max_{a \in \mathcal{A}} \left(C_t(S_t, a) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, a) F_{t+1}^*(s') \right).$$

Of course, we have that $F_{t+1}^*(s) \geq F_{t+1}^\pi(s)$ for an arbitrary π . Also let $A^\pi(S_t)$ be the decision that would be chosen by policy π when in state S_t . Then

$$\begin{aligned} V_t(S_t) &\geq \max_{a \in \mathcal{A}} \left(C_t(S_t, a) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, a) F_{t+1}^\pi(s') \right) \\ &\geq C_t(S_t, A^\pi(S_t)) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, A^\pi(S_t)) F_{t+1}^\pi(s') \\ &= F_t^\pi(S_t). \end{aligned}$$

This means that

$$V_t(S_t) \geq F_t^\pi(S_t) \quad \text{for all } \pi \in \Pi,$$

which proves part 1.

Part 2

Now we are going to prove the inequality from the other side. Specifically, we want to show that for any $\epsilon > 0$ there exists a policy π that satisfies

$$F_t^\pi(S_t) + (T - t)\epsilon \geq V_t(S_t). \quad (3.52)$$

To do this, we start with the definition

$$V_t(S_t) = \max_{a \in \mathcal{A}} \left(C_t(S_t, a) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, a) V_{t+1}(s') \right). \quad (3.53)$$

We let $a_t(S_t)$ be the decision rule that solves (3.53). This rule corresponds to the policy π . In general, the set \mathcal{A} may be infinite, whereupon we have to replace the “max” with a “sup” and handle the case where an optimal decision may not exist. For this case we know that we can design a decision rule $a_t(S_t)$ that returns a decision a that satisfies

$$V_t(S_t) \leq C_t(S_t, a) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, a) V_{t+1}(s') + \epsilon. \quad (3.54)$$

We can prove (3.52) by induction. We first note that (3.52) is true for $t = T$ since $F_T^\pi(S_t) = V_T(S_T)$. Now assume that it is true for $t' = t + 1, t + 2, \dots, T$. We already know that

$$F_t^\pi(S_t) = C_t(S_t, A^\pi(S_t)) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, A^\pi(S_t)) F_{t+1}^\pi(s').$$

We can use our induction hypothesis, which says that $F_{t+1}^\pi(s') \geq V_{t+1}(s') - (T - (t + 1))\epsilon$, to get

$$\begin{aligned} F_t^\pi(S_t) &\geq C_t(S_t, A^\pi(S_t)) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, A^\pi(S_t)) [V_{t+1}(s') - (T - (t + 1))\epsilon] \\ &= C_t(S_t, A^\pi(S_t)) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, A^\pi(S_t)) V_{t+1}(s') \\ &\quad - \sum_{s' \in \mathcal{S}} p_t(s'|S_t, A^\pi(S_t)) [(T - t - 1)\epsilon] \\ &= \left\{ C_t(S_t, A^\pi(S_t)) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, A^\pi(S_t)) V_{t+1}(s') + \epsilon \right\} - (T - t)\epsilon. \end{aligned}$$

Now, using equation (3.54), we replace the term in brackets with the smaller $V_t(S_t)$ (equation (3.54)):

$$F_t^\pi(S_t) \geq V_t(S_t) - (T - t)\epsilon,$$

which proves the induction hypothesis. We have shown that

$$F_t^*(S_t) + (T - t)\epsilon \geq F_t^\pi(S_t) + (T - t)\epsilon \geq V_t(S_t) \geq F_t^*(S_t).$$

This proves the result. \square

Now we know that solving the optimality equations also gives us the optimal value function. This is our most powerful result because we can solve the optimality equations for many problems that cannot be solved any other way.

3.10.2 Convergence of Value Iteration

We undertake here the proof that the basic value function iteration converges to the optimal solution. This is not only an important result; it is also an elegant one that brings some powerful theorems into play. The proof is also quite short. However, we will need some mathematical preliminaries:

Definition 3.10.1 *Let \mathcal{V} be a set of (bounded, real-valued) functions and define the norm of v by*

$$\|v\| = \sup_{s \in \mathcal{S}} v(s)$$

where we replace the “sup” with a “max” when the state space is finite. Since \mathcal{V} is closed under addition and scalar multiplication and has a norm, it is a normed linear space.

Definition 3.10.2 $T : \mathcal{V} \rightarrow \mathcal{V}$ is a **contraction mapping** if there exists a γ , $0 \leq \gamma < 1$, such that

$$\|Tv - Tu\| \leq \gamma \|v - u\|.$$

Definition 3.10.3 A sequence $v^n \in \mathcal{V}, n = 1, 2, \dots$, is said to be a **Cauchy sequence** if for all $\epsilon > 0$ there exists N such that for all $n, m \geq N$:

$$\|v^n - v^m\| < \epsilon.$$

Definition 3.10.4 A normed linear space is **complete** if every Cauchy sequence contains a limit point in that space.

Definition 3.10.5 A **Banach space** is a complete normed linear space.

Definition 3.10.6 We define the norm of a matrix Q as

$$\|Q\| = \max_{s \in S} \sum_{j \in S} |q(j|s)|,$$

that is, the largest row sum of the matrix. If Q is a one-step transition matrix, then $\|Q\| = 1$.

Definition 3.10.7 The **triangle inequality** means that given two vectors $a, b \in \mathbb{R}^n$:

$$\|a + b\| \leq \|a\| + \|b\|.$$

The triangle inequality is commonly used in proofs because it helps us establish bounds between two solutions (and in particular, between a solution and the optimum).

We now state and prove one of the famous theorems in applied mathematics and then use it immediately to prove convergence of the value iteration algorithm.

Theorem 3.10.2 (Banach Fixed-Point Theorem) Let \mathcal{V} be a Banach space, and let $T : \mathcal{V} \rightarrow \mathcal{V}$ be a contraction mapping. Then

- a. there exists a unique $v^* \in \mathcal{V}$ such that $Tv^* = v^*$.
- b. for an arbitrary $v^0 \in \mathcal{V}$, the sequence v^n defined by $v^{n+1} = Tv^n = T^{n+1}v^0$ converges to v^* .

Proof We start by showing that the distance between two vectors v^n and v^{n+m} goes to zero for sufficiently large n and by writing the difference $v^{n+m} - v^n$ using

$$\begin{aligned} v^{n+m} - v^n &= v^{n+m} - v^{n+m-1} + v^{n+m-1} - \dots - v^{n+1} + v^{n+1} - v^n \\ &= \sum_{k=0}^{m-1} (v^{n+k+1} - v^{n+k}). \end{aligned}$$

Taking norms of both sides and invoking the triangle inequality gives

$$\begin{aligned}
 \|v^{n+m} - v^n\| &= \left\| \sum_{k=0}^{m-1} (v^{n+k+1} - v^{n+k}) \right\| \\
 &\leq \sum_{k=0}^{m-1} \|v^{n+k+1} - v^{n+k}\| \\
 &= \sum_{k=0}^{m-1} \|(T^{n+k} v^1 - T^{n+k} v^0)\| \\
 &\leq \sum_{k=0}^{m-1} \gamma^{n+k} \|v^1 - v^0\| \\
 &= \frac{\gamma^n (1 - \gamma^m)}{(1 - \gamma)} \|v^1 - v^0\|. \tag{3.55}
 \end{aligned}$$

Since $\gamma < 1$, for sufficiently large n the right-hand side of (3.55) can be made arbitrarily small, which means that v^n is a Cauchy sequence. Since \mathcal{V} is *complete*, it must be that v^n has a limit point v^* . From this we conclude that

$$\lim_{n \rightarrow \infty} v^n \rightarrow v^*. \tag{3.56}$$

We now want to show that v^* is a fixed point of the mapping T . To show this, we observe that

$$0 \leq \|Tv^* - v^*\| \tag{3.57}$$

$$= \|Tv^* - v^n + v^n - v^*\| \tag{3.58}$$

$$\leq \|Tv^* - v^n\| + \|v^n - v^*\| \tag{3.59}$$

$$= \|Tv^* - Tv^{n-1}\| + \|v^n - v^*\| \tag{3.60}$$

$$\leq \gamma \|v^* - v^{n-1}\| + \|v^n - v^*\|. \tag{3.61}$$

Equation (3.57) comes from the properties of a norm. We play our standard trick in (3.58) of adding and subtracting a quantity (in this case, v^n), which sets up the triangle inequality in (3.59). Using $v^n = Tv^{n-1}$ gives us (3.60). The inequality in (3.61) is based on the assumption of the theorem that T is a contraction mapping. From (3.56) we know that

$$\lim_{n \rightarrow \infty} \|v^* - v^{n-1}\| = \lim_{n \rightarrow \infty} \|v^n - v^*\| = 0. \tag{3.62}$$

Combining (3.57), (3.61), and (3.62) gives

$$0 \leq \|Tv^* - v^*\| \leq 0,$$

from which we conclude that

$$\|Tv^* - v^*\| = 0,$$

which means that $Tv^* = v^*$.

We can prove uniqueness by contradiction. Suppose that there are two limit points that we represent as v^* and u^* . The assumption that T is a contraction mapping requires that

$$\|Tv^* - Tu^*\| \leq \gamma \|v^* - u^*\|.$$

But, if v^* and u^* are limit points, then $Tv^* = v^*$ and $Tu^* = u^*$, which means

$$\|v^* - u^*\| \leq \gamma \|v^* - u^*\|.$$

Since $\gamma < 1$, this is a contradiction, which means that it must be true that $v^* = u^*$. \square

We can now show that the value iteration algorithm converges to the optimal solution if we can establish that \mathcal{M} is a contraction mapping. So we need to show the following:

Proposition 3.10.2 *If $0 \leq \gamma < 1$, then \mathcal{M} is a contraction mapping on \mathcal{V} .*

Proof Let $u, v \in \mathcal{V}$, and assume that $\mathcal{M}v \geq \mathcal{Mu}$ where the inequality is applied elementwise. For a particular state s let

$$a_s^*(v) \in \arg \max_{a \in \mathcal{A}} \left(C(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a)v(s') \right),$$

where we assume that a solution exists. Then

$$0 \leq \mathcal{M}v(s) - \mathcal{Mu}(s) \tag{3.63}$$

$$\begin{aligned} &= C(s, a_s^*(v)) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a_s^*(v))v(s') \\ &\quad - \left(C(s, a_s^*(u)) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a_s^*(u))u(s') \right) \end{aligned} \tag{3.64}$$

$$\begin{aligned} &\leq C(s, a_s^*(v)) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a_s^*(v))v(s') \\ &\quad - \left(C(s, a_s^*(v)) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a_s^*(v))u(s') \right) \end{aligned} \tag{3.65}$$

$$= \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a_s^*(v)) [v(s') - u(s')] \quad (3.66)$$

$$\leq \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a_s^*(v)) \|v - u\| \quad (3.67)$$

$$= \gamma \|v - u\| \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a_s^*(v)) \quad (3.68)$$

$$= \gamma \|v - u\|. \quad (3.69)$$

Equation (3.63) is true by assumption, while (3.64) holds by definition. The inequality in (3.65) holds because $a_s^*(v)$ is not optimal when the value function is u , giving a reduced value in the second set of parentheses. Equation (3.66) is a simple reduction of (3.65). Equation (3.67) forms an upper bound because the definition of $\|v - u\|$ is to replace all the elements $[v(s) - u(s)]$ with the largest element of this vector. Since this is now a vector of constants, we can pull it outside of the summation, giving us (3.68), which then easily reduces to (3.69) because the probabilities add up to one.

This result states that if $\mathcal{M}v(s) \geq \mathcal{Mu}(s)$, then $\mathcal{M}v(s) - \mathcal{Mu}(s) \leq \gamma|v(s) - u(s)|$. If we start by assuming that $\mathcal{M}v(s) \leq \mathcal{Mu}(s)$, then the same reasoning produces $\mathcal{M}v(s) - \mathcal{Mu}(s) \geq -\gamma|v(s) - u(s)|$. This means that we have

$$|\mathcal{M}v(s) - \mathcal{Mu}(s)| \leq \gamma|v(s) - u(s)| \quad (3.70)$$

for all states $s \in \mathcal{S}$. From the definition of our norm, we can write

$$\begin{aligned} \sup_{s \in \mathcal{S}} |\mathcal{M}v(s) - \mathcal{Mu}(s)| &= \|\mathcal{M}v - \mathcal{Mu}\| \\ &\leq \gamma \|v - u\|. \end{aligned}$$

This means that \mathcal{M} is a contraction mapping, which means that the sequence v^n generated by $v^{n+1} = \mathcal{M}v^n$ converges to a unique limit point v^* that satisfies the optimality equations. \square

3.10.3 Monotonicity of Value Iteration

Infinite horizon dynamic programming provides a compact way to study the theoretical properties of these algorithms. The insights gained here are applicable to problems even when we cannot apply this model, or these algorithms, directly.

We assume throughout our discussion of infinite horizon problems that the reward function is bounded over the domain of the state space. This assumption is virtually always satisfied in practice, but notable exceptions exist. For example, the assumption is violated if we are maximizing a utility function that depends on the log of the resources we have at hand (the resources may be bounded, but the function is unbounded if the resources are allowed to hit zero).

Our first result establishes a monotonicity property that can be exploited in the design of an algorithm.

Theorem 3.10.3 *For a vector $v \in \mathcal{V}$,*

- a. *if v satisfies $v \geq \mathcal{M}v$, then $v \geq v^*$.*
- b. *if v satisfies $v \leq \mathcal{M}v$, then $v \leq v^*$.*
- c. *if v satisfies $v = \mathcal{M}v$, then v is the unique solution to this system of equations and $v = v^*$.*

Proof Part a requires that

$$v \geq \max_{\pi \in \Pi} \{c^\pi + \gamma P^\pi v\} \quad (3.71)$$

$$\geq c^{\pi_0} + \gamma P^{\pi_0} v \quad (3.72)$$

$$\geq c^{\pi_0} + \gamma P^{\pi_0} (c^{\pi_1} + \gamma P^{\pi_1} v) \quad (3.73)$$

$$= c^{\pi_0} + \gamma P^{\pi_0} c^{\pi_1} + \gamma^2 P^{\pi_0} P^{\pi_1} v.$$

Equation (3.71) is true by assumption (part a of the theorem) and equation (3.72) is true because π_0 is some policy that is not necessarily optimal for the vector v . By the similar reasoning, equation (3.73) is true because π_1 is another policy which, again, is not necessarily optimal. Using $P^{\pi,(t)} = P^{\pi_0} P^{\pi_1} \dots P^{\pi_t}$, we obtain by induction

$$v \geq c^{\pi_0} + \gamma P^{\pi_0} c^{\pi_1} + \dots + \gamma^{t-1} P^{\pi_0} P^{\pi_1} \dots P^{\pi_{t-1}} c^{\pi_t} + \gamma^t P^{\pi,(t)} v. \quad (3.74)$$

Recall that

$$v^\pi = \sum_{t=0}^{\infty} \gamma^t P^{\pi,(t)} c^{\pi_t}. \quad (3.75)$$

Breaking the sum in (3.75) into two parts allows us to rewrite the expansion in (3.74) as

$$v \geq v^\pi - \sum_{t'=t+1}^{\infty} \gamma^{t'} P^{\pi,(t')} c^{\pi_{t'+1}} + \gamma^t P^{\pi,(t)} v. \quad (3.76)$$

Taking the limit of both sides of (3.76) as $t \rightarrow \infty$ gives us

$$v \geq \lim_{t \rightarrow \infty} v^\pi - \sum_{t'=t+1}^{\infty} \gamma^{t'} P^{\pi,(t')} c^{\pi_{t'+1}} + \gamma^t P^{\pi,(t)} v \quad (3.77)$$

$$\geq v^\pi \quad \forall \pi \in \Pi. \quad (3.78)$$

The limit in (3.77) exists as long as the reward function c^π is bounded and $\gamma < 1$. Because (3.78) is true for all $\pi \in \Pi$, it is also true for the optimal policy, which means that

$$\begin{aligned} v &\geq v^{\pi^*} \\ &= v^*, \end{aligned}$$

which proves part a of the theorem. Part b can be proved in an analogous way. Parts a and b mean that $v \geq v^*$ and $v \leq v^*$. If $v = \mathcal{M}v$, then we satisfy the preconditions of both parts a and b, which means they are both true and therefore we must have $v = v^*$. \square

This result means that if we start with a vector that is higher than the optimal vector, then we will decline monotonically to the optimal solution (almost—we have not quite proved that we actually get to the optimal). Alternatively, if we start below the optimal vector, we will rise to it. Note that it is not always easy to find a vector v that satisfies either condition a or b of the theorem. In problems where the rewards can be positive and negative, this can be tricky.

3.10.4 Bounding the Error from Value Iteration

We now want to establish a bound on our error from value iteration, which will establish our stopping rule. We propose two bounds: one on the value function estimate that we terminate with and one for the long-run value of the decision rule that we terminate with. To define the latter, let π^ϵ be the policy that satisfies our stopping rule, and let v^{π^ϵ} be the infinite horizon value of following policy π^ϵ .

Theorem 3.10.4 *If we apply the value iteration algorithm with stopping parameter ϵ and the algorithm terminates at iteration n with value function v^{n+1} , then*

$$\|v^{n+1} - v^*\| \leq \frac{\epsilon}{2}, \quad (3.79)$$

and

$$\|v^{\pi^\epsilon} - v^*\| \leq \epsilon. \quad (3.80)$$

Proof We start by writing

$$\begin{aligned} \|v^{\pi^\epsilon} - v^*\| &= \|v^{\pi^\epsilon} - v^{n+1} + v^{n+1} - v^*\| \\ &\leq \|v^{\pi^\epsilon} - v^{n+1}\| + \|v^{n+1} - v^*\|. \end{aligned} \quad (3.81)$$

Recall that π^ϵ is the policy that solves $\mathcal{M}v^{n+1}$, which means that $\mathcal{M}^{\pi^\epsilon}v^{n+1} = \mathcal{M}v^{n+1}$. This allows us to rewrite the first term on the right-hand side of (3.81) as

$$\begin{aligned} \|v^{\pi^\epsilon} - v^{n+1}\| &= \|\mathcal{M}^{\pi^\epsilon}v^{\pi^\epsilon} - \mathcal{M}v^{n+1} + \mathcal{M}v^{n+1} - v^{n+1}\| \\ &\leq \|\mathcal{M}^{\pi^\epsilon}v^{\pi^\epsilon} - \mathcal{M}v^{n+1}\| + \|\mathcal{M}v^{n+1} - v^{n+1}\| \\ &= \|\mathcal{M}^{\pi^\epsilon}v^{\pi^\epsilon} - \mathcal{M}^{\pi^\epsilon}v^{n+1}\| + \|\mathcal{M}v^{n+1} - \mathcal{M}v^n\| \\ &\leq \gamma \|v^{\pi^\epsilon} - v^{n+1}\| + \gamma \|v^{n+1} - v^n\|. \end{aligned}$$

Solving for $\|v^{\pi^\epsilon} - v^{n+1}\|$ gives

$$\|v^{\pi^\epsilon} - v^{n+1}\| \leq \frac{\gamma}{1-\gamma} \|v^{n+1} - v^n\|.$$

We can use similar reasoning applied to the second term in equation (3.81) to show that

$$\|v^{n+1} - v^*\| \leq \frac{\gamma}{1-\gamma} \|v^{n+1} - v^n\|. \quad (3.82)$$

The value iteration algorithm stops when $\|v^{n+1} - v^n\| \leq \epsilon(1-\gamma)/2\gamma$. Substituting this in (3.82) gives

$$\|v^{n+1} - v^*\| \leq \frac{\epsilon}{2}. \quad (3.83)$$

Recognizing that the same bound applies to $\|v^{\pi^\epsilon} - v^{n+1}\|$ and combining these with (3.81) gives us

$$\|v^{\pi^\epsilon} - v^*\| \leq \epsilon,$$

which completes our proof. \square

3.10.5 Randomized Policies

We have implicitly assumed that for each state, we want a single action. An alternative would be to choose a policy probabilistically from a family of policies. If a state produces a single action, we say that we are using a *deterministic policy*. If we are randomly choosing an action from a set of actions probabilistically, we say we are using a *randomized policy*.

Randomized policies may arise because of the nature of the problem. For example, you wish to purchase something at an auction, but you are unable to attend yourself. You may have a simple rule (“purchase it as long as the price is under a specific amount”), but you cannot assume that your representative will apply the same rule. You can choose a representative, and in doing so, you are effectively choosing the probability distribution from which the action will be chosen.

Behaving randomly also plays a role in two-player games. If you make the same decision each time in a particular state, your opponent may be able to predict your behavior and gain an advantage. For example, as an institutional investor you may tell a bank that you are not willing to pay any more than \$14 for a new offering of stock, while in fact you are willing to pay up to \$18. If you always bias your initial prices by \$4, the bank will be able to guess what you are willing to pay.

When we can only influence the likelihood of an action, then we have an instance of a randomized MDP. Let

$$q_t^\pi(a|S_t) = \begin{aligned} &\text{probability that decision } a \text{ will be taken at time } t \text{ given state} \\ &S_t \text{ and policy } \pi \text{ (more precisely, decision rule } A^\pi). \end{aligned}$$

In this case our optimality equations look like

$$V_t^*(S_t) = \max_{\pi \in \Pi^{MR}} \sum_{a \in \mathcal{A}} \left[q_t^\pi(a|S_t) \left(C_t(S_t, a) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, a) V_{t+1}^*(s') \right) \right]. \quad (3.84)$$

Now let us consider the single best action that we could take. Calling this a^* , we can find it using

$$a^* = \arg \max_{a \in \mathcal{A}} \left[C_t(S_t, a) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, a) V_{t+1}^*(s') \right].$$

This means that

$$C_t(S_t, a^*) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, a^*) V_{t+1}^*(s') \geq C_t(S_t, a) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, a) V_{t+1}^*(s') \quad (3.85)$$

for all $a \in \mathcal{A}$. Substituting (3.85) back into (3.84) gives us

$$\begin{aligned} V_t^*(S_t) &= \max_{\pi \in \Pi^{MR}} \sum_{a \in \mathcal{A}} \left\{ q_t^\pi(a|S_t) \left(C_t(S_t, a) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, a) V_{t+1}^*(s') \right) \right\} \\ &\leq \max_{\pi \in \Pi^{MR}} \sum_{a \in \mathcal{A}} \left\{ q_t^\pi(a|S_t) \left(C_t(S_t, a^*) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, a^*) V_{t+1}^*(s') \right) \right\} \\ &= C_t(S_t, a^*) + \sum_{s' \in \mathcal{S}} p_t(s'|S_t, a^*) V_{t+1}^*(s'). \end{aligned}$$

What this means is that if you have a choice between picking exactly the action you want versus picking a probability distribution over potentially optimal and nonoptimal actions, you would always prefer to pick exactly the best action. Clearly, this is not a surprising result.

The value of randomized policies arises primarily in two-person games, where one player tries to anticipate the actions of the other player. In such situations part of the state variable is the estimate of what the other play will do when the game is in a particular state. By randomizing his behavior, a player reduces the ability of the other player to anticipate his moves.

3.10.6 Optimality of Monotone Policies

The foundational result that we use is the following technical lemma:

Lemma 3.10.2 *If a function $g(s, a)$ is supermodular, then*

$$a^*(s) = \max \left\{ a' \in \arg \max_a g(s, a) \right\} \quad (3.86)$$

is monotone and nondecreasing in s .

If the function $g(s, a)$ has a unique, optimal $a^*(s)$ for each value of s , then we can replace (3.86) with

$$a^*(s) = \max_a g(s, a). \quad (3.87)$$

Discussion The lemma is saying that if $g(s, a)$ is supermodular, then as s grows larger, the optimal value of a given s will grow larger. When we use the version of supermodularity given in equation (3.42), we see that the condition implies that as the state becomes larger, the value of increasing the decision also grows. As a result it is not surprising that the condition produces a decision rule that is monotone in the state vector.

Proof of the lemma Assume that $s^+ \geq s^-$, and choose $a \leq a^*(s^-)$. Since $a^*(s)$ is, by definition, the best value of a given s , we have

$$g(s^-, a^*(s^-)) - g(s^-, a) \geq 0. \quad (3.88)$$

The inequality arises because $a^*(s^-)$ is the best value of a given s^- . Supermodularity requires that

$$g(s^-, a) + g(s^+, a^*(s^-)) \geq g(s^-, a^*(s^-)) + g(s^+, a). \quad (3.89)$$

Rearranging (3.89) gives us

$$g(s^+, a^*(s^-)) \geq \underbrace{\{g(s^-, a^*(s^-)) - g(s^-, a)\}}_{\geq 0} + g(s^+, a) \quad \forall a \leq a^*(s^-) \quad (3.90)$$

$$\geq g(s^+, a) \quad \forall a \leq a^*(s^-). \quad (3.91)$$

We obtain equation (3.91) because the term in brackets in (3.90) is nonnegative (from (3.88)).

Clearly,

$$g(s^+, a^*(s^+)) \geq g(s^+, a^*(s^-))$$

because $a^*(s^+)$ optimizes $g(s^+, a)$. This means that $a^*(s^+) \geq a^*(s^-)$ since, otherwise, we would simply have chosen $a = a^*(s^-)$. \square

Just as the sum of concave functions is concave, we have the following:

Proposition 3.10.3 *The sum of supermodular functions is supermodular.*

The proof follows immediately from the definition of supermodularity, so we leave it as one of those proverbial exercises for the reader.

The main theorem regarding monotonicity is relatively easy to state and prove, so we will do it right away. The conditions required are what make it a little more difficult.

Theorem 3.10.5 *Assume that*

- a. $C_t(s, a)$ is supermodular on $\mathcal{S} \times \mathcal{A}$.
- b. $\sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a) v_{t+1}(s')$ is supermodular on $\mathcal{S} \times \mathcal{A}$.

Then there exists a decision rule $a(s)$ that is nondecreasing on \mathcal{S} .

Proof Let

$$w(s, a) = C_t(s, a) + \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a) v_{t+1}(s'). \quad (3.92)$$

The two terms on the right-hand side of (3.92) are assumed to be supermodular, and we know that the sum of two supermodular functions is supermodular, which tells us that $w(s, a)$ is supermodular. Let

$$a^*(s) = \arg \max_{a \in \mathcal{A}} w(s, a)$$

From lemma 3.10.2, we obtain the result that the decision $a^*(s)$ increases monotonically over \mathcal{S} , which proves our result. \square

The proof that the one-period reward function $C_t(s, a)$ is supermodular must be based on the properties of the function for a specific problem. Of greater concern is establishing the conditions required to prove condition b of the theorem because it involves the property of the value function, which is not part of the basic data of the problem.

In practice, it is sometimes possible to establish condition b directly based on the nature of the problem. These conditions usually require conditions on the monotonicity of the reward function (and hence the value function) along with properties of the one-step transition matrix. For this reason we will start by showing that if the one-period reward function is nondecreasing (or nonincreasing), then the value functions are nondecreasing (or nonincreasing). We will first need the following technical lemma:

Lemma 3.10.3 *Let $p_j, p'_j, j \in \mathcal{J}$ be probability mass functions defined over \mathcal{J} that satisfy*

$$\sum_{j=j'}^{\infty} p_j \geq \sum_{j=j'}^{\infty} p'_j \quad \forall j' \in \mathcal{J}, \quad (3.93)$$

and let $v_j, j \in \mathcal{J}$ be a nondecreasing sequence of numbers. Then

$$\sum_{j=0}^{\infty} p_j v_j \geq \sum_{j=0}^{\infty} p'_j v_j \quad (3.94)$$

We would say that the distribution represented by $\{p_j\}_{j \in \mathcal{J}}$ *stochastically dominates* the distribution $\{p'_j\}_{j \in \mathcal{J}}$. If we think of p_j as representing the probability a random variable $V = v_j$, then equation (3.94) is saying that $E^p V \geq E^{p'} V$. Although this is well known, a more algebraic proof is as follows:

Proof Let $v_{-1} = 0$, and write

$$\sum_{j=0}^{\infty} p_j v_j = \sum_{j=0}^{\infty} p_j \sum_{i=0}^j (v_i - v_{i-1}) \quad (3.95)$$

$$= \sum_{j=0}^{\infty} (v_j - v_{j-1}) \sum_{i=j}^{\infty} p_i \quad (3.96)$$

$$= \sum_{j=1}^{\infty} (v_j - v_{j-1}) \sum_{i=j}^{\infty} p_i + v_0 \sum_{i=0}^{\infty} p_i \quad (3.97)$$

$$\geq \sum_{j=1}^{\infty} (v_j - v_{j-1}) \sum_{i=j}^{\infty} p'_j + v_0 \sum_{i=0}^{\infty} p'_j \quad (3.98)$$

$$= \sum_{j=0}^{\infty} p'_j v_j. \quad (3.99)$$

In equation (3.95) we replace v_j with an alternating sequence that sums to v_j . Equation (3.96) involves one of those painful change of variable tricks with summations. Equation (3.97) is simply getting rid of the term that involves v_{-1} . In equation (3.98) we replace the cumulative distributions for p_j with the distributions for p'_j , which gives us the inequality. Finally, we simply reverse the logic to get back to the expectation in (3.99). \square

We stated that lemma 3.10.3 is true when the sequences $\{p_j\}$ and $\{p'_j\}$ are probability mass functions because it provides an elegant interpretation as expectations. For example, we may use $v_j = j$, in which case equation (3.94) gives us the familiar result that when one probability distribution stochastically dominates another, it has a larger mean. If we use an increasing sequence v_j instead of j , then this can be viewed as nothing more than the same result on a transformed axis.

In our presentation, however, we need a more general statement of the lemma, which follows:

Lemma 3.10.4 *Lemma 3.10.3 holds for any real valued, nonnegative (bounded) sequences $\{p_j\}$ and $\{p'_j\}$.*

The proof involves little more than realizing that the proof of lemma 3.10.3 never required that the sequences $\{p_j\}$ and $\{p'_j\}$ be probability mass functions.

Proposition 3.10.4 *Suppose that*

- a. $C_t(s, a)$ is nondecreasing (nonincreasing) in s for all $a \in \mathcal{A}$ and $t \in \mathcal{T}$.
- b. $C_T(s)$ is nondecreasing (nonincreasing) in s .

c. $q_t(\bar{s}|s, a) = \sum_{s' \geq \bar{s}} \mathbb{P}(s'|s, a)$, the reverse cumulative distribution function for the transition matrix, is nondecreasing in s for all $s \in \mathcal{S}$, $a \in \mathcal{A}$ and $t \in \mathcal{T}$.

Then $v_t(s)$ is nondecreasing (nonincreasing) in s for $t \in \mathcal{T}$.

Proof As always, we use a proof by induction. We will prove the result for the nondecreasing case. Since $v_T(s) = C_t(s)$, we obtain the result by assumption for $t = T$. Now assume the result is true for $v_{t'}(s)$ for $t' = t + 1, t + 2, \dots, T$. Let $a_t^*(s)$ be the decision that solves

$$\begin{aligned} v_t(s) &= \max_{a \in \mathcal{A}} C_t(s, a) + \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a) v_{t+1}(s') \\ &= C_t(s, a_t^*(s)) + \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a_t^*(s)) v_{t+1}(s'). \end{aligned} \quad (3.100)$$

Let $\hat{s} \geq s$. Condition c of the proposition implies that

$$\sum_{s' \geq s} \mathbb{P}(s'|s, a) \leq \sum_{s' \geq \hat{s}} \mathbb{P}(s'|\hat{s}, a). \quad (3.101)$$

Lemma 3.10.4 tells us that when (3.101) holds, and if $v_{t+1}(s')$ is nondecreasing (the induction hypothesis), then

$$\sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a) v_{t+1}(s') \leq \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|\hat{s}, a) v_{t+1}(s'). \quad (3.102)$$

Combining equation (3.102) with condition a of proposition 3.10.4 into equation (3.100) gives us

$$\begin{aligned} v_t(s) &\leq C_t(\hat{s}, a^*(s)) + \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|\hat{s}, a^*(s)) v_{t+1}(s') \\ &\leq \max_{a \in \mathcal{A}} C_t(\hat{s}, a) + \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|\hat{s}, a) v_{t+1}(s') \\ &= v_t(\hat{s}), \end{aligned}$$

which proves the proposition. \square

With this result we can establish condition b of theorem 3.10.5:

Proposition 3.10.5 *If*

- a. $q_t(\bar{s}|s, a) = \sum_{s' \geq \bar{s}} \mathbb{P}(s'|s, a)$ is supermodular on $\mathcal{S} \times \mathcal{A}$, and
- b. $v(s)$ is nondecreasing in s ,

then $\sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a) v(s')$ is supermodular on $\mathcal{S} \times \mathcal{A}$.

Proof Supermodularity of the reverse cumulative distribution means that

$$\sum_{s' \geq \bar{s}} \mathbb{P}(s'|s^+, a^+) + \sum_{s' \geq \bar{s}} \mathbb{P}(s'|s^-, a^-) \geq \sum_{s' \geq \bar{s}} \mathbb{P}(s'|s^+, a^-) + \sum_{s' \geq \bar{s}} \mathbb{P}(s'|s^-, a^+)$$

We can apply lemma 3.10.4 using $p_{\bar{s}} = \sum_{s' \geq \bar{s}} \mathbb{P}(s'|s^+, a^+) + \sum_{s' \geq \bar{s}} \mathbb{P}(s'|s^-, a^-)$ and $p'_{\bar{s}} = \sum_{s' \geq \bar{s}} \mathbb{P}(s'|s^+, a^-) + \sum_{s' \geq \bar{s}} \mathbb{P}(s'|s^-, a^+)$, which gives

$$\sum_{s' \in \mathcal{S}} (\mathbb{P}(s'|s^+, a^+) + \mathbb{P}(s'|s^-, a^-)) v(s') \geq \sum_{s' \in \mathcal{S}} (\mathbb{P}(s'|s^+, a^-) + \mathbb{P}(s'|s^-, a^+)) v(s'),$$

which implies that $\sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a)v(s')$ is supermodular. \square

Remark Supermodularity of the reverse cumulative distribution $\sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a)$ may seem like a bizarre condition at first, but a little thought suggests that it is often satisfied in practice. As stated, the condition means that

$$\sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s^+, a^+) - \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s^+, a^-) \geq \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s^-, a^+) - \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s^-, a^-).$$

Assume that the state s is the water level in a dam, and the decision a controls the release of water from the dam. Because of random rainfalls, the amount of water behind the dam in the next time period, given by s' , is random. The reverse cumulative distribution gives us the probability that the amount of water is greater than s^+ (or s^-). Our supermodularity condition can now be stated: if the amount of water behind the dam is higher one month (s^+), then the effect of the decision of how much water to release (a) has a greater impact than when the amount of water is initially at a lower level (s^-). This condition is often satisfied because a control frequently has more of an impact when a state is at a higher level than a lower level.

For another example of supermodularity of the reverse cumulative distribution, assume that the state represents a person's total wealth, and the control is the level of taxation. The effect of higher or lower taxes is going to have a bigger impact on wealthier people than on those who are not as fortunate (but not always: think about other forms of taxation that affect less affluent people more than the wealthy, and use this example to create an instance of a problem where a monotone policy may not apply).

We now have the result that if the reward function $C_t(s, a)$ is nondecreasing in s for all $a \in \mathcal{A}$ and the reverse cumulative distribution $\sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a)$ is supermodular, then $\sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a)v(s')$ is supermodular on $\mathcal{S} \times \mathcal{A}$. Combine this with the supermodularity of the one-period reward function, and we obtain the optimality of a nondecreasing decision function.

3.11 BIBLIOGRAPHIC NOTES

This chapter presents the classic view of Markov decision processes, for which the literature is extensive. Beginning with the seminal text of Bellman (Bellman, 1957), there have been numerous, significant textbooks on the subject, including Howard (1960), Nemhauser (1966), White (1969), Derman (1970), Bellman (1971), Dreyfus and Law (1977), Dynkin and Yushkevich (1979), Denardo (1982), Ross (1983), and Heyman and Sobel (1984). As of this writing, the current high watermark for textbooks in this area is the landmark volume by Puterman (2005). Most of this chapter is based on Puterman (2005), modified to our notational style.

Section 3.8 The linear programming method was first proposed in Manne (1960) (see subsequent discussions in Derman (1962) and Puterman (2005)). The so-called linear programming method was ignored for many years because of the large size of the linear programs that were produced, but the method has seen a resurgence of interest using approximation techniques. Recent research into algorithms for solving problems using this method are discussed in Section 10.8.

Section 3.10.6 In addition to Puterman (2005), see also Topkins (1978).

PROBLEMS

- 3.1** A classical inventory problem works as follows: Assume that our state variable R_t is the amount of product on hand at the end of time period t and that D_t is a random variable giving the demand during time interval $(t-1, t)$ with distribution $p_d = \mathbb{P}(D_t = d)$. The demand in time interval t must be satisfied with the product on hand at the beginning of the period. We can then order a quantity a_t at the end of period t that can be used to replenish the inventory in period $t+1$.
- Give the transition function that relates R_{t+1} to R_t if the order quantity is a_t (where a_t is fixed for all R_t).
 - Give an algebraic version of the one-step transition matrix $P^\pi = \{p_{ij}^\pi\}$ where $p_{ij}^\pi = \mathbb{P}(R_{t+1} = j | R_t = i, A^\pi = a_t)$.
- 3.2** Repeat the previous exercise, but now assume that we have adopted a policy π that says we should order a quantity $a_t = 0$ if $R_t \geq s$ and $a_t = Q - R_t$ if $R_t < q$ (we assume that $R_t \leq Q$). Your expression for the transition matrix will now depend on our policy π (which describes both the structure of the policy and the control parameter s).
- 3.3** We are going to use a very simple Markov decision process to illustrate how the initial estimate of the value function can affect convergence behavior. In

fact we are going to use a Markov reward process to illustrate the behavior because our process does not have any decisions. Suppose that we have a two-stage Markov chain with one-step transition matrix

$$P = \begin{bmatrix} 0.7 & 0.3 \\ 0.05 & 0.95 \end{bmatrix}.$$

The contribution from each transition from state $i \in \{1, 2\}$ to state $j \in \{1, 2\}$ is given by the matrix

$$\begin{bmatrix} 10 & 30 \\ 20 & 5 \end{bmatrix}.$$

That is, a transition from state 1 to state 2 returns a contribution of 30. Apply the value iteration algorithm for an infinite horizon problem (note that you are not choosing a decision so there is no maximization step). The calculation of the value of being in each state will depend on your previous estimate of the value of being in each state. The calculations can be easily implemented in a spreadsheet. Assume that your discount factor is 0.8.

- (a) Plot the value of being in state 1 as a function of the number of iterations if your initial estimate of the value of being in each state is 0. Show the graph for 50 iterations of the algorithm.
- (b) Repeat this calculation using initial estimates of 100.
- (c) Repeat the calculation using an initial estimate of the value of being in state 1 of 100, and use 0 for the value of being in state 2. Contrast the behavior with the first two starting points.

- 3.4** Show that $\mathbb{P}(S_{t+\tau}|S_t)$, given that we are following a policy π (for stationary problems), is given by (3.14). [Hint: First show it for $\tau = 1, 2$, and then use inductive reasoning to show that it is true for general τ .]
- 3.5** Apply policy iteration to the problem given in exercise 3.3. Plot the average value function (i.e., average the value of being in each state) after each iteration alongside the average value function found using value iteration after each iteration (for value iteration, initialize the value function to zero). Compare the computation time for one iteration of value iteration and one iteration of policy iteration.
- 3.6** Now apply the hybrid value-policy iteration algorithm to the problem given in exercise 3.3. Show the average value function after each major iteration (update of n) with $M = 1, 2, 3, 5, 10$. Compare the convergence rate to policy iteration and value iteration.
- 3.7** An oil company will order tankers to fill a group of large storage tanks. One full tanker is required to fill an entire storage tank. Orders are placed at the beginning of each four-week accounting period but do not arrive until the

end of the accounting period. During this period the company may be able to sell 0, 1, or 2 tanks of oil to one of the regional chemical companies (orders are conveniently made in units of storage tanks). The probability of a demand of 0, 1 or 2 is 0.40, 0.40, and 0.20, respectively.

A tank of oil costs \$1.6 million (M) to purchase and sells for \$2M. It costs \$0.020M to store a tank of oil during each period (oil ordered in period t , which cannot be sold until period $t + 1$, is not charged to any holding cost in period t). Storage is only charged on oil that is in the tank at the beginning of the period and remains unsold during the period. It is possible to order more oil than can be stored. For example, the company may have two full storage tanks, order three more, and then only sell one. This means that at the end of the period, they will have four tanks of oil. Whenever they have more than two tanks of oil, the company must sell the oil directly from the ship for a price of \$0.70M. There is no penalty for unsatisfied demand.

An order placed in time period t must be paid for in time period t even though the order does not arrive until $t + 1$. The company uses an interest rate of 20 percent per accounting period (i.e., a discount factor of 0.80).

- (a) Give an expression for the one-period reward function $r(s, d)$ for being in state s and making decision d . Compute the reward function for all possible states (0, 1, 2) and all possible decisions (0, 1, 2).
- (b) Find the one-step probability transition matrix when your action is to order one or two tanks of oil. The transition matrix when you order zero is given by

From-to	0	1	2
0	1	0	0
1	0.6	0.4	0
2	0.2	0.4	0.4

- (c) Write out the general form of the optimality equations and solve this problem in steady state.
 - (d) Solve the optimality equations using the value iteration algorithm, starting with $V(s) = 0$ for $s = 0, 1$, and 2. You may use a programming environment, but the problem can be solved in a spreadsheet. Run the algorithm for 20 iterations. Plot $V^n(s)$ for $s = 0, 1, 2$, and give the optimal action for each state at each iteration.
 - (e) Give a bound on the value function after each iteration.
- 3.8** Every day a salesman visits N customers in order to sell the R identical items he has in his van. Each customer is visited exactly once, and each customer buys zero or one item. Upon arrival at a customer location, the salesman quotes one of the prices $0 < p_1 \leq p_2 \leq \dots \leq p_m$. Given that the quoted price is p_i , a customer buys an item with probability r_i . Naturally r_i is decreasing in i . The salesman is interested in maximizing the total expected revenue for the day. Show that if $r_i p_i$ is increasing in i , then it is always optimal to quote the highest price p_m .

- 3.9** You need to decide when to replace your car. If you own a car of age y years, then the cost of maintaining the car that year will be $c(y)$. Purchasing a new car (in constant dollars) costs P dollars. If the car breaks down, which it will do with probability $b(y)$ (the breakdown probability), it will cost you an additional K dollars to repair it, after which you immediately sell the car and purchase a new one. At the same time you express your enjoyment with owning a new car as a negative cost $-r(y)$, where $r(y)$ is a declining function with age. At the beginning of each year, you may choose to purchase a new car ($z = 1$) or to hold onto your old one ($z = 0$). You anticipate that you will actively drive a car for another T years.
- (a) Identify all the elements of a Markov decision process for this problem.
 - (b) Write out the objective function which will allow you to find an optimal decision rule.
 - (c) Write out the one-step transition matrix.
 - (d) Write out the optimality equations that will allow you to solve the problem.

- 3.10** You are trying to find the best parking space to use that minimizes the time needed to get to your restaurant. There are 50 parking spaces, and you see spaces 1, 2, ..., 50 in order. As you approach each parking space, you see whether it is full or empty. We assume, somewhat heroically, that the probability that each space is occupied follows an independent Bernoulli process, which is to say that each space will be occupied with probability p but will be free with probability $1 - p$, and that each outcome is independent of the other.

It takes 2 seconds to drive past each parking space and it takes 8 seconds to walk past. That is, if we park in space n , it will require $8(50 - n)$ seconds to walk to the restaurant. Furthermore it would have taken you $2n$ seconds to get to this space. If you get to the last space without finding an opening, then you will have to drive into a special lot down the block, adding 30 seconds to your trip.

We want to find an optimal strategy for accepting or rejecting a parking space.

- (a) Give the sets of state and action spaces and the set of decision epochs.
- (b) Give the expected reward function for each time period and the expected terminal reward function.
- (c) Give a formal statement of the objective function.
- (d) Give the optimality equations for solving this problem.
- (e) You have just looked at space 45, which was empty. There are five more spaces remaining (46 through 50). What should you do? Using $p = 0.6$, find the optimal policy by solving your optimality equations for parking spaces 46 through 50.
- (f) Give the optimal value of the objective function in part (e) corresponding to your optimal solution.

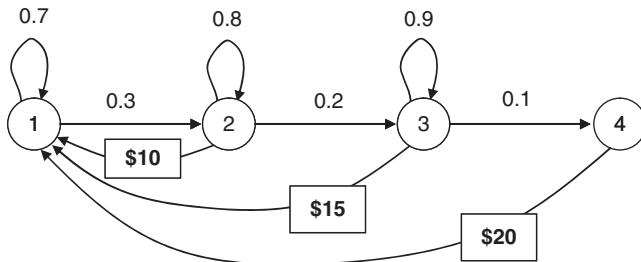


Figure 3.8

- 3.11** We have a four-state process (shown in Figure 3.8). In state 1, we will remain in the state with probability 0.7 and will make a transition to state 2 with probability 0.3. In states 2 and 3, we may choose between two policies: remain in the state waiting for an upward transition or make the decision to return to state 1 and receive the indicated reward. In state 4, we return to state 1 immediately and receive \$20. We wish to find an optimal long run policy using a discount factor $\gamma = .8$. Set up and solve the optimality equations for this problem.
- 3.12** Say that you have been applying value iteration to a four-state Markov decision process, and that you have obtained the values over iterations 8 through 12 shown in the table below (assume a discount factor of 0.90). Say that you stop after iteration 12. Give the tightest possible (valid) bounds on the optimal value of being in each state.

State	Iteration				
	8	9	10	11	12
1	7.42	8.85	9.84	10.54	11.03
2	4.56	6.32	7.55	8.41	9.01
3	11.83	13.46	14.59	15.39	15.95
4	8.13	9.73	10.85	11.63	12.18

- 3.13** In the proof of theorem 3.10.3 we showed that if $v \geq \mathcal{M}v$, then $v \geq v^*$. Go through the steps of proving the converse, that if $v \leq \mathcal{M}v$, then $v \leq v^*$.
- 3.14** Theorem 3.10.3 states that if $v \leq \mathcal{M}v$, then $v \leq v^*$. Show that if $v^n \leq v^{n+1} = \mathcal{M}v^n$, then $v^{m+1} \geq v^m$ for all $m \geq n$.
- 3.15** Consider a finite-horizon MDP with the following properties:
- $\mathcal{S} \in \mathfrak{N}^n$, the action space \mathcal{A} is a compact subset of \mathfrak{N}^n , $\mathcal{A}(s) = \mathcal{A}$ for all $s \in \mathcal{S}$.

- $C_t(S_t, a_t) = c_t S_t + g_t(a_t)$, where $g_t(\cdot)$ is a known scalar function, and $C_T(S_T) = c_T S_T$.
- If action a_t is chosen when the state is S_t at time t , the next state is

$$S_{t+1} = A_t S_t + f_t(a_t) + \omega_{t+1},$$

where $f_t(\cdot)$ is scalar function, and A_t and ω_t are, respectively, $n \times n$ and $n \times 1$ -dimensional random variables whose distributions are independent of the history of the process prior to t .

- (a) Show that the optimal value function is linear in the state variable.
- (b) Show that there exists an optimal policy $\pi^* = (a_1^*, \dots, a_{T-1}^*)$ composed of constant decision functions. That is, $A_t^{\pi^*}(s) = A_t^*$ for all $s \in \mathcal{S}$ for some constant A_t^* .

- 3.16** You have invested R_0 dollars in a stock market that evolves according to the equation

$$R_t = \gamma R_{t-1} + \varepsilon_t,$$

where ε_t is a discrete, positive random variable that is independent and identically distributed and where $0 < \gamma < 1$. If you sell the stock at the end of period t , it will earn a riskless return r until time T , which means it will evolve according to

$$R_t = (1+r)R_{t-1}.$$

You have to sell the stock, all on the same day, some time before T .

- (a) Write a dynamic programming recursion to solve the problem.
- (b) Show that there exists a point in time τ such that it is optimal to sell for $t \geq \tau$, and optimal to hold for $t < \tau$.
- (c) How does your answer to (b) change if you are allowed to sell only a portion of the assets in a given period? That is, if you have R_t dollars in your account, you are allowed to sell $a_t \leq R_t$ at time t .

- 3.17** An airline has to decide when to bring an aircraft in for a major engine overhaul. Let s_t represent the state of the engine in terms of engine wear, and let d_t be a nonnegative amount by which the engine deteriorates during period t . At the beginning of period t the airline may decide to continue operating the aircraft ($z_t = 0$) or to repair the aircraft ($z_t = 1$) at a cost of c^R , which has the effect of returning the aircraft to $s_{t+1} = 0$. If the airline does not repair the aircraft, the cost of operation is $c^o(s_t)$, which is a nondecreasing, convex function in s_t .

- (a) Define what is meant by a control limit policy in dynamic programming, and show that this is an instance of a monotone policy.

- (b) Formulate the one-period reward function $C_t(s_t, z_t)$, and show that it is submodular.
- (c) Show that the decision rule is monotone in s_t . (Outline the steps in the proof, and then fill in the details.)
- (d) Assume that a control limit policy exists for this problem, and let γ be the control limit. Now write $C_t(s_t, z_t)$ as a function of one variable: the state s . Using the control limit structure of Section 8.2.2, we can write the decision z_t as the decision rule $z^\pi(s_t)$. Illustrate the shape of $C_t(s_t, z^\pi(s))$ by plotting it over the range $0 \leq s \leq 3\gamma$ (in theory, we may be given an aircraft with $s > \gamma$ initially).
- 3.18** A dispatcher controls a finite capacity shuttle that works as follows: In each time period a random number A_t arrives. After the arrivals occur, the dispatcher must decide whether to call the shuttle to remove up to M customers. The cost of dispatching the shuttle is c , which is independent of the number of customers on the shuttle. Each time period that a customer waits costs h . If we let $z = 1$ if the shuttle departs and 0 otherwise, then our one-period reward function is given by

$$c_t(s, z) = cz + h[s - Mz]^+,$$

where M is the capacity of the shuttle. Show that $c_t(s, a)$ is submodular where we would like to minimize r . Note that we are representing the state of the system after the customers arrive.

- 3.19** Assume that a control limit policy exists for our shuttle problem in exercise 8.8 that allows us to write the optimal dispatch rule as a function of s , as in $z^\pi(s)$. We may write $r(s, z)$ as a function of one variable, the state s .
- (a) Illustrate the shape of $r(s, z(s))$ by plotting it over the range $0 < s < 3M$ (since we are allowing there to be more customers than can fill one vehicle, assume that we are allowed to send $z = 0, 1, 2, \dots$ vehicles in a single time period).
- (b) Let $c = 10$, $h = 2$, and $M = 5$, and assume that $A_t = 1$ with probability 0.6 and is 0 with probability 0.4. Set up and solve a system of linear equations for the optimal value function for this problem in steady state.

C H A P T E R 4

Introduction to Approximate Dynamic Programming

In Chapter 3 we saw that we could solve stochastic optimization problems of the form

$$\max_{\pi} \mathbb{E} \left\{ \sum_{t=0}^T \gamma^t C_t(S_t, A_t^\pi(S_t)) \right\} \quad (4.1)$$

by recursively computing the optimality equations

$$V_t(S_t) = \max_{a_t} (C_t(S_t, a_t) + \gamma \mathbb{E}\{V_{t+1}(S_{t+1})|S_t\}) \quad (4.2)$$

backward through time. As before, we use the shorthand notation $S_{t+1} = S^M(S_t, a_t, W_{t+1})$ to express the dependency of state S_{t+1} on the previous state S_t , the action a_t and the new information W_{t+1} . Equation (4.1) can be computationally intractable even for very small problems. The optimality equations (4.2) give us a mechanism for solving these stochastic optimization problems in a simple and elegant way. Unfortunately, in the vast majority of real applications we cannot solve Bellman's equation exactly.

Approximate dynamic programming offers a powerful set of strategies for problems that are hard because they are large, but this is not the only application. Even small problems may be hard to solve if we lack a formal model of the information process, or if we do not know the transition function. For example, we may have observations of changes in prices in an asset but we do not have a mathematical model that describes these changes. We may observe an opponent playing poker, but we are unable to describe his logic for making decisions. In this case we are not able to compute the expectation. Alternatively, consider the problem of modeling how the economy of a small country responds to loans from the International

Approximate Dynamic Programming: Solving the Curses of Dimensionality, Second Edition.

Warren B. Powell.

© 2011 John Wiley & Sons, Inc. Published 2011 by John Wiley & Sons, Inc.

Monetary Fund. If we do not know how the economy responds to the size of a loan, then this means that we do not know the transition function.

There are also many problems which are intractable simply because of size. Imagine a multi-skill call center that handles questions about personal computers. People who call in identify the nature of their question through the options they select on the phone's menu. Calls are then routed to people with expertise in those areas. Assigning phone calls to technicians is a dynamic assignment problem involving multiattribute resources, creating a dynamic program with a state space that is effectively infinite.

We begin our presentation by revisiting the “curses of dimensionality.” The remainder of the chapter provides an overview of the basic principles and vocabulary of approximate dynamic programming. As with the previous chapters, we retain the basic notation where our system occupies a discrete state $S_t \in \mathcal{S} = \{1, 2, \dots, s, \dots\}$. In Chapter 5 we introduce a much richer notational framework that makes it easier to develop approximation strategies.

4.1 THE THREE CURSES OF DIMENSIONALITY (REVISITED)

The concept of the backward recursion of dynamic programming is so powerful that we have to remind ourselves again why its usefulness can be so limited for many problems. Consider, again, our simple asset acquisition problem, but this time assume that we have multiple asset classes. For example, we may wish to purchase stocks each month for our retirement account using the money that is invested in it each month. Our online brokerage charges \$50 for each purchase order we place, so we have an incentive to purchase stocks in reasonably sized lots. Let p_{tk} be the purchase price of asset type $k \in \mathcal{K}$ in period t . Since our decision is a vector, we model our decisions using $x_t = (x_{tk})_{k \in \mathcal{K}}$, where x_{tk} is the number of shares of asset k we purchase in period t . The total purchase cost of our order is

$$C^P(x_t) = \sum_{k \in \mathcal{K}} - (50I_{\{x_{tk} > 0\}} + p_{tk}x_{tk}),$$

where $I_{\{x_{tk} > 0\}} = 1$ if $x_{tk} > 0$ is true, and 0 otherwise. Each month we have \$2000 to invest in our retirement plan. We have to make sure that $\sum_{k \in \mathcal{K}} p_{tk}x_{tk} \leq \2000 and, of course, $x_{tk} \geq 0$. While it is important to diversify the portfolio, transaction costs rise as we purchase a wider range of stocks, which reduces how much we can buy.

Let R_{tk} be the number of shares of stock k that we have on hand at the end of period t . The value of the portfolio reflects the prices p_{tk} which evolve over time according to

$$p_{t,k} = p_{t-1,k} + \hat{p}_{t,k},$$

where $\hat{p}_{t,k}$ is the exogenous change in the price of stock k between $t-1$ and t . In addition each stock returns a random dividend \hat{d}_{tk} (between $t-1$ and t) given in dollars per share that are reinvested. Thus the information we receive in each time period is

$$W_t = (\hat{p}_t, \hat{d}_t).$$

If we were to reinvest our dividends, then our transition function would be

$$R_{t+1,k} = R_{tk} + \frac{\hat{d}_{t+1,k}}{p_{t+1,k}} R_{tk} + x_{tk}.$$

The state of our system is given by

$$S_t = (R_t, p_t).$$

Let the total value of the portfolio be

$$Y_t = \sum_{k \in \mathcal{K}} p_{tk} R_{tk},$$

which we evaluate using a concave utility function $U(Y_t)$. Our goal is to maximize total expected discounted utility over a specified horizon.

We now have a problem where our state S_t variable has $2|\mathcal{K}|$ dimensions (the resource allocation R_t and the prices p_t), the decision variable x_t has $|\mathcal{K}|$ dimensions, and our information process W_t has $2|\mathcal{K}|$ dimensions (the changes in prices and the dividends). To illustrate how bad this gets, assume that we are restricting our attention to 50 different stocks and that we always purchase shares in blocks of 100, but that we can own as many as 20,000 shares (or 200 blocks) of any one stock at a given time. This means that the number of possible portfolios R_t is 200^{50} . Now assume that we can discretize each price into 100 different prices. This means that we have up to 100^{50} different price vectors p_t . Thus we would have to loop over equation (4.2) $200^{50} \times 100^{50}$ times to compute $V_t(S_t)$ for all possible values of S_t . This is the classical “curse of dimensionality” that is widely cited as the Achilles heel of dynamic programming.

As we look at our problem, we see the situation is much worse. For each of the $200^{50} \times 100^{50}$ states, we have to compute the expectation in (4.2). Since our random variables might not be independent (the prices generally will not be) we could conceive of finding a joint probability distribution and performing 2×50 nested summations to complete the expectation (we have 50 prices and 50 dividends). It seems that this can be avoided if we work with the one-step transition matrix and use the form of the optimality equations expressed in (3.3). This form of the recursion, however, only hides the expectation (the one-step transition matrix is an expectation).

We are not finished. This expectation has to be computed for each action x . Assume we are willing to purchase up to 10 blocks of 100 shares of any one of the 50 stocks. Since we can also purchase 0 shares of a stock, we have upwards of 11^{50} different combinations of x_t that we might have to consider (the actual number is smaller because we have a budget constraint of \$2000 to spend).

So we see that we have three curses of dimensionality if we wish to work in discrete quantities. In other words, vectors really cause problems. In this chapter we are going to provide an introduction to approximate dynamic programming, which has evolved to address these issues. We are not ready to handle vector-valued decisions, so we are going to stay with our action notation a and assume that we are choosing among a relatively small number of discrete actions.

4.2 THE BASIC IDEA

The foundation of approximate dynamic programming is based on an algorithmic strategy that steps *forward* through time (earning it the name “forward dynamic programming” in some communities). If we wanted to solve this problem using classical dynamic programming, we would have to find the value function $V_t(S_t)$ using

$$\begin{aligned} V_t(S_t) &= \max_{a_t \in \mathcal{A}_t} (C(S_t, a_t) + \gamma \mathbb{E}\{V_{t+1}(S_{t+1})|S_t\}) \\ &= \max_{a_t \in \mathcal{A}_t} \left(C(S_t, a_t) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|S_t, a_t) V_{t+1}(s') \right) \end{aligned}$$

for each value of S_t . We have written our maximization problem as one of choosing the best $a_t \in \mathcal{A}_t$. Note that \mathcal{A}_t depends on our state variable S_t ; we express this dependence by indexing the action set by time t .

This problem cannot be solved using the techniques we presented in Chapter 3, which require that we loop over all possible states (in fact we usually have three nested loops, two of which require that we enumerate all states while the third enumerates all actions). With approximate dynamic programming we step forward in time. In order to simulate the process forward in time, we need to solve two problems. The first is that we need a way to randomly generate a sample of what *might* happen (in terms of the various sources of random information). The second is that we need a way to make decisions. We start with the problem of making decisions first, and then turn to the problem of simulating random information.

4.2.1 Making Decisions (Approximately)

When we used exact dynamic programming, we stepped backward in time, exactly computing the value function that we then used to produce optimal decisions. When we step forward in time, we have not computed the value function, so we have to turn to an approximation in order to make decisions.

Let $\bar{V}_t(S_t)$ be an approximation of the value function. This is easiest to understand if we assume that we have an estimate $\bar{V}_t(S_t)$ for each state S_t , but since it is an approximation, we may use any functional form we wish. For our portfolio problem above, we might create an approximation $\bar{V}_t(R_t)$ that depends only on the number of shares we own, rather than the prices. In fact we might even use a separable approximation of the form

$$V_t(S_t) \approx \sum_{k \in \mathcal{K}} \bar{V}_{tk}(R_{tk}),$$

where $\bar{V}_{tk}(R_{tk})$ is a nonlinear function giving the value of holding R_{tk} shares of stock k . Obviously, assuming such a structure (where we are both ignoring the price vector p_t as well as assuming separability) will introduce errors. Welcome to the field of approximate dynamic programming! The challenge, of course, is finding approximations that are good enough for the purpose at hand.

Approximate dynamic programming proceeds by estimating the approximation $\bar{V}_t(S_t)$ iteratively. We assume we are given an initial approximation \bar{V}_t^0 for all t (we often use $\bar{V}_t^0 = 0$ when we have no other choice). Let \bar{V}_t^{n-1} be the value function approximation after $n-1$ iterations, and consider what happens during the n th iteration. We are going to start at time $t = 0$ in state S_0 . We determine a_0 by solving

$$\begin{aligned} a_0 &= \arg \max_{a \in A_0} (C(S_0, a) + \gamma \mathbb{E}\{\bar{V}_1(S_1) | S_0\}) \\ &= \arg \max_{a \in A_0} (C(S_0, a) + \gamma \sum_{s' \in S} \mathbb{P}_0(s' | S_0, a) \bar{V}_1(s')), \end{aligned} \quad (4.3)$$

where a_0 is the value of a that maximizes the right-hand side of (4.3), and $\mathbb{P}(s'|S_0, a)$ is the one-step transition matrix (which we temporarily assume is given). Assuming this can be accomplished (or at least approximated), we can use (4.3) to determine a_0 .

For the moment, we are going to assume that we now have a way of making decisions.

4.2.2 Stepping Forward through Time

We are now going to step forward in time from S_0 to S_1 . This starts by implementing the decision a_0 , which we are going to do using our approximate value function. Given our decision, we next need to know the information that arrived between $t = 0$ and $t = 1$. At $t = 0$, this information is unknown, and therefore random. Our strategy will be to simply pick a sample realization of the information (for our example, this would be \hat{d}_1 and \hat{p}_1) at random, a process that is often referred to as *Monte Carlo simulation*. Monte Carlo simulation refers to the popular practice of generating random information, using some sort of artificial process to pick a random observation from a population. For example, imagine that we think a price will be uniformly distributed between 60 and 70. Most computer languages have a random number generator that produces a number that is uniformly distributed between 0 and 1. If we let RAND() denote this function, then the statement

$$U = \text{RAND}()$$

produces a value for U that is some number between 0 and 1 (e.g., 0.804663917). We can then create our random price using the equation

$$\begin{aligned} \hat{p}_t &= 60 + 10 * U \\ &= 68.04663917. \end{aligned}$$

While this is not the only way to obtain random samples of information, it is one of the more popular approaches.

Using our random sample of new information, we can now compute the next state we visit, given by S_1 . We do this by assuming that we have been given a

transition function, which we represent using

$$S_{t+1} = S^M(S_t, a_t, W_{t+1}),$$

where W_{t+1} is a set of random variables representing the information that arrived between t and $t+1$. We first introduced this notation in Chapter 2, which provides a number of examples.

Keeping in mind that we assume that we have been given \bar{V}_t for all t , we simply repeat the process of making a decision by solving

$$a_1 = \arg \max_{a \in \mathcal{A}_1} \left(C(S_1, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}_1(s'|S_1, a) \bar{V}_2(s') \right). \quad (4.4)$$

Once we determine a_1 , we, again, sample the new information (\hat{p}_2, \hat{d}_2) , compute S_2 , and repeat the process. Whenever we make a decision based on a value function approximation (as we do in equation (4.4)), we refer to this as a *greedy strategy*. This term, which is widely used in dynamic programming, can be somewhat misleading since the value function approximation is trying to produce decisions that balance rewards now with rewards in the future.

Fundamental to approximate dynamic programming is the idea of following a *sample path*. A sample path refers to a particular sequence of exogenous information. To illustrate, suppose that we have a problem with two assets, where the price of each asset at time t is given by (p_{t1}, p_{t2}) . Assume that these changes occur randomly over time according to

$$p_{ti} = p_{t-1,i} + \hat{p}_{ti}, \quad i = 1, 2.$$

Further assume that from history, we have constructed 10 potential realizations of the price changes \hat{p}_t , $t = 1, 2, \dots, 8$, which we show in Table 4.1. Each sample path is a particular set of outcomes of the vector \hat{p}_t for all time periods. For historical reasons we index each potential set of outcomes by ω , and let Ω be the set of all sample paths, where for our example $\Omega = \{1, 2, \dots, 10\}$. Let W_t be a generic random variable representing the information arriving at time t , where for this example we would have $W_t = (\hat{p}_{t1}, \hat{p}_{t2})$. Then we would let $W_t(\omega)$ be a particular sample realization of the prices in time period t . For example, using the data in Table 4.1, we see that $W_3(5) = (-9, 25)$. In an iterative algorithm, we might choose ω^n in iteration n , and then follow the sample path $W_1(\omega^n), W_2(\omega^n), \dots, W_t(\omega^n)$. In each iteration we have to choose a new outcome ω^n .

The notation $W_t(\omega)$ matches how we have laid out the data in Table 4.1. We go to row ω in column t to get the information we need. This layout seems to imply that we have all the random data generated in advance, and we just pick out what we want. In practice, this may not be how the code is actually written. It is often the case that we are randomly generating information as the algorithm progresses. Readers need to realize that when we write $W_t(\omega)$, we mean a random observation of the information that arrived at time t , regardless of whether we just generated

Table 4.1 Illustration of a set of sample paths

	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$	$t = 6$	$t = 7$	$t = 8$
ω	\hat{p}_1	\hat{p}_2	\hat{p}_3	\hat{p}_4	\hat{p}_5	\hat{p}_6	\hat{p}_7	\hat{p}_8
1	(-14, -26)	(7, 16)	(-10, 20)	(41, 7)	(15, -11)	(8, -29)	(-6, -50)	(13, -36)
2	(11, -33)	(-36, -50)	(-23, 0)	(3, -50)	(7, -23)	(12, -10)	(35, 46)	(-18, -10)
3	(-50, -12)	(-2, -18)	(-15, 12)	(-31, 3)	(5, -2)	(-24, 33)	(10, 2)	(38, -19)
4	(1, -13)	(41, 20)	(32, -2)	(3, -4)	(-46, 34)	(-15, 14)	(38, 16)	(48, -49)
5	(2, -34)	(-24, 34)	(-9, 25)	(-19, -40)	(1, -28)	(34, -7)	(36, -3)	(-46, -35)
6	(41, -49)	(-24, -33)	(-5, -25)	(16, 36)	(8, 47)	(-17, -4)	(-29, 45)	(-48, -30)
7	(44, 37)	(7, -19)	(49, -40)	(-13, 5)	(38, 37)	(-30, 45)	(-48, -47)	(19, 41)
8	(-19, 37)	(-50, -35)	(-28, 32)	(-13, -17)	(2, -2)	(-10, -22)	(-2, 47)	(2, -24)
9	(-13, 48)	(-48, 25)	(-37, 39)	(-2, 30)	(-28, 33)	(-35, -49)	(-44, -13)	(-6, -18)
10	(48, 5)	(37, -39)	(43, 34)	(-13, -6)	(28, -37)	(-47, -12)	(13, 28)	(26, -35)

the data, looked it up in a table, or were given the data from an exogenous process (perhaps an actual price realization from the Internet).

Our sample path notation is useful because it is quite general. For example, it does not require that the random observations be independent across time (by contrast, Bellman's equation explicitly assumes that the distribution of W_{t+1} depends only on S_t and possibly a_t). In approximate dynamic programming we do not impose any restrictions of this sort. ADP allows us to simulate arbitrarily complex processes as we step forward in time. We do not have to approximate the physics. We are only approximating how we make decisions.

4.2.3 Sampling Random Variables

Our ADP algorithm depends on having access to a sequence of sample realizations of our random variable. How this is done depends on the setting. There are three ways that we can obtain random samples:

- 1. Real world.** Random realizations may come from real physical processes. For example, we may be trying to estimate average demand using sequences of actual demands. We may also be trying to estimate prices, costs, travel times, or other system parameters from real observations.
- 2. Computer simulations.** The random realization may be a calculation from a computer simulation of a complex process. The simulation may be of a physical system such as a supply chain or an asset allocation model. Some simulation models can require extensive calculations (a single sample realization could take hours or days on a computer).
- 3. Sampling from a known distribution.** This is the easiest way to sample a random variable. We can use existing tools available in most software languages or spreadsheet packages to generate samples from standard probability distributions. These tools can be used to generate many thousands of random observations extremely quickly.

The ability of ADP algorithms to work with real data (or data coming from a complex computer simulation) means that we can solve problems without actually knowing the underlying probability distribution. We may have multiple random variables that exhibit correlations. For example, observations of interest rates or currency exchange rates can exhibit complex interdependencies that are difficult to estimate. As a result we may not be able to compute an expectation because we do not know the underlying probability distribution. That we can still solve such a problem (approximately) greatly expands the scope of problems that we can address.

When we do have a probability model describing our information process, we can use the power of computers to generate random observations from a distribution using a process that is generally referred to as Monte Carlo sampling. Although most software tools come with functions to generate observations from major distributions, it is often necessary to customize tools to handle more general distributions. When this is the case, we often find ourselves relying on sampling techniques that depend on functions for generating random variables that are uniformly distributed between 0 and 1, which we denote by U (often called RAND() in many computer languages), or for generating random variables that are normally distributed with mean 0 and variance 1 (the standard normal distribution), which we denote by Z . Using the computer to generate random variables in this way is known as *Monte Carlo simulation*. A “Monte Carlo sample” (or “Monte Carlo realization”) refers to a particular observation of a random variable. Not surprisingly, it is much easier to use a single observation of a random variable than the entire distribution.

If we need a random variable X that is uniformly distributed between a and b , then we use the internal random number generator to produce U and calculate

$$X = a + (b - a)U.$$

Similarly, if we wish to randomly generate a random variable that is normally distributed with mean μ and variance σ^2 , then we can usually depend on an internal random number generator to compute a random variable Z that has mean 0 and variance 1. We can use the result to perform the transformation

$$X = \mu + \sigma Z$$

and obtain a random variable with mean μ and variance σ^2 .

If we need a random variable X with a cumulative distribution function $F_X(x) = \mathbb{P}(X \leq x)$, then we can obtain observations of X using a simple property. Let $Y = F_X(X)$ be a random variable that we compute by taking the original random variable X , and then computing $F_X(X)$. It is possible to show that Y is a random variable with a uniform distribution between 0 and 1. This implies that $F_X^{-1}(U)$ is a random variable that has the same distribution as X , which means that we can generate observations of X using

$$X = F_X^{-1}(U).$$

For example, consider the case of an exponential density function $\lambda e^{-\lambda x}$ with cumulative distribution function $1 - e^{-\lambda x}$. Setting $U = 1 - e^{-\lambda x}$ and solving for x

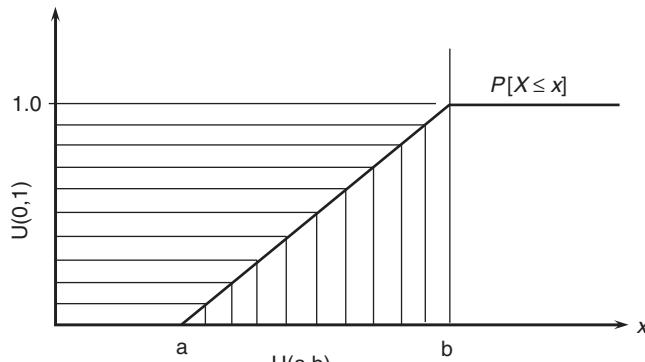
gives

$$X = -\frac{1}{\lambda} \ln(1 - U).$$

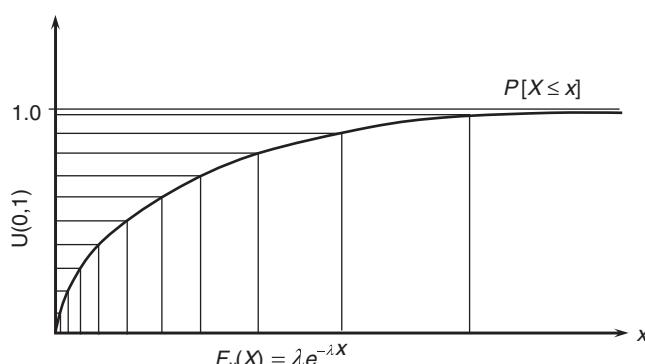
Since $1 - U$ is also uniformly distributed between 0 and 1, we can use

$$X = -\frac{1}{\lambda} \ln(U).$$

Figure 4.1 illustrates using the inverse cumulative-distribution method to generate both uniformly distributed and exponentially distributed random numbers. After generating a uniformly distributed random number in the interval $[0, 1]$ (denoted $U(0, 1)$ in the figure), we map this number from the vertical axis to the horizontal axis. If we want to find a random number that is uniformly distributed between a and b , the cumulative distribution simply stretches (or compresses) the uniform $(0, 1)$ distribution over the range (a, b) .



(a) Generating uniform random variables



(b) Generating exponentially distributed random variables

Figure 4.1 Generating uniformly and exponentially distributed random variables using the inverse cumulative distribution method.

There is an extensive literature on generating Monte Carlo random variables that goes well beyond the scope of this book. This section provides only a brief introduction.

4.2.4 An ADP Algorithm

Stepping forward through time following a single set of sample realizations would not, of course, produce anything of value. The price of using approximate value functions is that we have to do this over and over, using a fresh set of sample realizations each time. Using our vocabulary of the previous section, we would say that we follow a new sample path each time.

When we run our algorithm iteratively, we index everything by the iteration counter n . We would let ω^n represent the specific value of ω that we sampled for iteration n . At time t , we would be in state S_t^n , and make decision a_t^n using the value function approximation \bar{V}^{n-1} . The value function is indexed $n-1$ because it was computed using information from iterations $1, \dots, n-1$. After finding a_t^n , we would observe the information $W_{t+1}(\omega^n)$ to obtain S_{t+1}^n . After reaching the end of our horizon, we would increment n and start over again.

A basic approximate dynamic programming algorithm is summarized in Figure 4.2. This is very similar to backward dynamic programming (see Figure 3.1), except that we are stepping forward in time. We no longer have the dangerous “loop over all states” requirement, but we have introduced a fresh set of challenges. As this book should make clear, this basic algorithmic strategy is

Step 0. Initialization.

Step 0a. Initialize $\bar{V}_t^0(S_t)$ for all states S_t .

Step 0b. Choose an initial state S_0^1 .

Step 0c. Set $n = 1$.

Step 1. Choose a sample path ω^n .

Step 2. For $t = 0, 1, 2, \dots, T$ do:

Step 2a. Solve

$$\hat{v}_t^n = \max_{a_t \in \mathcal{A}_t^n} \left(C_t(S_t^n, a_t) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|S_t^n, a_t) \bar{V}_{t+1}^{n-1}(s') \right),$$

and let a_t^n be the value of a_t that solves the maximization problem.

Step 2b. Update $\bar{V}_t^{n-1}(S_t)$ using

$$\bar{V}_t^n(S_t) = \begin{cases} \hat{v}_t^n, & S_t = S_t^n, \\ \bar{V}_t^{n-1}(S_t), & \text{otherwise.} \end{cases}$$

Step 2c. Compute $S_{t+1}^n = S^M(S_t^n, a_t^n, W_{t+1}(\omega^n))$.

Step 3. Let $n = n+1$. If $n < N$, go to step 1.

Figure 4.2 Approximate dynamic programming algorithm using the one-step transition matrix.

exceptionally powerful but introduces a number of technical challenges that have to be overcome (indeed this is what fills up the remainder of this book).

A brief note on notation is in order here. We use $V_t(S_t)$ as the true value of being in a state at time t , while $\bar{V}_t^n(s)$ is our statistical estimate after n sample observations. When we are following sample path ω^n , we make decisions using $\bar{V}_t^{n-1}(s)$. The value \hat{v}_t^n is computed using information from sample path ω^n (e.g., the fact that we are in state S_t^n), and $\bar{V}_t^{n-1}(s)$. \hat{v}_t^n is then used to update our value function approximation to produce $\bar{V}_t^n(s)$. Our indexing tells us the information content of a variable, so if we use $\bar{V}_t^{n-1}(s)$ to make a decision, we know this variable depends on the sample information from $\omega^1, \dots, \omega^{n-1}$.

4.2.5 Discussion

Our generic approximate dynamic programming algorithm introduces several problems that we have to solve:

- Forward dynamic programming avoids the problem of looping over all possible states, but it still requires the use of a one-step transition matrix, with the equally dangerous $\sum_{s' \in \mathcal{S}} \mathbb{P}(s'|S_t^n, a_t)[\dots]$.
- We only update the value of states we visit, but we need the value of states that we *might* visit. We need a way to estimate the value of being in states that we have not visited.
- We might get caught in a circle where states we have never visited look bad (relative to states we have visited), so we never visit them. We may never discover that a state that we have never visited is actually quite good!
- The algorithmic strategy is quite general, but at the same time it does not provide any mechanism for taking advantage of problem structure.

In the remainder of this chapter, we provide an introduction to some of the major strategies that have evolved. We limit our attention in this chapter to value function approximations known as lookup tables, which means we store a value $V(s)$ for each discrete state s . Not surprisingly, this approach does not scale, and we are going to need a much richer set of approximation strategies. Also we only consider policies that depend on value function approximations, whereas there is a much wider range of strategies that are used in this community.

In the remainder of the chapter, we consider the following algorithmic strategies:

- *Q-learning*. We present this popular strategy for problems with small state and action spaces, but where we do not have a mathematical model for how the system evolves over time.
- *Real-time dynamic programming* (RTDP). This is a minor variation of the algorithm we presented in Figure 4.2, but with a twist that allows us to claim that the algorithm actually converges.
- *Approximate value iteration*. Here we relax the assumption that we can compute the one-step transition matrix.

- *Approximate value iteration using the post-decision state variable.* We first introduce the concept of the post-decision state variable, and then demonstrate how this simplifies the algorithm.

4.3 Q-LEARNING AND SARSA

One of the oldest algorithms from the reinforcement learning community is known as Q -learning, named after the variable $Q(s, a)$, which is an estimate of the value of being in a state s and taking action a . Q -learning, and its sister algorithm SARSA, are not only the simplest introductions to approximate dynamic programming and reinforcement learning, they also provide a nice mechanism for introducing the important issue of exploration.

4.3.1 Q-Learning

Let $\bar{Q}^l(s, a)$ be a statistical estimate of the true value $Q(s, a)$ after n iterations. Suppose that we are in state S^n . We choose an action a^n using

$$a^n = \arg \max_{a \in \mathcal{A}} \bar{Q}^{n-1}(S^n, a). \quad (4.5)$$

Note that in iteration n , we use the estimates $\bar{Q}^{n-1}(S^n, a)$ from the previous iteration. The important feature of Q -learning is that to find an action a^n using equation (4.5), we do not need to compute an expectation, and nor do we need an estimate of the value of being in a downstream state in the future.

Some physical systems are so complex that we cannot describe them with mathematical models, but we are able to observe behaviors directly. Such applications arise in operational settings where a model is running in production, allowing us to observe exogenous outcomes and state transitions from physical processes rather than depending on mathematical equations.

There are three key mathematical models used in dynamic programming. In engineering communities, “model” refers to the transition function (also known as the system model or plant model). Model-free dynamic programming refers to applications where we do not have an explicit transition function (the physical problem may be quite complex). In such problems we may make a decision but then have to observe the results of the decision from a physical process.

The second “model” refers to the exogenous information process, where we may not have a probability law describing the outcomes. The result is that we will not be able to compute a one-step transition matrix, but it also means that we cannot even run simulations because we do not know the likelihood of an outcome. In such settings we assume that we have an exogenous process generating outcomes (e.g., stock prices or demands).

The third use of the term model refers to the cost or contribution function. For example, we may have a process where decisions are being made by a human maximizing an unknown utility function. We may have a system where we observe the behavior of a human and infer what action we should be taking given a state.

In the reinforcement learning community, the term model often refers to the one-step transition matrix. Computing the one-step transition matrix not only means that you know the transition function and the probability law behind the exogenous information, it also means that your state space is discrete and small enough that it makes sense to compute the transition matrix.

Once we choose an action a^n , we assume that we are allowed to observe a contribution $\hat{C}(S^n, a^n)$ (which may be random), and the next state S^{n+1} . We then compute an updated value of being in state S^n and taking action a^n using

$$\hat{q}^n = \hat{C}(S^n, a^n) + \gamma \max_{a' \in \mathcal{A}} \bar{Q}^{n-1}(S^{n+1}, a'). \quad (4.6)$$

We use this information to update our Q -factors as follows:

$$\bar{Q}_t^n(S^n, a^n) = (1 - \alpha_{n-1}) \bar{Q}^{n-1}(S^n, a^n) + \alpha_{n-1} \hat{q}^n,$$

where α_{n-1} is a stepsize between 0 and 1. Chapter 11 addresses the issue of stepsize selection in considerable depth. Common practice is to use a constant stepsize (e.g., $\alpha_n = 0.05$) or a declining rule such as $a/(a + n - 1)$. However, special care is required in the choice of stepsize rule for this algorithm, and we urge the reader to consider the issues raised in Chapter 11.

Given a set of Q -factors, we can quickly compute an estimate of the value of being in a state s using

$$\bar{V}^n(s) = \max_{a \in \mathcal{A}} \bar{Q}^n(s, a).$$

We can use this simple substitution to replace (4.6) with

$$\hat{q}^n = \hat{C}(S^n, a^n) + \gamma \bar{V}^{n-1}(S^{n+1}). \quad (4.7)$$

Now contrast this calculation with how we computed \hat{v}_t^n in our ADP algorithm in Figure 4.2:

$$\hat{v}_t^n = \max_{a_t \in \mathcal{A}_t^n} \left(C_t(S_t^n, a_t) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|S_t^n, a_t) \bar{V}_{t+1}^{n-1}(s') \right).$$

We quickly see that to find \hat{v}_t^n , we have to compute the embedded expectation over the potential downstream states that result from our action a_t . By contrast, since Q -learning depends on observations from an exogenous process, we do not need a transition function (known as a “model”) $S^M(S^n, a^n, W^{n+1})$. For example, we might be trying to optimize our play at online poker. Given our cards (in our hand, and anything displayed on the table), we make a decision, observe the play of our opponent, and then observe the next state. We do not receive a contribution until the end of the game.

If we choose the action using equation (4.5), the resulting algorithm would not be guaranteed to lead to an optimal solution. The problem is that the factors $\bar{Q}^n(s, a)$ might underestimate the value of a state-action pair. As a result we may

not choose actions that take us to this state, which means that we do not correct the error, and we may end up ignoring actions that might be quite attractive. What we need is a rule that forces us to *explore* states and actions that may not look attractive, because we have not visited them often enough. One of the simplest ways of overcoming this is to modify our policy for choosing an action using a rule such as the ϵ -greedy policy. Using this policy, with probability ϵ we choose an action at random from the set \mathcal{A} . With probability $1 - \epsilon$ we choose the action according to equation (4.5), in which case we say that we are *exploiting* our current knowledge of the value of each state-action pair.

The ϵ -greedy policy is simple and intuitive, and produces a guarantee that we will visit every (reachable) state and action infinitely often. This can work well in practice for some problems, but not for others. Solving the problem of when to explore and when to exploit is known as the *exploration versus exploitation* problem. This is a difficult problem that is an active area of research. See Chapter 12 for a more complete discussion of this problem.

Not surprisingly, Q -learning is difficult to apply to problems with even modest state and action spaces, but its value lies in its ability to solve problems without a model.

4.3.2 SARSA

SARSA is an algorithm that is closely related to Q -learning that allows us to illustrate an important issue that pervades approximate dynamic programming. SARSA works as follows: Say we are in a state s , choose an action a , after which we observe a reward r , observe the next state s' , then choose an action a' . The sequence s, a, r, s', a' makes up the name of the algorithm.

Say we use some policy A^π for choosing $a^n = A^\pi(S^n)$. We then simulate the next state S^{n+1} , perhaps by using our transition function $S^{n+1} = S^M(S^n, a^n, W^{n+1})$, where W^{n+1} is a sampled observation of our random noise. We choose the next action using $a^{n+1} = A^\pi(S^{n+1})$. For the purpose of this section only, we use $r(s, a)$ to represent our reward (rather than a contribution $C(s, a)$). We then compute a sample estimate of the value of being in state S^n and taking action $a^n = A^\pi(S^n)$ as follows:

$$\hat{q}^n(S^n, a^n) = r(S^n, a^n) + \gamma \bar{Q}^{n-1}(S^{n+1}, a^{n+1}).$$

We can then use $\hat{q}^n(S^n, a^n)$ to update $\bar{Q}^{n-1}(S^n, a^n)$ as we did with Q -learning.

SARSA is an algorithm for learning the value of a fixed policy $A^\pi(s)$. The policy has to guarantee that all actions are tested infinitely often, and by construction, it means that actions are tested with a frequency that is fixed by the policy. Also, by following the state from S^n to S^{n+1} , we ensure that we are going to sample states with the correct distribution determined by the policy. However, we are not searching for an optimal (or even good) policy; the algorithm only estimates the value of a policy. While this seems like a modest result, such algorithms play an important role in policy iteration algorithms, where we have to estimate the value of following a policy before updating the policy.

4.3.3 On-policy, Off-policy, and the Exploration Problem

Q-learning and SARSA provide a nice mechanism for introducing the idea of *off-policy* and *on-policy* algorithms. With SARSA, we use the same policy $A^\pi(s)$ for choosing action a^n when we are in state s^n as we do when we reach into the future and choose action $a' = A^\pi(S^{n+1})$. As long as our policy ensures that all actions will be chosen infinitely often, we can guarantee that our *Q*-factors will converge to the correct value of being in state s and choose action a while following policy $A^\pi(s)$. Also, by following the trajectory from state S^n to state $S^{n+1} = S^M(S^n, A^\pi(S^n), W^{n+1})$, we ensure that we are sampling states in the proportion determined by the policy.

Q-learning, to the contrary, may use a policy such as ϵ -greedy to choose an action a , but downstream, it uses $\max_a \bar{Q}^{n-1}(S^{n+1}, a)$ to choose the next action. In other words, one policy is used to decide which action to evaluate, and a separate policy is used to choose the action in the future.

In approximate dynamic programming, we often like to choose what we think is the best action given a state. Once a certain policy appears best, we would like to evaluate this policy. But, if we insist on choosing actions we think are best, we may get stuck visiting a small number of states that look good, while avoiding states that may not look good, and simply end up with poor estimates. To overcome this problem, we may introduce exploration strategies such as ϵ -greedy, but this means we are using one policy to force exploration, while tracking the performance of a policy that chooses what action to take.

The policy that determines which action to take, from which we determine the next state to visit, is often referred to as the *behavior policy*, since it often describes how a physical system is actually behaving. If we have an online system, the behavior policy is describing how a system behaves, but in a simulation we may choose this policy to control the process of sampling states. In this setting we prefer to use the more descriptive term *sampling policy*.

The policy that determines the action that appears to be best is then called the *target policy* in the reinforcement learning community, but is also sometimes known by the more descriptive name *learning policy* (literally, the policy we are trying to learn). Our goal is typically to improve the target policy, while using a sampling policy to ensure that we visit states often enough. When the learning policy and the sampling policy are the same (as they are with SARSA), this is called *on-policy learning*. When the learning and sampling policies are different, this is called *off-policy learning*.

The use of on-policy or off-policy learning is an issue that pervades approximate dynamic programming and reinforcement learning. With on-policy learning we may not be able to ensure that we are sampling different parts of the state space often enough to estimate the value of being in these states. By contrast, off-policy learning may interfere with our ability to learn the value of a learning policy, although this is primarily an issue when we use parametric models to approximate a policy (see Chapter 9 for a further discussion of this issue).

4.4 REAL-TIME DYNAMIC PROGRAMMING

The next algorithm we consider assumes that we can compute the one-step transition matrix (as we did in algorithm 4.2), but this time we are going to make a small modification that enables us to guarantee that the algorithm will eventually converge to the optimal policy. The strategy was introduced in the research literature under the name *real-time dynamic programming*, abbreviated RTDP.

The best way to understand RTDP is through its deterministic analogue known as the A^* algorithm, which is a type of shortest path algorithm. Imagine a deterministic network where c_{ij} is the deterministic contribution of traversing link (i, j) . We want to get from an origin q to a destination r while maximizing the total contribution. Suppose that we are given a value v_i that overestimates the contribution from node i to the destination node r . Now, imagine that we are at some node i , choose the link (i, j) out of i that maximizes $c_{ij} + v_j$ over all j . If (i, j^*) is the best link, let $v_i = c_{ij^*} + v_{j^*}$, traverse to j^* , and repeat the process. The algorithm runs by starting the process at node q , restarting at node q each time we reach the destination (or otherwise become stuck). If we start with optimistic estimates of the value of each node, we can guarantee that every time we update the estimate, the resulting revised estimate is also optimistic. If we are at a node i and do not traverse a link (i, j) (which would allow us to update $v(j)$), even if $v(j)$ is incorrect, we can ensure that we have chosen an even better path.

RTDP is a stochastic version of the A^* algorithm. Again, assume that we have an optimistic estimate of $V(s)$, the value of being at a state s . However, if we take an action a , we now assume there is a known probability $P(s'|s, a)$ that gives the probability that we will transition to state s' . If we are in state S^n , we choose an action a^n by solving

$$\hat{v}^n = \max_{a \in \mathcal{A}^n} \left(C(S^n, a) + \gamma \sum_{s'} \mathbb{P}(s'|S^n, a) V^{n-1}(s') \right).$$

and let a^n be the resulting optimal action. We update $V^n(S^n) = \hat{v}^n$, leaving the values of all other states unchanged. We then choose the next state using

$$S^n = S^M(S^n, a^n, W^{n+1}), \quad (4.8)$$

where W^{n+1} is a sample realization of a possible transition. A version of an RTDP algorithm (there are different variations) is given in Figure 4.3.

As with the A^* algorithm, if $V^{n-1}(s)$ is optimistic, then we can guarantee that $V^n(s)$ is also optimistic (for all states). As long as the values are optimistic, we will explore actions that appear to be best. The use of optimistic estimates means that if we do not select an action (even with an optimistic estimate), then we can safely ignore the action. If we are choosing an action only because the downstream value is optimistic, eventually this downstream value will be corrected.

Unfortunately, we pay a significant price for this guarantee. First, we have to compute the one-step transition matrix so that we can calculate the expectation exactly. Second, the use of optimistic estimates means that we may be visiting every state (multiple times), which is an issue with large state spaces.

Step 0. Initialization.

Step 0a. Initialize $\bar{V}^0(s)$ for all states s .

Step 0b. Choose an initial state S^1 .

Step 0c. Set $n = 1$.

Step 1. Choose a sample path ω^n .

Step 2a. Solve

$$\hat{v}^n = \max_{a \in \mathcal{A}^n} \left(C(S^n, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|S^n, a) \bar{V}^{n-1}(s') \right),$$

and let a^n be the value of a that solves the maximization problem.

Step 2b. Update $\bar{V}^{n-1}(S^n)$ using

$$\bar{V}^n(S) = \begin{cases} \hat{v}^n, & S = S^n, \\ \bar{V}^{n-1}(S), & \text{otherwise.} \end{cases}$$

Step 2c. Compute $S^n = S^M(S^n, a^n, W(\omega^n))$.

Step 3. Let $n = n+1$. If $n < N$, go to step 1.

Figure 4.3 Illustrative real-time dynamic programming algorithm.

4.5 APPROXIMATE VALUE ITERATION

For our next algorithm we are going to eliminate the assumption that we can compute the one-step transition matrix. One strategy is to randomly generate a sample of outcomes $\hat{\Omega}^n$, in iteration n , of what W might be. Let $p^n(\omega)$ be the probability of outcome $\hat{\omega} \in \hat{\Omega}^n$, where, if we choose N observations in the set $\hat{\Omega}^n$, it might be the case that $p^n(\omega) = 1/N$. We could then approximate the expectation using

$$\begin{aligned} \mathbb{E}\bar{V}^{n-1}(S^M(S^n, a, W)) &\approx \sum_{\omega \in \hat{\Omega}^n} p^n(\omega) \bar{V}^{n-1}(S^M(S^n, a, W(\omega))), \\ \hat{v}^n &= \max_{a \in \mathcal{A}} \left(C(S^n, a) + \gamma \sum_{\hat{\omega} \in \hat{\Omega}^n} p^n(\hat{\omega}) \bar{V}^{n-1}(S^M(S^n, a, W(\hat{\omega}))) \right), \end{aligned} \quad (4.9)$$

which is an estimate of the value of being in state S^n . We then update our estimate of the value of being in state S^n using

$$\bar{V}^n(S^n) = (1 - \alpha_{n-1}) \bar{V}^{n-1}(S^n) + \alpha_{n-1} \hat{v}^n. \quad (4.10)$$

Here α_{n-1} is known as a “stepsize” (among other things), and generally takes on values between 0 and 1 (see Chapter 11 for an in-depth discussion of equation (4.10)). Equation (4.10) is an operation that we see a lot of in approximate dynamic

programming. It is variously described as “smoothing,” (or “exponential smoothing”), a “linear filter,” or “stochastic approximation.” Sometimes we use a constant stepsize (e.g., $\alpha_n = 0.1$), a deterministic formula (e.g., $\alpha_n = 1/n$), or one that adapts to the data. All are trying to do the same thing: use observations of noisy data (\hat{v}^n) to approximate the mean of the distribution from which the observations are being drawn. $\bar{V}^n(S^n)$ is viewed as the best estimate of the value of being in state S^n after iteration n .

The smoothing is needed only because of the randomness in \hat{v}^n due to the way we approximated the expectation. If we were able to compute the expectation exactly, we would compute

$$\bar{V}^n(S^n) = \max_{a \in \mathcal{A}^n} \left(C(S^n, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|S^n, a) V^{n-1}(s') \right).$$

This is similar to equation (4.11) using $\alpha_{n-1} = 1$. It is also the same as our value iteration algorithm (Section 3.4), except that now we are updating a single state S^n rather than all states.

An outline of an approximate dynamic programming algorithm is provided in Figure 4.4. In contrast with the algorithm in Figure 4.2, we now have completely eliminated any step that requires looping over all the states in the state space. In theory at least, we can now tackle problems with state spaces that are infinitely large. We face only the practical problem of filling in the details so that the algorithm actually works on a particular application.

Step 0. Initialization.

Step 0a. Initialize $\bar{V}^0(S)$ for all states S .

Step 0b. Choose an initial state S_0^1 .

Step 0c. Set $n = 1$.

Step 1. Choose a random sample of outcomes $\hat{\Omega}^n \subset \Omega$ representing possible realizations of the information arriving between t and $t+1$.

Step 2. Solve

$$\hat{v}^n = \max_{a \in \mathcal{A}^n} \left(C(S^n, a) + \gamma \sum_{\hat{\omega} \in \hat{\Omega}^n} p^n(\hat{\omega}) \bar{V}^{n-1}(S^M(S^n, a, W(\hat{\omega}))) \right). \quad (4.11)$$

Let a^n be the value of a that solves the maximization problem.

Step 3. Update $\bar{V}^{n-1}(S^n)$:

$$\bar{V}^n(S) = \begin{cases} (1 - \alpha_{n-1}) \bar{V}^{n-1}(S^n) + \alpha_{n-1} \hat{v}^n, & S = S^n, \\ \bar{V}^{n-1}(S), & \text{otherwise.} \end{cases}$$

Step 4. Compute $S^{n+1} = S^M(S^n, a^n, W(\omega^n))$.

Step 5. Let $n = n+1$. If $n < N$, go to step 1.

Figure 4.4 Forward dynamic programming algorithm approximating the expectation.

4.6 THE POST-DECISION STATE VARIABLE

There is a simple and elegant way of avoiding the ugly step of approximating the expectation that will work for many applications. The idea uses a powerful construct known as the *post-decision state variable*. The post-decision state variable is the state of the system after we have made a decision but before any new information has arrived. For a wide range of applications, the post-decision state is no more complicated than the pre-decision state, and for many problems is it much simpler. In fact the post-decision state is a device that opens up the door to solving vector-valued decisions, a largely overlooked class of problems in approximate dynamic programming.

Given the importance and power of the post-decision state, this section develops this idea in considerable depth. We close by showing how the post-decision state simplifies a basic approximate value iteration algorithm.

4.6.1 Finding the Post-decision State Variable

For many problems it is possible to break down the effect of decisions and information on the state variable. For example, we might write the transition function for an inventory problem as

$$R_{t+1} = \max\{R_t + a_t - \hat{D}_{t+1}, 0\}.$$

We can break this into two steps, given by

$$\begin{aligned} R_t^a &= R_t + a_t, \\ R_{t+1} &= \max\{R_t^a - \hat{D}_{t+1}, 0\}. \end{aligned}$$

R_t^a captures the pure effect of the decision a_t to order additional assets, while R_{t+1} captures the effect of serving demands \hat{D}_{t+1} .

If this is possible, we can break our original transition function

$$S_{t+1} = S^M(S_t, a_t, W_{t+1}) \quad (4.12)$$

into the two steps

$$\begin{aligned} S_t^a &= S^{M,a}(S_t, a_t), \\ S_{t+1} &= S^{M,W}(S_t^a, W_{t+1}). \end{aligned}$$

S_t is the state of the system immediately before we make a decision, while S_t^a is the state immediately after we make a decision (which is why it is indexed by t). For this reason we sometimes refer to S_t as the *pre-decision state variable* while S_t^a is the *post-decision state variable*. Whenever we refer to a “state variable,” we always mean the pre-decision state S_t . We refer to the pre-decision state only when there is a specific need to avoid confusion.

We use the function $S^{M,a}(S_t, a_t)$ (or $S^{M,x}(S_t, x_t)$) if we are using x as our decision vector) extensively in our presentation. However, while there is some elegance in defining the “post- to pre-” transition function $S^{M,W}(S_t^a, W_{t+1})$, we do not actually need it in our algorithms. We use the pre- to post-transition function $S^{M,a}(S_t, a_t)$ purely for the purpose of computing the value function, and then we use the classical pre- to pre-transition in (4.12) when we step forward in time.

We first saw pre- and post-decision state variables in Section 2.2.1 on decision trees. Figure 4.5 illustrates a generic decision tree with decision nodes (squares) and outcome nodes (circles). The information available at a decision node is the pre-decision state, and the information available at an outcome node is the post-decision state. The function $S^{M,a}(S_t, a_t)$ takes us from a decision node (pre-decision state) to an outcome node (post-decision state). The function $S^{M,W}(S_t^a, W_{t+1})$ takes us from an outcome node to a decision node.

We revisit the use of post-decision state variables repeatedly through this book. Its value is purely computational. The benefits are problem specific, but in some settings they are significant.

4.6.2 A Perspective of the Post-decision State Variable

The decision of when to estimate the value function around a post-decision state variable is fairly simple. If you can easily compute the expectation $\mathbb{E}V_{t+1}(S_{t+1})$,

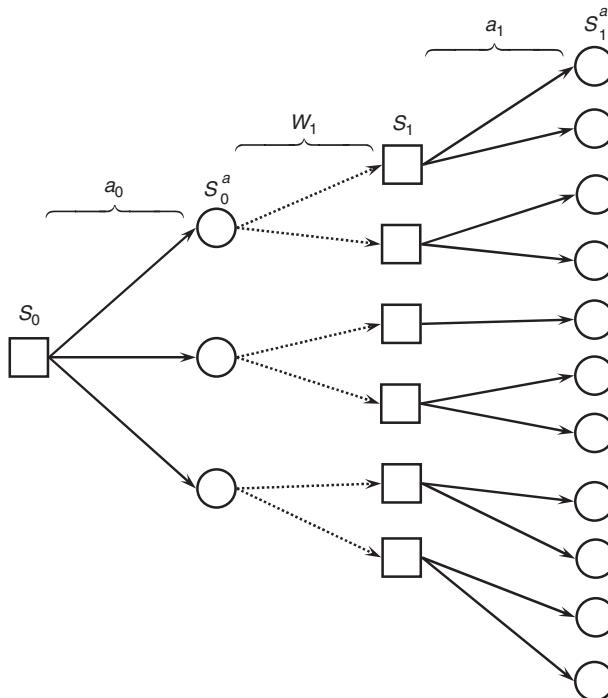


Figure 4.5 Generic decision tree, showing decision nodes (squares) and outcome nodes (circles). Solid lines are decisions, and dotted lines are random outcomes.

you should use the pre-decision state variable. If the expectation is hard (the “second curse of dimensionality”), you might consider seeing if you can identify a compact post-decision state variable. If you can compute an expectation exactly, taking advantage of the structure of the problem, you are going to produce a more reliable estimate than using Monte Carlo based methods.

The vast majority of papers in the ADP community use the pre-decision state variable, reflecting, we believe, the classical development of Markov decision processes where the ability to take expectations is taken for granted. A simple example illustrates the potential of using the post-decision state variable. Imagine that we are trying to predict unemployment for year t as a function of interest rates in year t . If U_t is the unemployment rate in year t and I_t is the interest rate, we might create a model of the form

$$U_t = \theta_0 + \theta_1(I_t)^2, \quad (4.13)$$

where we would estimate θ_0 and θ_1 using historical data on unemployment and interest rates. Now imagine that we wish to use our model to predict unemployment next year. We would first have to predict interest rates and then use our model to predict unemployment.

If we use the same data to estimate the model, then

$$U_t = \theta_0 + \theta_1(I_{t-1})^2. \quad (4.14)$$

Estimating this model requires no additional work, but now we can use interest rates this year to predict next year’s unemployment. Not surprisingly, this is a much easier model to use.

Approximating a value function around the pre-decision state S_{t+1} is comparable to using equation (4.14). At time t , we first have to forecast S_{t+1} and then compute $\bar{V}_{t+1}(S_{t+1})$. Approximating the value function around the post-decision state variable is comparable to using (4.14), which allows us to compute a variable in the future (U_{t+1}) based on what we know now (I_t).

4.6.3 Examples of Post-decision State Variables

The power of the post-decision state depends largely on the nature of the underlying application. Some problems offer considerable structure, but it takes some time to develop a sense of how to construct a post-decision state. Below we provide some examples that illustrate different settings.

Decision Trees—Outcome Nodes

In Figure 4.6 we see a typical decision tree representing a sequence of decisions (solid lines) and information (dashed lines). At each node we show the current estimate of the value of being at that node, written as $\bar{v}_1^n, \dots, \bar{v}_4^n$. Now assume that at node 1, we have made a decision that takes us to outcome node 2, after which a random event takes us to node 3, and we then made a decision that takes

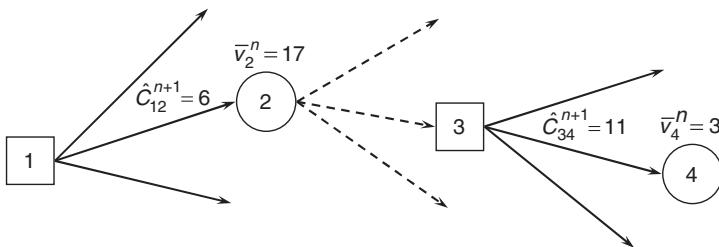


Figure 4.6 Sequence of decision nodes and outcome nodes in a decision tree, with estimates of the value of each node prior to observing costs \hat{C} .

us to node 4. The decision at node 1 produced a contribution of $\hat{C}_{12}^{n+1} = 6$, while the decision at node 3 produced a contribution $\hat{C}_{34}^{n+1} = 11$.

The decision nodes (squares) represent the states where decisions are made. The circles are typically referred to as outcome nodes, but these are also equivalent to post-decision states. Let $\hat{v}_3^{n+1} = \hat{C}_{34}^{n+1} + \bar{v}_4^n = 11 + 3 = 14$. We can use \hat{v}_3^{n+1} to update the value \bar{v}_2^n of the previous, post-decision state using $\bar{v}_2^{n+1} = (1 - \alpha)\bar{v}_2^n + \alpha\hat{v}_3^{n+1} = (0.9)17 + (0.1)14 = 16.7$.

In a classical decision tree an outcome node is equivalent to specifying a post-decision state $S_t^a = (S_t, a_t)$, which is to say a state-action pair. It is *always* possible to specify a post-decision state as a state-action pair, but this means that the post-decision state is higher dimensional than the pre-decision state.

A Stochastic Graph

Imagine that you are traversing a graph with random arc costs. As you arrive to node i , you are allowed to see the actual cost on links out of node i that you will incur if you traverse one of those links. If \hat{c}_{tij} is the sample realization of the cost on link (i, j) when you reach node i , the pre-decision state is $S_t = (i, (\hat{c}_{tij})_j)$. Now assume that you choose to traverse the link (i, k) . Once you make the decision, and while you are still at node i , your post-decision state would be $S_t^a = (k)$. Your next pre-decision state would be $S_{t+1} = (k, (\hat{c}_{k\ell})_\ell)$. In this example the post-decision state is much simpler than the pre-decision state. In fact this situation is quite common when the pre-decision state includes information needed to make a decision (e.g., the costs on links out of a node), but otherwise are not needed to model the transition.

Selling an Asset

The post-decision state is widely used to evaluate the value of an option to sell an asset. Let

$$R_t = \begin{cases} 1 & \text{if we are holding the asset at time } t, \\ 0 & \text{otherwise,} \end{cases}$$

$$a_t = \begin{cases} 1 & \text{if we sell the asset at time } t, \\ 0 & \text{otherwise,} \end{cases}$$

where

- p_t = price that we obtain for the asset if we sell at time t ,
- c^{hold} = cost of holding the asset during time interval t ,
- \hat{p}_t = change in price that arrives during time interval t ,
- T = time by which the asset must be sold.

The pre-decision state variable is given by

$$S_t = (R_t, p_t).$$

The post-decision state is given by $S_t^a = (R_t^a, p_t^a)$, given by

$$R_t^a = \begin{cases} 1 & \text{if } R_t = 1 \text{ and } a_t = 0, \\ 0 & \text{otherwise,} \end{cases}$$

$$p_t^a = p_t.$$

The next pre-decision state is then given by

$$R_{t+1} = R_t^a,$$

$$p_{t+1} = p_t^a + \hat{p}_{t+1}.$$

This is an important special case of a problem where there is a controllable dimension of the state variable that evolves deterministically and an uncontrollable dimension that evolves stochastically.

A Water Reservoir Problem

Water reservoirs have become an important component of an energy system that is depending increasingly on energy from wind and solar, both notoriously variable sources. As wind increases, it is relatively easy to cut back on the output of a hydroelectric generator; similarly, if the wind drops, we can increase the output. We also have the ability to pump water back into the reservoir during periods of high wind and low demand.

At the foundation of these problems is a simple inventory equation given by

$$R_{t+1} = R_t - x_t + \hat{R}_{t+1},$$

where R_t is the amount of water in the reservoir at time t , x_t is the amount of water we choose to pump out at time t (if $x_t > 0$) or pump back in (if $x_t < 0$). \hat{R}_{t+1} represents random, exogenous inflows to the reservoir from precipitation and snowmelt. Assume for this exercise that the sequence $\hat{R}_1, \hat{R}_2, \dots, \hat{R}_t$ is independent and identically distributed.

Given this structure, the pre-decision state is given by $S_t = R_t$, and we might model the post-decision state as $S_t^x = R_t^x = R_t - x_t$. However, there is another choice. Let \bar{R}_t be a forecast of \hat{R}_{t+1} given what we know at time t . We might also

represent the post-decision state as

$$S_t^x = R_t^x = R_t - x_t + \bar{R}_t.$$

Thus S_t^x can be viewed as our *expected state* at time $t+1$ given what we know at time t .

Meeting Energy Demands—Creating Low-Dimensional Problems

Companies that operate our electric power grid have to match supplies of energy from coal, nuclear, natural gas, wind, and solar to meet demands on a minute by minute basis. Demands from electricity come from sectors such as residential, commercial, light industrial, heavy industrial, and transportation. Let

$$D_{tj} = \text{demand for energy by demand sector } j.$$

We assume that the demand for energy can be forecasted up to an exogenous noise term using

$$D_{t+1,j} = \mu_{tj}^D + \epsilon_{t+1,j}^D,$$

where we are going to assume that the error terms ϵ_{tj}^D are independent and identically distributed over time.

We can represent the flows of energy from each supply to each demand using

$$x_{tij} = \text{flow of energy from source } i \text{ to demand sector } j \text{ at time } t.$$

The supply of energy is governed by the physics of ramping up and shutting down nuclear plants, coal plants, and natural gas facilities.

Assume also that we have a single reservoir attached to the grid, where we can move energy into the reservoir by pumping water uphill during low demand periods or withdraw energy from the reservoir (just as we did in the previous section). We represent these flows using

$$x_{ti0} = \text{flow of energy from source } i \text{ into the reservoir.}$$

$$x_{t0j} = \text{flow of energy from the reservoir to serve demand sector } j.$$

Finally let R_t be a scalar variable representing the amount of energy held in the reservoir (there will be a conversion from megawatthours to gallons of water). The transition equations would be given by

$$R_{t+1} = R_t + \sum_i x_{ti0} - \sum_j x_{t0j} + \hat{R}_{t+1},$$

where we again let \hat{R}_{t+1} be random inputs to the reservoir such as rainfall (but now measured in megawatt-hours). The pre-decision state is given by

$$S_t = (R_t, D_t),$$

where R_t is a scalar, but D_t is a vector (note that all these quantities are continuous). Note that since μ_t^D is a deterministic vector (representing the predictable patterns of energy demand), we do not view it as part of the state variable. The post-decision state variable would now be $S_t^x = R_t^x$, where

$$R_t^x = R_{t+1} = R_t + \sum_i x_{ti0} - \sum_j x_{t0j},$$

or

$$R_t^x = R_{t+1} = R_t + \sum_i x_{ti0} - \sum_j x_{t0j} + \bar{R}_t,$$

where again \bar{R}_t would be a forecast of \bar{R}_{t+1} made at time t . Note that while we have high-dimensional state and decision variables, the post-decision state is a scalar.

This example illustrates how the post-decision state, for some problems, can have much lower dimensionality than the pre-decision state. It also illustrates that a process such as the demands D_t can vary stochastically in a way that makes them drop out of the post-decision state variable. This would not be the case if we modeled the evolution of demands using

$$D_{t+1,j} = D_{tj} + \epsilon_{tj}.$$

In this case we would need to keep the vector D_t as part of the post-decision state variable.

Selling Stocks—Action-Dependent Processes

Investment funds face a challenge when selling large blocks of stock because the simple act of selling can depress the stock. Let

R_t = number of shares of the stock that are held at time t ,

x_t = number of shares to be sold at time t ,

p_t = current market price.

We might start with a simple model described using

$$R_{t+1} = R_t - x_t,$$

$$p_{t+1} = p_t + \hat{p}_{t+1},$$

where \hat{p}_{t+1} is the exogenous change in the price. Our pre-decision state would be $S_t = (R_t, p_t)$ while the post-decision state would be $S_t^x = (R_t - x_t, p_t)$. However, what if we introduce the dimension that \hat{p}_{t+1} depends on x_t ? We might assume, for example, that $\mathbb{E}\{\hat{p}_{t+1}|p_t, x_t\} = -\beta x_t$, where β captures the downward pressure of selling on the price. A brute-force way of handling this problem is to simply add x_t to the post-decision state, giving us $S_t^x = (R_t - x_t, p_t, x_t)$ or, more compactly, $S_t^x = (S_t, x_t)$ (we would write this as $S_t^a = (S_t, a_t)$ if we are using our notation

for discrete actions). We saw this earlier in our decision-tree example in Figure 4.5, where the post-decision state is given by the outcome node. Note that there is one outcome node for each state-action pair.

One attraction of this way of writing a post-decision state is that it is very general—it applies to any problem. However, the price of this strategy is that the post-decision state space, rather than possibly being more compact, was instead multiplied by the size of the action space. If we have a discrete state space \mathcal{S} and a discrete action space \mathcal{A} , the post-decision state space is now $|\mathcal{S}| \times |\mathcal{A}|$.

Sometimes this growth of the post-decision state space cannot be avoided. However, for this particular application, we can use a more compact representation if we take advantage of the deterministic way that actions influence future randomness. If x_t has only the effect of shifting the mean of the price process downward (without otherwise affecting the distribution), then our post-decision state for this problem would be

$$S_t^x = (R_t - x_t, p_t - \beta x_t).$$

Moving a Robot

Imagine that we have a robot that is moving left to right along a line. The robot may occupy any one of six positions, and may move left or right at positions 2–5, but must move right at position 1 and must move left at position 5. If the robot is in position 2 and moves right, his intended position is 3, but there is noise in the transition and the robot may end up at position 2, 3, or 4. Table 4.2 summarizes all the states, actions, and the probabilities of landing in a subsequent state.

Given this table, we can represent the *intended* state as the post-decision state. It is important that if we intend to land in state 2, the probability that it eventually lands in state 1, ..., 6 is independent of where the robot came from.

Now assume that the probabilities are as given in Table 4.3. This time, if we are in location 3 and decide to go right to 4, the probability of where we

Table 4.2 Pre-decision state (location), action, post-decision state (intended destination), and next pre-decision state

Location	Action	Post-decision State	Probability of Next Pre-decision State					
			1	2	3	4	5	6
2	Left	1	0.8	0.15	0.05	0	0	0
3	Left	2	0.1	0.8	0.1	0	0	0
4	Left	3	0	0.1	0.8	0.1	0	0
5	Left	4	0	0	0.1	0.8	0.1	0
6	Left	5	0	0	0	0.1	0.8	0.1
1	Right	2	0.1	0.8	0.1	0	0	0
2	Right	3	0	0.1	0.8	0.1	0	0
3	Right	4	0	0	0.1	0.8	0.1	0
4	Right	5	0	0	0	0.1	0.8	0.1
5	Right	6	0	0	0	0.05	0.15	0.8

Table 4.3 Pre-decision state (location), action, post-decision state (intended destination), and next pre-decision state

Location	Action	Post-decision State	Probability of Next Pre-decision State					
			1	2	3	4	5	6
2	Left	(2, Left)	0.8	0.15	0.05	0	0	0
3	Left	(3, Left)	0.13	0.8	0.07	0	0	0
4	Left	(4, Left)	0	0.13	0.8	0.07	0	0
5	Left	(5, Left)	0	0	0.13	0.8	0.07	0
6	Left	(6, Left)	0	0	0	0.13	0.8	0.07
1	Right	(1, Right)	0.07	0.8	0.13	0	0	0
2	Right	(2, Right)	0	0.07	0.8	0.13	0	0
3	Right	(3, Right)	0	0	0.07	0.8	0.13	0
4	Right	(4, Right)	0	0	0	0.07	0.8	0.13
5	Right	(5, Right)	0	0	0	0.05	0.15	0.8

eventually end up depends on the fact that we arrived at location 4 from location 3. If we land in location 4 from location 5, the next transition depends on where we came from. In this case we would normally represent the post-decision state as the state-action pair.

Homoscedastic Noise

Our robot problem is a special case of a more general stochastic system that evolves according to the dynamics

$$S_{t+1} = A_t S_t + B_t x_t + \varepsilon_{t+1}, \quad (4.15)$$

where S_t is a continuous n -dimensional vector, x_t is a continuous m -dimensional decision (control), A_t and B_t are suitably defined matrices, and ε_{t+1} is an n -dimensional noise term that is independent and identically distributed. When the noise does not depend on the state, it is called *homoscedastic*. It is fairly easy to see that $S_t^x = A_t S_t + B_t x_t$ is the post-decision state.

State-Dependent Noise

There are, of course, many applications where the noise depends on the state, which means the noise is *heteroscedastic*. For example, if S_t is the speed of wind, the variation in the wind is much larger when the wind is faster. If the wind is fairly quiet, the variation is smaller. In this setting the state transition is still given by equation (4.14), but now ε_{t+1} depends on S_t . However, even if this is the case, the post-decision state is still given by $S_t^x = A_t S_t + B_t x_t$, since ε_{t+1} does not depend on x_t .

There may be problems where ε_{t+1} is a random variable whose distribution depends on S_t and x_t . For example, imagine a mutual fund whose holding of a stock is given by S_t , and x_t describes decisions to buy and sell. The mutual fund may be large enough that a large decision to buy ($x_t \gg 0$) may increase the price.

Let \hat{p}_{t+1} be the change in price so that our price dynamics are governed by

$$p_{t+1} = p_t + \hat{p}_{t+1}.$$

The distribution of \hat{p}_{t+1} may depend on x_t , but perhaps we are willing to write the random variable in the form

$$\hat{p}_{t+1} = \theta_0 \operatorname{sign}(x_t) \left(\frac{x_t}{\theta_1} \right)^2 + \xi_{t+1},$$

where θ_0 and θ_1 are parameters, $\operatorname{sign}(x_t) = -1$ if $x_t < 0$, and $+1$ if $x_t > 0$. Here ξ_{t+1} is a random variable with mean 0 that does not depend on p_t or x_t . Of course, all we are doing is recognizing that our random variable \hat{p}_{t+1} has a deterministic mean and a homoscedastic error term. Pulling out the mean leaves us with an error term that does not depend on the state or action, which means we are back with our homoscedastic setting. The point of this illustration is that it is important to separate the truly random component of a random variable from any component that is a deterministic function of the state and/or action.

4.6.4 The Optimality Equations Using the Post-decision State Variable

Just as we earlier defined $V_t(S_t)$ to be the value of being in state S_t (just before we made a decision), let $V_t^a(S_t^a)$ be the value of being in state S_t^a immediately after we made a decision. There is a simple relationship between $V_t(S_t)$ and $V_t^a(S_t^a)$ that is summarized as follows

$$V_{t-1}^a(S_{t-1}^a) = \mathbb{E} \{ V_t(S_t) | S_{t-1}^a \}, \quad (4.16)$$

$$V_t(S_t) = \max_{a_t \in \mathcal{A}_t} (C_t(S_t, a_t) + \gamma V_t^a(S_t^a)), \quad (4.17)$$

$$V_t^a(S_t^a) = \mathbb{E} \{ V_{t+1}(S_{t+1}) | S_t^a \}. \quad (4.18)$$

where $S_t = S^{M,W}(S_{t-1}^a, W_t)$ in (4.15), $S_t^a = S^{M,a}(S_t, a_t)$ in equation (4.17), and $S_{t+1} = S^{M,W}(S_t^a, W_{t+1})$ in equation (4.18). Equation (4.16) writes $V_{t-1}^a(S_{t-1}^a)$ as a function of $V_t(S_t)$. Equation (4.18) does the same for the next time period, whereas equation (4.17) writes $V_t(S_t)$ as a function of $V_t^a(S_t^a)$. Note that we put the discount factor γ only when we take the expectation, since this is where we step backward in time.

If we substitute (4.17) into (4.16), we obtain the standard form of Bellman's equation

$$V_t(S_t) = \max_{a_t \in \mathcal{A}_t} (C_t(S_t, a_t) + \gamma \mathbb{E} \{ V_{t+1}(S_{t+1}) | S_t \}).$$

By contrast, if we substitute (4.16) into (4.15), we obtain the optimality equations around the post-decision state variable

$$V_{t-1}^a(S_{t-1}^a) = \mathbb{E} \left\{ \max_{a_t \in \mathcal{A}_t} (C_t(S_t, a_t) + V_t^a(S_t^a)) \middle| S_{t-1}^a \right\}. \quad (4.19)$$

What should immediately stand out is that the expectation is now outside of the max operator. While this should initially appear to be a much more difficult equation to solve (we have to solve an optimization problem within the expectation), this is going to give us a tremendous computational advantage. Equation (4.17) is a deterministic optimization problem that can still be quite challenging for some problem classes (especially those involving the management of discrete resources), but there is a vast research base that we can draw on to solve these problems. By contrast, the expectation in equation (4.16) or (4.18) may be easy, but it is often computationally intractable. This is the step that typically requires the use of approximation methods.

Writing the value function in terms of the post-decision state variable provides tremendous computational advantages (as well as considerable simplicity). Since Bellman's equation, written around the pre-decision state variable, is absolutely standard even in the field of approximate dynamic programming, this represents a significant point of departure of our treatment of approximate dynamic programming.

4.6.5 An ADP Algorithm Using the Post-decision State

Assume, as we did in Section 4.2, that we have found a suitable approximation $\bar{V}_t(S_t^a)$ for the value function around the post-decision state S_t^a . As before, we are going to run our algorithm iteratively. Assume that we are in iteration n and that at time t that we are in state S_t^n .

Our optimization problem at time t can now be written

$$\hat{v}_t^n = \max_{a_t \in \mathcal{A}_t^n} (C_t(S_t^n, a_t) + \bar{V}_t^{n-1}(S_t^{M,a}(S_t^n, a_t))). \quad (4.20)$$

We use the notation S_t^n to indicate that we are at a particular state rather than at a set of all possible states. We also note that the feasible region \mathcal{A}_t^n depends on the state S_t^n (the subscript t indicates that we are at time t with access to the information in S_t , while the superscript n indicates that we are depending on the n th sample realization).

Let a_t^n be the value of a_t that solves (4.19). What is very important about this problem is that it is deterministic. We do not require directly computing or even approximating an expectation as we did in equation (4.9) (the expectation is already captured in the deterministic function $\bar{V}_t^{n-1}(S_t^n)$). For large-scale problems, which we consider later, this will prove critical.

We next need to update our value function approximation. \hat{v}_t^n is a sample realization of the value of being in state S_t^n so that we can update our value function approximation using

$$\bar{V}_t^n(S_t^n) = (1 - \alpha_{n-1})\bar{V}_t^{n-1}(S_t^n) + \alpha_{n-1}\hat{v}_t^n.$$

There is nothing wrong with this expression, except that it will give us an estimate of the value of being in the pre-decision state S_t . When we are solving the decision

problem at time t (in iteration n), we would use $\bar{V}_{t+1}^{n-1}(S_{t+1})$, but since S_{t+1} is a random variable at time t , we have to compute (or at least approximate) the expectation.

A much more effective alternative is to use \hat{v}_t^n to update the value of being in the post-decision state $S_{t-1}^{a,n}$. Keep in mind that the previous decision a_{t-1}^n put us in state $S_{t-1}^{a,n}$, after which a random outcome, $W_t(\omega^n)$, put us in state S_t^n . So, while \hat{v}_t^n is a sample of the value of being in state S_t^n , it is also a sample of the value of the decision that put us in state $S_{t-1}^{a,n}$. Thus we can update our post-decision value function approximation using

$$\bar{V}_{t-1}^n(S_{t-1}^{a,n}) = (1 - \alpha_{n-1})\bar{V}_{t-1}^{n-1}(S_{t-1}^{a,n}) + \alpha_{n-1}\hat{v}_t^n. \quad (4.21)$$

Note that we have chosen not to use a superscript “ a ” for the value function approximation $\bar{V}_{t-1}^n(S_{t-1}^{a,n})$. In the remainder of this book, we are going to use the value function around the post-decision state almost exclusively, so we suppress the superscript “ a ” notation to reduce notational clutter. By contrast, we have to retain the superscript for our state variable, since we will use both pre-decision and post-decision state variables throughout.

The smoothing in equation (4.20) is where we are taking our expectation. If we write

$$V_t^a(S_t^a) = \mathbb{E} \{ V_{t+1}(S_{t+1}) | S_t^a \},$$

then the classical form of Bellman’s equation becomes

$$V_t(S_t) = \max_{a_t} (C_t(S_t, a_t) + V_t^a(S_t^a)).$$

The key step is that we are writing $\mathbb{E} \{ V_{t+1}(S_{t+1}) | S_t^a \}$ as a function of S_t^a rather than S_t , where we take advantage of the fact that S_t^a is a deterministic function of a_t .

A complete sketch of an ADP algorithm using the post-decision state variable is given in Figure 4.7.

Forward dynamic programming using the post-decision state variable is more elegant because it avoids the need to approximate the expectation explicitly within the optimization problem. It also gives us another powerful device. Since the decision function “sees” $\bar{V}_t(S_t^a)$ directly (rather than indirectly through the approximation of the expectation), we are able to control the structure of $\bar{V}_t(S_t^a)$. This feature is especially useful when the myopic problem $\max_{a_t \in \mathcal{A}_t} C_t(S_t, a_t)$ is an integer program or a difficult linear or nonlinear program that requires special structure. This feature comes into play in the context of more complex problems, such as those discussed in Chapter 14.

Although it may not be entirely apparent now, this basic strategy makes it possible to produce production quality models for large-scale industrial problems that are described by state variables with millions of dimensions. The process of stepping forward in time uses all the tools of classical simulation, where it is

Step 0. Initialization.

Step 0a. Initialize \bar{V}_t^0 , $t \in \mathcal{T}$.

Step 0b. Set $n = 1$.

Step 0c. Initialize S_0^1 .

Step 1. Choose a sample path ω^n .

Step 2. Do for $t = 0, 1, 2, \dots, T$:

Step 2a. Solve:

$$\hat{v}_t^n = \max_{a_t \in \mathcal{A}_t^n} (C_t(S_t^n, a_t) + \bar{V}_t^{n-1}(S^{M,a}(S_t^n, a_t))),$$

and let a_t^n be the value of a_t that solves the maximization problem.

Step 2b. If $t > 0$, update \bar{V}_{t-1}^{n-1} using

$$\bar{V}_{t-1}^n(S_{t-1}^{a,n}) = (1 - \alpha_{n-1})\bar{V}_{t-1}^{n-1}(S_{t-1}^{a,n}) + \alpha_{n-1}\hat{v}_t^n.$$

Step 2c. Find the post-decision state

$$S_t^{a,n} = S^{M,a}(S_t^n, a_t^n)$$

and the next pre-decision state

$$S_{t+1}^n = S^M(S_t^n, a_t^n, W_{t+1}(\omega^n)).$$

Step 3. Increment n . If $n \leq N$, go to step 1.

Step 4. Return the value functions $(\bar{V}_t^N)_{t=0}^T$.

Figure 4.7 Forward dynamic programming using the post-decision state variable.

possible to capture an arbitrarily high level of detail as we simulate forward in time. Our interest in this book is obtaining the highest quality decisions as measured by our objective function. The key here is creating approximate value functions that accurately capture the impact of decisions now on the future. For problems with possibly high-dimensional decision vectors, the optimization problem

$$\max_{a_t \in \mathcal{A}_t^n} (C_t(S_t^n, a_t) + \bar{V}_t^{n-1}(S^{M,a}(S_t^n, a_t)))$$

will have to be solved using the various tools of mathematical programming (linear programming, nonlinear programming, and integer programming). In order to use these tools, we need to identify in the value function approximation certain properties that may play a role in the design of the approximation strategy.

4.6.6 The Post-decision State and Q -Learning

There is a very important relationship between approximate dynamic programming using the post-decision state and Q -learning. It stems from the simple observation

that a state-action pair is a form of post-decision state. In equations (4.16) and (4.17) we set up Bellman's equations around the pre- and post-decision states. For a steady-state problem (but retaining the time-indexing on the states and actions, so the sequencing is clear), these equations are given by

$$\begin{aligned} V(S_t) &= \max_{a_t \in \mathcal{A}} (C(S_t, a_t) + \gamma V^a(S^{M,a}(S_t, a_t))), \\ V^a(S_t^a) &= \mathbb{E} \{V(S_{t+1})|S_t^a\}. \end{aligned}$$

We next replace the value functions with value function approximations, where we are going to use $\bar{V}(s)$ to approximate $V(s)$ around the pre-decision state, and $\bar{V}^a(S^a)$ to approximate the value around the post-decision state. Let

$$\hat{v}_t^a = \bar{V}^a(S_{t+1}(\omega)),$$

where $S_{t+1}(\omega)$ is a sample realization of S_{t+1} , which is to say $S_{t+1}(\omega) = S^M(S_t, a_t, W_{t+1}(\omega))$. \hat{v}_t^a is a sample realization of the value of being at a post-decision state $S_t^a = S^{M,a}(S_t, a_t)$ and then evolving to $S_{t+1}(\omega)$ under sample realization ω . This would be the same as \hat{q} (in equation (4.6)) but we did not capture the direct contribution from taking action a_t . For this reason we write \hat{q}_t as

$$\hat{q}_t = C(S_t, a_t) + \gamma \hat{v}_t^a.$$

If our contributions are random, we would use $\hat{C}(S_t, a_t) = C(S_t, a_t, W_{t+1}(\omega))$ instead of $C(S_t, a_t)$.

If we were using the approximate value iteration around the post-decision state, we would perform smoothing of \hat{v}_t^a to estimate $\bar{V}^a(S^a)$. With Q -learning (or SARSA) we would smooth \hat{q}_t to estimate $\bar{Q}(S_t, a_t)$. We easily see that $\bar{V}(S_t) = \max_{a_t} \bar{Q}(S_t, a_t)$. This means that if our problem lends itself to a compact post-decision state, Q -learning and approximate value iteration are basically the same.

4.6.7 The Post-decision State and Multidimensional Decision Vectors

The use of the post-decision state variable offers the advantage of eliminating the embedded expectation, which is important for problems where this expectation is difficult to compute. For some problem classes it also offers the advantage that it may be much simpler than the pre-decision state variable. But it also opens up the door to handling multi-dimensional decision vectors.

Imagine that we have a fleet of vehicles (robots, trucks, unmanned aerial vehicles), with each vehicle described by its location $i \in \mathcal{I}$ (we can make this a lot more complicated, but this simple setting will do for now). Assume that we move a vehicle to a location to serve some task at the location. Assume that the opportunity for earning a reward \hat{C}_{ti} at location i is random, but it becomes known at time t . Let

$$R_{ti} = \text{number of vehicles at location } i \text{ at time } t,$$

$$x_{tij} = \text{number of vehicles we send from } i \text{ to } j \text{ at time } t,$$

$$c_{ij} = \text{cost of sending a vehicle from } i \text{ to } j,$$

\hat{C}_{ti} = contribution earned for each vehicle at i at time t ,
which is random before time t .

Our state vector at time t is given by $S_t = (R_t, \hat{C}_t)$ where $R_t = R_{ti})_{i \in \mathcal{I}}$ and $\hat{C}_t = (\hat{C}_{ti})_{i \in \mathcal{I}}$. The dynamics are given by

$$R_{t+1,j} = \sum_i x_{tij}, \quad (4.22)$$

and our decisions are constrained by

$$\sum_i x_{tij} = R_{ti}, \quad (4.23)$$

$$x_{tij} \geq 0. \quad (4.24)$$

Let \mathcal{X}_t be the region defined by constraints (4.22) and (4.23). Equation (4.21) defines the transition function for our resources, which we can also represent using $R_{t+1} = R^M(R_t, x_t)$, which is deterministic. The system transition function $S^M(S_t, x_t, W_{t+1})$ is stochastic, where W_{t+1} includes the realization of the contributions \hat{C}_{t+1} , but this does not affect our resource transition function.

The contribution function is given by

$$C(S_t, x) = \sum_i \hat{C}_{ti} R_{ti} - \sum_i \sum_j c_{ij} x_{tij}.$$

We can only receive the contribution \hat{C}_{ti} if the vehicles are already at i .

Using approximate dynamic programming, we could in principle find a decision x_t via

$$x_t = \arg \min_{x \in \mathcal{X}_t} (C(S_t, x) + \gamma \mathbb{E}\{V_{t+1}(S_{t+1})|S_t\}).$$

The post-decision state for this problem is $S_t = R_t^x$, where $R_t^x = R^M(R_t, x_t)$, given by equation (4.21). Note that since the resource transition function is deterministic, $R_{t+1} = R_t^x$. If we approximate the value function around the post-decision state, we would solve

$$x_t = \arg \min_{x \in \mathcal{X}_t} (C(S_t, x) + \bar{V}_t(R_t^x)).$$

Now we have to find a good approximation for $\bar{V}_t(R_t^x)$. A simple strategy is to use a function that is linear in R_t^x , given by

$$\bar{V}_t(R_t^x) = \sum_{i \in \mathcal{I}} \bar{v}_{ti} R_{ti}^x.$$

This would then produce the following optimization problem:

$$x_t = \arg \min_{x \in \mathcal{X}_t} \left(\sum_i \hat{C}_{ti} R_{ti} - \sum_i \sum_j c_{ij} x_{tij} + \sum_{i \in \mathcal{I}} \bar{v}_{ti} R_{ti}^x \right). \quad (4.25)$$

The optimization problem in (4.24) is a fairly simple linear program. So handling vectors x_t with hundreds or thousands of dimensions is relatively easy. We have made the sudden transition from problems that are characterized by a small number of discrete actions to problems with decision vectors with a thousand dimensions. Chapter 14 describes a number of applications, including the use of ADP in production applications in industry.

4.7 LOW-DIMENSIONAL REPRESENTATIONS OF VALUE FUNCTIONS

Classical dynamic programming typically assumes a discrete representation of the value function. This means that for every state $s \in \mathcal{S}$, we have to estimate a parameter v_s that gives the value of being in state s . Forward dynamic programming may eliminate the loop over all states that is required in backward dynamic programming, but it does not solve the classic “curse of dimensionality” in the state space. Forward dynamic programming focuses attention on the states that we actually visit, but it also requires that we have some idea of the value of being in a state that we *might* visit (we need this estimate to conclude that we should not visit the state).

Virtually every large-scale problem in approximate dynamic programming will focus on determining how to approximate the value function with a smaller number of parameters. In backward discrete dynamic programming, we have one parameter per state, and we want to avoid searching over a large number of states. In forward dynamic programming, we depend on Monte Carlo sampling, and the major issue is statistical error. It is simply easier to estimate a function that is characterized by fewer parameters.

In practice, approximating value functions always requires understanding the structure of the problem. However, there are general strategies that emerge. Below we discuss two of the most popular.

4.7.1 Aggregation

In the early days of dynamic programming, aggregation was quickly viewed as a way to provide a good approximation with a smaller state space, allowing the tools described in Chapter 3 to be used. A major problem with aggregation when we use the framework in Chapter 3 is that we have to solve the entire problem in the aggregated state space. With approximate dynamic programming, we only have to aggregate for the purpose of computing the value function approximation. It is extremely valuable that we can retain the full state variable for the purpose of computing the transition function, costs, and constraints. In the area of approximate dynamic programming, aggregation provides a mechanism for reducing statistical error. Thus, while it may introduce structural error, it actually makes a model more accurate by improving statistical robustness.

■ EXAMPLE 4.1

Consider the problem of evaluating a basket of securities where \hat{p}_{ti} is the price of the i th security at time t , where the state variable $S_t = (\hat{p}_{ti})_i$ is the vector of all the prices. We can discretize the prices, but we can dramatically simplify the problem of estimating $\bar{V}_t(S_t)$ if we use a fairly coarse discretization of the prices. We can discretize prices for the purpose of estimating the value function without changing how we represent prices of each security as we step forward in time. ■

■ EXAMPLE 4.2

A trucking company that moves loads over long distances has to consider the profitability of assigning a driver to a particular load. Estimating the value of the load requires estimating the value of a driver at the destination of the load. We can estimate the value of the driver at the level of the 5-digit zip code of the location, or the 3-digit level, or at the level of a region (companies typically represent the United States using about 100 regions, which is far more aggregate than a 3-digit zip). The value of a driver in a location may also depend on his home domicile, which can also be represented at several levels of aggregation. ■

Aggregation is particularly powerful when there are no other structural properties to exploit, which often arises when the state contains dimensions that are categorical rather than numerical. When this is the case, we typically find that the state space \mathcal{S} does not have any metric to provide a “distance” between two states. For example, we have an intuitive sense that a disk drive company and a company that makes screens for laptops both serve the personal computer industry. We would expect that valuations in these two segments would be more closely correlated than they would be with a textile manufacturer. But we do not have a formal metric that measures this relationship.

In Chapter 12 we investigate the statistics of aggregation in far greater detail.

4.7.2 Continuous Value Function Approximations

In many problems the state variable does include continuous elements. Consider a resource allocation problem where R_{ti} is the number of resources of type $i \in \mathcal{I}$. R_{ti} may be either continuous (how much money is invested in a particular asset class, and how long has it been invested there) or discrete (the number of people with a particular skill set), but it is always numerical. The number of possible values of a vector $R_t = (R_{ti})_i$ can be huge (the curse of dimensionality). Now consider what happens when we replace the value function $V_t(R_t)$ with a linear approximation of the form

$$\bar{V}_t = \sum_{i \in \mathcal{I}} \bar{v}_{ti} R_{ti}.$$

Instead of having to estimate the value $V_t(R_t)$ for each possible vector R_t , we have only to estimate \bar{v}_{ti} , one for each value of $i \in \mathcal{I}$. For some problems, this reduces the size of the problem from 10^{100} or greater to one with several hundred or several thousand parameters.

Not all problems lend themselves to linear-in-the-resource approximations, but other approximations may emerge. Early in the development of dynamic programming, Bellman realized that a value function could be represented using statistical models and estimated with regression techniques. To illustrate, let $\bar{V}(R|\theta)$ be a statistical model where the elements of R_t are used to create the independent variables, and θ is the set of parameters. For example, we might specify

$$\bar{V}(R|\theta) = \sum_{i \in \mathcal{I}} (\theta_{1i} R_i + \theta_{2i} (R_i)^2).$$

The formulation and estimation of continuous value function approximations is one of the most powerful tools in approximate dynamic programming. The fundamentals of this approach are presented in considerably more depth in Chapters 8, 9 and 10.

4.7.3 Algorithmic Issues

The design of an approximation strategy involves two algorithmic challenges. First, we have to be sure that our value function approximation does not unnecessarily complicate the solution of the myopic problem. We have to assume that the myopic problem is solvable in a reasonable period of time. If we are choosing our decision a_t by enumerating all possible decisions (a “lookup table” decision function), then this is not an issue, but if a_t is a vector, then this is not going to be possible. If our myopic problem is a linear or nonlinear program, it is usually impossible to consider a value function that is of the discrete lookup table variety. If our myopic problem is continuously differentiable and concave, we do not want to introduce a potentially nonconcave value function. By contrast, if our myopic problem is a discrete scheduling problem that is being solved with a search heuristic, then lookup table value functions can work just fine.

Once we have decided on the structure of our functional approximation, we have to devise an updating strategy. Value functions are basically statistical models that are updated using classical statistical techniques. However, it is very convenient when our updating algorithm is in a recursive form. A strategy that fits a set of parameters by implementing a sequence of observations using standard regression techniques may be too expensive for many applications.

4.8 SO JUST WHAT IS APPROXIMATE DYNAMIC PROGRAMMING?

Approximate dynamic programming can be viewed from four very different perspectives. Depending on the problem you are trying to solve, ADP can be viewed as a way of improving an observable, physical process; an algorithmic strategy for solving complex dynamic programs; a way of making classical simulations more intelligent; and a decomposition technique for large-scale mathematical programs.

4.8.1 Reinforcement Learning

Approximate dynamic programming, as it is practiced under the umbrella of “reinforcement learning,” has long been viewed as a way of learning the state-action pairs that produce the best results. In this view we have a physical process that allows us to observe transitions from S_t to S_{t+1} (model-free dynamic programming). In a state S_t , we can choose actions according to a policy $A^\pi(s)$, and then learn the value of following this policy. We can use various strategies to help us update the policy $A^\pi(s)$.

4.8.2 ADP for Solving Complex Dynamic Programs

In the research community, approximate dynamic programming is primarily viewed as a way of solving dynamic programs that suffer from “the curse of dimensionality” (in this book, the three curses). Faced with the need to solve Bellman’s equation, we have to overcome the classical problem of computing the value function $V(S)$ when the number of states S is too large to enumerate. We also consider problems where the expectation cannot be computed, and where the action space is too large to enumerate. The remainder of this book presents modeling and algorithmic strategies for overcoming these problems.

4.8.3 ADP as an “Optimizing Simulator”

Part of the power of approximate dynamic programming is that it is basically a form of classical simulation with more intelligent decision rules. Almost any ADP algorithm looks something like the algorithm depicted in Figure 4.8, which includes two major steps: an optimization step, where we choose a decision, and a simulation step, where we capture the effects of random information. Viewed in this way, ADP is basically an “optimizing simulator.” The complexity of the state variable S_t and the underlying physical processes, captured by the transition function $S^M(S_t^a, x_t, W_{t+1})$, are virtually unlimited (as is the case with any simulation model). We are, as a rule, limited in the complexity of the value function approximation $\bar{V}_t(S_t^a)$. If S_t^a is too complex, we have to design a functional approximation

Step 0. Given an initial state S_0^1 and value function approximations $\bar{V}_t^0(S_t)$ for all S_t and t , set $n = 1$.

Step 1. Choose a sample path ω^n .

Step 2. For $t = 0, 1, 2, \dots, T$ do:

Step 2a. Optimization: Compute a decision $a_t^n = A_t^\pi(S_t)$ and find the post-decision state $S_t^{a,n} = S^{M,a}(S_t^n, a_t^n)$.

Step 2b. Simulation: Find the next pre-decision state using $S_t^n = S^M(S_t^n, a_t^n, W_{t+1}(\omega^n))$.

Step 3. Update the value function approximation to obtain $\bar{V}_t^n(S_t)$ for all t .

Step 4. If we have not met our stopping rule, let $n = n+1$ and go to step 1.

Figure 4.8 Generic approximate dynamic programming algorithm.

that uses only the most important features. But it is critical in many applications that we do not have to similarly simplify the state variable used to compute the transition function.

There are many complex, industrial applications where it is very important to capture the physical process at a high level of detail, but where the quality of the decisions (i.e., how close the decisions are to the optimal decisions) is much harder to measure. A modest relaxation in the optimality of the decisions may be difficult or impossible to measure, whereas simplifications in the state variable are quickly detected because the system simply does not evolve correctly. It has been our repeated experience in many industrial applications that it is far more important to capture a high degree of realism in the transition function than it is to produce truly optimal decisions.

4.8.4 ADP as a Decomposition Technique for Large-Scale Math Programs

Our foray into approximate dynamic programming began with our efforts to solve very large-scale linear (and integer) programs that arise in a range of transportation applications. Expressed as a linear program (and switching to a decision vector x), these would look like

$$\max_{(x_t)_{t=0,\dots,T}} \sum_{t=0}^T c_t x_t$$

subject to

$$\begin{aligned} A_0 x_0 &= R_0, \\ A_t x_t - B_{t-1} x_{t-1} &= \hat{R}_t, \quad t = 1, \dots, T, \\ D_t x_t &\leq u_t, \quad t = 1, \dots, T, \\ x &\geq 0 \quad \text{an integer.} \end{aligned}$$

Here R_0 is the initial inventories of vehicles and drivers (or crews or pilots), while \hat{R}_t is the (deterministic) net inflow or outflow of resources to or from the system. A_t captures flow conservation constraints, while B_{t-1} tells us where resources that were moved at time $t-1$ end up in the future. D_t tells us which elements of x_t are limited by an upper bound u_t .

In industrial applications, x_t might easily have 10,000 elements, defined over 50 time periods. A_t might have thousands of rows, or hundreds of thousands (depending on whether we were modeling equipment or drivers).

Formulated as a single, large linear (or integer) program (over 50 time periods), we obtain mathematical programs with hundreds of thousands of rows and upward of millions of columns. As this book is being written, there are numerous researchers attacking these problems using a wide range of heuristic algorithms and decomposition strategies. Yet our finding has been that even these large-scale models ignore a number of operating issues (even if we ignore uncertainty).

ADP would solve this problem by solving sequences of problems that look like

$$\max_{x_t} (c_t x_t + \bar{V}_{t+1}(R_{t+1}(x_t)))$$

subject to, for $t = 1, \dots, T$,

$$\begin{aligned} A_t x_t &= R_t + \hat{R}_t, \\ R_{t+1} - B_t x_t &= 0, \\ D_t x_t &\leq u_t, \\ x &\geq 0 \quad \text{an integer}, \end{aligned}$$

where $R_t = B_{t-1} x_{t-1}$ and $\bar{V}_{t+1}(R_{t+1})$ is some sort of approximation that captures the impact of decision at time t on the future. After solving the problem at time t , we would compute R_{t+1} , increment t , and solve the problem again. We would run repeated simulations over the time horizon while improving our approximation $\bar{V}_t(R_t)$.

In this setting it is quite easy to introduce uncertainty, but the real value is that instead of solving one extremely large linear program over many time periods (something that even modern optimization packages struggle with), we solve the problem one time period at a time. Our finding is that commercial solvers handle these problems quite easily, even for the largest and most complex industrial applications. The price is that we have to design an effective approximation for $\bar{V}_t(R_t)$.

4.9 EXPERIMENTAL ISSUES

Once we have developed an approximation strategy, we have the problem of testing the results. Two issues tend to dominate the experimental side: rate of convergence and solution quality. The time required to execute a forward pass can range between microseconds and hours, depending on the underlying application. You may be able to assume that you can run millions of iterations, or you may have to work with a few dozen. The answer depends on both your problem and the setting. Are you able to run for a week on a large parallel processor to get the very best answer? Or do you need real-time response on a purchase order or to run a “what if” simulation with a user waiting for the results?

Within our time constraints we usually want the highest quality solution possible. Here the primary interest tends to be in the rate of convergence. Monte Carlo methods are extremely flexible, but the price of this flexibility is a slow rate of convergence. If we are estimating a parameter from statistical observations drawn from a stationary process, the quality of the solution tends to improve at a rate proportional to the square root of the number of observations. Unfortunately, we typically do not enjoy the luxury of stationarity. We usually face the problem of estimating a value function that is steadily increasing or decreasing (it might well be increasing for some states and decreasing for others) as the learning process evolves.

4.9.1 The Initialization Problem

All of the strategies above depend on setting an initial estimate \bar{V}^0 for the value function approximation. In Chapter 3 the initialization of the value function did not affect our ability to find the optimal solution; it only affected the rate of convergence. In approximate dynamic programming it is often the case that the value function approximation \bar{V}^{n-1} used in iteration n affects the choice of the action taken and therefore the next state visited.

Consider the shortest path problem illustrated in Figure 4.9a, where we want to find the best path from node 1 to node 7.

Assume that at any node i , we choose the link that goes to node j , which minimizes the cost c_{ij} from i to j , plus the cost \bar{v}_j of getting from node j to the destination. If $\hat{v}_i = \min_j(c_{ij} + \bar{v}_j)$ is the value of the best decision out of node i , then we will set $\bar{v}_i = \hat{v}_i$. If we start with a low initial estimate for each \bar{v} (Figure 4.9a), we first explore a more expensive path (Figure 4.9b) before finding the best path (Figure 4.9c) where the algorithm stops.

Now assume we start with values for \bar{v}_j that are too high, as illustrated in Figure 4.9d. We first explore the lower path (Figure 4.9e), but we never find the better path.

If we have a deterministic problem and start with an optimistic estimate of the value of being in a state (too low if we are minimizing, too high if we are maximizing), then we are guaranteed to eventually find the best solution. This is a popular algorithm in the artificial intelligence community known as the A* algorithm (pronounced “A star”). However, we generally can never guarantee this in the presence of uncertainty. Just the same, the principle of starting with optimistic estimates remains the same. If we have an optimistic estimate of the value of being in a state, we are more likely to explore that state. The problem is that if the estimates are too optimistic, we may end up exploring too many states.

4.9.2 State Sampling Strategies

There are several strategies for sampling states in dynamic programming. In Chapter 3 we used *synchronous* updating because we updated the value of all states at the same time. The term is derived from parallel implementations of the algorithm, where different processors might independently (but at the same time, hence synchronously) update the value of being in each state. In approximate dynamic programming, synchronous updating would imply that we are going to loop over all the states, updating the value of being in each state (potentially on different processors). An illustration of synchronous approximate dynamic programming is given in Figure 4.10 for an infinite horizon problem. Note that at iteration n , all decisions are made using $\bar{V}^{n-1}(s)$.

Asynchronous dynamic programming, to the contrary, assumes that we are updating one state at a time, after which we update the entire value function. In asynchronous ADP we might choose states at random, allowing us to ensure that we sample all states infinitely often. The term “asynchronous” is based on parallel implementations where different processors might be updating different states

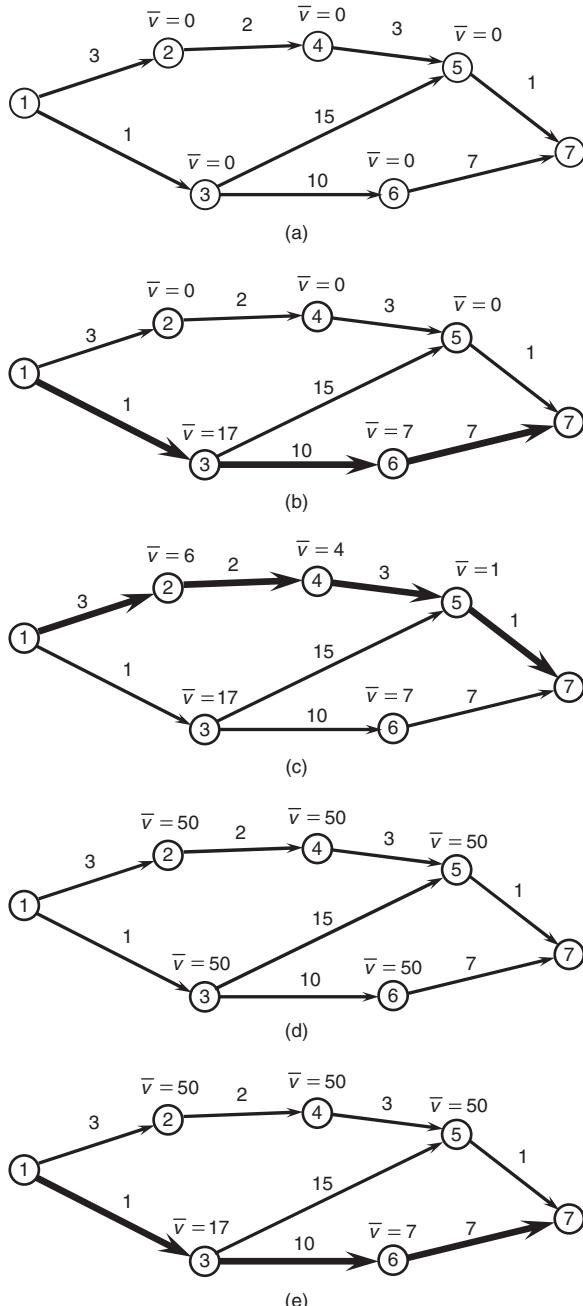


Figure 4.9 Shortest path problems with low (a, b, c) and high (d, e) initial estimates of the cost from each node to the destination. (a) Shortest path problem with low initial estimates for the value at each node. (b) After one pass; updates cost to destination along this path. (c) After second pass; finds optimal path. (d) Shortest path problem with high initial estimates for the value at each node. (e) After one pass, finds cost along more expensive path, but never finds best path.

Step 0. Initialize an approximation for the value function $\bar{V}^0(S)$ for all states S . Let $n = 1$.

Step 1. For each state $s \in \mathcal{S}$, do

$$\hat{v}^n(s) = \max_{a \in \mathcal{A}} \left(C(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a) \bar{V}^{n-1}(s') \right).$$

Step 2. Use the estimate $\hat{v}^n(s)$ to update $\bar{V}^{n-1}(s)$, giving us $\bar{V}^n(s)$.

Step 3. Let $n = n+1$. If $n < N$, go to step 1.

Figure 4.10 Synchronous dynamic programming.

Step 0. Initialize an approximation for the value function $\bar{V}^0(S)$ for all states S . Let $n = 1$.

Step 1. Randomly choose a state s^n .

Step 2. Solve

$$\hat{v}^n = \max_{a \in \mathcal{A}} \left(C(s^n, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s^n, a) \bar{V}^{n-1}(s') \right).$$

Step 3. Use \hat{v}^n to update the approximation $\bar{V}^{n-1}(s)$ for all s .

Step 4. Let $n = n+1$. If $n < N$, go to step 1.

Figure 4.11 Asynchronous approximate dynamic programming.

without any guarantee of the order or timing with which states are being updated. The procedure is illustrated in Figure 4.11.

4.9.3 Exploration versus Exploitation

Closely related to the initialization problem is the classical issue of “exploration” versus “exploitation.” Exploration implies visiting states specifically to obtain better information about the value of being in a state, regardless of whether the decision appears to be the best given the current value function approximation. Exploitation means using our best estimate of the contributions and values, and making decisions that seem to be the best given the information we have (we are “exploiting” our estimates of the value function). Of course, exploiting our value function to visit a state is also a form of exploration, but the literature typically uses the term “exploration” to mean that we are visiting a state in order to improve our estimate of the value of being in that state.

There are two ways to explore. After we update the value of one state, we might then choose another state at random, without regard to the state that we just visited (or the action we chose). The second is to randomly choose an action (even it is not the best) that leads us to a somewhat randomized state. The value of the latter is that it constrains our search to states that can be reasonably reached, avoiding the problem of visiting states that can never be reached.

There is a larger issue when considering different forms of exploration. We have introduced several problems (the blood management problem in Section 1.2, the dynamic assignment problem in Section 2.2.10, and the asset acquisition problem in Section 4.1) where both the state variable and the decision variable are multi-dimensional vectors. For these problems the state space and the action space are effectively infinite. Randomly sampling a state or even an action for these problems makes little sense.

Determining the right balance between exploring states just to estimate their values, along with using current value functions to visit the states that appear to be the most profitable, represents one of the great unsolved problems in approximate dynamic programming. If we only visit states that appear to be the most profitable given the current estimates (a pure exploitation strategy), then we run the risk of landing in local optima unless the problem has special properties. There are strategies that help minimize the likelihood of being trapped in a local optima but at a cost of very slow convergence. Finding good strategies with fairly fast convergence appears to depend on taking advantage of natural problem structure. This topic is covered in considerably more depth in Chapter 12.

4.9.4 Evaluating Policies

A common question is whether a policy A^{π_1} is better than another policy A^{π_2} . Suppose that we are facing a finite horizon problem that can be represented by the objective function

$$F^\pi = \mathbb{E} \left\{ \sum_{t=0}^T \gamma^t C_t(S_t, A_t^\pi(S_t)) \right\}.$$

Since we cannot compute the expectation, we might choose a sample $\hat{\Omega} \subseteq \Omega$ and then calculate

$$\begin{aligned} \hat{F}^\pi(\omega) &= \sum_{t=0}^T \gamma^t C_t(A_t^\pi(S_t(\omega))), \\ \bar{F}^\pi &= \sum_{\omega \in \hat{\Omega}} \hat{p}(\omega) \hat{F}^\pi(\omega), \end{aligned}$$

where $\hat{p}(\omega)$ is the probability of the outcome $\omega \in \hat{\Omega}$. If we have chosen the outcomes in $\hat{\Omega}$ at random from within Ω , then, letting $N = |\hat{\Omega}|$ be our sample size, we would use

$$\hat{p}(\omega) = \frac{1}{|\hat{\Omega}|} = \frac{1}{N}.$$

Alternatively, we may choose $\hat{\Omega}$ so that we control the types of outcomes in Ω that are represented in $\hat{\Omega}$. Such sampling strategies fall under names such as stratified sampling or importance sampling. They require that we compute the

sample probability distribution \hat{p} to reflect the proper frequency of an outcome $\omega \in \hat{\Omega}$ within the larger sample space Ω .

The choice of the size of $\hat{\Omega}$ should be based on a statistical analysis of \bar{F}^π . For a given policy π , it is possible to compute the variance of $\bar{F}^\pi(\hat{\Omega})$ using

$$(\bar{\sigma}^\pi)^2 = \frac{1}{N} \left(\frac{1}{N-1} \right) \sum_{\omega \in \hat{\Omega}} (\hat{F}^\pi(\omega) - \bar{F}^\pi)^2.$$

This formula assumes that we are sampling outcomes “at random” from Ω , which means that they should be equally weighted. More effective strategies will use sampling strategies that will overrepresent certain types of outcomes.

In most applications it is reasonable to assume that $(\bar{\sigma}^\pi)^2$ is independent of the policy, allowing us to use a single policy to estimate the variance of our estimate. If we treat the estimates of \bar{F}^{π_1} and \bar{F}^{π_2} as independent random variables the variance of the difference is $2(\bar{\sigma}^\pi)^2$. If we are willing to assume normality, we can then compute a confidence interval on the difference using

$$[(\bar{F}^{\pi_1} - \bar{F}^{\pi_2}) - z_{\alpha/2} \sqrt{2(\bar{\sigma}^\pi)^2}, (\bar{F}^{\pi_1} - \bar{F}^{\pi_2}) + z_{\alpha/2} \sqrt{2(\bar{\sigma}^\pi)^2}], \quad (4.26)$$

where $z_{\alpha/2}$ is the standard normal deviate for a confidence level α .

Typically we can obtain a much tighter confidence interval by using the same sample $\hat{\Omega}$ to test both policies. In this case \bar{F}^{π_1} and \bar{F}^{π_2} will not be independent, and may in fact be highly correlated (in a way we can use to our benefit). Instead of computing an estimate of the variance of the value of each policy, we should compute a sample realization of the difference

$$\hat{\Delta}^{\pi_1, \pi_2}(\omega) = \hat{F}^{\pi_1}(\omega) - \hat{F}^{\pi_2}(\omega),$$

from which we can compute an estimate of the difference

$$\begin{aligned} \bar{\Delta}^{\pi_1, \pi_2} &= \sum_{\omega \in \hat{\Omega}} \hat{p}(\omega) \hat{\Delta}^{\pi_1, \pi_2}(\omega) \\ &= \bar{F}^{\pi_1} - \bar{F}^{\pi_2}. \end{aligned}$$

When comparing two policies, it is important to compute the variance of the estimate of the difference to see if it is statistically significant. If we evaluate each policy using a different set of random outcomes (e.g., ω_1 and ω_2), the variance of the difference would be given by

$$\text{Var}[\hat{\Delta}^{\pi_1, \pi_2}] = \text{Var}[\hat{F}^{\pi_1}] + \text{Var}[\hat{F}^{\pi_2}]. \quad (4.27)$$

This is generally not the best way to estimate the variance of the difference between two policies. It is better to evaluate two policies using the same random sample for each policy. In this case $\hat{F}^{\pi_1}(\omega)$ and $\hat{F}^{\pi_2}(\omega)$ are usually correlated, which means the variance would be

$$\text{Var}[\hat{\Delta}^{\pi_1, \pi_2}] = \text{Var}[\hat{F}^{\pi_1}] + \text{Var}[\hat{F}^{\pi_2}] - 2 \text{Cov}[\hat{F}^{\pi_1}, \hat{F}^{\pi_2}].$$

The covariance is typically positive, so this estimate of the variance will be smaller (and possibly much smaller). One way to estimate the variance is to compute $\hat{\Delta}^{\pi_1, \pi_2}(\omega)$ for each $\omega \in \hat{\Omega}$ and then compute

$$\bar{\sigma}^{\pi_1, \pi_2} = \frac{1}{N} \left(\frac{1}{N-1} \right) \sum_{\omega \in \hat{\Omega}} \left(\hat{\Delta}^{\pi_1, \pi_2}(\omega) - \bar{\Delta}^\pi \right)^2.$$

In general, $\bar{\sigma}^{\pi_1, \pi_2}$ will be much smaller than $2(\bar{\sigma}^\pi)^2$, which we would obtain if we chose independent estimates.

For some large-scale experiments it will be necessary to perform comparisons using a single sample realization ω . In fact this is the strategy that would typically be used if we were solving a steady-state problem. However, the strategy can be used for any problem where the horizon is sufficiently long that the variance of an estimate of the objective function is not too large. We again emphasize that the variance of the difference in the estimates of the contribution of two different policies may be much smaller than would be expected if we used multiple sample realizations to compute the variance of an estimate of the value of a policy.

4.10 BUT DOES IT WORK?

The technique of stepping forward through time using Monte Carlo sampling is a powerful strategy, but it effectively replaces the challenge of looping over all possible states with the problem of statistically estimating the value of being in “important states.” Furthermore it is not enough just to get reasonable estimates for the value of being in a state. We have to get reasonable estimates for the value of being in states we *might* want to visit.

There is a growing body of theory addressing the issue of convergence proofs.

As of this writing, formal proofs of convergence are limited to a small number of very specialized algorithms. For lookup table representations of the value function, it is not possible to obtain convergence proofs without a guarantee that all states will be visited infinitely often. This precludes pure exploitation algorithms where we only use the decision that appears to be the best (given the current value function approximation).

Compounding this lack of proofs is experimental work that illustrates cases where the methods simply do not work. What has emerged from the various laboratories doing experimental work are two themes:

- The functional form of an approximation has to reasonably capture the true value function.
- For large problems it is essential to exploit the structure of the problem so that a visit to one state provides improved estimates of the value of visiting a large number of other states.

For example, a discrete lookup table function will always capture the general shape of a (discrete) value function, but it does little to exploit what we have learned

from visiting one state in terms of updated estimates of the value of visiting other states. As a result it is quite easy to design an approximate dynamic programming strategy (e.g., using a lookup table value function) that either does not work at all or provides a suboptimal solution that is well below the optimal solution.

At the same time approximate dynamic programming has proved itself to be an exceptionally powerful tool in the context of specific problem classes. This chapter has illustrated ADP in the context of very simple problems, but it has been successfully applied to very complex resource allocation problems that arise in some of the largest transportation companies. Approximate dynamic programming (and reinforcement learning, as it is still called in artificial intelligence) has proved to be very effective in problems ranging from playing games (e.g., backgammon) to solving engine control problems (e.g., managing fuel mixture ratios in engines).

It is our belief that general-purpose results in approximate dynamic programming will be few and far between. Our experience suggests instead that most results will involve taking advantage of the structure of a particular problem class. Identifying a value function approximation, along with a sampling and updating strategy, that produces a high-quality solution represents a major contribution to the field in which the problem arises. The best we can offer in a general textbook on the field is to provide guiding principles and general tools, allowing domain experts to devise the best possible solution for a particular problem class. We suspect that an ADP strategy applied to a problem context is probably a patentable invention.

4.11 BIBLIOGRAPHIC NOTES

This chapter can be viewed as an introduction to approximate dynamic programming, but very much from the perspective adopted in the reinforcement learning community. The entire chapter assumes discrete states and actions, deferring until later the algorithms for handling more complex states and actions.

Section 4.2 The core ideas of approximate dynamic programming and reinforcement learning date to the 1950s, but primarily to work done in the 1980s and 1990s. See Bertsekas and Tsitsiklis (1996) and Sutton and Barto (1998). There is a very rich literature on generating random variables. An easy introduction to simulation is given in Ross (2002). More extensive treatments are given in Banks et al. (1996) and Law and Kelton (2000).

Section 4.3 *Q*-learning was first introduced by Watkins (1989) and Watkins and Dayan (1992), and it is probably the most widely studied algorithm in reinforcement learning. See Sutton and Barto (1998), Bertsekas and Tsitsiklis (1996), and the references cited there.

Section 4.4 Real-time dynamic programming was introduced in Barto et al. (1995).

Section 4.5 Approximate value iteration has been widely studied under the name temporal-difference learning, and it is closely related to *Q*-learning, especially when lookup tables are used. Bertsekas and Tsitsiklis (1996)

Table 4.4 Sampling of convergence proofs for ADP/RP

	State	Action	Reward	Exp.	Noise	Policy	Conv.
<i>Fixed Policy</i>							
Bradtko and Barti (1996)	D	D	G	N	S	FP	Y
Tsitsiklis (1994)	D	D	G	Y	S	FP	Y
Tsitsiklis and van Roy (1997)	D	D	G	Y	S	FP	Y
Papavassiliou and Russell (1999)	D	D	G	N	S	FP	Y
Precup (2001)	D	D	G	N	S	FP	Y
Melo and Ribeiro (2007)	C	D	G	N	S	FP	Y
Sutton et al. (2009b)	D	D	G	N	S	FP	Y
Sutton et al. (2009a)	D	D	G	N	S	FP	Y
<i>Optimizing Policies, Discrete States</i>							
Watkins and Dayan (1992)	D	D	G	S	S	VI	Y
Baird (1995)	D	D	G	N	D	VI	Y
Tsitsiklis and Roy (1996)	D	D	G	Y	S	VI	Y
Gordon (2001)	D	D	G	N	S	VI	Y
Lagoudakis and Parr (2003)	D	D	G	N	S	API	N
Precup and Perkins (2003)	D	D	G	N	S	API	Y
Deisenroth et al. (2008)	D	D	G	Y	S	VI	N
Gordon (1995)	D	D	G	Y	S	VI	Y
<i>Optimizing Policies, Continuous States</i>							
Bradtko et al. (1994)	C	C	Q	N	D	API	Y
Landelius and Knutsson (1997)	C	C	Q	N	D	VI/API	Y
Meyn (1997)	C	D	G	Y	S	EPI	Y
Ormoneit and Sen (2002)	C	D	G	N	S	VI	Y
Engel et al. (2005)	C	C	G	N	S	API	N
Munos and Szepesvari (2008)	C	D	G	N	S	VI	B
Szita (2007)	C	C	Q	N	S(G)	VI	Y
Antos et al. (2007)	C	C	G	N	S	API	B
Antos et al. (2008a)	C	D	G	N	S	API	B
Antos et al. (2008b)	C	D	G	N	S	API	B

Source: Adapted from Ma and Powell (2010b).

discuss several variations of approximate value iteration using lookup tables or state aggregations (where the value of being in a set of states is a constant) and establish conditions for convergence.

Section 4.6 Virtually every textbook on dynamic programming (or approximate dynamic programming) sets up Bellman's equations around the pre-decision state variable. However, a number of authors have found that some problems are naturally formulated around the post-decision state variable. The first use of the term "post-decision state variable" that we have found is in Bertsekas et al. (1997), although uses of the post-decision state variable date as far back as Bellman (1957). The first presentation describing the post-decision

state as a general solution strategy (as opposed to a technique for a special problem class) appears to be Powell and van Roy (2004). This book is the first book to systematically use the post-decision state variable as a way of approximating dynamic programs.

Section 4.7 The vast majority of work in approximate dynamic programming involves representing functions with fewer parameters. The use of aggregation has been suggested since the origins of dynamic programming in the 1950s, as has the use of statistical models, in Bellman and Dreyfus (1959).

Section 4.8 The A* algorithm is presented in a number of books on artificial intelligence. See, for example, Pearl (1984). The concept of synchronous and asynchronous dynamic programming is based on Bertsekas (1982) and in particular Bertsekas and Tsitsiklis (1989).

PROBLEMS

- 4.1** Let U be a random variable that is uniformly distributed between 0 and 1. Let $R = -\frac{1}{\lambda} \ln U$. Show that $\mathbb{P}[R \leq x] = 1 - e^{-\lambda x}$, which shows that R has an exponential distribution.
- 4.2** Let $R = U_1 + U_2$, where U_1 and U_2 are independent, uniformly distributed random variables between 0 and 1. Derive the probability density function for R .
- 4.3** Let Z be a normally distributed random variable with mean 0 and variance 1, and let U be a uniform $[0, 1]$ random variable. Let $\Phi(z) = \mathbb{P}(Z \leq z)$ be the cumulative distribution, and let $\Phi^{-1}(u)$ be the inverse cumulative. Then $R = \Phi^{-1}(U)$ is also normally distributed with mean 0 and variance 1. Use a spreadsheet to randomly generate 10,000 observations of U and the associated observations of R . Let $N(z)$ be the number of observations of R that are less than z . Compare $N(z)$ to $10000\Phi(z)$ for $z = -2, -1, -0.5, 0, 0.5, 1, 2$. [Note that in a spreadsheet, RAND() generates a uniform $[0,1]$ random variable, $\Phi(z) = \text{NORMSDIST}(z)$ and $\Phi^{-1}(u) = \text{NORMSINV}(u)$.]
- 4.4** Let X be a continuous random variable. Let $F(x) = 1 - \mathbb{P}[X \leq x]$, and let $F^{-1}(y)$ be its inverse (i.e., if $y = F(x)$, then $F^{-1}(y) = x$). Show that $F^{-1}(U)$ has the same distribution as X .
- 4.5** You are holding an asset that can be sold at time $t = 1, 2, \dots, 10$ at a price \hat{p}_t , that fluctuates from period to period. Assume that \hat{p}_t is uniformly distributed between 0 and 10. Further assume that \hat{p}_t is independent of prior prices. You have 10 opportunities to sell the asset, and you must sell it by the end of the 10th time period. Let x_t be the decision variable, where $x_t = 1$ means sell and $x_t = 0$ means hold. Assume that the reward you receive is the price that you receive when selling the asset ($\sum_{t=1}^{10} x_t p_t$).

- (a) Define the pre- and post-decision state variables for this problem. Plot the shape of the value function around the pre- and post-decision state variables.
- (b) Set up the optimality equations, and show that there exists a price \bar{p}_t where we will sell if $p_t > \bar{p}_t$. Also show that $\bar{p}_t \geq \bar{p}_{t+1}$.
- (c) Use backward dynamic programming to compute the optimal value function for each time period.
- (d) Use an approximate dynamic programming algorithm with a pre-decision state variable to estimate the optimal value function, and give your estimate of the optimal policy (i.e., at each price, should we sell or hold). Note that even if you sell the asset at time t , you need to stay in the state that you are holding the asset for $t+1$ (we are not interested in estimating the value of being in the state that we have sold the asset). Run the algorithm for 1000 iterations using a stepsize of $\alpha_{n-1} = 1/n$, and compare the results to that obtained using a constant stepsize $\alpha = 0.20$.
- 4.6** Suppose that we have an asset selling problem where we can sell at price p_t that evolves according to

$$p_{t+1} = p_t + 0.5(120 - p_t) + \hat{p}_{t+1},$$

where $p_0 = 100$ and \hat{p}_{t+1} is uniformly distributed between -10 and $+10$. Say we have to sell the asset within the first 10 time periods. Solve the problem using approximate dynamic programming. Unlike 4.5, now p_t is part of the state variable, and we have to estimate $V_t(p_t)$ (the value of holding the asset when the price is p_t). Since p_t is continuous, define your value function approximation by discretizing p_t to obtain \bar{p}_t (e.g., rounded to the nearest dollar or nearest five dollars). After training the value function for 10,000 iterations, run 1000 samples (holding the value function fixed) and determine when the model decided to sell the asset. Plot the distribution of times that the asset was held over these 1000 realizations. If δ is your discretization parameter (e.g., $\delta = 5$ means rounding to the nearest 5 dollars), compare your results for $\delta = 1, 5$, and 10 .

- 4.7** Here we are going to solve a variant of the asset-selling problem using a post-decision state variable. Suppose that we are holding a real asset and we are responding to a series of offers. Let \hat{p}_t be the t th offer, which is uniformly distributed between 500 and 600 (all prices are in thousands of dollars). Assume that each offer is independent of all prior offers. You are willing to consider up to 10 offers, and your goal is to get the highest possible price. If you have not accepted the first nine offers, you must accept the 10th offer.
- (a) Write out the decision function you would use in an approximate dynamic programming algorithm in terms of a Monte Carlo sample of the latest price and a current estimate of the value function approximation.

- (b) Use the knowledge that \hat{p}_t is uniform between 500 and 600 and derive the exact value of holding the asset after each offer.
- (c) Write out the updating equations (for the value function) you would use after solving the decision problem for the t th offer using Monte Carlo sampling.
- (d) Implement an approximate dynamic programming algorithm using *synchronous* state sampling (you sample both states at every iteration). Using 100 iterations, write out your estimates of the value of being in each state immediately after each offer.
- (e) From your value functions, infer a decision rule of the form “sell if the price is greater than \bar{p}_t .”

4.8 We are going to use approximate dynamic programming to estimate

$$F^T = \mathbb{E} \sum_{t=0}^T \gamma^t R_t,$$

where R_t is a random variable that is uniformly distributed between 0 and 100 and $\gamma = 0.7$. We assume that R_t is independent of prior history. We can think of this as a single-state Markov decision process with no decisions.

- (a) Using the fact that $\mathbb{E}R_t = 50$, give the exact value for F^{20} .
- (b) Let $\hat{v}_t^n = \sum_{t'=t}^T \gamma^{t'-t} R_{t'}(\omega^n)$, where ω^n represents the n th sample path and $R_{t'}(\omega^n)$ is the realization of the random variable $R_{t'}$ for the n th sample path. Show that $\hat{v}_t^n = R_t^n + \gamma \hat{v}_{t+1}^n$, which means that \hat{v}_0^n is a sample realization of R^T .
- (c) Propose an approximate dynamic programming algorithm to estimate F^T . Give the value function updating equation, using a stepsize $\alpha_{n-1} = 1/n$ for iteration n .
- (d) Perform 100 iterations of the approximate dynamic programming algorithm to produce an estimate of F^{20} . How does this compare to the true value?
- (e) Repeat part (d) 10 times, but now use a discount factor of 0.9. Average the results to obtain an averaged estimate. Now use these 10 calculations to produce an estimate of the standard deviation of your average.
- (f) From your answer to part (e), estimate how many iterations would be required to obtain an estimate where 95 percent confidence bounds would be within 2 percent of the true number.

4.9 Consider a batch replenishment problem where we satisfy demand in a period from the available inventory at the beginning of the period and then order more inventory at the end of the period. Define both the pre- and post-decision state variables, and write the pre-decision and post-decision forms of the transition equations.

4.10 A mutual fund has to maintain a certain amount of cash on hand for redemptions. The cost of adding funds to the cash reserve is \$250 plus \$0.005 per dollar transferred into the reserve. The demand for cash redemptions is uniformly distributed between \$5,000 and \$20,000 per day. The mutual fund manager has been using a policy of transferring in \$500,000 whenever the cash reserve goes below \$250,000. He thinks he can lower his transaction costs by transferring in \$650,000 whenever the cash reserve goes below \$250,000. However, he pays an opportunity cost of \$0.00005 per dollar per day for cash in the reserve account.

- (a) Use a spreadsheet to set up a simulation over 1000 days to estimate total transaction costs plus opportunity costs for the policy of transferring in \$500,000, assuming that you start with \$500,000 in the account. Perform this simulation 10 times, and compute the sample average and variance of these observations. Then compute the variance and standard deviation of your sample average.
- (b) Repeat (a), but now test the policy of transferring in \$650,000. Repeat this 1000 times and compute the sample average and the standard deviation of this average. Use a new set of observations of the demands for capital.
- (c) Can you conclude which policy is better at a 95 percent confidence level? Estimate the number of observations of each that you think you would need to conclude there is a difference at a 95 percent confidence level.
- (d) How does your answer to (d) change if instead of using a fresh set of observations for each policy, you use the same set of random demands for each policy?

4.11 Let \mathcal{M} be the dynamic programming “max” operator:

$$\mathcal{M}v(s) = \max_a \left(C(s, a) + \gamma \sum_{s'} \mathbb{P}(s'|s, a)v(s') \right).$$

Let

$$\bar{c} = \max_s (\mathcal{M}v(s) - v(s)),$$

and define a policy $\pi(v)$ using

$$a(s) = \arg \max_a \left(C(s, a) + \gamma \mathbb{E} \sum_{s'} \mathbb{P}(s'|s, a)v(s') \right).$$

Let v^π be the value of this policy. Show that

$$v^\pi \leq v + \frac{\bar{c}}{1 - \gamma} e,$$

where e is a vector of 1's.

- 4.12** As a broker, you sell shares of a company you think your customers will buy. Each day you start with s_{t-1} shares left over from the previous day, and then buyers for the brokerage add r_t new shares to the pool of shares you can sell that day (r_t fluctuates from day to day). Each day customer $i \in \mathcal{I}$ calls in asking to purchase q_{ti} units at price p_{ti} , but you do not confirm orders until the end of the day. Your challenge is to determine a_t , which is the number of shares you wish to sell to the market on that day. You will collect all the orders. Then at the end of the day you must choose a_t , which you will allocate among the customers willing to pay the highest price. Any shares you do not sell on one day are available for sale the next day. Let $p_t = (p_{ti})_{i \in \mathcal{I}}$ and $q_t = (q_{ti})_{i \in \mathcal{I}}$ be the vectors of price/quantity offers from all the customers on a given day.
- What is the exogenous information process for this system? What is a history for the process?
 - Give a general definition of a state variable. Is s_t a valid state variable given your definition?
 - Set up the optimality equations for this problem using the post-decision state variable. Be precise!
 - Set up the optimality equations for this problem using the pre-decision state variable. Contrast the two formulations from a computational perspective.

- 4.13** The network depicted in Figure 4.12 has an origin at node 1 and a destination at node 4. Each link has two possible costs with a probability of each outcome.
- Write out the dynamic programming recursion to solve the shortest path problem from 1 to 4. Assuming that the driver does not see the cost on a link until he arrives at the link (i.e., he will not see the cost on link (2,4) until he arrives to node 2). Solve the dynamic program and give the expected cost of getting from 1 to 4.
 - Set up and solve the dynamic program to find the expected shortest path from 1 to 4, assuming that the driver sees all the link costs before he starts the trip.

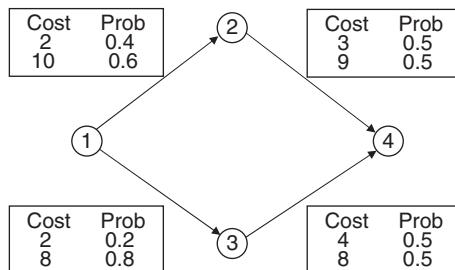


Figure 4.12 Illustration of a stochastic graph with origin node 1 and destination node 4.

- (c) Set up and solve the dynamic program to find the expected shortest path, assuming that the driver does not see the cost on a link until after he traverses the link.
- (d) Give a set of inequalities relating the results from parts (a), (b), and (c) and provide a coherent argument to support your relationships.
- 4.14** Here we are going to again solve a variant of the asset selling problem using a post-decision state variable, but this time we are going to use asynchronous state sampling (in Chapter 4 we used synchronous approximate dynamic programming). We assume that we are holding a real asset and that we are responding to a series of offers. Let \hat{p}_t be the t th offer, which is uniformly distributed between 500 and 600 (all prices are in thousands of dollars). We also assume that each offer is independent of all prior offers. You are willing to consider up to 10 offers, and your goal is to get the highest possible price. If you have not accepted the first nine offers, you must accept the 10th offer.
- (a) Implement an approximate dynamic programming algorithm using *asynchronous* state sampling, initializing all value functions to zero. Using 100 iterations, write out your estimates of the value of being in each state immediately after each offer. Use a stepsize rule of $5/(5+n-1)$. Summarize your estimate of the value of each state after each offer.
- (b) Compare your results against the estimates you obtain using synchronous sampling. Which produces better results?
- 4.15** The taxicab problem is a famous learning problem in the artificial intelligence literature. The cab enters a grid (see Figure 4.13), and at each cell, it can go up/down or left/right. The challenge is to teach it to find the exit as quickly as possible. Write an approximate dynamic programming algorithm to learn the best path to the exit where at every iteration you are allowed to loop over every cell and update the value of being in that cell.

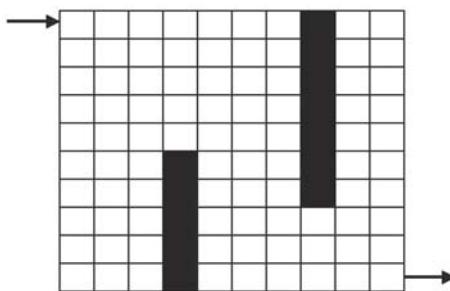


Figure 4.13 The taxi problem with barriers, where the goal is to get from the upper left hand corner to the lower right hand corner while avoiding the barriers.

- 4.16** Repeat the taxicab exercise, but now assume you can only update the value of the cell that you are in, using the following initialization strategies:
- (a) Initialize your value function with 0 everywhere.

- (b) Initialize your value function where the value of being in cell (i,j) is $10 - i$, where i is the row, and $i = 1$ is the bottom row.
- (c) Initialize your value function using the formula $(10 - i) + j$. This estimate pushes the system to look down and to the right.

- 4.17 Nomadic trucker project.** We are going to create a version of the nomadic trucker problem using randomly generated data. Suppose that our trucker may visit any of 20 cities in a set \mathcal{I} . The trucker may move loaded from i to j earning a positive reward r_{ij} as long as there is a demand \hat{D}_{tij} at time t to move a load from i to j . Alternatively, the trucker may move empty from i to j at a cost c_{ij} .

Assume that the demands \hat{D}_{tij} follow a Poisson distribution with mean λ_{ij} . Randomly generate these means by first generating a parameter ρ_i for each i , where ρ_i is uniformly distributed between 0 and 1. Then set $\lambda_{ij} = \theta\rho_i(1 - \rho_j)$, where μ is a scaling parameter (for this exercise, use $\theta = 2$). Let d_{ij} be the distance between i and j . Randomly generate distances from a uniform distribution between 100 and 1500 miles. Now let $r_{ij} = 4\rho_i d_{ij}$ and $c_{ij} = 1.2d_{ij}$. Assume that if our trucker is in location i , he can only serve demands out of location i , and that any demands not served at one point in time are lost. Further assume that it takes one day (with one time period per day) to get from location i to location j (regardless of the distance). We wish to solve our problem over a horizon $T = 21$ days.

At location i , our trucker may choose to move loaded to j if $\hat{D}_{tij} > 0$, or to move empty to j . Let $x_{tij}^L = 1$ if he moves loaded from i to j on day t , and 0 otherwise. Similarly let $x_{tij}^E = 1$ if he moves empty from i to j on day t , and 0 otherwise. Of course, $\sum_j (x_{tij}^L + x_{tij}^E) = 1$. We make the decision by solving

$$\max_x \sum_j \left((r_{ij} + \bar{V}^{n-1}(j))x_{tij}^D + (-c_{ij} + \bar{V}^{n-1}(j))x_{tij}^E \right)$$

subject to the constraint that $x_{tij}^L = 0$ if $\hat{D}_{tij} = 0$.

- (a) Say our trucker starts in city 1. Use a temporal-differencing algorithm with $\lambda = 0$ using a stepsize of $\alpha_{n-1} = a/(a + n - 1)$ with $a = 10$. Train the value functions for 1000 iterations. Then, holding the value functions constant, perform an additional 1000 simulations, and report the mean and standard deviation of the profits, as well as the number of times the trucker visits each city.
- (b) Repeat part (a), but this time insert a loop over all cities, so that for each iteration n and time t , we pretend that we are visiting every city to update the value of being in the city. Again, perform 1000 iterations to estimate the value function (this will now take 10 times longer), and then perform 1000 testing iterations.

- (c) Repeat part (a), but this time, after solving the decision problem for location i and updating the value function, randomly choose the next city to visit.
- (d) Compare your results in terms of solution quality and computational requirements (measured by how many times you solve a decision problem).

4.18 Connect-4 project. Connect-4 is a game played where players alternatively drop in chips with two colors (one for each player), as illustrated in Figure 4.14. The grid is six high by seven wide. Chips are dropped into the top of a column where they then fall to the last available spot. The goal is to get four in a row (vertically, horizontally, or diagonally) of the same color. Since this is a two-player game, you will have to train two value functions: one for the player that starts first, and one for the player who starts second. Once these are trained, you should be able to play against it.

Training a value function for board games will require playing the game a few times, and identifying specific patterns that should be captured. These patterns become your features (also known as *basis functions*). These could be indicator variables (does your opponent have three in a row?) or a numerical quantity (how many pieces are in a given row or column?). The simplest representation is a separate indicator variable for each cell, producing a value function with 42 parameters.

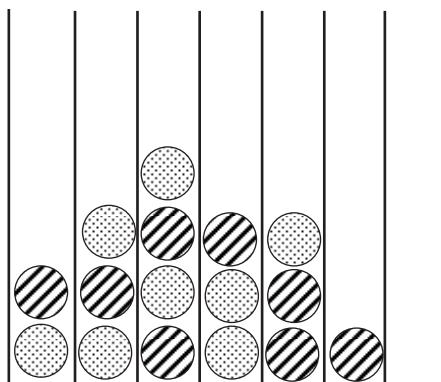


Figure 4.14 Connect-4 playing grid.

You must write software that allows a human to play your model after your value functions have been trained. Allow a person to select into which column to place his piece (no need for fancy graphics—we can keep track of the game board on a sheet of paper or using a real game piece).

C H A P T E R 5

Modeling Dynamic Programs

Perhaps one of the most important skills to develop in approximate dynamic programming is the ability to write down a model of the problem. Everyone who wants to solve a linear program learns to write out

$$\min_x c^T x$$

subject to

$$Ax = b,$$

$$x \geq 0.$$

This modeling framework allows people around the world to express their problem in a standard format.

Stochastic, dynamic problems are much richer than a linear program, and require the ability to model the flow of information and complex system dynamics. Just the same, there is a standard framework for modeling dynamic programs. We provided a taste of this framework in Chapter 2, but that chapter only hinted at the richness of the problem class.

In Chapters 2, 3, and 4 we used fairly standard notation, and we avoided discussing some important subtleties that arise in the modeling of stochastic, dynamic systems. We intentionally overlooked trying to define a state variable, which we have viewed as simply S_t , where the set of states was given by the indexed set $\mathcal{S} = \{1, 2, \dots, |\mathcal{S}|\}$. We have avoided discussions of how to properly model time or more complex information processes. This style has facilitated introducing some basic ideas in dynamic programming, but would severely limit our ability to apply these methods to real problems.

The goal of this chapter is to describe a standard modeling framework for dynamic programs, providing a vocabulary that will allow us to take on a much

Approximate Dynamic Programming: Solving the Curses of Dimensionality, Second Edition.

Warren B. Powell.

© 2011 John Wiley & Sons, Inc. Published 2011 by John Wiley & Sons, Inc.

wider set of applications. Notation is not as critical for simple problems, as long as it is precise and consistent. But what seems like benign notational decisions for a simple problem can cause unnecessary difficulties, possibly making the model completely intractable as problems become more complex. Complex problems require considerable discipline in notation because they combine the details of the original physical problem with the challenge of modeling sequential information and decision processes. The modeling of time can be particularly subtle. In addition to a desire to model problems accurately, we also need to be able to understand and exploit the structure of the problem, which can become lost in a sea of complex notation.

Good modeling begins with good notation. The choice of notation has to balance traditional style with the needs of a particular problem class. Notation is easier to learn if it is mnemonic (the letters look like what they mean) and compact (avoiding a profusion of symbols). Notation also helps bridge communities. For example, it is common in dynamic programming to refer to actions using “ a ” (where a is discrete); in control theory a decision (control) is “ u ” (which may be continuous). For high-dimensional problems it is essential to draw on the field of mathematical programming, where decisions are typically written as “ x ” and resource constraints are written in the standard form $Ax = b$. In this text many of our problems involve managing resources where we are trying to maximize or minimize an objective subject to constraints. For this reason, we adopt, as much as possible, the notation of math programming to help us bridge the fields of math programming and dynamic programming.

Sections 5.1 to 5.3 provide some foundational material. Section 5.1 begins by describing some basic guidelines for notational style. Section 5.2 addresses the critical question of modeling time, and Section 5.3 provides notation for modeling resources that we will use throughout the remainder of the book.

The general framework for modeling a dynamic program is covered in Sections 5.4 to 5.8. There are five elements to a dynamic program, consisting of the following:

1. *State variables*. These variables describe what we need to know at a point in time (Section 5.4).
2. *Decision variables*. These are the variables we control. Choosing these variables (“making decisions”) represents the central challenge of dynamic programming (Section 5.5).
3. *Exogenous information processes*. These variables describe information that arrives to us exogenously, representing the sources of randomness (Section 5.6).
4. *Transition function*. This is the function that describes how the state evolves from one point in time to another (Section 5.7).
5. *Objective function*. We are either trying to maximize a contribution function (profits, revenues, rewards, utility) or minimize a cost function. This function describes how well we are doing at a point in time (Section 5.8).

This chapter describes modeling in considerable depth, and as a result it is quite long. A number of sections are marked with a asterisk (*), indicating that these can be skipped on a first read. There is a single section marked with a double asterisks (**) which, as with all sections marked this way, is material designed for readers with more advanced training in probability and stochastic processes.

5.1 NOTATIONAL STYLE

Notation is a language: the simpler the language, the easier it is to understand the problem. As a start it is useful to adopt notational conventions to simplify the style of our presentation. For this reason we adopt the following notational conventions:

Variables. Variables are *always* a single letter. We would never use, for example, CH for “holding cost.”

Modeling time. We always use t to represent a point in time, while we use τ to represent an interval over time. When we need to represent different points in time, we might use t, t', \bar{t}, t^{\max} , and so on.

Indexing vectors. Vectors are almost always indexed in the subscript, as in x_{ij} . Since we use discrete time models throughout, an activity at time t can be viewed as an element of a vector. When there are multiple indexes, they should be ordered from outside in the general order over which they might be summed (think of the outermost index as the most detailed information). So, if x_{tij} is the flow from i to j at time t with cost c_{tij} , we might sum up the total cost using $\sum_t \sum_i \sum_j c_{tij} x_{tij}$. Dropping one or more indices creates a vector over the elements of the missing indexes to the right. So $x_t = (x_{tj})_{\forall i, \forall j}$ is the vector of all flows occurring at time t . If we write x_{ti} , this would be the vector of flows out of i at time t to all destinations j . Time, when present, is always the innermost index.

Indexing time. If we are modeling activities in discrete time, then t is an index and should be put in the subscript. So x_t would be an activity at time t , with the vector $x = (x_1, x_2, \dots, x_t, \dots, x_T)$ giving us all the activities over time. When modeling problems in continuous time, it is more common to write t as an argument, as in $x(t)$. x_t is notationally more compact (try writing a complex equation full of variables as $x(t)$ instead of x_t).

Flavors of variables. It is often the case that we need to indicate different flavors of variables, such as holding costs and order costs. These are always indicated as superscripts, where we might write c^h or c^{hold} as the holding cost. Note that while variables must be a single letter, superscripts may be words (although this should be used sparingly). We think of a variable like “ c^h ” as a single piece of notation. It is better to write c^h as the holding cost and c^p as the purchasing cost than to use h as the holding cost and p as the purchasing cost (the first approach uses a single letter c for cost, while the second approach

uses up two letters—the roman alphabet is a scarce resource). Other ways of indicating flavors is hats (\hat{x}), bars (\overline{x}), tildes (\tilde{x}), and primes (x').

Iteration counters. We place iteration counters in the superscript, and we primarily use n as our iteration counter. So x^n is our activity at iteration n . If we are using a descriptive superscript, we might write $x^{h,n}$ to represent x^h at iteration n . Sometimes algorithms require inner and outer iterations. In this case we use n to index the outer iteration and m for the inner iteration. While this will prove to be the most natural way to index iterations, there is potential for confusion where it may not be clear if the superscript n is an index (as we view it) or raising a variable to the n th power. We make one notable exception to our policy of indexing iterations in the superscript. In approximate dynamic programming we make wide use of a parameter known as a stepsize α where $0 \leq \alpha \leq 1$. We often make the stepsize vary with the iterations. However, writing α^n looks too much like raising the stepsize to the power of n . Instead, we write α_n to indicate the stepsize in iteration n . This is our only exception to this rule.

Sets. To represent sets, we use capital letters in a calligraphic font, such as \mathcal{X}, \mathcal{F} or \mathcal{I} . We generally use the lowercase roman letter as an element of a set, as in $x \in \mathcal{X}$ or $i \in \mathcal{I}$.

Exogenous information. Information that first becomes available (from outside the system) at time t is denoted using hats, for example, \hat{D}_t or \hat{p}_t . Our only exception to this rule is W_t which is our generic notation for exogenous information (since W_t always refers to exogenous information, we do not use a hat).

Statistics. Statistics computed using exogenous information are generally indicated using bars, for example \overline{x}_t or \overline{V}_t . Since these are functions of random variables, they are also random.

Index variables. Throughout, i, j, k, l, m , and n are always scalar indexes.

Of course, there are exceptions to every rule. It is extremely common in the transportation literature to model the flow of a type of resource (called a commodity and indexed by k) from i to j using x_{ij}^k . Following our convention, this should be written x_{kij} . Authors need to strike a balance between a standard notational style and existing conventions.

5.2 MODELING TIME

A survey of the literature reveals different styles toward modeling time. When using discrete time, some authors start at 1 whereas others start at zero. When solving finite horizon problems, it is popular to index time by the number of time periods remaining, rather than elapsed time. Some authors index a variable, say S_t , as being a function of information up through $t - 1$, while others assume it includes

information up through time t . t may be used to represent when a physical event actually happens, or when we first know about a physical event.

The confusion over modeling time arises in large part because there are two processes that we have to capture: the flow of information, and the flow of physical and financial resources. There are many applications of dynamic programming to deterministic problems where the flow of information does not exist (everything is known in advance). Similarly there are many models where the arrival of the information about a physical resource, and when the information takes effect in the physical system, are the same. For example, the time at which a customer physically arrives to a queue is often modeled as being the same as when the information about the customer first arrives. Similarly we often assume that we can sell a resource at a market price as soon as the price becomes known.

There is a rich collection of problems where the information process and physical process are different. A buyer may purchase an option now (an information event) to buy a commodity in the future (the physical event). Customers may call an airline (the information event) to fly on a future flight (the physical event). An electric power company has to purchase equipment now to be used one or two years in the future. All these problems represent examples of *lagged information processes* and force us to explicitly model the informational and physical events (see Section 2.2.7 for an illustration).

Notation can easily become confused when an author starts by writing down a deterministic model of a physical process and then adds uncertainty. The problem arises because the proper convention for modeling time for information processes is different than what should be used for physical processes.

We begin by establishing the relationship between discrete and continuous time. All the models in this book assume that decisions are made in discrete time (sometimes referred to as *decision epochs*). However, the flow of information, and many of the physical processes being modeled, are best viewed in continuous time. A common error is to assume that when you model a dynamic program in discrete time then all events (information events and physical events) are also occurring in discrete time (in some applications, this is the case). Throughout this text decisions are made in discrete time, while all other activities occur in continuous time.

The relationship of our discrete time approximation to the real flow of information and physical resources is depicted in Figure 5.1. Above the line, “ t ” refers to a time interval, whereas below the line, “ t ” refers to a point in time. When we are modeling information, time $t = 0$ is special; it represents “here and now” with the information that is available at the moment. The discrete time t refers to the time interval from $t - 1$ to t (illustrated in Figure 5.1a). This means that the first new information arrives during time interval 1. This notational style means that any variable indexed by t , say S_t or x_t , is assumed to have access to the information that arrived up to time t , which means up through time interval t . This property will dramatically simplify our notation in the future. For example, assume that f_t is our forecast of the demand for electricity. If \hat{D}_t is the observed demand during

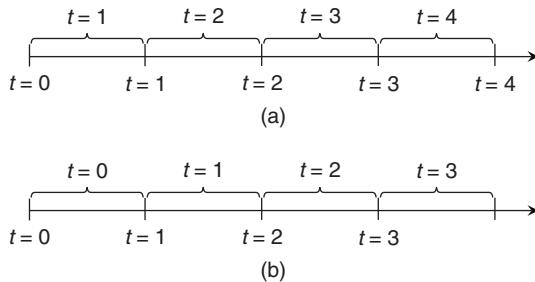


Figure 5.1 Relationship between discrete and continuous time for information processes (a) and physical processes (b).

time interval t , we would write our updating equation for the forecast using

$$f_t = (1 - \alpha)f_{t-1} + \alpha\hat{D}_t. \quad (5.1)$$

We refer to this form as the *informational representation*. Note that the forecast f_t is written as a function of the information that became available during time interval t .

When we are modeling a physical process, it is more natural to adopt a different convention (illustrated in Figure 5.1b): discrete time t refers to the time interval between t and $t + 1$. This convention arises because it is most natural in deterministic models to use time to represent when something is happening or when a resource can be used. For example, let R_t be our cash on hand that we can use during day t (implicitly, this means that we are measuring it at the beginning of the day). Let \hat{D}_t be the demand for cash during the day, and let x_t represent additional cash that we have decided to add to our balance (to be used during day t). We can model our cash on hand using

$$R_{t+1} = R_t + x_t - \hat{D}_t. \quad (5.2)$$

We refer to this form as the *actionable representation*. Note that the left-hand side is indexed by $t + 1$, while all the quantities on the right-hand side are indexed by t . This equation makes perfect sense when we interpret time t to represent when a quantity can be used. For example, many authors would write our forecasting equation (5.1) as

$$f_{t+1} = (1 - \alpha)f_t + \alpha\hat{D}_t. \quad (5.3)$$

This equation is correct if we interpret f_t as the forecast of the demand that will happen in time interval t .

A review of the literature quickly reveals that both modeling conventions are widely used. It is important to be aware of the two conventions and how to interpret them. We handle the modeling of informational and physical processes by using two time indexes, a form that we refer to as the “ (t, t') ” notation. For example,

$\hat{D}_{tt'} =$ demands that first become known during time interval t to be served during time interval t' .

$f_{tt'} =$ forecast for activities during time interval t' made using the information available up through time t .

$R_{tt'} =$ resources on hand at time t that cannot be used until time t' .

$x_{tt'} =$ decision to purchase futures at time t to be exercised during time interval t' .

For each variable, t indexes the information content (literally, when the variable is measured or computed), while t' represents the time at which the activity takes place. Each of these variables can be written as vectors, such as

$$\hat{D}_t = (\hat{D}_{tt'})_{t' \geq t},$$

$$f_t = (f_{tt'})_{t' \geq t},$$

$$x_t = (x_{tt'})_{t' \geq t},$$

$$R_t = (R_{tt'})_{t' \geq t}.$$

Note that these vectors are now written in terms of the information content. For stochastic problems, this style is the easiest and most natural. If we were modeling a deterministic problem, we would drop the first index “ t ” and model the entire problem in terms of the second index “ t' .”

Each one of these quantities is computed at the end of time interval t (i.e., with the information up through time interval t), and each represents a quantity that can be used at time t' in the future. We could adopt the convention that the first time index uses the indexing system illustrated in Figure 5.1a, while the second time index uses the system in Figure 5.1b. While this convention would allow us to easily move from a natural deterministic model to a natural stochastic model, we suspect most people will struggle with an indexing system where time interval t in the information process refers to time interval $t - 1$ in the physical process. Instead, we adopt the convention to model information in the most natural way, and live with the fact that product arriving at time t can only be used during time interval $t + 1$.

Using this convention it is instructive to interpret the special case where $t = t'$. \hat{D}_{tt} is simply demands that arrive during time interval t , where we first learn of them when they arrive. f_{tt} makes no sense because we would never forecast activities during time interval t after we have this information. R_{tt} represents resources that we know about during time interval t and that can be used during time interval t . Finally, x_{tt} is a decision to purchase resources to be used during time interval t given the information that arrived during time interval t . In financial circles this is referred to as purchasing on the spot market.

The most difficult notational decision arises when first starting to work on a problem. It is natural at this stage to simplify the problem (often the problem *appears* simple) and then choose notation that seems simple and natural. If the problem is deterministic and you are quite sure that you will never solve a stochastic version of the problem, then the actionable representation (Figure 5.1b) and equation (5.2) is going to be the most natural. Otherwise, it is best to choose the informational format. If you do not have to deal with lagged information processes (e.g., ordering at time t to be used at some time t' in the future), then you should be able to get by with a single time index, but you need to remember that x_t may mean purchasing product to be used during time interval $t + 1$.

Care has to be used when taking expectations of functions. Consider what happens when we want to know the expected costs to satisfy customer demands \hat{D}_t that arose during time interval t given the decision x_{t-1} we made at time $t - 1$. We would have to compute $\mathbb{E}\{C_t(x_{t-1}, \hat{D}_t)\}$, where the expectation is over the random variable \hat{D}_t . The function that results from taking the expectation is now a function of information up through time $t - 1$. Thus we might use the notation

$$\overline{C}_{t-1}(x_{t-1}) = \mathbb{E}\{C_t(x_{t-1}, \hat{D}_t)\}.$$

This can take a little getting used to. The costs are incurred during time interval t , but now we are indexing the function with time $t - 1$. The problem is that if we use a single time index, we are not capturing when the activity is actually happening. An alternative is to switch to a double time index, as in

$$\overline{C}_{t-1,t}(x_{t-1}) = \mathbb{E}\{C_t(x_{t-1}, \hat{D}_t)\},$$

where $\overline{C}_{t-1,t}(x_{t-1})$ is the expected costs that will be incurred during time interval t using the information known at time $t - 1$.

5.3 MODELING RESOURCES

There is a vast range of problems that can be broadly described in terms of managing “resources.” Resources can be equipment, people, money, robots, or even games such as backgammon or chess. Depending on the setting, we might use the term asset (financial applications, expensive equipment) or entity (managing a robot, playing a board game). It is very common to start with fairly simple models of these problems, but the challenge is to solve complex problems.

There are four important problem classes that we consider in this volume, each offering unique computational challenges. These can be described along two dimensions: the number of resources or entities being managed, and the complexity of the attributes of each resource or entity. We may be managing a single entity (a robot, an aircraft, an electrical power plant) or multiple entities (a fleet of aircraft, trucks or locomotives, funds in different asset classes, groups of people). The attributes of each entity or resource may be simple (the truck may be described by its location,

the money by the asset class into which it has been invested) or complex (all the characteristics of an aircraft or pilot).

The computational implications of each problem class can be quite different. Not surprisingly, different communities tend to focus on specific problem classes, making it difficult for people to make the connection between mathematical notation (which can be elegant but vague) and the characteristics of a problem. Problems involving a single, simple entity can usually be solved using the classical techniques of Markov decision processes (Chapter 3), although even here some problems can be difficult. The artificial intelligence community often works on problems involving a single, complex entity (games, e.g., Connect-4 and backgammon, moving a robot arm, or flying an aircraft). The operations research community has major subcommunities working on problems that involve multiple, simple entities (multicommodity flow problems, inventory problems), while portions of the simulation community and math programming community (for deterministic problems) will work on applications with multiple, complex entities.

In this section we describe notation that allows us to evolve from simple to complex problems in a natural way. Our goal is to develop mathematical notation that does a better job of capturing which problem class we are working on.

5.3.1 Modeling a Single, Discrete Resource

Many problems in dynamic programming involve managing a single resource such as flying an aircraft, planning a path through a network, or planning a game of chess or backgammon. These problems are distinctly different than those involving the management of fleets of vehicles, inventories of blood, or groups of people. For this reason we adopt specific notation for the single entity problem.

If we are managing a single, discrete resource, we find it useful to introduce specific notation to model the attributes of the resource. For this purpose we use

$$r_t = \text{attribute vector of the resource at time } t$$

$$= (r_{t1}, r_{t2}, \dots, r_{tN}),$$

$$\mathcal{R} = \text{set of all possible attribute vectors.}$$

Attributes might be discrete (0, 1, 2, …), continuous ($0 \leq r_i \leq 1$) or categorical ($r_i = \text{red}$). We typically assume that the number of dimensions of r_t is not too large. For example, if we are modeling the flow of product through a supply chain, the attribute vector might consist of the product type and location. If we are playing chess, the attribute vector would have 64 dimensions (the piece on each square of the board).

5.3.2 Modeling Multiple Resources

Imagine that we are modeling a fleet of unmanned aerial vehicles (UAVs), which are robotic aircraft used primarily for collecting information. We can let r_t be the attributes of a single UAV at time t , but we would like to describe the collective

attributes of a fleet of UAVs. There is more than one way to do this, but one way that will prove to be notationally convenient is to define

$$R_{tr} = \text{number of resources with attribute } r \text{ at time } t.$$

$$R_t = (R_{tr})_{r \in \mathcal{R}}.$$

R_t is known as the *resource state vector*. If r is a vector, then $|\mathcal{R}|$ may be quite large. It is not hard to create problems where R_t has hundreds of thousands of dimensions. If elements of r_t are continuous, then in theory at least, R_t is infinite-dimensional. It is important to emphasize that in such cases we would never enumerate the entire vector of elements in R_t .

We note that we can use this notation to model a single resource. Instead of letting r_t be our state vector (for the single resource), we let R_t be the state vector, where $\sum_{r \in \mathcal{R}} R_{tr} = 1$. This may seem clumsy, but it offers notational advantages we will exploit from time to time.

5.3.3 Illustration: The Nomadic Trucker

The “nomadic trucker” is a colorful illustration of a multiattribute resource that helps us clarify some of the modeling conventions being introduced in this chapter. We use this example to illustrate different issues that arise in approximate dynamic programming, leading up to the solution of large-scale resource management problems later in our presentation.

The problem of the nomadic trucker arises in what is known as the truckload trucking industry. In this industry a truck driver works much like a taxicab. A shipper will call a truckload motor carrier and ask it to send over a truck. The driver arrives, loads up the shipper’s freight, and takes it to the destination where it is unloaded. The shipper pays for the entire truck, so the carrier is not allowed to consolidate the shipment with freight from other shippers. In this sense the trucker works much like a taxicab for people. However, as we will soon see, our context of the trucking company adds an additional level of richness that offers some relevant lessons for dynamic programming.

Our trucker crosses the United States, so we can assume that his location is one of the 48 contiguous states. When he arrives in a state, he sees the customer demands for loads to move from that state to other states. There may be none, one, or several. He may choose a load to move if one is available; alternatively, he has the option of doing nothing or moving empty to another state (even if a load is available). Once he moves out of a state, all other customer demands (in the form of loads to be moved) are assumed to be picked up by other truckers and are therefore lost. He is not able to see the availability of loads out of states other than where he is located.

Although truckload motor carriers can boast fleets of over 10,000 drivers, our model focuses on the decisions made by a single driver. There are in fact thousands of trucking “companies” that consist of a single driver. In Chapter 14 we will show that the concepts we develop here form the foundation for managing the largest and

most complex versions of this problem. For now, our “nomadic trucker” represents a particularly effective way of illustrating some important concepts in dynamic programming.

A Basic Model

The simplest model of our nomadic trucker assumes that his only attribute is his location, which we assume has to be one of the 48 contiguous states. We let

$$\mathcal{I} = \text{set of “states” (locations) at which the driver can be located.}$$

We use i and j to index elements of \mathcal{I} . His attribute vector then consists of

$$r = \{i\}.$$

In addition to the attributes of the driver, we also have to capture the attributes of the loads that are available to be moved. For our basic model, loads are characterized only by where they are going. Let

$$\begin{aligned} b &= \text{vector of characteristics of a load} \\ &= \left(\begin{array}{c} b_1 \\ b_2 \end{array} \right) = \left(\begin{array}{c} \text{Origin of the load} \\ \text{Destination of the load} \end{array} \right). \end{aligned}$$

We let \mathcal{R} be the set of all possible values of the driver attribute vector r , and we let \mathcal{B} be the set of all possible load attribute vectors b .

A More Realistic Model

We need a richer set of attributes to capture some of the realism of an actual truck driver. To begin, we need to capture the fact that at a point in time, a driver may be in the process of moving from one location to another. If this is the case, we represent the attribute vector as the attribute that we expect when the driver arrives at the destination (which is the next point at which we can make a decision). We then have to include as an attribute the time at which we expect the driver to arrive.

Second, we introduce the dimension that the equipment may fail, requiring some level of repair. A failure can introduce additional delays before the driver is available.

A third important dimension covers the rules that limit how much a driver can be on the road. In the United States, drivers are governed by a set of rules set by the Department of Transportation (DOT). There are three basic limits: the amount a driver can be behind the wheel in one shift, the amount of time a driver can be “on duty” in one shift (includes time waiting), and the amount of time that a driver can be on duty over any contiguous eight-day period. As of this writing, these rules are as follows: a driver can drive at most 11 hours at a stretch, he may be on duty for at most 14 continuous hours (there are exceptions to this rule), and the driver can work at most 70 hours in any eight-day period. The last clock is reset if the driver is off-duty for 34 successive hours during any stretch (known as the “34-hour reset”).

A final dimension involves getting a driver home. In truckload trucking, drivers may be away from home for several weeks at a time. Companies work to assign drivers that get them home in a reasonable amount of time.

If we include these additional dimensions, our attribute vector grows to

$$r_t = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \\ r_8 \end{pmatrix} = \begin{pmatrix} \text{Current or future location of the driver} \\ \text{Time at which the driver is expected to arrive at his future location} \\ \text{Maintenance status of the equipment} \\ \text{Number of hours a driver has been behind the wheel during his current shift} \\ \text{Number of hours a driver has been on duty during his current shift} \\ \text{An eight-element vector giving the number of hours the driver was on duty over each of the previous eight days} \\ \text{Drivers home domicile} \\ \text{Number of days a driver has been away from home} \end{pmatrix}.$$

We note that element r_6 is actually a vector that holds the number of hours the driver was on duty during each calendar day over the last eight days.

A single attribute such as location (including the driver's domicile) might have 100 outcomes, or over 1000. The number of hours a driver has been on the road might be measured to the nearest hour, while the number of days away from home can be as large as 30 (in rare cases). Needless to say, the number of potential attribute vectors is extremely large.

5.4 THE STATES OF OUR SYSTEM

The most important quantity in a dynamic program is the state variable. The state variable captures what we need to know, but just as important it is the variable around which we construct value function approximations. Success in developing a good approximation strategy depends on a deep understanding of what is important in a state variable to capture the future behavior of a system.

5.4.1 Defining the State Variable

Surprisingly, other presentations of dynamic programming spend little time defining a state variable. Bellman's seminal text (Bellman 1957, p. 81) says: "... we have a physical system characterized at any stage by a small set of parameters, the *state variables*." In a much more modern treatment, Puterman first introduces a state variable by saying (Puterman 2005, p. 18): "At each decision epoch, the system occupies a *state*." In both cases the italics are in the original manuscript, indicating that the term "state" is being introduced. In effect both authors are saying that given a system, the state variable will be apparent from the context.

Interestingly different communities appear to interpret state variables in slightly different ways. We adopt an interpretation that is fairly common in the control theory community, but offer a definition that appears to be somewhat tighter than what typically appears in the literature. We suggest the following definition:

Definition 5.4.1 A state variable is the minimally dimensioned function of history that is necessary and sufficient to compute the decision function, the transition function, and the contribution function.

Later in the chapter we discuss the decision function (Section 5.5), the transition function (Section 5.7), and the objective function (Section 5.8). In plain English, a state variable is all the information you need to know (at time t) to model the system from time t onward. Initially it is easiest to think of the state variable in terms of the physical state of the system (the status of the pilot, the amount of money in our bank account), but ultimately it is necessary to think of it as the “state of knowledge.”

This definition provides a very quick test of the validity of a state variable. If there is a piece of data in either the decision function, the transition function, or the contribution function that is not in the state variable, then we do not have a complete state variable. Similarly, if there is information in the state variable that is never needed in any of these three functions, then we can drop it and still have a valid state variable.

We use the term “minimally dimensioned function” so that our state variable is as compact as possible. For example, we could argue that we need the entire history of events up to time t to model future dynamics. But this is not practical. As we start doing computational work, we are going to want S_t to be as compact as possible. Furthermore there are many problems where we simply do not need to know the entire history. It might be enough to know the status of all our resources at time t (the resource variable R_t). But there are examples where this is not enough.

Assume, for example, that we need to use our history to forecast the price of a stock. Our history of prices is given by $(\hat{p}_1, \hat{p}_2, \dots, \hat{p}_t)$. If we use a simple exponential smoothing model, our estimate of the mean price w_t can be computed using

$$\bar{p}_t = (1 - \alpha)\bar{p}_{t-1} + \alpha\hat{p}_t,$$

where α is a stepsize satisfying $0 \leq \alpha \leq 1$. With this forecasting mechanism we do not need to retain the history of prices, but rather only the latest estimate \bar{p}_t . As a result \bar{p}_t is called a *sufficient statistic*, which is a statistic that captures all relevant information needed to compute any additional statistics from new information. A state variable, according to our definition, is always a sufficient statistic.

Consider what happens when we switch from exponential smoothing to an N -period moving average. Our forecast of future prices is now given by

$$\bar{p}_t = \frac{1}{N} \sum_{\tau=0}^{N-1} \hat{p}_{t-\tau}.$$

Now we have to retain the N -period rolling set of prices $(\hat{p}_t, \hat{p}_{t-1}, \dots, \hat{p}_{t-N+1})$ in order to compute the price estimate in the next time period. With exponential smoothing, we could write

$$S_t = \bar{p}_t.$$

If we use the moving average, our state variable would be

$$S_t = (\hat{p}_t, \hat{p}_{t-1}, \dots, \hat{p}_{t-N+1}). \quad (5.4)$$

Many authors say that if we use the moving average model, we no longer have a proper state variable. Rather, we would have an example of a “history-dependent process” where the state variable needs to be augmented with history. By our definition of a state variable, the concept of a history-dependent process has no meaning. The state variable is simply the minimal information required to capture what is needed to model future dynamics. Needless to say, having to explicitly retain history, as we did with the moving average model, produces a much larger state variable than the exponential smoothing model.

5.4.2 The Three States of Our System

To set up our discussion, assume that we are interested in solving a relatively complex resource management problem, one that involves multiple (possibly many) different types of resources which can be modified in various ways (changing their attributes). For such a problem, it is necessary to work with three types of states:

Physical state. This is a snapshot of the status of the physical resources we are managing and their attributes. The physical state might include the amount of water in a reservoir, the price of a stock, or the location of a sensor on a network.

Information state. This encompasses the physical state as well as any other information we need to make a decision, compute the transition, or compute the objective function.

Belief/knowledge state. If the information state is what we know, the belief state (also known as the knowledge state) captures how well we know it. This concept is largely absent from most dynamic programs but arises in the setting of partially observable processes (when we cannot observe a portion of the state variable).

There are many problems in dynamic programming that involve the management of a single resource or entity (or asset—the best terminology depends on the context), such as using a computer to play backgammon, routing a single aircraft, controlling a power plant, or selling an asset. There is nothing wrong with letting S_t be the state of this entity. When we are managing multiple entities (which often puts us in the domain of “resource management”), it is often useful to distinguish between the state of a single entity, which we represent as r_t , and the state of all the entities, which we represent as R_t .

We can use S_t to be the state of a single resource (if this is all we are managing), or let $S_t = R_t$ be the state of all the resources we are managing. There are many problems where the state of the system consists only of r_t or R_t . We suggest using S_t as a generic state variable when it is not important to be specific, but it must

be used when we may wish to include other forms of information. For example, we might be managing resources (consumer products, equipment, people) to serve customer demands \hat{D}_t that become known at time t . If R_t describes the state of the resources we are managing, our state variable would consist of $S_t = (R_t, \hat{D}_t)$, where \hat{D}_t represents additional information we need to solve the problem.

Alternatively, other information might include estimates of parameters of the system (costs, speeds, times, prices). To represent this, let

$\bar{\theta}_t$ = a vector of estimates of different problem parameters at time t .

$\hat{\theta}_t$ = new information about problem parameters that arrive during time interval t .

We can think of $\bar{\theta}_t$ as the state of our information about different problem parameters at time t . We can now write a more general form of our state variable as

$$S_t = \text{information state at time } t$$

$$= (R_t, \bar{\theta}_t).$$

In Chapter 12 we will show that it is important to include not just the point estimate $\bar{\theta}_t$, but the entire distribution (or the parameters needed to characterize the distribution, e.g., the variance).

A particularly important version of this more general state variable arises in approximate dynamic programming. Recall that in Chapter 4 we used an approximation of the value function to make a decision, as in

$$x_t^n = \arg \max_{x_t \in \mathcal{X}_t^n} (C_t(R_t^n, x_t) + \bar{V}_t^{n-1}(R_t^x)) \quad (5.5)$$

Here $\bar{V}^{n-1}(\cdot)$ is an estimate of the value function if our decision takes us from resource state R_t to R_t^x , and x_t^n is the value of x_t that solves the right-hand side of (5.5). In this case our state variable would consist of

$$S_t = (R_t, \bar{V}^{n-1}).$$

The idea that the value function is part of our state variable is quite important in approximate dynamic programming.

5.4.3 The Post-decision State Variable

We can view our system as evolving through sequences of new information followed by a decision followed by new information (and so on). Although we have not yet discussed decisions, for the moment let the decisions (which may be a vector) be represented generically using a_t (we discuss our choice of notation for a decision in the next section). So a history of the process might be represented using

$$h_t = (S_0, a_0, W_1, a_1, W_2, a_2, \dots, a_{t-1}, W_t).$$

h_t contains all the information we need to make a decision a_t at time t . As we discussed before, h_t is sufficient but not necessary. We expect our state variable to capture what is needed to make a decision, allowing us to represent the history as

$$h_t = (S_0, a_0, W_1, S_1, a_1, W_2, S_2, a_2, \dots, a_{t-1}, W_t, S_t). \quad (5.6)$$

The sequence in equation (5.6) defines our state variable as occurring after new information arrives and before a decision is made. For this reason we call S_t the *pre-decision state variable*. This is the most natural place to write a state variable because the point of capturing information from the past is to make a decision.

For most problem classes we can design more effective computational strategies using the *post-decision state variable*. This is the state of the system after a decision a_t . For this reason we denote this state variable S_t^a , which produces the history

$$h_t = (S_0, a_0, S_0^a, W_1, S_1, a_1, S_1^a, W_2, S_2, a_2, S_2^a, \dots, a_{t-1}, S_{t-1}^a, W_t, S_t). \quad (5.7)$$

We again emphasize that our notation S_t^a means that this function has access to all the exogenous information up through time t , along with the decision a_t (which also has access to the information up through time t).

The examples below provide some illustrations of pre- and post-decision states.

■ EXAMPLE 5.1

A traveler is driving through a network, where the travel time on each link of the network is random. As she arrives at node i , she is allowed to see the travel times on each of the links out of node i , which we represent by $\hat{\tau}_i = (\hat{\tau}_{ij})_j$. As she arrives at node i , her pre-decision state is $S_t = (i, \hat{\tau}_i)$. Assume that she decides to move from i to k . Her post-decision state is $S_t^a = (k)$ (note that she is still at node i ; the post-decision state captures the fact that she will next be at node k , and we no longer have to include the travel times on the links out of node i). ■

■ EXAMPLE 5.2

The nomadic trucker revisited. Let $R_{tr} = 1$ if the trucker has attribute vector r at time t and 0 otherwise. Now let D_{tb} be the number of customer demands (loads of freight) of type b available to be moved at time t . The pre-decision state variable for the trucker is $S_t = (R_t, D_t)$, which tells us the state of the trucker and the demands available to be moved. Assume that once the trucker makes a decision, all the unserved demands in D_t are lost, and new demands become available at time $t + 1$. The post-decision state variable is given by $S_t^a = R_t^a$, where $R_{tr}^a = 1$ if the trucker has attribute vector r after a decision has been made. ■

■ EXAMPLE 5.3

Imagine playing backgammon where R_{ti} is the number of your pieces on the i th “point” on the backgammon board (there are 24 points on a board). The transition from S_t to S_{t+1} depends on the player’s decision a_t , the play of the opposing player, and the next roll of the dice. The post-decision state variable is simply the state of the board after a player moves but before his opponent has moved. ■

The importance of the post-decision state variable, and how to use it, depends on the problem at hand. We saw in Chapter 4 that the post-decision state variable allowed us to make decisions without having to compute the expectation within the max or min operator. Later we will see that this allows us to solve some very large scale problems.

There are three ways of finding a post-decision state variable:

Decomposing Decisions and Information

There are many problems where we can create functions $S^{M,a}(\cdot)$ and $S^{M,W}(\cdot)$ from which we can compute

$$S_t^a = S^{M,a}(S_t, a_t), \quad (5.8)$$

$$S_{t+1} = S^{M,W}(S_t^a, W_{t+1}). \quad (5.9)$$

The structure of these functions is highly problem-dependent. However, there are sometimes significant computational benefits, primarily when we face the problem of approximating the value function. Recall that the state variable captures all the information we need to make a decision, compute the transition function, and compute the contribution function. S_t^a only has to carry the information needed to compute the transition function. For some applications, S_t^a has the same dimensionality as S_t , but in many settings, S_t^a is dramatically simpler than S_t , simplifying the problem of approximating the value function.

State-Action Pairs

A very generic way of representing a post-decision state is to simply write

$$S_t^a = (S_t, a_t).$$

Figure 5.2 provides a nice illustration using our tic-tac-toe example. Figure 5.2a shows a tic-tac-toe board just before player O makes his move. Figure 5.2b shows the augmented state-action pair, where the decision (O decides to place his move in the upper right-hand corner) is distinct from the state. Finally, Figure 5.2c shows the post-decision state. For this example the pre- and post-decision state spaces are the same, while the augmented state-action pair is nine times larger.

The augmented state (S_t, a_t) is closely related to the post-decision state S_t^a (not surprising, since we can compute S_t^a deterministically from S_t and a_t). But

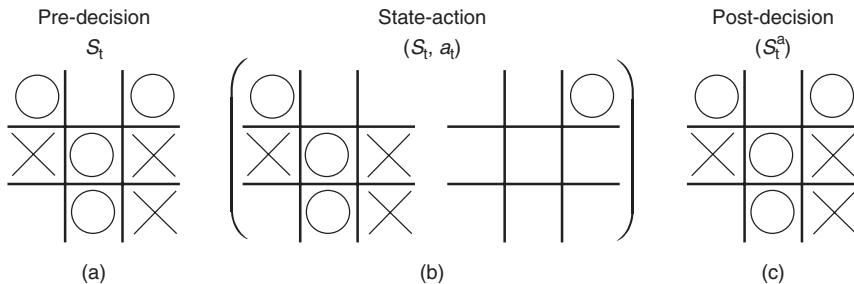


Figure 5.2 Pre-decision state, augmented state-action, and post-decision state for tic-tac-toe.

computationally the difference is significant. If \mathcal{S} is the set of possible values of S_t , and \mathcal{A} is the set of possible values of a_t , then our augmented state space has size $|\mathcal{S}| \times |\mathcal{A}|$, which is obviously much larger.

The augmented state variable is used in a popular class of algorithms known as Q -learning (introduced in Chapter 6), where the challenge is to statistically estimate Q -factors that give the value of being in state S_t and taking action a_t . The Q -factors are written $Q(S_t, a_t)$, in contrast to value functions $V_t(S_t)$, which provide the value of being in a state. This allows us to directly find the best action by solving $\min_a Q(S_t, a_t)$. This is the essence of Q -learning, but the price of this algorithmic step is that we have to estimate $Q(S_t, a_t)$ for each S_t and a_t . It is not possible to determine a_t by optimizing a function of S_t^a alone, since we generally cannot determine which action a_t brought us to S_t^a .

The Post-decision as a Point Estimate

Suppose that we have a problem where we can compute a point estimate of future information. Let $\overline{W}_{t,t+1}$ be a point estimate, computed at time t , of the outcome of W_{t+1} . If W_{t+1} is a numerical quantity, we might use $\overline{W}_{t,t+1} = \mathbb{E}(W_{t+1}|S_t)$ or $\overline{W}_{t,t+1} = 0$. W_{t+1} might be a discrete outcome such as the number of equipment failures. It may not make sense to use an expectation (we may have problems working with 0.10 failures), so in this setting W_{t+1} might be the most likely outcome. Finally, we might simply assume that W_{t+1} is empty (a form of “null” field). For example, a taxi picking up a customer may not know the destination of the customer before the customer gets in the cab. In this case, if W_{t+1} represents the destination, we might use $\overline{W}_{t,t+1} = \cdot$.

If we can create a reasonable estimate $\overline{W}_{t,t+1}$, we can compute post- and pre-decision state variables using

$$S_t^a = S^M(S_t, a_t, \overline{W}_{t,t+1}), \\ S_{t+1} = S^M(S_t, a_t, W_{t+1}).$$

Measured this way, we can think of S_t^a as a point estimate of S_{t+1} , but this does not mean that S_t^a is necessarily an approximation of the expected value of S_{t+1} .

5.4.4 Partially Observable States*

There are many applications where we are not able to observe (or measure) the state of the system precisely, as illustrated in the examples. These problems are referred to as *partially observable Markov decision processes*, and require introducing a new class of exogenous information representing the difference between the true state and the observed state.

■ EXAMPLE 5.4

A retailer may have to order inventory without being able to measure the precise current inventory. It is possible to measure sales, but theft and breakage introduce errors. ■

■ EXAMPLE 5.5

The military has to make decisions about sending out aircraft to remove important military targets that may have been damaged in previous raids. These decisions typically have to be made without knowing the precise state of the targets. ■

■ EXAMPLE 5.6

Policy makers have to decide how much to reduce CO₂ emissions, and would like to plan a policy over 200 years that strikes a balance between costs and the rise in global temperatures. Scientists cannot measure temperatures perfectly (in large part because of natural variations), and the impact of CO₂ on temperature is unknown and not directly observable. ■

To model this class of applications, let

$$\tilde{S}_t = \text{true state of the system at time } t,$$

$$\tilde{W}_t = \text{errors that arise when measuring the state } \tilde{S}_t.$$

In this context we assume that our state variable S_t is the observed state of the system. Now, our history is given by

$$h_t = (S_0, a_0, S_0^a, W_1, \tilde{S}_1, \tilde{W}_1, S_1, a_1, S_1^a, W_2, \tilde{S}_2, \tilde{W}_2, S_2, a_2, S_2^a, \dots, \\ a_{t-1}, S_{t-1}^a, W_t, \tilde{S}_t, \tilde{W}_t, S_t).$$

We view our original exogenous information W_t as representing information such as the change in price of a resource, a customer demand, equipment failures, or delays in the completion of a task. By contrast, \tilde{W}_t , which captures the difference between \tilde{S}_t and S_t , represents measurement error or the inability to observe information. Examples of measurement error might include differences between

actual and calculated inventory of product on a store shelf (e.g., due to theft or breakage), the error in estimating the location of a vehicle (due to errors in the GPS tracking system), or the difference between the actual state of a piece of machine such as an aircraft (which might have a failed part) and the observed state (we do not yet know about the failure). A different form of observational error arises when there are elements we simply cannot observe (e.g., we know the location of the vehicle but not its fuel status).

It is important to realize that there are two transition functions at work here. The “real” transition function models the dynamics of the true (unobservable) state, as in

$$\tilde{S}_{t+1} = \tilde{S}^M(\tilde{S}_t, a_t, \tilde{W}_{t+1}).$$

In practice, not only do we have the problem that we cannot perfectly measure \tilde{S}_t , we may not know the transition function $\tilde{S}^M(\cdot)$. Instead, we are working with our “engineered” transition function

$$S_{t+1} = S^M(S_t, a_t, W_{t+1}),$$

where W_{t+1} is capturing some of the effects of the observation error. When building a model where observability is an issue, it is important to try to model \tilde{S}_t , the transition function $\tilde{S}^M(\cdot)$, and the observation error \tilde{W}_t as much as possible. However, anything we cannot measure may have to be captured in our generic noise vector W_t .

5.4.5 Flat versus Factored State Representations*

It is very common in the dynamic programming literature to define a discrete set of states $\mathcal{S} = \{1, 2, \dots, |\mathcal{S}|\}$, where $s \in \mathcal{S}$ indexes a particular state. For example, consider an inventory problem where S_t is the number of items we have in inventory (where S_t is a scalar). Here our state space \mathcal{S} is the set of integers, and $s \in \mathcal{S}$ tells us how many products are in inventory. This is the style we used in Chapters 3 and 4.

Now suppose that we are managing a set of K product types. The state of our system might be given by $S_t = (S_{t1}, S_{t2}, \dots, S_{tk}, \dots)$, where S_{tk} is the number of items of type k in inventory at time t . Assume that $S_{tk} \leq M$. Our state space \mathcal{S} would consist of all possible values of S_t , which could be as large as K^M . A state $s \in \mathcal{S}$ corresponds to a particular vector of quantities $(S_{tk})_{k=1}^K$.

Modeling each state with a single scalar index is known as a flat or unstructured representation. Such a representation is simple and elegant, and produces very compact models that have been popular in the operations research community. The presentation in Chapter 3 depends on this representation. However, the use of a single index completely disguises the structure of the state variable, and often produces intractably large state spaces.

In the arena of approximate dynamic programming, it is often essential that we exploit the structure of a state variable. For this reason we generally find it

necessary to use what is known as a *factored* representation, where each factor represents a *feature* of the state variable. For example, in our inventory example we have K factors (or features). It is possible to build approximations that exploit the structure that each dimension of the state variable is a particular quantity.

Our attribute vector notation, which we use to describe a single entity, is an example of a factored representation. Each element r_i of an attribute vector represents a particular feature of the entity. The resource state variable $R_t = (R_{tr})_{r \in \mathcal{R}}$ is also a factored representation, since we explicitly capture the number of resources with a particular attribute. This is useful when we begin developing approximations for problems such as the dynamic assignment problem that we introduced in Section 2.2.10.

5.5 MODELING DECISIONS

Fundamental to dynamic programs is the characteristic that we are making decisions over time. For stochastic problems we have to model the sequencing of decisions and information, but there are many applications of dynamic programming that address deterministic problems.

Virtually all problems in approximate dynamic programming have large state spaces; it is hard to devise a problem with a small state space that is hard to solve. But problems can vary widely in terms of the nature of the decisions we have to make. In this section we introduce two notational styles (which are illustrated in Chapters 2 and 4) to help communicate the breadth of problem classes.

5.5.1 Decisions, Actions, and Controls

A survey of the literature reveals a distressing variety of words used to mean “decisions.” The classical literature on Markov decision process talks about choosing an action $a \in \mathcal{A}$ (or $a \in \mathcal{A}_s$, where \mathcal{A}_s is the set of actions available when we are in state s) or a policy (a rule for choosing an action). The optimal control community chooses a control $u \in \mathcal{U}_x$ when the system is in state x . The math programming community wants to choose a decision represented by the vector x , and the simulation community wants to apply a rule.

The proper notation for decisions will depend on the specific application. Rather than use one standard notation, we use two. Following the widely used convention in the reinforcement learning community, we use a whenever we have a finite number of discrete actions. The action space \mathcal{A} might have 10, 100, or perhaps even 1000 actions, but we are not aware of actual applications with, say, 10,000 actions or more.

There are many applications where a decision is either continuous or vector-valued. For example, in Chapters 2 and 14 we describe applications where a decision at time t involves the assignment of resources to tasks. Let $x = (x_d)_{d \in \mathcal{D}}$ be the vector of decisions, where $d \in \mathcal{D}$ is a *type* of decision, such as assigning resource i to task j , or purchasing a particular type of equipment. It is not hard to create problems with hundreds, thousands, and even tens of thousands of *dimensions* (enumerating the number of potential actions, even if x_t is discrete,

is meaningless). These high-dimensional decision vectors arise frequently in the types of resource allocation problems addressed in operations research.

There is an obvious equivalence between the “ a ” notation and the “ x ” notation. Let $d = a$ represent the decision to make a decision of “type” a . Then $x_d = 1$ corresponds to the decision to take action a . We would like to argue that one representation is better than the other, but the simple reality is that both are useful. We could simply stay with the “ a ” notation, allowing a to be continuous or a vector, as needed. However, there are many algorithms, primarily developed in the reinforcement learning community, where we really need to insist that the action space be finite, and we feel it is useful to let the use of a for action communicate this property. By contrast, when we switch to x notation, we are communicating that we can no longer enumerate the action space, and have to turn to other types of search algorithms to find the best action. In general, this means that we cannot use lookup table approximations for the value function, as we did in Chapter 4.

It is important to realize that problems with small, discrete action spaces, and problems with continuous and/or multidimensional decisions, both represent important problem classes, and both can be solved using the algorithmic framework of approximate dynamic programming.

5.5.2 Making Decisions

The challenge of dynamic programming is making decisions. In a deterministic setting we can pose the problem as one of making a sequence of decisions a_0, a_1, \dots, a_T (if we are lucky enough to have a finite horizon problem). However, in a stochastic problem the challenge is finding the best *policy* for making a decision.

It is common in the dynamic programming community to represent a policy by π . Given a state S_t , an action would be given by $a_t = \pi(S_t)$ or $x_t = \pi(S_t)$ (depending on our notation). We feel that this notation can be a bit abstract, and that it also limits our ability to specify classes of policies. We prefer instead to emphasize that a policy is in fact a *function* that returns a decision. As a result we let $a_t = A^\pi(S_t)$ (or $x_t = X^\pi(S_t)$) represent the function (or decision function) that returns an action a_t given state S_t . We use π as a superscript to capture the fact that $A^\pi(S_t)$ is one element in a family of functions that we represent by writing $\pi \in \Pi$.

We refer to $A^\pi(S_t)$ (or $X^\pi(S_t)$) and π interchangeably as a policy. A policy can be a simple function. For example, in an inventory problem, let S_t be the number of items that we have in inventory. We might use a reorder policy of the form

$$A(S_t) = \begin{cases} Q - S_t & \text{if } S_t < q, \\ 0 & \text{otherwise.} \end{cases}$$

We might let Π^{OUT} be the set of “order-up-to” decision rules. Searching for the best policy $\pi \in \Pi^{OUT}$ means searching for the best set of parameters (Q, q) . Finding the best policy within a particular set does not mean that we are finding the optimal policy, but in many cases that will be the best we can do.

Finding good policies is the challenge of dynamic programming. We provided a glimpse of how to do this in Chapter 4, but we defer to Chapter 6 a more thorough discussion of policies, which sets the tone for the rest of the book.

5.6 THE EXOGENOUS INFORMATION PROCESS

An important dimension of many of the problems that we address is the arrival of exogenous information, which changes the state of our system. While there are many important deterministic dynamic programs, exogenous information processes represent an important dimension in many problems in resource management.

5.6.1 Basic Notation for Information Processes

The central challenge of dynamic programs is dealing with one or more exogenous information processes, forcing us to make decisions before all the information is known. These might be stock prices, travel times, equipment failures, or the behavior of an opponent in a game. There might be a single exogenous process (the price at which we can sell an asset) or a number of processes. For a particular problem we might model this process using notation that is specific to the application. Here we introduce generic notation that can handle any problem.

Consider a problem of tracking the value of an asset. Assume the price evolves according to

$$p_{t+1} = p_t + \hat{p}_{t+1}.$$

Here, \hat{p}_{t+1} is an exogenous random variable representing the chance in the price during time interval $t + 1$. At time t , p_t is a number, while (at time t) p_{t+1} is random. We might assume that \hat{p}_{t+1} comes from some probability distribution. For example, we might assume that it is normally distributed with mean 0 and variance σ^2 . However, rather than work with a random variable described by some probability distribution, we are going to primarily work with sample realizations. Table 5.1 shows 10 sample realizations of a price process that starts with $p_0 = 29.80$ but then evolves according to the sample realization. Following standard mathematical convention, we index each path by the Greek letter ω (in the example below, ω runs from 1 to 10). At time $t = 0$, p_t and \hat{p}_t are random variables (for $t \geq 1$), while $p_t(\omega)$ and $\hat{p}_t(\omega)$ are *sample realizations*. We refer to the sequence

$$p_1(\omega), p_2(\omega), p_3(\omega), \dots,$$

as a *sample path* (for the prices p_t).

We are going to use “ ω ” notation throughout this book, so it is important to understand what it means. As a rule we will primarily index exogenous random variables such as \hat{p}_t using ω , as in $\hat{p}_t(\omega)$. $\hat{p}_{t'}$ is a random variable if we are sitting at a point in time $t < t'$. $\hat{p}_t(\omega)$ is not a random variable; it is a sample realization. For example, if $\omega = 5$ and $t = 2$, then $\hat{p}_t(\omega) = -0.73$. We are going to create

Table 5.1 Set of sample realizations of prices (p_t) and the changes in prices (\hat{p}_t)

Sample Path	$t = 0$	$t = 1$		$t = 2$		$t = 3$		
	ω	p_0	\hat{p}_1	p_1	\hat{p}_2	p_2	\hat{p}_3	p_3
1		29.80	2.44	32.24	1.71	33.95	-1.65	32.30
2		29.80	-1.96	27.84	0.47	28.30	1.88	30.18
3		29.80	-1.05	28.75	-0.77	27.98	1.64	29.61
4		29.80	2.35	32.15	1.43	33.58	-0.71	32.87
5		29.80	0.50	30.30	-0.56	29.74	-0.73	29.01
6		29.80	-1.82	27.98	-0.78	27.20	0.29	27.48
7		29.80	-1.63	28.17	0.00	28.17	-1.99	26.18
8		29.80	-0.47	29.33	-1.02	28.31	-1.44	26.87
9		29.80	-0.24	29.56	2.25	31.81	1.48	33.29
10		29.80	-2.45	27.35	2.06	29.41	-0.62	28.80

randomness by choosing ω at random. To make this more specific, we need to define

Ω = set of all possible sample realizations (with $\omega \in \Omega$).

$p(\omega)$ = probability that outcome ω will occur.

A word of caution is needed here. We will often work with continuous random variables, in which case we have to think of ω as being continuous. With continuous ω , we cannot say $p(\omega)$ is the “probability of outcome ω .” However, in all of our work, we will use discrete samples. For this purpose we can define

$\hat{\Omega}$ = a set of discrete sample observations of $\omega \in \Omega$.

In this case we can talk about $p(\omega)$ being the probability that we sample ω from within the set $\hat{\Omega}$.

For more complex problems we may have an entire family of random variables. In such cases it is useful to have a generic “information variable” that represents all the information that arrives during time interval t . For this purpose we define

W_t = exogenous information becoming available during interval t .

W_t may be a single variable, or a collection of variables (travel times, equipment failures, customer demands). We note that while we use the convention of putting hats on variables representing exogenous information (\hat{D}_t , \hat{p}_t), we do not use a hat for W_t because this is our only use for this variable, whereas D_t and p_t have other meanings. We always think of information as arriving in continuous time; hence W_t is the information arriving during time interval t , rather than at time t . This eliminates the ambiguity over the information available when we make a decision at time t .

The choice of notation W_t as a generic “information function” is not standard, but it is mnemonic (it looks like ω_t). We would then write $\omega_t = W_t(\omega)$ as a sample

realization of the information arriving during time interval t . This notation adds a certain elegance when we need to write decision functions and information in the same equation.

Some authors use ω to index a particular sample path, where $W_t(\omega)$ is the information that arrives during time interval t . Other authors view ω as the information itself, as in

$$\omega = (-0.24, 2.25, 1.48).$$

Obviously, both are equivalent. Sometimes it is convenient to define

$$\begin{aligned}\omega_t &= \text{information that arrives during time period } t \\ &= W_t(\omega), \\ \omega &= (\omega_1, \omega_2, \dots).\end{aligned}$$

We sometimes need to refer to the *history* of our process, for which we define

$$\begin{aligned}H_t &= \text{history of the process, consisting of all the information known through time } t, \\ &= (W_1, W_2, \dots, W_t), \\ \mathcal{H}_t &= \text{set of all possible histories through time } t, \\ &= \{H_t(\omega) | \omega \in \Omega\}, \\ h_t &= \text{a sample realization of a history,} \\ &= H_t(\omega), \\ \Omega(h_t) &= \{\omega \in \Omega | H_t(\omega) = h_t\}.\end{aligned}$$

In some applications we might refer to h_t as the state of our system, but this is usually a very clumsy representation. However, we will use the history of the process for a specific modeling and algorithmic strategy.

5.6.2 Outcomes and Scenarios

Some communities prefer to use the term *scenario* to refer to a sample realization of random information. For most purposes “outcome,” “sample path,” and “scenario” can be used interchangeably (although sample path refers to a sequence of outcomes over time). The term scenario causes problems of both notation and interpretation. First, “scenario” and “state” create an immediate competition for the interpretation of the letter “s.” Second, “scenario” is often used in the context of major events. For example, we can talk about the scenario that the Chinese might revalue their currency (a major question in the financial markets at this time). We could talk about two scenarios: (1) the Chinese hold the current relationship between the yuan and the dollar, and (2) they allow their currency to float. For

each scenario we could talk about the fluctuations in the exchange rates between all currencies.

Recognizing that different communities use “outcome” and “scenario” to mean the same thing, we suggest that we may want to reserve the ability to use both terms simultaneously. For example, we might have a set of scenarios that determine if and when the Chinese revalue their currency (but this would be a small set). We recommend denoting the set of scenarios by Ψ , with $\psi \in \Psi$ representing an individual scenario. Then, for a particular scenario ψ , we might have a set of outcomes $\omega \in \Omega$ (or $\Omega(\psi)$) representing various minor events (e.g., currency exchange rates).

■ EXAMPLE 5.7

Planning spare transformers - In the electric power sector, a certain type of transformer was invented in the 1960s. As of this writing, the industry does not really know the failure rate curve for these units (is their lifetime roughly 50 years? 60 years?). Let ψ be the scenario that the failure curve has a particular shape (e.g., where failures begin happening at a higher rate around 50 years). For a given scenario (failure rate curve), ω represents a sample realization of failures (transformers can fail at any time, although the likelihood they will fail depends on ψ). ■

■ EXAMPLE 5.8

Energy resource planning - The federal government has to determine energy sources that will replace fossil fuels. As research takes place, there are random improvements in various technologies. However, the planning of future energy technologies depends on the success of specific options, notably whether we will be able to sequester carbon underground. If this succeeds, we will be able to take advantage of vast stores of coal in the United States. Otherwise, we have to focus on options such as hydrogen and nuclear. ■

In Section 13.7 we provide a brief introduction to the field of stochastic programming where “scenario” is the preferred term to describe a set of random outcomes. Often applications of stochastic programming apply to problems where the number of outcomes (scenarios) is relatively small.

5.6.3 Lagged Information Processes*

There are many settings where the information about a new arrival comes before the new arrival itself as illustrated in the examples.

■ EXAMPLE 5.9

An airline may order an aircraft at time t and expect the order to be filled at time t' . ■

■ EXAMPLE 5.10

An orange juice products company may purchase futures for frozen concentrated orange juice at time t that can be exercised at time t' . ■

■ EXAMPLE 5.11

A programmer may start working on a piece of coding at time t with the expectation that it will be finished at time t' . ■

This concept is important enough that we offer the following term:

Definition 5.6.1 *The actionable time of a resource is the time at which a decision may be used to change its attributes (typically generating a cost or reward).*

The actionable time is simply one attribute of a resource. For example, if at time t we own a set of futures purchased in the past with exercise dates of $t + 1, t + 2, \dots, t'$, then the exercise date would be an attribute of each futures contract (the exercise dates do not need to coincide with the discrete time instances when decisions are made). When writing out a mathematical model, it is sometimes useful to introduce an index just for the actionable time (rather than having it buried as an element of the attribute vector r). As before, we let R_{tr} be the number of resources that we know about at time t with attribute vector r . The attribute might capture that the resource is not actionable until time t' in the future. If we need to represent this explicitly, we might write

$R_{t,t'r} =$ number of resources that we know about at time t that will be actionable with attribute vector r at time t' ,

$$R_{tt'} = (R_{t,t'r})_{r \in \mathcal{R}},$$

$$R_t = (R_{t,t'})_{t' \geq t}.$$

It is important to emphasize that while t is discrete (representing when decisions are made), the actionable time t' may be continuous. When this is the case, it is generally best to simply leave it as an element of the attribute vector.

5.6.4 Models of Information Processes*

Information processes come in varying degrees of complexity. Needless to say, the structure of the information process plays a major role in the models and algorithms used to solve the problem. Below we describe information processes in increasing levels of complexity.

State-Independent Processes

A large number of problems involve processes that evolve independently of the state of the system, such as wind (in an energy application), stock prices (in the context of small trading decisions), and demand for a product (assuming inventories are small relative to the market).

■ EXAMPLE 5.12

A publicly traded index fund has a price process that can be described (in discrete time) as $p_{t+1} = p_t + \sigma\delta$, where δ is normally distributed with mean μ , variance 1, and σ is the standard deviation of the change over the length of the time interval. ■

■ EXAMPLE 5.13

Requests for credit card confirmations arrive according to a Poisson process with rate λ . This means that the number of arrivals during a period of length Δt is given by a Poisson distribution with mean $\lambda\Delta t$, which is independent of the history of the system. ■

The practical challenge we typically face in these applications is that we do not know the parameters of the system. In our price process, the price may be trending upward or downward, as determined by the parameter μ . In our customer arrival process, we need to know the rate λ (which can also be a function of time).

State-Dependent Information Processes

The standard dynamic programming models allow the probability distribution of new information (e.g., the chance in price of an asset) to be a function of the state of the system (the mean change in the price might be negative if the price is high enough, positive if the price is low enough). This is a more general model than one with independent increments, where the distribution is independent of the state of the system.

■ EXAMPLE 5.14

Customers arrive at an automated teller machine according to a Poisson process. As the line grows longer, an increasing proportion decline to join the queue (a property known as *balking* in the queueing literature). The apparent arrival rate at the queue is a process that depends on the length of the queue. ■

■ EXAMPLE 5.15

A market with limited information may respond to price changes. If the price drops over the course of a day, the market may interpret the change

as a downward movement, increasing sales and putting further downward pressure on the price. Conversely, upward movement may be interpreted as a signal that people are buying the stock, encouraging more buying behavior. ■

Interestingly many models of Markov decision processes use information processes that do exhibit independent increments. For example, we could have a queueing problem where the state of the system is the number of customers in the queue. The number of arrivals may be Poisson, and the number of customers served in an increment of time is determined primarily by the length of the queue. It is possible, however, that our arrival process is a function of the length of the queue itself (see the examples for illustrations).

State-dependent information processes are more difficult to model and introduce additional parameters that must be estimated. However, from the perspective of dynamic programming, they do not introduce any fundamental complexities. As long as the distribution of outcomes is dependent purely on the state of the system, we can apply our standard models. In fact approximate dynamic programming algorithms simply need some mechanism to sample information. It does not even matter if the exogenous information depends on information that is not in the state variable (although this will introduce errors).

It is also possible that the information arriving to the system depends on its state, as depicted in the next set of examples.

■ EXAMPLE 5.16

A driver is planning a path over a transportation network. When the driver arrives at intersection i of the network, he is able to determine the transit times of each of the segments (i, j) emanating from i . Thus the transit times that are observed by the driver depend on the path taken by the driver. ■

■ EXAMPLE 5.17

A private equity manager learns information about a company only by investing in the company and becoming involved in its management. The information arriving to the manager depends on the state of his portfolio. ■

This is a different form of state-dependent information process. Normally an outcome ω is assumed to represent *all* information available to the system. A probabilist would insist that this is still the case with our driver; the fact that the *driver* does not know the transit times on all the links is simply a matter of modeling the information the driver uses. However, many will find it more natural to think of the information as depending on the state.

Action-Dependent Information Processes

Imagine that a mutual fund is trying to optimize the process of selling a large position in a company. If the mutual fund makes the decision to sell a large number of shares, the effect may be to depress the stock price because the act of selling sends a negative signal to the market. Thus the action may influence what would normally be an exogenous stochastic process.

More Complex Information Processes

Now consider the problem of modeling currency exchange rates. The change in the exchange rate between one pair of currencies is usually followed quickly by changes in others. If the Japanese yen rises relative to the US dollar, it is likely that the euro will also rise relative to it, although not necessarily proportionally. As a result we have a vector of information processes that are correlated.

In addition to correlations between information processes, we can have correlations over time. An upward push in the exchange rate between two currencies in one day is likely to be followed by similar changes for several days while the market responds to new information. Sometimes the changes reflect long-term problems in the economy of a country. Such processes may be modeled using advanced statistical models that capture correlations between processes as well as over time.

An *information model* can be thought of as a probability density function $\phi_t(\omega_t)$ that gives the density (we would say the probability of ω if it were discrete) of an outcome ω_t in time t . If the problem has independent increments, we would write the density simply as $\phi_t(\omega_t)$. If the information process is Markovian (dependent on a state variable), then we would write it as $\phi_t(\omega_t | S_{t-1})$.

In some cases with complex information models, it is possible to proceed without any model at all. We can use instead, realizations drawn from history. For example, we may take samples of changes in exchange rates from different periods in history and assume that these are representative of changes that may happen in the future. The value of using samples from history is that they capture all of the properties of the real system. This is an example of planning a system without a model of an information process.

5.6.5 Supervisory Processes*

We are often trying to control systems where we have access to a set of decisions from an exogenous source. These may be decisions from history, or they may come from a knowledgeable expert. Either way, this produces a dataset of states $(S^m)_{m=1}^n$ and decisions $(x^m)_{m=1}^n$. In some cases we can use this information to fit a statistical model that we use to try to predict the decision that would have been made given a state.

The nature of such a statistical model depends very much on the context, as illustrated in the examples.

■ EXAMPLE 5.18

Consider our nomadic trucker where we measure his state s^n (his state) and his decision a^n , which we represent in terms of the destination of his next move. We could use a historical file $(s^m, a^m)_{m=1}^n$ to build a probability distribution $\rho(s, a)$ that gives the probability that we make decision a given his state s . We can use $\rho(s, a)$ to predict decisions in the future. ■

■ EXAMPLE 5.19

A mutual fund manager adds x_t dollars in cash at the end of day t (to be used to cover withdrawals on day $t + 1$) when there are R_t dollars in cash left over at the end of the day. We can use a series of observations of x_{t_m} and R_{t_m} on days t_1, t_2, \dots, t_m to fit a model of the form $X(R) = \theta_0 + \theta_1 R + \theta_2 R^2 + \theta_3 R^3$. ■

We can use supervisory processes to statistically estimate a decision function that forms an initial policy. We can then use this policy in the context of an approximate dynamic programming algorithm to help fit value functions that can be used to improve the decision function. The supervisory process helps provide an initial policy that may not be perfect but at least is reasonable.

5.6.6 Policies in the Information Process*

The sequence of information $(\omega_1, \omega_2, \dots, \omega_t)$ is assumed to be driven by some sort of exogenous process. However, we are generally interested in quantities that are functions of both exogenous information as well as the decisions. It is useful to think of decisions as *endogenous information*. But where do the decisions come from? We now see that decisions come from policies. In fact it is useful to represent our sequence of information and decisions as

$$H_t^\pi = (S_0, X_0^\pi, W_1, S_1, X_1^\pi, W_2, S_2, X_2^\pi, \dots, X_{t-1}^\pi, W_t, S_t). \quad (5.10)$$

Now our history is characterized by a family of functions: the information variables W_t , the decision functions (policies) X_t^π , and the state variables S_t . We see that to characterize a particular history h_t , we have to specify both the sample outcome ω as well as the policy π . Thus we might write a sample realization as

$$h_t^\pi = H_t^\pi(\omega).$$

We can think of a complete history $H_\infty^\pi(\omega)$ as an outcome in an expanded probability space (if we have a finite horizon, we would denote this by $H_T^\pi(\omega)$). Let

$$\omega^\pi = H_\infty^\pi(\omega)$$

be an outcome in our expanded space, where ω^π is determined by ω and the policy π . Let Ω^π be the set of all outcomes of this expanded space. The probability of an outcome in Ω^π obviously depends on the policy we are following. Thus, computing expectations (e.g., expected costs or rewards) requires knowing the policy as well as the set of exogenous outcomes. For this reason, if we are interested, say, in the expected costs during time period t , some authors will write $E_t^\pi\{C_t(S_t, x_t)\}$ to express the dependence of the expectation on the policy. However, even if we do not explicitly index the policy, it is important to understand that we need to know how we are making decisions if we are going to compute expectations or other quantities.

5.7 THE TRANSITION FUNCTION

The next step in modeling a dynamic system is the specification of the *transition function*. This function describes how the system evolves from one state to another as a result of decisions and information. We begin our discussion of system dynamics by introducing some general mathematical notation. While useful, this generic notation does not provide much guidance into how specific problems should be modeled. We then describe how to model the dynamics of some simple problems, followed by a more general model for complex resources.

5.7.1 A General Model

The dynamics of our system are represented by a function that describes how the state evolves as new information arrives and decisions are made. The dynamics of a system can be represented in different ways. The easiest is through a simple function that works as follows:

$$S_{t+1} = S^M(S_t, X_t^\pi, W_{t+1}). \quad (5.11)$$

The function $S^M(\cdot)$ goes by different names such as “plant model” (literally, the model of a physical production plant), “plant equation,” “law of motion,” “transfer function,” “system dynamics,” “system model,” “transition law,” and “transition function.” We prefer “transition function” because it is the most descriptive. We use the notation $S^M(\cdot)$ to reflect that this is the *state* transition function, which represents a *model* of the dynamics of the system. Below we reinforce the “ M ” superscript with other modeling devices.

The arguments of the function follow standard notational conventions in the control literature (state, action, information), but different authors will follow one of two conventions for modeling time. While equation (5.11) is fairly common, many authors will write the recursion as

$$S_{t+1} = S^M(S_t, X_t^\pi, W_t). \quad (5.12)$$

If we use the form in equation (5.12), we would say “the state of the system at the beginning of time interval $t + 1$ is determined by the state at time t , plus the

decision that is made at time t and the information that arrives during time interval t ." In this representation, t indexes when we are using the information. We refer to (5.12) as the *actionable representation*, since it captures when we can act on the information. This representation is always used for deterministic models, and many authors adopt it for stochastic models as well. We prefer the form in equation (5.11) because we are measuring the information available at time t when we are about to make a decision. If we are making a decision x_t at time t , it is natural to index by t all the variables that can be measured at time t . We refer to this style as the *informational representation*.

In equation (5.11) we have written the function assuming that the *function* does not depend on time (it does depend on data that depends on time). A common notational error is to write a function, say, $S_t^M(S_t, x_t)$ as if it depends on time, when in fact the function is stationary but depends on data that depends on time. If the parameters (or structure) of the function depend on time, then we would use $S_t^M(S_t, x_t, W_{t+1})$ (or possibly $S_{t+1}^M(S_t, x_t, W_{t+1})$). If not, the transition function should be written $S^M(S_t, x_t, W_{t+1})$.

This is a very general way of representing the dynamics of a system. In many problems the information W_{t+1} arriving during time interval $t + 1$ depends on the state S_t at the end of time interval t , but is conditionally independent of all prior history given S_t . For example, a driver moving over a road network may only learn about the travel times on a link from i to j when he arrives at node i . When this is the case, we say that we have a Markov information process. When the decisions depend only on the state S_t , then we have a Markov decision process. In this case we can store the system dynamics in the form of a one-step transition matrix using

$$P(s'|s, x) = \text{probability that } S_{t+1} = s' \text{ given } S_t = s \text{ and } X_t^\pi = x,$$

P^π = matrix of elements where $P(s'|s, x)$ is the element in row s and column s' and where the decision x to be made in each state is determined by a policy π .

There is a simple relationship between the transition function and the one-step transition matrix. Let

$$1_X = \begin{cases} 1 & X \text{ is true,} \\ 0 & \text{otherwise.} \end{cases}$$

Assuming that the set of outcomes Ω is discrete, the one-step transition matrix can be computed using

$$\begin{aligned} P(s'|s, x) &= \mathbb{E}\{1_{\{s'=S^M(S_t, x, W_{t+1})\}}|S_t = s\} \\ &= \sum_{\omega_{t+1} \in \Omega_{t+1}} P(W_{t+1} = \omega_{t+1})1_{\{s'=S^M(S_t, x, W_{t+1})\}}. \end{aligned} \quad (5.13)$$

It is common in the field of Markov decision processes to assume that the one-step transition is given as data. Often it can be quickly derived (for simple

problems) using assumptions about the underlying process. For example, consider a financial asset selling problem with state variable $S_t = (R_t, p_t)$, where

$$R_t = \begin{cases} 1 & \text{we are still holding the asset,} \\ 0 & \text{asset has been sold,} \end{cases}$$

and where p_t is the price at time t . We assume the price process is described by

$$p_t = p_{t-1} + \epsilon_t,$$

where ϵ_t is a random variable with distribution

$$\epsilon_t = \begin{cases} +1 & \text{with probability 0.3,} \\ 0 & \text{with probability 0.6,} \\ -1 & \text{with probability 0.1.} \end{cases}$$

Assume that the prices are integer and range from 1 to 100. We can number our states from 0 to 100 using

$$\mathcal{S} = \{(0, -), (1, 1), (1, 2), \dots, (1, 100)\}.$$

We propose that our rule for determining when to sell the asset is of the form

$$X^\pi(R_t, p_t) = \begin{cases} \text{sell asset} & \text{if } p_t < \bar{p}, \\ \text{hold asset} & \text{if } p_t \geq \bar{p}. \end{cases}$$

Assume that $\bar{p} = 60$. A portion of the one-step transition matrix for the rows and columns corresponding to the state $(0, -)$ and $(1, 58), (1, 59), (1, 60), (1, 61), (1, 62)$ looks like

$$P^{60} = \begin{array}{c|cccccc} (0, -) & 1 & 0 & 0 & 0 & 0 & 0 \\ (1, 58) & 1 & 0 & 0 & 0 & 0 & 0 \\ (1, 59) & 1 & 0 & 0 & 0 & 0 & 0 \\ (1, 60) & 0 & 0 & 0.1 & 0.6 & 0.3 & 0 \\ (1, 61) & 0 & 0 & 0 & 0.1 & 0.6 & 0.3 \\ (1, 62) & 0 & 0 & 0 & 0 & 0.1 & 0.6 \end{array}.$$

As we saw in Chapter 3, this matrix plays a major role in the theory of Markov decision processes, although its value is more limited in practical applications. By representing the system dynamics as a one-step transition matrix, it is possible to exploit the rich theory surrounding matrices in general and Markov chains in particular.

In engineering problems it is far more natural to develop the transition function first. Given this, it may be possible to compute the one-step transition matrix exactly or to estimate it using simulation. The techniques in this book do not, in general, use the one-step transition matrix, but instead use the transition function directly. However, formulations based on the transition matrix provide a powerful foundation for proving convergence of both exact and approximate algorithms.

5.7.2 The Resource Transition Function

There are many dynamic programming problems that can be modeled in terms of managing “resources” where the state vector is denoted R_t . We use this notation when we want to specifically exclude other dimensions that might be in a state variable (e.g., the challenge of making decisions to better estimate a quantity, which was first introduced in Section 2.3). If we are using R_t as the state variable, the general representation of the transition function would be written

$$R_{t+1} = R^M(R_t, x_t, W_{t+1}).$$

Our notation is exactly analogous to the notation for a general state variable S_t , but it opens the door to other modeling dimensions. For now, we illustrate the resource transition equation using some simple applications.

Resource Acquisition I—Purchasing Resources for Immediate Use

Let R_t be the quantity of a single resource class we have available at the end of a time period, but before we have acquired new resources (for the following time period). The resource may be money available to spend on an election campaign, or the amount of oil, coal, grain, or other commodities available to satisfy a market. Let \hat{D}_t be the demand for the resource that occurs over time interval t , and let x_t be the quantity of the resource that is acquired at time t to be used during time interval $t + 1$. The transition function would be written

$$R_{t+1} = \max\{0, R_t + x_t - \hat{D}_{t+1}\}.$$

Resource Acquisition II—Purchasing Futures

Assume that we are purchasing futures at time t to be exercised at time $t' > t$. At the end of time period t , we let $R_{tt'}$ be the number of futures we are holding that can be exercised during time period t' (where $t' > t$). Now assume that we purchase $x_{tt'}$ additional futures to be used during time period t' . Our system dynamics would look like

$$R_{t+1,t'} = \begin{cases} \max\{0, (R_{tt} + R_{t,t+1}) + x_{t,t+1} - \hat{D}_{t+1}\}, & t' = t + 1, \\ R_{tt'} + x_{tt'}, & t' \geq t + 2 \end{cases}.$$

In many problems we can purchase resources on the spot market, which means we are allowed to see the actual demand before we make the decision. This decision would be represented by $x_{t+1,t+1}$, which means the amount purchased using the information that arrived during time interval $t + 1$ to be used during time interval $t + 1$ (of course, these decisions are usually the most expensive). In this case the dynamics would be written

$$R_{t+1,t'} = \begin{cases} \max\{0, (R_{tt} + R_{t,t+1}) + x_{t,t+1} + x_{t+1,t+1} - \hat{D}_{t+1}\}, & t' = t + 1, \\ R_{tt'} + x_{tt'}, & t' \geq t + 2. \end{cases}$$

Planning a Path through College

Consider a student trying to satisfy a set of course requirements (number of science courses, language courses, departmentals, etc.). Let R_{tc} be the number of courses taken that satisfy requirement c at the end of semester t . Let x_{tc} be the number of courses the student enrolled in at the end of semester t for semester $t + 1$ to satisfy requirement c . Finally, let $\hat{F}_{tc}(x_{t-1})$ be the number of courses in which the student received a failing grade during semester t given x_{t-1} . This information depends on x_{t-1} , since a student cannot fail a course that she was not enrolled in. The system dynamics would look like

$$R_{t+1,c} = R_{t,c} + x_{t,c} - \hat{F}_{t+1,c}.$$

Playing Backgammon

In backgammon there are 24 points on which pieces may sit, plus a bar when you bump your opponent's pieces off the board, and plus the state that a piece has been removed from the board (you win when you have removed all of your pieces). Let $i \in \{1, 2, \dots, 26\}$ represent these positions. Let R_{ti} be the number of your pieces on location i at time t , where $R_{ti} < 0$ means that your opponent has $|R_{ti}|$ pieces on i . Let $x_{tii'}$ be the number of pieces moved from point i to i' at time t . If $\omega_t = W_t(\omega)$ is the outcome of the roll of the dice, then the allowable decisions can be written in the general form $x_t \in \mathcal{X}_t(\omega)$, where the feasible region $\mathcal{X}_t(\omega)$ captures the rules on how many pieces can be moved given $W_t(\omega)$. For example, if we only have two pieces on a point, \mathcal{X}_t would restrict us from moving more than two pieces from this point. Let δx_t be a column vector with element i given by

$$\delta x_{ti} = \sum_{i'} x_{tii'} - \sum_{i''} x_{tii''}.$$

Now let \hat{y}_{t+1} be a variable similar to x_t representing the moves of the opponent after we have finished our moves, and let $\delta \hat{y}_{t+1}$ be a column vector similar to δx_t . The transition equation would look like

$$R_{t+1} = R_t + \delta x_t + \delta \hat{y}_{t+1}.$$

A Portfolio Problem

Let R_{tk} be the amount invested in asset $k \in \mathcal{K}$, where \mathcal{K} may be individual stocks, mutual funds, or asset classes such as bonds and money market instruments. Suppose that each month we examine our portfolio and shift money from one asset to another. Let $x_{tkk'}$ be the amount we wish to move from asset k to k' , where x_{tkk} is the amount we hold in asset k . We assume the transition is made instantly (the issue of transaction costs are not relevant here). Now let $\hat{\rho}_{t+1,k}$ be the return for asset k between t and $t + 1$. The transition equation would be given by

$$R_{t+1,k} = \hat{\rho}_{t+1,k} \left(\sum_{k' \in \mathcal{K}} x_{tk'k} \right).$$

We note that it is virtually always the case that the returns $\hat{\rho}_{tk}$ are correlated across the assets. When we use a sample realization $\hat{\rho}_{t+1}(\omega)$, we assume that these correlations are reflected in the sample realization.

5.7.3 Transition Functions for Complex Resources*

When we are managing multiple, complex resources, each of which are described by an attribute vector r . It is useful to adopt special notation to describe the evolution of the system. Recall that the state variable S_t might consist of both the resource state variable R_t as well as other information. We need notation that specifically describes the evolution of R_t separately from the other variables.

We define the *attribute transition function* that describes how a specific entity with attribute vector r is modified by a decision of type a . This is modeled using

$$r_{t+1} = r^M(r_t, a_t, W_{t+1}).$$

The function $r^M(\cdot)$ parallels the state transition function $S^M(\cdot)$, but it works at the level of a decision of type a acting on a resource of type r . It is possible that there is random information affecting the outcome of the decision. For example, in Section 5.3.3 we introduced a realistic version of our nomadic trucker where the attributes include dimensions such as the estimated time that the driver will arrive in a city (random delays can change this), and the maintenance status of the equipment (the driver may identify an equipment problem while moving).

As with our state variable, we let r_t be the attribute just before we make a decision. Although we are acting on the resource with a decision of type a , we retain our notation for the post-decision state and let r_t^a be the post-decision attribute vector. A simple example illustrates the pre- and post-decision attribute vectors for our nomadic trucker. Assume our driver has two attributes: location and the expected time at which he can be assigned to a new activity. Equation (5.14) shows pre-and post-decision attributes for a nomadic trucker. At time $t = 40$, we expect the driver to be available in St. Louis at time $t = 41.4$. At $t = 40$, we make the decision that as soon as the driver is available, we are going to assign him to a load going to Los Angeles, where we expect him (again at $t = 40$) to arrive at time $t = 65.0$. At time $t = 60$, he is still expected to be heading to Los Angeles, but we have received information that he has been delayed and now expect him to be available at time $t = 70.4$ (the delay is the new information).

$$\begin{array}{lll} t = 40 & t = 40 & t = 60 \\ \text{pre-decision} & \text{post-decision} & \text{pre-decision} \\ \left(\begin{array}{c} \text{St.Louis} \\ 41.4 \end{array} \right) & \left(\begin{array}{c} \text{LosAngeles} \\ 65.0 \end{array} \right) & \left(\begin{array}{c} \text{LosAngeles} \\ 70.4 \end{array} \right) \end{array} \quad (5.14)$$

As before, we can break down the effect of decisions and information using

$$\begin{aligned} r_t^a &= r^{M,a}(r_t, a_t), \\ r_{t+1} &= r^{M,W}(r_t^a, W_{t+1}). \end{aligned}$$

For algebraic purposes it is also useful to define the indicator function:

$$\delta_{r'}^a(r, a) = \begin{cases} 1, & r_t^a = r' = r^{M,a}(r_t, a_t), \\ 0 & \text{otherwise,} \end{cases}$$

$\Delta^a = \text{matrix with } \delta_{r'}^a(r, a) \text{ in row } r' \text{ and column } (r, a).$

The function $\delta^a(\cdot)$ (or matrix Δ^a) gives the post-decision attribute vector resulting from a decision a (in the case of Δ^a , a set of decisions represented by a_t).

It is convenient to think of acting on a single resource with an action a . If we have multiple resources (or types of resources), it also helps to let x_{tra} be the number of times we act on a resource of type r with action a , and let x_t be the vector of these decisions. We can then describe the evolution of an entire set of resources, represented by the vector R_t , using

$$R_{t+1} = R^M(R_t, x_t, W_{t+1}). \quad (5.15)$$

In this setting we would represent the random information W_{t+1} as exogenous changes to the resource vector, given by

$\hat{R}_{t+1,r} = \text{change in the number of resources with attribute vector } r \text{ due to information arriving during time interval } t + 1.$

We can now write out the transition equation using

$$R_{t+1,r'} = \sum_{r \in \mathcal{R}} \sum_{a \in \mathcal{A}} \delta_{r'}^a(r, a) x_{tra} + \hat{R}_{t+1,r'}$$

or in matrix-vector form

$$R_{t+1} = \Delta^x R_t + \hat{R}_{t+1}.$$

5.7.4 Some Special Cases*

It is important to realize when special cases offer particular modeling challenges (or opportunities). Below we list a few we have encountered.

Deterministic Resource Transition Functions

It is quite common in resource allocation problems that the attribute transition function is deterministic (the equipment never breaks down, there are no delays, resources do not randomly enter and leave the system). The uncertainty may arise not in the evolution of the resources that we are managing but in the “other information” such as customer demands (e.g., the loads that the driver might move) or our estimates of parameters (where we use our decisions to collect information on random variables with unknown distributions). If the attribute transition

function is deterministic, then we would write $r_t^a = r^M(r_t, a_t)$ and we would have that $r_{t+1} = r_t^a$. In this case we simply write

$$r_{t+1} = r^M(r_t, a_t).$$

This is an important special case because it arises quite often in practice. It does not mean the transition function is deterministic. For example, a driver moving over a network faces the decision, at node i , whether to go to node j . If he makes this decision, he will go to node j deterministically, but the cost or time over the link from i to j may be random.

Another example arises when we are managing resources (people, equipment, blood) to serve demands that arrive randomly over time. The effect of a decision acting on a resource is deterministic. The only source of randomness is in the demands that arise in each time period. Let R_t be the vector describing our resources, and let \hat{D}_t be the demands that arose during time interval t . The state variable is $S_t = (R_t, \hat{D}_t)$ where \hat{D}_t is purely exogenous. If x_t is our decision vector that determines which resources are assigned to each demand, then the resource transition function $R_{t+1} = R^M(R_t, x_t)$ would be deterministic.

Gaining and Losing Resources

In addition to the attributes of the modified resource we sometimes have to capture the fact that we may gain or lose resources in the process of completing a decision. We might define

$\rho_{t+1,r,a}$ = multiplier giving the quantity of resources with attribute vector r available after being acted on with decision a at time t .

The multiplier may depend on the information available at time t (in which case we would write it as ρ_{tra}), but it is often random and depends on information that has not yet arrived (in which case we use $\rho_{t+1,r,a}$). Illustrations of gains and losses are given in the next set of examples.

■ EXAMPLE 5.20

A corporation is holding money in an index fund with a 180-day holding period (money moved out of this fund within the period incurs a 4 percent load) and would like to transfer them into a high-yield junk bond fund. The attribute of the resource would be $r = (\text{Type}, \text{Age})$. There is a transaction cost (the cost of executing the trade) and a gain ρ , which is 1.0 for funds held more than 180 days, and 0.96 for funds held less than 180 days. ■

■ EXAMPLE 5.21

A company transporting liquefied natural gas would like to purchase 500,000 tons of liquefied natural gas in southeast Asia for consumption in North

America. Although in liquified form, the gas evaporates at a rate of 0.2 percent per day, implying $\rho = 0.998$. ■

5.8 THE OBJECTIVE FUNCTION

The final dimension of our model is the objective function. We divide this presentation from a summary of the contribution function (or cost function if we are minimizing) and the function we optimize to find the best policy.

5.8.1 The Contribution Function

We assume that we have some measure of how well we are doing. This might be a cost that we wish to minimize, a contribution or reward if we are maximizing, or a more generic utility (which we typically maximize). We assume that we are maximizing a contribution. In many problems the contribution is a deterministic function of the state and action, in which case we would write

$$C(S_t, a_t) = \text{contribution (cost if we are minimizing) earned by taking action } a_t \\ \text{while in state } S_t \text{ at time } t.$$

We often write the contribution function as $C_t(S_t, a_t)$ to emphasize that it is being measured at time t and therefore depends on information in the state variable S_t . Our contribution may be random. For example, we may invest a dollars in an asset that earns a random return ρ_{t+1} that we do not know until time $t + 1$. We may think of the contribution as $\rho_{t+1}a_t$, which means that the contribution is random. In this case we typically will write

$$\hat{C}_{t+1}(S_t, a_t, W_{t+1}) = \text{contribution at time } t \text{ from being in state } S_t, \text{ making} \\ \text{decision } a_t, \text{ which also depends on the information } W_{t+1}.$$

We emphasize that the decision a_t does not have access to the information W_{t+1} (this is where our time indexing style eliminates any ambiguity about the information content of a variable). As a result the decision a_t has to work with the expected contribution, which we write

$$C_t(S_t, a_t) = \mathbb{E}\{\hat{C}_{t+1}(S_t, a_t, W_{t+1})|S_t\}.$$

The role that W_{t+1} plays is problem-dependent, as illustrated in the examples below.

■ EXAMPLE 5.22

In asset acquisition problems we order a_t in time period t to be used to satisfy demands \hat{D}_{t+1} in the next time period. Our state variable is $S_t = R_t$ = the product on hand after demands in period t have been satisfied. We pay a cost

$c^p a_t$ in period t and receive a revenue $p \min(R_t + a_t, \hat{D}_{t+1})$ in period $t + 1$. Our total one-period contribution function is then

$$\hat{C}_{t,t+1}(R_t, a_t, \hat{D}_{t+1}) = p \min(R_t + a_t, \hat{D}_{t+1}) - c^p a_t.$$

The expected contribution is

$$C_t(S_t, a_t) = \mathbb{E}\{p \min(R_t + a_t, \hat{D}_{t+1}) - c^p a_t\}. \quad \blacksquare$$

■ EXAMPLE 5.23

Consider again the asset acquisition problem above, but this time we place our orders in period t to satisfy the known demand in period t . Our cost function contains both a fixed cost c^f (which we pay for placing an order of any size) and a variable cost c^p . The cost function would look like

$$C_t(S_t, a_t) = \begin{cases} p \min(R_t + a_t, \hat{D}_t), & a_t = 0, \\ p \min(R_t + a_t, \hat{D}_t) - c^f - c^p a_t, & a_t > 0, \end{cases}$$

Note that our contribution function no longer contains information from the next time period. If we did not incur a fixed cost c^f , then we would simply look at the demand D_t and order the quantity needed to cover demand (as a result there would never be any product left over). However, since we incur a fixed cost c^f with each order, there is a benefit to ordering enough to cover the demand now and future demands. This benefit is captured through the value function. ■

There are many resource allocation problems where the contribution of a decision can be written using

$$c_{tra} = \text{unit contribution of acting on a resource with attribute vector } r \text{ with action } a.$$

This contribution is incurred in period t using information available in period t . In this case our total contribution at time t could be written

$$C_t(S_t, a_t) = \sum_{r \in \mathcal{R}} \sum_{a \in \mathcal{A}} c_{tra} x_{tra}.$$

It is surprisingly common for us to want to work with two contributions. The common view of a contribution function is that it contains revenues and costs that we want to maximize or minimize. In many operational problems there can be a mixture of “hard dollars” and “soft dollars.” The hard dollars are our quantifiable revenues and costs. But there are often other issues that are important in an operational setting and cannot always be easily quantified. For example, if we cannot cover all of the demand, we may wish to assess a penalty for not satisfying it.

We can then manipulate this penalty to reduce the amount of unsatisfied demand. Examples of the use of soft-dollar bonuses and penalties abound in operational problems (see examples).

■ EXAMPLE 5.24

A trucking company has to pay the cost of a driver to move a load, but wants to avoid using inexperienced drivers for their high-priority accounts (yet has to accept the fact that it is sometimes necessary). An artificial penalty can be used to reduce the number of times this happens. ■

■ EXAMPLE 5.25

A charter jet company requires that in order for a pilot to land at night, he/she has to have landed a plane at night three times in the last 60 days. If the third time a pilot landed at night is at least 50 days ago, the company wants to encourage assignments of these pilots to flights with night landings so that they can maintain their status. A bonus can be assigned to encourage these assignments. ■

■ EXAMPLE 5.26

A student planning her schedule of courses has to face the possibility of failing a course, which may require taking either an extra course one semester or a summer course. She wants to plan out her course schedule as a dynamic program but use a penalty to reduce the likelihood of having to take an additional course. ■

■ EXAMPLE 5.27

An investment banker wants to plan a strategy to maximize the value of an asset and minimize the likelihood of a very poor return. She is willing to accept lower overall returns in order to achieve this goal and can do it by incorporating an additional penalty when the asset is sold at a significant loss. ■

Given the presence of these so-called soft dollars, it is useful to think of two contribution functions. We can let $C_t(S_t, a_t)$ be the hard dollars and $C_t^\pi(S_t, a_t)$ be the contribution function with the soft dollars included. The notation captures the fact that a set of soft bonuses and penalties represents a form of policy. So we can think of our policy as making decisions that maximize $C_t^\pi(S_t, a_t)$, but measure the value of the policy (in hard dollars), using $C_t(S_t, A^\pi(S_t))$.

5.8.2 Finding the Best Policy

We are now ready to optimize to find the best policy. Let $A_t^\pi(S_t)$ be a decision function (equivalent to a policy) that determines what decision we make given that we are in state S_t . Our optimization problem is to choose the best policy by choosing the best decision function from the family $(A_t^\pi(S_t))_{\pi \in \Pi}$. We wish to choose the best function that maximizes the total expected (discounted) contribution over a finite (or infinite) horizon. This would be written as

$$F_0^* = \max_{\pi \in \Pi} \mathbb{E}^\pi \left\{ \sum_{t=0}^T \gamma^t C_t^\pi(S_t, A_t^\pi(S_t)) | S_0 \right\}, \quad (5.16)$$

where γ discounts the money into time $t = 0$ values. We write the value of policy π as

$$F_0^\pi = \mathbb{E} \left\{ \sum_{t=0}^T \gamma^t C_t^\pi(S_t, A_t^\pi(S_t)) | S_0 \right\}.$$

In some communities, it is common to use an interest rate r , in which case the discount factor is

$$\gamma = \frac{1}{1+r}.$$

Important variants of this objective function are the infinite horizon problem ($T = \infty$), the undiscounted finite horizon problem ($\gamma = 1$), and the average reward, given by

$$F_0^\pi = \mathbb{E} \left\{ \lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^{T-1} C_t^\pi(S_t, A_t^\pi(S_t)) | S_0 \right\}. \quad (5.17)$$

A word of explanation is in order when we write the expectation (5.16) as $\mathbb{E}^\pi(\cdot)$. If we can pre-generate all the potential outcomes $W(\omega)$ in advance (without regard to the dynamics of the system), then we would normally write the expectation $\mathbb{E}(\cdot)$, since the exogenous events are not affected by the policy. Of course, the policy does affect the dynamics, but as long as it does not affect the random events, the expectation does not depend on the policy. However, there are many problems where the exogenous events depend on the states that we visit, and possibly the actions we take. For this reason it is safest to express the expectation as dependent on the policy.

Our optimization problem is to choose the best policy. In most practical applications we can write the optimization problem as one of choosing the best policy, or

$$F_0^* = \max_{\pi \in \Pi} F_0^\pi. \quad (5.18)$$

It might be the case that a policy is characterized by a continuous parameter (the speed of a car, the temperature of a process, the price of an asset). In theory, we could have a problem where the optimal policy corresponds to a value of a parameter being equal to infinity. It is possible that F_0^* exists but that an optimal “policy” does not exist (because it requires finding a parameter equal to infinity). While this is more of a mathematical curiosity, we handle these situations by writing the optimization problem as

$$F_0^* = \sup_{\pi \in \Pi} F_0^\pi, \quad (5.19)$$

where “sup” is the supremum operator, which finds the smallest number greater than or equal to F_0^π for any value of π . If we were minimizing, we would use “inf,” which stands for “infimum,” which is the largest value less than or equal to the value of any policy. It is common in more formal treatments to use “sup” instead of “max” or “inf” instead of “min” since these are more general. Our emphasis is on computation and approximation, where we consider only problems where a solution exists. For this reason we use “max” and “min” throughout our presentation.

The expression (5.16) contains one important but subtle assumption that will prove to be critical later and that will limit the applicability of our techniques in some problem classes. Specifically, we assume the presence of what is known as *linear, additive utility*. That is, we have added up contributions for each time period. It does not matter if the contributions are discounted or if the contribution functions are nonlinear. However, we will not be able to handle functions that look like

$$F^\pi = \mathbb{E}^\pi \left\{ \left(\sum_{t \in T} \gamma^t C_t(S_t, A_t^\pi(S_t)) \right)^2 \right\}. \quad (5.20)$$

The assumption of linear, additive utility means that the total contribution is a separable function of the contributions in each time period. While this works for many problems, it certainly does not work for all of them, as presented in the examples below.

■ EXAMPLE 5.28

We may value a policy of managing a resource using a nonlinear function of the number of times the price of a resource dropped below a certain amount. ■

■ EXAMPLE 5.29

Suppose that we have to find the route through a network where the traveler is trying to arrive at a particular point in time. The value function is a nonlinear function of the total lateness, which means that the value function is not a separable function of the delay on each link. ■

■ EXAMPLE 5.30

Consider a mutual fund manager who has to decide how much to allocate among aggressive stocks, conservative stocks, bonds, and money market instruments. Let the allocation of assets among these alternatives represent a policy π . The mutual fund manager wants to maximize long-term return, but needs to be sensitive to short-term swings (the risk). He can absorb occasional downswings, but wants to avoid sustained downswings over several time periods. Thus his value function must consider not only his return in a given time period but also how his return looks over one-year, three-year, and five-year periods. ■

In some cases these apparent instances of violations of linear, additive utility can be solved using a creatively defined state variable.

5.9 A MEASURE-THEORETIC VIEW OF INFORMATION**

For researchers interested in proving theorems or reading theoretical research articles, it is useful to have a more fundamental understanding of information.

When we work with random information processes and uncertainty, it is standard in the probability community to define a probability space, which consists of three elements. The first is the set of outcomes Ω , which is generally assumed to represent all possible outcomes of the information process (actually, Ω can include outcomes that can never happen). If these outcomes are discrete, then all we would need is the probability of each outcome $p(\omega)$.

It is nice to have a terminology that allows for continuous quantities. We want to define the probabilities of our events, but if ω is continuous, we cannot talk about the probability of an outcome ω . However, we can talk about a set of outcomes \mathcal{E} that represent some specific event (if our information is a price, the event \mathcal{E} could be all the prices that constitute the event that the price is greater than some number). In this case we can define the probability of an outcome \mathcal{E} by integrating the density function $p(\omega)$ over all ω in the event \mathcal{E} .

Probabilists handle continuous outcomes by defining a set of events \mathfrak{F} , which is literally a “set of sets” because each element in \mathfrak{F} is itself a set of outcomes in Ω . This is the reason that we resort to the script font \mathfrak{F} as opposed to our calligraphic font for sets; it is easy to read \mathcal{E} as “calligraphic E” and \mathfrak{F} as “script F.” The set \mathfrak{F} has the property that if an event \mathcal{E} is in \mathfrak{F} , then its complement $\Omega \setminus \mathcal{E}$ is in \mathfrak{F} , and the union of any two events $\mathcal{E}_X \cup \mathcal{E}_Y$ in \mathfrak{F} is also in \mathfrak{F} . \mathfrak{F} is called a “sigma-algebra” (which may be written “ σ -algebra”), and is a countable union of outcomes in Ω . An understanding of sigma-algebras is not important for computational work but can be useful in certain types of proofs, as we see in the “why does it work” sections at the end of several chapters. Sigma-algebras are without question one of the more arcane devices used by the probability community, but once they are mastered, they are a powerful theoretical tool.

Finally, it is required that we specify a probability measure denoted \mathcal{P} , which gives the probability (or density) of an outcome ω , which can then be used to compute the probability of an event in \mathfrak{F} .

We can now define a formal probability space for our exogenous information process as $(\Omega, \mathfrak{F}, \mathcal{P})$. If we wish to take an expectation of some quantity that depends on the information, say, $Ef(W_t)$, then we would sum (or integrate) over the set ω multiplied by the probability (or density) \mathcal{P} .

It is important to emphasize that ω represents *all* the information that will become available, over all time periods. As a rule we are solving a problem at time t , which means we do not have the information that will become available after time t . To handle this, we let \mathfrak{F}_t be the sigma-algebra representing events that can be created using only the information up to time t . To illustrate, consider an information process W_t consisting of a single 0 or 1 in each time period. W_t may be the information that a customer purchases a jet aircraft, or the event that an expensive component in an electrical network fails. If we look over three time periods, there are eight possible outcomes, as shown in Table 5.2.

Let $\mathcal{E}_{\{W_1\}}$ be the set of outcomes ω that satisfy some logical condition on W_1 . If we are at time $t = 1$, we only see W_1 . The event $W_1 = 0$ would be written

$$\mathcal{E}_{\{W_1=0\}} = \{\omega | W_1 = 0\} = \{1, 2, 3, 4\}.$$

The sigma-algebra \mathfrak{F}_1 would consist of the events

$$\{\mathcal{E}_{\{W_1=0\}}, \mathcal{E}_{\{W_1=1\}}, \mathcal{E}_{\{W_1 \in \{0,1\}\}}, \mathcal{E}_{\{W_1 \notin \{0,1\}\}}\}.$$

Now assume that we are at time $t = 2$ and have access to W_1 and W_2 . With this information, we are able to divide our outcomes Ω into finer subsets. Our history H_2 consists of the elementary events $\mathcal{H}_2 = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$. Let $h_2 = (0, 1)$ be an element of H_2 . The event $\mathcal{E}_{\{h_2=(0,1)\}} = \{3, 4\}$. At time $t = 1$, we could not tell the difference between outcomes 1, 2, 3, and 4; now that we are at time 2, we can differentiate between $\omega \in \{1, 2\}$ and $\omega \in \{3, 4\}$. The sigma-algebra \mathfrak{F}_2 consists of all the events $\mathcal{E}_{h_2}, h_2 \in \mathcal{H}_2$, along with all possible unions and complements.

Table 5.2 Set of demand outcomes

Outcome ω	Time Period		
	1	2	3
1	0	0	0
2	0	0	1
3	0	1	0
4	0	1	1
5	1	0	0
6	1	0	1
7	1	1	0
8	1	1	1

Another event in \mathfrak{F}_2 is $\{\omega | (W_1, W_2) = (0, 0)\} = \{1, 2\}$. A third event in \mathfrak{F}_2 is the union of these two events, which consists of $\omega = \{1, 2, 3, 4\}$ that, of course, is one of the events in \mathfrak{F}_1 . In fact every event in \mathfrak{F}_1 is an event in \mathfrak{F}_2 , but not the other way around. The reason is that the additional information from the second time period allows us to divide Ω into finer set of subsets. Since \mathfrak{F}_2 consists of all unions (and complements), we can always take the union of events, which is the same as ignoring a piece of information. By contrast, we cannot divide \mathfrak{F}_1 into a finer subsets. The extra information in \mathfrak{F}_2 allows us to filter Ω into a finer set of subsets than was possible when we only had the information through the first time period. If we are in time period 3, \mathfrak{F} will consist of each of the individual elements in Ω as well as all the unions needed to create the same events in \mathfrak{F}_2 and \mathfrak{F}_1 .

From this example we see that more information (i.e., the ability to see more elements of W_1, W_2, \dots) allows us to divide Ω into finer-grained subsets. For this reason we can always write $\mathfrak{F}_{t-1} \subseteq \mathfrak{F}_t$. \mathfrak{F}_t always consists of every event in \mathfrak{F}_{t-1} in addition to other finer events. As a result of this property, \mathfrak{F}_t is termed a *filtration*. It is because of this interpretation that the sigma-algebras are typically represented using the script letter F (which literally stands for filtration) rather the more natural letter H (which stands for history). The fancy font used to denote a sigma-algebra is used to designate that it is a set of sets (rather than just a set).

It is *always* assumed that information processes satisfy $\mathfrak{F}_{t-1} \subseteq \mathfrak{F}_t$. Interestingly this is not always the case in practice. The property that information forms a filtration requires that we never “forget” anything. In real applications this is not always true. Suppose that we are doing forecasting using a moving average. This means that our forecast f_t might be written as $f_t = (1/T) \sum_{t'=1}^T \hat{D}_{t-t'}$. Such a forecasting process “forgets” information that is older than T time periods.

There are numerous textbooks on measure theory. For a nice introduction to measure-theoretic thinking (and in particular the value of measure-theoretic thinking), see Pollard (2002).

5.10 BIBLIOGRAPHIC NOTES

Section 5.2 Figure 5.1, which describes the mapping from continuous to discrete time, was outlined for me by Erhan Cinlar.

Section 5.3 The multiattribute notation for multiple resource classes is based primarily on Simao et al. (2001).

Section 5.4 The definition of states is amazingly confused in the stochastic control literature. The first recognition of the difference between the physical state and the state of knowledge appears to be in Bellman and Kalaba (1959), which used the term “hyperstate” to refer to the state of knowledge. The control literature has long used state to represent a sufficient statistic (e.g., see Kirk, 1998), representing the information needed to model the system forward in time. For an introduction to partially observable Markov decision processes, see White (1991). An excellent description of the modeling of Markov decision processes from an AI perspective is given in Boutilier

et al. (1999), including a very nice discussion of factored representations. See also Guestrin et al. (2003) for an application of the concept of factored state spaces to a Markov decision process.

Section 5.5 Our notation for decisions represents an effort to bring together the fields of dynamic programming and math programming. We believe this notation was first used in Simao et al. (2001). For a classical treatment of decisions from the perspective of Markov decision processes, see Puterman (2005). For examples of decisions from the perspective of the optimal control community, see Kirk (1998) and Bertsekas (2005). For examples of treatments of dynamic programming in economics, see Stokey and R. E. Lucas (1989) and Chow (1997).

Section 5.6 Our representation of information follows classical styles in the probability literature (see, for example, Chung (1974)). Considerable attention has been given to the topic of supervisory control. An example includes Werbos (1992b).

PROBLEMS

- 5.1** A college student must plan what courses she takes over each of eight semesters. To graduate, she needs 34 total courses, while taking no more than five and no less than three courses in any semester. She also needs two language courses, one science course, eight departmental courses in her major and two math courses.
- (a) Formulate the state variable for this problem in the most compact way possible.
 - (b) Give the transition function for our college student assuming that she successfully passes any course she takes. You will need to introduce variables representing her decisions.
 - (c) Give the transition function for our college student, but now allow for the random outcome that she may not pass every course.
- 5.2** Suppose that we have N discrete resources to manage, where R_a is the number of resources of type $a \in \mathcal{A}$ and $N = \sum_{a \in \mathcal{A}} R_a$. Let \mathcal{R} be the set of possible values of the vector R . Show that

$$|\mathcal{R}| = \binom{N + |\mathcal{A}| - 1}{|\mathcal{A}| - 1},$$

where

$$\binom{X}{Y} = \frac{X!}{Y!(X - Y)!}$$

is the number of combinations of X items taken Y at a time.

- 5.3** A broker is working in thinly traded stocks. He must make sure that he does not buy or sell in quantities that would move the price, and he feels that if he works in quantities that are no more than 10 percent of the average sales volume, he should be safe. He tracks the average sales volume of a particular stock over time. Let \hat{v}_t be the sales volume on day t , and assume that he estimates the average demand f_t using $f_t = (1 - \alpha)f_{t-1} + \alpha\hat{v}_t$. He then uses f_t as his estimate of the sales volume for the next day. Assuming he started tracking demands on day $t = 1$, what information would constitute his state variable?
- 5.4** How would your previous answer change if our broker used a 10-day moving average to estimate his demand? That is, he would use $f_t = 0.10 \sum_{i=1}^{10} \hat{v}_{t-i+1}$ as his estimate of the demand.
- 5.5** The pharmaceutical industry spends millions managing a sales force to push the industry's latest and greatest drugs. Suppose that one of these salesmen must move between a set \mathcal{I} of customers in his district. He decides which customer to visit next only after he completes a visit. For this exercise, assume that his decision does not depend on his prior history of visits (that is, he may return to a customer he has visited previously). Let S_n be his state immediately after completing his n th visit that day.
- Assume that it takes exactly one time period to get from any customer to any other customer. Write out the definition of a state variable, and argue that his state is only his current location.
 - Assume that τ_{ij} is the (deterministic and integer) time required to move from location i to location j . What is the state of our salesman at any time t ? Be sure to consider both the possibility that he is at a location (having just finished with a customer) or between locations.
 - Assume that the travel time τ_{ij} follows a discrete uniform distribution between a_{ij} and b_{ij} (where a_{ij} and b_{ij} are integers)?
- 5.6** Consider a simple asset acquisition problem where x_t is the quantity purchased at the end of time period t to be used during time interval $t + 1$. Let D_t be the demand for the assets during time interval t . Let R_t be the pre-decision state variable (the amount on hand before you have ordered x_t) and R_t^x be the post-decision state variable.
- Write the transition function so that R_{t+1} is a function of R_t , x_t , and D_{t+1} .
 - Write the transition function so that R_t^x is a function of R_{t-1}^x , D_t , and x_t .
 - Write R_t^x as a function of R_t , and write R_{t+1} as a function of R_t^x .
- 5.7** As a buyer for an orange juice products company, you are responsible for buying futures for frozen concentrate. Let $x_{tt'}$ be the number of futures you purchase in year t that can be exercised during year t' .

- (a) What is your state variable in year t ?
 (b) Write out the transition function.
- 5.8** A classical inventory problem works as follows. Assume that our state variable R_t is the amount of product on hand at the end of time period t and that D_t is a random variable giving the demand during time interval $(t - 1, t)$ with distribution $p_d = P(D_t = d)$. The demand in time interval t must be satisfied with the product on hand at the beginning of the period. We can then order a quantity x_t at the end of period t that can be used to replenish the inventory in period $t + 1$. Give the transition function that relates R_{t+1} to R_t .
- 5.9** Many problems involve the movement of resources over networks. The definition of the state of a single resource, however, can be complicated by different assumptions for the probability distribution for the time required to traverse a link. For each example below, give the state of the resource:
- (a) You have a deterministic, static network, and you want to find the shortest path from an origin node q to a destination node r . There is a known cost c_{ij} for traversing each link (i, j) .
 - (b) Next assume that the cost c_{ij} is a random variable with an unknown distribution. Each time you traverse a link (i, j) , you observe the cost \hat{c}_{ij} , which allows you to update your estimate \bar{c}_{ij} of the mean of c_{ij} .
 - (c) Finally assume that when the traveler arrives at node i , he sees \hat{c}_{ij} for each link (i, j) out of node i .
 - (d) A taxicab is moving people in a set of cities \mathcal{C} . After dropping a passenger off at city i , the dispatcher may have to decide to reposition the cab from i to j , $(i, j) \in \mathcal{C}$. The travel time from i to j is τ_{ij} , which is a random variable with a discrete uniform distribution (that is, the probability that $\tau_{ij} = t$ is $1/T$, for $t = 1, 2, \dots, T$). Assume that the travel time is known before the trip starts.
 - (e) Same as (d), but now the travel times are random with a geometric distribution (i.e., the probability that $\tau_{ij} = t$ is $(1 - \theta)\theta^{t-1}$ for $t = 1, 2, 3, \dots$).
- 5.10** In Figure 5.3, a sailboat is making its way upwind from point A to point B. To do this, the sailboat must tack, whereby it sails generally at a 45-degree angle to the wind. The problem is that the angle of the wind tends to shift randomly over time. The skipper decides to check the angle of the wind each minute and must decide whether the boat should be on port or starboard tack. Note that the proper decision must consider the current location of the boat, which we may indicate by an (x, y) coordinate.
- (a) Formulate the problem as a dynamic program. Carefully define the state variable, decision variable, exogenous information, and the contribution function.

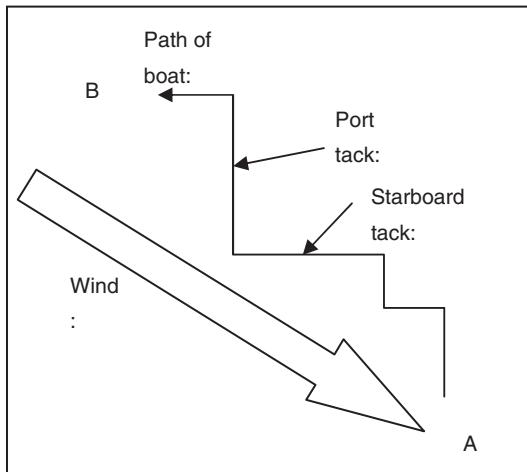


Figure 5.3

- (b) Use δ to discretize any continuous variables (in practice, you might choose different levels of discretization for each variable, but we are going to keep it simple). In terms of δ , give the size of the state space, the number of exogenous outcomes (in a single time period), and the action space. If you need an upper bound on a variable (e.g., wind speed), simply define an appropriate variable and express your answer in terms of this variable. All your answers should be expressed algebraically.
- (c) Using a maximum wind speed of 30 miles per hour and $\delta = 0.1$, compute the size of your state, outcome and action spaces.
- 5.11** Implement your model from exercise 5.10 as a Markov decision process, and solve it using the techniques of 3 (Section 3.2). Choose a value of δ that makes your program computationally reasonable (run times under 10 minutes). Let $\bar{\delta}$ be the smallest value of δ that produces a run time (for your computer) of under 10 minutes, and compare your solution (in terms of the total contribution) for $\delta = \bar{\delta}^N$ for $N = 2, 4, 8, 16$. Evaluate the quality of the solution by simulating 1000 iterations using the value functions obtained using backward dynamic programming. Plot your average contribution function as a function of δ .
- 5.12** What is the difference between the *history* of a process, and the state of a process?
- 5.13** As the purchasing manager for a major citrus juice company, you have the responsibility of maintaining sufficient reserves of oranges for sale or conversion to orange juice products. Let x_{ti} be the amount of oranges that you decide to purchase from supplier i in week t to be used in week $t + 1$.

Each week you can purchase up to \hat{q}_{ti} oranges (i.e., $x_{ti} \leq \hat{q}_{ti}$) at a price \hat{p}_{ti} from supplier $i \in \mathcal{I}$, where the price/quantity pairs $(\hat{p}_{ti}, \hat{q}_{ti})_{i \in \mathcal{I}}$ fluctuate from week to week. Let s_0 be your total initial inventory of oranges, and let D_t be the number of oranges that the company needs for production during week t (this is our demand). If we are unable to meet demand, the company must purchase additional oranges on the spot market at a spot price \hat{p}_{ti}^{spot} .

- (a) What is the exogenous stochastic process for this system?
 - (b) What are the decisions you can make to influence the system?
 - (c) What would be the state variable for your problem?
 - (d) Write out the transition equations.
 - (e) What is the one-period contribution function?
 - (f) Propose a reasonable structure for a decision rule for this problem, and call it X^π . Your decision rule should be in the form of a function that determines how much to purchase in period t .
 - (g) Carefully and precisely, write out the objective function for this problem in terms of the exogenous stochastic process. Clearly identify what you are optimizing over.
 - (h) For your decision rule, what do we mean by the space of policies?
- 5.14** Customers call in to a service center according to a (nonstationary) Poisson process. Let \mathcal{E} be the set of events representing phone calls, where t_e , $e \in \mathcal{E}$ is the time that the call is made. Each customer makes a request that will require time τ_e to complete and will pay a reward r_e to the service center. The calls are initially handled by a receptionist who determines τ_e and r_e . The service center does not have to handle all calls and obviously favors calls with a high ratio of reward per time unit required (r_e/τ_e). For this reason the company adopts a policy that the call will be refused if $(r_e/\tau_e) < \gamma$. If the call is accepted, then it is placed in a queue to wait for one of the available service representatives. Assume that the probability law driving the process is known, where we would like to find the right value of γ .
 - (a) This process is driven by an underlying exogenous stochastic process with element $\omega \in \Omega$. What is an instance of ω ?
 - (b) What are the decision epochs?
 - (c) What is the state variable for this system? What is the transition function?
 - (d) What is the action space for this system?
 - (e) Give the one-period reward function.
 - (f) Give a full statement of the objective function that defines the Markov decision process. Clearly define the probability space over which the expectation is defined, and what you are optimizing over.
- 5.15** A major oil company is looking to build up its storage tank reserves, anticipating a surge in prices. It can acquire 20 million barrels of oil, and it would like to purchase this quantity over the next 10 weeks (starting in week 1).

At the beginning of the week, the company contacts its usual sources, and each source $j \in \mathcal{J}$ is willing to provide \hat{q}_{tj} million barrels at a price \hat{p}_{tj} . The price/quantity pairs $(\hat{p}_{tj}, \hat{q}_{tj})$ fluctuate from week to week. The company would like to purchase (in discrete units of millions of barrels) x_{tj} million barrels (where x_{tj} is discrete) from source j in week $t \in \{1, 2, \dots, 10\}$. Your goal is to acquire 20 million barrels while spending the least amount possible.

- (a) What is the exogenous stochastic process for this system?
 - (b) What would be the state variable for your problem? Give an equation(s) for the system dynamics.
 - (c) Propose a structure for a decision rule for this problem and call it X^π .
 - (d) For your decision rule, what do we mean by the space of policies? Give examples of two different decision rules.
 - (e) Write out the objective function for this problem using an expectation over the exogenous stochastic process.
 - (f) You are given a budget of \$300 million to purchase the oil, but you absolutely must end up with 20 million barrels at the end of the 10 weeks. If you exceed the initial budget of \$300 million, you may get additional funds, but each additional \$1 million will cost you \$1.5 million. How does this affect your formulation of the problem?
- 5.16** You own a mutual fund where at the end of each week t you must decide whether to sell the asset or hold it for an additional week. Let \hat{r}_t be the one-week return (e.g., $\hat{r}_t = 1.05$ means the asset gained five percent in the previous week), and let p_t be the price of the asset if you were to sell it in week t (so $p_{t+1} = p_t \hat{r}_{t+1}$). We assume that the returns \hat{r}_t are independent and identically distributed. You are investing this asset for eventual use in your college education, which will occur in 100 periods. If you sell the asset at the end of time period t , then it will earn a money market rate q for each time period until time period 100, at which point you need the cash to pay for college.
- (a) What is the state space for our problem?
 - (b) What is the action space?
 - (c) What is the exogenous stochastic process that drives this system? Give a five time period example. What is the history of this process at time t ?
 - (d) You adopt a policy that you will sell if the asset falls below a price \bar{p} (which we are requiring to be independent of time). Given this policy, write out the objective function for the problem. Clearly identify exactly what you are optimizing over.

C H A P T E R 6

Policies

Perhaps one of the most widely used and poorly understood terms in dynamic programming is *policy*. A simple definition of a policy is:

Definition 6.0.1 *A policy is a rule (or function) that determines a decision given the available information in state S_t .*

The problem with the concept of a policy is that it refers to *any* method for determining an action given a state, and as a result it covers a wide range of algorithmic strategies, each suited to different problems with different computational requirements. Given the vast diversity of problems that fall under the umbrella of dynamic programming, it is important to have a strong command of the types of policies that can be used, and how to identify the best type of policy for a particular decision.

In this chapter we review a range of policies, grouped under four broad categories:

Myopic policies. These are the most elementary policies. They optimize costs/rewards now, but do not explicitly use forecasted information or any direct representation of decisions in the future. However, they may use tunable parameters to produce good behaviors over time.

Lookahead policies. These policies make decisions now. They explicitly optimize over some horizon by combining an approximation of future information, with an approximation of future actions.

Policy function approximations. These are functions which directly return an action given a state, without resorting to any form of imbedded optimization, and without directly using any forecast of future information.

Value function approximations. These policies are often referred to as *greedy* policies. They depend on an approximation of the value of being in a future state as a result of a decision made now. The impact of a decision now on the future is captured purely through a value function that depends on the state that results from a decision now.

These four broad categories span a wide range of strategies for making decisions. Further complicating the design of clean categories is the ability to form hybrids combined of strategies from two or even three of these categories. However, we feel that these four fundamental categories offer a good starting point. We note that the term “value function approximation” is widely used in approximate dynamic programming, while “policy function approximation” is a relatively new phrase, although the idea behind policy function approximations is quite old. We use this phrase to emphasize the symmetry between approximating the value of being in a state, and approximating the action we should take given a state. Recognizing this symmetry will help synthesize what is currently a disparate literature on solution strategies.

Before beginning our discussion, it is useful to briefly discuss the nature of these four strategies. Myopic policies are the simplest, since they make no explicit attempt to capture the impact of a decision now on the future. Lookahead policies capture the effect of decisions now by explicitly optimizing in the future (perhaps in an approximate way) using some approximation of information that is not known now. A difficulty is that lookahead strategies can be computationally expensive. For this reason considerable energy has been devoted to developing approximations. The most popular strategy involves developing an approximation $\bar{V}(s)$ of the value around the pre- or post-decision state. If we use the more convenient form of approximating the value function around the post-decision state, we can make decisions by solving

$$a_t = \arg \max_a (C(S_t, a) + \bar{V}(S^{M,a}(S_t, a))). \quad (6.1)$$

If we have a vector-valued decision x_t with feasible region \mathcal{X}_t , we would solve

$$x_t = \arg \max_{x \in \mathcal{X}_t} (C(S_t, x) + \bar{V}(S^{M,x}(S_t, x))). \quad (6.2)$$

Equations (6.1) and (6.2) both represent policies. For example, we can write

$$A^\pi(S_t) = \arg \max_a (C(S_t, a) + \bar{V}(S^{M,a}(S_t, a))). \quad (6.3)$$

When we characterize a policy by a value function approximation, π refers to a specific approximation in a set Π that captures the family of value function approximations. For example, we might use $\bar{V}(s) = \theta_0 + \theta_1 s + \theta_2 s^2$. In this case we might write the space of policies as Π^{VFA-LR} to represent value function approximation-based policies using linear regression. $\pi \in \Pi^{VFA-LR}$ would represent a particular setting of the vector $(\theta_0, \theta_1, \theta_2)$.

Remark Some authors like to write the decision function in the form

$$A_t^\pi(S_t) \in \arg \max_a (C(S_t, a) + \bar{V}(S^{M,a}(S_t, a))).$$

The “ \in ” captures the fact that there may be more than one optimal solution, which means the $\arg \max_a(\cdot)$ is really a set. If this is the case, then we need some rule for choosing which member of the set we are going to use. Most algorithms make this

choice at random (typically we find a best solution, and replace it only when we find a solution that is even better, rather than just as good). In rare cases we do care. However, if we use $A_t^\pi(S_t) \in \arg \max$, then we have to introduce additional logic to solve this problem. We assume that the “ $\arg \max$ ” operator includes whatever logic we are going to use to solve this problem (since this is typically what happens in practice).

Alternatively, we may try to directly estimate a decision function $A^\pi(S_t)$ (or $X^\pi(S_t)$), which does not have the embedded \max_a (or \max_x) within the function. We can think of $\bar{V}(S_t^a)$ (where $S_t^a = S^{M,a}(S_t, a)$) as an approximation of the value of being in (post-decision) state S_t^a (while following some policy in the future). At the same time we can think of $A^\pi(S_t)$ as an approximation of the decision we should make given that we are in (pre-decision) state S_t . $\bar{V}(s)$ is almost always a real-valued function. Since we have defined a to always refer to discrete actions, $A^\pi(s)$ is a discrete function, while $X^\pi(s)$ may be a scalar continuous function, or a vector-valued function (discrete or continuous). For example, we might use $X^\pi(s) = \theta_0 + \theta_1 s + \theta_2 s^2$ to compute a decision x directly from a continuous state s .

Whether we are trying to estimate a value function approximation $\bar{V}(s)$, or a decision function $A^\pi(s)$ (or $X^\pi(s)$), it is useful to identify three categories of approximation strategies:

Lookup tables. Also referred to as tabular functions, lookup tables mean that we have a discrete value $\bar{V}(s)$ (or action $A^\pi(s)$) for each discrete state s .

Parametric representations. These are explicit, analytic functions for $\bar{V}(s)$ or $A^\pi(s)$ that generally involve a vector of parameters that we typically represent by θ .

Nonparametric representations. Nonparametric representations offer a more general way of representing functions, but at a price of greater complexity.

We have a separate category for lookup tables because they represent a special case. Primarily they are the foundation of the field of Markov decision processes on which the original theory is based. Also lookup tables can be represented as a parametric model (with one parameter per state), but they share important characteristics of nonparametric models (since both focus on the local behavior of functions).

Some authors in the community take the position (sometimes implicitly) that approximate dynamic programming is equivalent to finding value function approximations. While designing policies based on value function approximations arguably remains one of the most powerful tools in the ADP toolbox, it is virtually impossible to create boundaries between a policy based on a value function approximation, and a policy based on direct search. Direct search is effectively equivalent to maximizing a value function, even if we are not directly trying to estimate the value function. Lookahead policies are little more than an alternative method for approximating the value of being in a state, and these are often used as heuristics within other algorithmic strategies.

In the remainder of this chapter, we illustrate the four major categories of policies. For the case of value function and policy function approximations, we draw

on the three major ways of representing functions. However, we defer to Chapter 8 an in-depth presentation of methods for approximating value functions. Chapter 7 discusses methods for finding policy function approximations in greater detail.

6.1 MYOPIC POLICIES

The most elementary class of policy does not explicitly use any forecasted information, or any attempt to model decisions that might be implemented in the future. In its most basic form, a myopic policy is given by

$$A^{Myopic}(S_t) = \arg \max_a C(S_t, a). \quad (6.4)$$

Myopic policies are actually widely used in resource allocation problems. Consider the dynamic assignment problem introduced in Section 2.2.10, where we are trying to assign resources (e.g., medical technicians) to tasks (equipment that needs to be installed). Let \mathcal{I} be the set of technicians, and let r_i be the attributes of the i th technician (location, skills, time away from home). Let \mathcal{J}_t be the tasks that need to be handled at time t , and let b_j be the attributes of task j , which can include the time window in which it has to be served and the reward earned from serving the task. Finally, let $x_{tij} = 1$ if we assign technician i to task j at time t and let c_{tij} be the contribution of assigning technician i to task j at time t that captures the revenue we receive from serving the task minus the cost of assigning a particular technician to the task. We might solve our problem at time t using

$$x_t = \arg \max_{x \in \mathcal{X}_t} \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}_t} c_{tij} x_{tij},$$

where the feasible region \mathcal{X}_t is defined by

$$\begin{aligned} \sum_{j \in \mathcal{J}_t} x_{tij} &\leq 1 \quad \forall i \in \mathcal{I}, \\ \sum_{i \in \mathcal{I}} x_{tij} &\leq 1 \quad \forall j \in \mathcal{J}_t, \\ x_{tij} &\geq 0. \end{aligned}$$

Note that we have no difficulty handling a high-dimensional decision vector x_t , since it requires only that we solve a simple linear program. The decision x_t ignores its effect on the future, hence its label as a myopic policy.

6.2 LOOKAHEAD POLICIES

Lookahead policies make a decision now by solving an approximation of the problem over some horizon. These policies are distinguished from value function

and policy function approximations by the explicit representation of both future information and future decisions. In this section we review tree search, roll-out heuristics, and rolling horizon procedures, all of which are widely used in engineering practice.

6.2.1 Tree Search

Tree search is a brute force strategy that enumerates all possible actions and all possible outcomes over some horizon. As a rule this can only be used for fairly short horizons, and even then only when the number of actions and outcomes is not too large. Tree search is identical to solving decision trees, as we did in Section 2.2.1.

If a full tree search can be performed for some horizon T , we would use the results to pick the action to take right now. The calculations are exact, subject to the single limitation that we are only solving the problem over T time periods rather than the full (and possibly infinite) horizon.

6.2.2 Sparse Sampling Tree Search

If the action space is not too large, we may limit the explosion due to the number of outcomes by replacing the enumeration of all outcomes with a Monte Carlo sample of outcomes. It is possible to actually bound the error from the optimal solution if enough outcomes are generated, but in practice this quickly produces the same explosion in the size of the outcome tree. Instead, this is best viewed as a potential heuristic that provides a mechanism for limiting the number of outcomes in the tree.

6.2.3 Roll-out Heuristics

For problems where the number of actions per state is fairly large, we may replace the explicit enumeration of the entire tree with some sort of heuristic policy to evaluate what might happen after we reach a state. Imagine that we are trying to evaluate if we should take action a_0 , which takes us to location A . Instead of enumerating the tree out of location A , we might follow some simple policy (perhaps a myopic policy) starting at A and extending a reasonable number of time periods in the future. The value of this trajectory is then used purely to develop an estimate of the value of being in location A . This process is repeated for each action a_0 so that we get a rough estimate of the downstream value of each of these actions. From this we choose the best value for a_0 given these rough estimates. The idea is illustrated in Figure 6.1, assuming that we are using a myopic policy to evaluate future trajectories.

The advantage of this strategy is that it is simple and fast. It provides a better approximation than using a pure myopic policy, since in this case we are using a myopic policy to provide a rough estimate of the value of being in a state. This can work quite well, but it can work very poorly. The performance is very dependent on the application.

Procedure MyopicRollOut(a)

Step 0. Initialize: Given initial state s , increment tree depth $m = m + 1$.

Step 1. Find the best myopic decision using

$$\bar{a} = \arg \max_{a \in \mathcal{A}} C(s, a).$$

Step 2. If $m < M$, do:

Step 3a. Sample the random information W given state s .

Step 3b. Compute $s' = S^M(s, \bar{a}, W)$

Step 3c. Compute

$$\bar{V}^n(s') = \text{MyopicRollOut}(s').$$

Step 3d. Find the value of the decision

$$\hat{v} = C(s, \bar{a}) + \gamma \bar{V}^n(s').$$

Else

$$\hat{v} = C(s, \bar{a}).$$

End if:

Step 4. Approximate the value of the state using:

$$\text{MyopicRollOut} = \hat{v}.$$

Figure 6.1 Approximating the value of being in a state using a myopic policy.

It is also possible to choose a decision using the current value function approximation, as in

$$\bar{a} = \arg \max_{a \in \mathcal{A}} (C(s, a) + \bar{V}^{n-1}(S^M(s, a))).$$

We note that we are most dependent on this logic in the early iterations when the estimates \bar{V}^n may be poor, but some estimate may be better than nothing.

In approximate dynamic programming (and in particular the reinforcement learning community), it is common practice to introduce an artificial discount factor. This factor is given its own name λ , reflecting the common (and occasionally annoying) habit of the reinforcement learning community to name algorithms after the notation used. If we introduce this factor, step 3c of Figure 6.1 would be replaced by

$$\hat{v} = C(s, \bar{a}) + \gamma \lambda \bar{V}^n(s').$$

In this setting γ is viewed as a fixed model parameter, while λ is a tunable algorithmic parameter. Its use in this setting reflects the fact that we are trying to estimate the value of a state s' by following some heuristic policy from s' onward. For example, imagine that we are using this idea to play a game, and we find that our

heuristic policy results in a loss. Should this loss be reflected in a poor value for state s' (because it led to a loss), or is it possible that state s' is actually a good state to be in, but the loss resulted from poor decisions made when we tried to search forward from s' ? Since we do not know the answer to this question, we discount values further into the future so that a negative outcome (or positive outcome) does not have too large of an impact on the results.

6.2.4 Rolling Horizon Procedures

Typically the reason we cannot solve our original stochastic optimization problem to optimality is the usual explosion of problem size when we try to solve a stochastic optimization problem over an infinite or even a long horizon. As a result a natural approximation strategy is to try to solve the problem over a shorter horizon. This is precisely what we were doing in Section 6.2.1 using tree search, but that method was limited to discrete (and typically small) action spaces.

Imagine that we are at time t (in state S_t), and that we can solve the problem optimally over the horizon from t to $t + H$ for a sufficiently small horizon H . We can then implement the decision a_t (or x_t if we are dealing with vectors), after which we simulate our way to a single state S_{t+1} . We then repeat the process by optimizing over the interval $t + 1$ to $t + H + 1$. In this sense we are “rolling” the horizon one time period forward. This method also goes under the names receding horizon procedure (common in operations research), or model predictive control (common in engineering control theory literature).

Our hope is that we can solve the problem over a sufficiently long horizon that the decision a_t (or x_t) “here and now” is a good one. The problem is that if we want to solve the full stochastic optimization problem, we might be limited to extremely short horizons. For this reason we typically have to resort to approximations even to solve shorter horizon problems. These strategies can be divided based on whether we are using a deterministic forecast of the future, or a stochastic one. A distinguishing feature of these methods is that we can handle vector-valued problems.

Deterministic Forecasts

A popular method in practice for building a policy for stochastic, dynamic problems is to use a point forecast of future exogenous information to create a deterministic model over a H -period horizon. In fact the term “rolling horizon procedure” (or model predictive control) is often interpreted to specifically refer to the use of deterministic forecasts. We illustrate the idea using a simple example drawn from energy systems analysis. We switch to vector-valued decisions x , since a major feature of deterministic forecasts is that it makes it possible to use standard math programming solvers that scale to large problems.

Consider the problem of managing how much energy we should store in a battery to help power a building that receives energy from the electric power grid (but at a random price) or solar panels (but with random production due to cloud cover).

We assume that we can always purchase power from the grid, but the price may be quite high.

Let

R_t = amount of energy stored in the battery at time t ,

p_t = price of electricity purchased at time t from the grid,

h_t = energy production from the solar panel at time t ,

D_t = demand for electrical power in the building at time t .

The state of our system is given by $S_t = (R_t, p_t, h_t, D_t)$. The system is controlled using

x_t^{gb} = amount of energy stored in the battery from the grid at price p_t at time t ,

x_t^{sb} = amount of energy stored in the battery from the solar panels at time t ,

x_t^{sd} = amount of energy directed from the solar panels to serve demand at time t ,

x_t^{bd} = amount of energy drawn from the battery to serve demand at time t ,

x_t^{gd} = amount of energy drawn from the grid to serve demand at time t .

We let $x_t = (x_t^{gb}, x_t^{sb}, x_t^{sd}, x_t^{bd})$ be the decision vector. The cost to meet demand at time t is given by

$$C(S_t, x_t) = p_t(x_t^{gb} + x_t^{gd}).$$

The challenge with deciding what to do right now is that we have to think about demands, prices, and solar production in the future. Demand and solar production tend to follow a daily pattern, although demand rises early in the morning more quickly than solar production, and can remain high in the evening after solar production has disappeared. Prices tend to be highest in the middle of the afternoon, and as a result we try to have energy stored in the battery to reduce our demand for expensive electricity at this time.

We can make the decision x_t by optimizing over the horizon from t to $t + H$. While $p_{t'}, h_{t'}$ and $D_{t'}$, for $t' > t$, are all random variables, we are going to replace them with forecasts $\bar{p}_{tt'}$, $\bar{h}_{tt'}$, and $\bar{D}_{tt'}$, all made with information available at time t . Since these are deterministic, we can formulate the following deterministic optimization problem:

$$\min_{x_{tt}, \dots, x_{t,t+H}} \sum_{t'=t}^{t+H} \bar{p}_{tt'} (x_{tt'}^{gb} + x_{tt'}^{gd}) \quad (6.5)$$

subject to

$$R_{t'+1} = R_{t'} + x_{tt'}^{gb} + x_{tt'}^{sb} - x_{tt'}^{bd}, \quad (6.6)$$

$$x_{tt'}^{bd} \leq R_{t'}, \quad (6.7)$$

$$x_{tt'}^{gb}, x_{tt'}^{sb}, x_{tt'}^{bd}, x_{tt'}^{sd} \geq 0. \quad (6.8)$$

We note that we are letting $x_{tt'}$ be a type of forecast of what we think we will do at time t' when we solve the optimization problem at time t . It is useful to think of this as a forecast of a decision. We project decisions over this horizon because we need to know what we would do in the future in order to know what we should do right now.

The optimization problem 6.5–6.8 is a deterministic linear program. We can solve this using, say, 1 minute increments over the next 12 hours (720 time periods) without difficulty. However, we are not interested in the values of x_{t+1}, \dots, x_{t+H} . We are only interested in x_t , which we implement at time t . As we advance the clock from t to $t + 1$, we are likely to find that the random variables have not evolved precisely according to the forecast, giving us a problem starting at time $t + 1$ (extending through $t + H + 1$) that is slightly different than what we thought would be the case.

Our deterministic model offers several advantages. First, we have no difficulty handling the property that x_t is a continuous vector. Second, the model easily handles the highly nonstationary nature of this problem, with daily cycles in demand, prices, and solar production. Third, if a weather forecast tells us that the solar production will be less than normal six hours from now, we have no difficulty taking this into account. Thus knowledge about the future does not complicate the problem.

At the same time, the inability of the model to handle uncertainty in the future introduces significant weaknesses. One problem is that we would not feel that we need to store energy in the battery in the event that solar production *might* be lower than we expect. Second, we may wish to store electricity in the battery during periods when electricity prices are lower than normal, something that would be ignored in a forecast of future prices, since we would not forecast stochastic variations from the mean.

Deterministic rolling horizon policies are widely used in operations research, where a deterministic approximation provides value in many applications. The real value of this strategy is that it opens the door for using commercial solvers that can handle vector-valued decisions. These policies are rarely seen in the classical examples of reinforcement learning that focus on small action spaces but that also focus on problems where a deterministic approximation of the future would not produce an interesting model.

Stochastic Forecasts

The tree search and roll-out heuristics were both methods that captured, in different ways, uncertainty in future exogenous events. In fact tree search is, technically speaking, a rolling horizon procedure. However, neither of these methods can handle vector-valued decisions, something that we had no difficulty handling if we were willing to assume we knew future events were known deterministically.

We can extend our deterministic, rolling horizon procedure to handle uncertainty in an approximate way. The techniques have evolved under the umbrella of a field known as *stochastic programming*, which describes the mathematics of introducing uncertainty in math programs. Introducing uncertainty in multistage optimization

problems in the presence of vector-valued decisions is intrinsically difficult, and not surprisingly, there does not exist a computationally tractable algorithm to provide an exact solution to this problem. However, some practical approximations have evolved. We illustrate the simplest strategy here.

We start by dividing the problem into two *stages*. The first stage is time t , representing our “here and now” decision. We assume we know the state S_t at time t . The second stage starts at time $t + 1$, at which point we assume we know all the information that will arrive from $t + 1$ through $t + H$. The approximation here is that the decision x_{t+1} is allowed to “see” the entire future.

Even this fairly strong approximation is not enough to allow us to solve this problem, since we still cannot compute the expectation over all possible realizations of future information exactly. Instead, we resort to Monte Carlo sampling. Let ω be a sample path representing a sample realization of all the random variables $(p_{t'}, h_{t'}, D_{t'})_{t' > t}$ from $t + 1$ until $t + H$. We use Monte Carlo methods to generate a finite sample Ω of potential outcomes. If we fix ω , then this is like solving the deterministic optimization problem above, but instead of using forecasts $\bar{p}_{tt'}$, $\bar{h}_{tt'}$, and $\bar{D}_{tt'}$, we use sample realizations $p_{tt'}(\omega)$, $h_{tt'}(\omega)$, and $D_{tt'}(\omega)$. Let $p(\omega)$ be the probability that ω happens, where this might be as simple as $p(\omega) = 1/|\Omega|$. We further note that $p_{tt}(\omega) = p_{tt}$ for all ω (the same is true for the other random variables). We are going to create decision variables $x_{tt}(\omega), \dots, x_{t,t+H}(\omega)$ for all $\omega \in \Omega$. We are going to allow $x_{tt'}(\omega)$ to depend on ω for $t' > t$, but then we are going to require that $x_{tt}(\omega)$ be the same for all ω .

We now solve the optimization problem

$$\min_{\bar{x}_{tt}, x_{tt}(\omega), \dots, x_{t,t+H}(\omega), \forall \omega} \bar{p}_{tt}(x_{tt}^{gb} + x_{tt}^{gd}) + \sum_{\omega \in \Omega} p(\omega) \sum_{t'=t+1}^{t+H} \bar{p}_{tt'}(\omega)(x_{tt'}^{gb}(\omega) + x_{tt'}^{gd}(\omega))$$

subject to the constraints

$$R_{t'+1}(\omega) = R_{t'}(\omega) + x_{tt'}^{gb}(\omega) + x_{tt'}^{sb}(\omega)x_{tt'}^{bd}(\omega), \quad (6.9)$$

$$x_{tt'}^{bd}(\omega) \leq R_{t'}(\omega), \quad (6.10)$$

$$x_{tt'}^{gb}(\omega), x_{tt'}^{sb}(\omega), x_{tt'}^{bd}(\omega), x_{tt'}^{sd}(\omega) \geq 0. \quad (6.11)$$

If we only imposed the constraints (6.9)–(6.11), then we would have a different solution $x_{tt}(\omega)$ for each scenario, which means we are allowing our decision now to see into the future. To prevent this, we impose *nonanticipativity* constraints

$$x_{tt}(\omega) - \bar{x}_{tt} = 0. \quad (6.12)$$

Equation (6.12) means that we have to choose one decision \bar{x}_{tt} regardless of the outcome ω , which means that we are not allowed to see into the future. However, we are allowing $x_{t,t+1}(\omega)$ to depend on ω , which specifies not only what happens between t and t' but also what happens for the entire future. This means that at time $t + 1$, we are allowing our decision to see future energy prices, future solar output, and future demands. However, this is only for the purpose of approximating

the problem so that we can make a decision at time t , which is not allowed to see into the future.

This formulation allows us to make a decision at time t while modeling the stochastic variability in future time periods. In the stochastic programming community, the outcomes ω are often referred to as *scenarios*. If we model 20 scenarios, then our optimization problem becomes roughly 20 times larger, so the introduction of uncertainty in the future comes at a significant computational cost. However, this formulation allows us to capture the variability in future outcomes, which can be a significant advantage over using simple forecasts. Furthermore, while the problem is certainly much larger, it is still a deterministic optimization problem that can be solved with standard commercial solvers.

There are numerous applications where there is a desire to break the second stage into additional stages, so we do not have to make the approximation that x_{t+1} sees the entire future. However, if we have $|\Omega|$ outcomes in each stage, then if we have K stages, we would be modeling $|\Omega|^K$ scenario paths. Needless to say, this grows quickly with K . Modelers overcome this by reducing the number of scenarios in the later stages, but it is hard to assess the errors that are being introduced.

Rolling Horizon with Discounting

A common objective function in dynamic programming uses discounted costs, whether over a finite or infinite horizon. If $C(S_t, x_t)$ is the contribution earned from implementing decision x_t when we are in state S_t , our objective function might be written

$$\max_{\pi} \mathbb{E} \sum_{t=0}^{\infty} \gamma^t C(S_t, X^{\pi}(S_t)),$$

where γ is a discount factor that captures, typically the time value of money. If we choose to use a rolling horizon policy (with deterministic forecasts), our policy might be written

$$X^{\pi}(S_t) = \arg \max_{x_t, \dots, x_{t+H}} \sum_{t'=t}^{t+H} \gamma^{t'-t} C(S_{t'}, x_{t'}).$$

Let's consider what a discount factor might look like if it only captures the time value of money. Imagine that we are solving an operational problem where each time step is one hour (there are many applications where time steps are in minutes or even seconds). If we assume a time value of money of 10 percent per year, then we would use $\gamma = 0.999989$ for our hourly discount factor. If we use a horizon of, say, 100 hours, then we might as well use $\gamma = 1$. Not surprisingly, it is common to introduce an artificial discount factor to reflect the fact that decisions made at $t' = t + 10$ should not carry the same weight as the decision x_t that we are going to implement right now. After all, $x_{t'}$ for $t' > t$ is only a forecast of a decision that might be implemented.

In our presentation of roll-out heuristics, we introduced an artificial discount factor λ to reduce the influence of poor decisions backward through time. In our rolling horizon model, we are actually making optimal decisions, but only for a deterministic approximation (or perhaps an approximation of a stochastic model). When we introduce this new discount factor, our rolling horizon policy would be written

$$X^\pi(S_t) = \arg \max_{x_t, \dots, x_{t+H}} \sum_{t'=t}^{t+H} \gamma^{t'-t} \lambda^{t'-t} C(S_{t'}, x_{t'}).$$

In this setting, γ plays the role of the original discount factor (which may be equal to 1.0, especially for finite horizon problems), while λ is a tunable parameter. An obvious motivation for the use of λ in a rolling horizon setting is the recognition that we are solving a deterministic approximation over a horizon in which events are uncertain.

6.2.5 Discussion

Lookahead strategies have long been a popular method for designing policies for stochastic, dynamic problems. Note that in each case we use both a model of exogenous information and some sort of method for making decisions in the future. Since optimizing over the future, even for a shorter horizon, quickly becomes computationally intractable, different strategies are used for simplifying the outcomes of future information, and simplifying how decisions might be made in the future.

Even with these approximations, lookahead procedures can be computationally demanding. It is largely as a result of a desire to find faster algorithms that researchers have developed methods based on value function approximations and policy function approximations.

6.3 POLICY FUNCTION APPROXIMATIONS

It is often the case that we have a very good idea of how to make a decision, and we can design a function (i.e., a policy) that returns a decision which captures the structure of the problem. For example:

■ EXAMPLE 6.1

A policeman would like to give tickets to maximize the revenue from the citations he writes. Stopping a car requires about 15 minutes to write up the citation, and the fines on violations within 10 miles per hour of the speed limit are fairly small. Violations of 20 miles per hour over the speed limit are significant, but relatively few drivers fall in this range. The policeman can formulate the problem as a dynamic program, but it is clear that the best policy will be to choose a speed, say \bar{s} , above which he writes out a citation. The problem is choosing \bar{s} . ■

■ EXAMPLE 6.2

A utility wants to maximize the profits earned by storing energy in a battery when prices are lowest during the day, and releasing the energy when prices are highest. There is a fairly regular daily pattern to prices. The optimal policy can be found by solving a dynamic program, but it is fairly apparent that the policy is to charge the battery at one time during the day, and discharge it at another. The problem is identifying these times. ■

Policies come in many forms, but all are functions that return a decision given an action. In this section we are limiting ourselves to functions that return a policy directly from the state, without solving an embedded optimization problem. As with value function approximations, which return an estimate of the value of being in a state, policy functions return an action given a state. Policy function approximations come in three basic styles:

Lookup tables. When we are in a discrete state s , the function returns a discrete action a directly from a matrix.

Parametric representations. These are explicit, analytic functions that are designed by the analyst. Similar to the process of choosing basis functions for value function approximations, designing a parametric representation of a policy is an art form.

Nonparametric representations. Nonparametric statistics offer the potential of producing very general behaviors without requiring the specification of basis functions.

Since a policy is *any* function that returns an action given a state, it is important to recognize when designing an algorithm if we are using (1) a rolling horizon procedure, (2) a value function approximation, or (3) a parametric representation of a policy. Readers need to understand that when a reference is made to “policy search” and “policy gradient methods,” this usually means that a parametric representation of a policy is being used.

We briefly describe each of these classes below.

6.3.1 Lookup Table Policies

We have already seen lookup table representations of policies in Chapter 3. By a lookup table, we specifically refer to a policy where every state has its own action (similar to having a distinct value of being in each state). This means we have one parameter (an action) for each state. We exclude from this class any policies that can be parameterized by a smaller number of parameters.

It is fairly rare that we would use a lookup table representation in an algorithmic setting, since this is the least compact form. However, lookup tables are relatively common in practice, since they are easy to understand. The Transportation Safety Administration (TSA) has specific rules that determine when and how a passenger should be searched. Call-in centers use specific rules to govern how a call should

be routed. A utility will use rules to govern how power should be released to the grid to avoid a system failure. Lookup tables are easy to understand, and easy to enforce. But, in practice, they can be very hard to optimize.

6.3.2 Parametric Policy Models

Without question the most widely used form of policy function approximation is the simple analytic models parameterized by a low dimensional vector. Some examples include:

- We are holding a stock, and would like to sell it when it goes over a price θ .
- In an inventory policy, we will order new product when the inventory S_t falls below θ_1 . When this happens, we place an order $a_t = \theta_2 - S_t$, which means we “order up to” θ_2 .
- We might choose to set the output x_t from a water reservoir, as a function of the state (the level of the water) S_t of the reservoir, as a linear function of the form $x_t = \theta_0 + \theta_1 S_t$. Or we might desire a nonlinear relationship with the water level, and use a basis function $\phi(S_t)$ to produce a policy $x_t = \theta_0 + \theta_1 \phi(S_t)$. These are known in the control literature as *affine policies* (literally, linear in the parameters).

Designing parametric policy function approximations is an art form. The science arises in determining the parameter vector θ .

To choose θ , assume that we have a parametric policy $A^\pi(S_t|\theta)$ (or $X^\pi(S_t|\theta)$), where we express the explicit dependence of the policy on θ . We would then approach the problem of finding the best value of θ as a stochastic optimization problem, written in the form

$$\max_{\theta} F(\theta) = \mathbb{E} \sum_{t=0}^T \gamma^t C(S_t, A^\pi(S_t, \theta)).$$

Here we write \max_{θ} whereas before we would write $\max_{\pi \in \Pi}$ to express the fact that we are optimizing over a class of policies Π that captures the parametric structure. $\max_{\pi \in \Pi}$ is more generic, but in this setting, it is equivalent to \max_{θ} .

The major challenge we face is that we cannot compute $F(\theta)$ in any compact form, primarily because we cannot compute the expectation. We have to depend instead on Monte Carlo samples. Fortunately, there is an entire field known as stochastic search (see Spall, 2003, for an excellent overview) to help us with this process. We describe these algorithms in more detail in Chapter 7.

There is another strategy we can use to fit a parametric policy model. Let $\bar{V}^{n-1}(s)$ be our current value function approximation at state s . Assume that we are currently at a state S^n and then use our value function approximation to compute an action a^n :

$$a^n = \arg \max_a \left(C(S^n, a) + \bar{V}^{n-1}(S^{M,a}(S^n, a)) \right).$$

We can now view a^n as an “observation” of the action a^n corresponding to state S^n . When we fit value function approximations, we used the observation \hat{v}^n of the value of being in state S^n to fit our value function approximation. With policies, we can use a^n as the observation corresponding to state S^n to fit a function that “predicts” the action we should be taking when in state S^n .

6.3.3 Nonparametric Policy Models

The strengths and weaknesses of using parametric models for policy function approximations is the same as we encountered when using them for approximating value functions. Not surprisingly, there has been some interest in using nonparametric models for policies. We provide a brief introduction to nonparametric methods in Chapter 8.

The basic idea of a nonparametric policy is to use nonparametric statistics, such as kernel regression, to fit a function that predicts the action a when we are in state s . Just as we described above with a parametric policy model, we can compute a^n using a value function approximation $\bar{V}(s)$ when we are in state $s = S^N$. We then use the pairs of responses a^n and independent variables (covariates) S^n to fit the model.

6.4 VALUE FUNCTION APPROXIMATIONS

The most powerful and visible method for solving complex dynamic programs involves replacing the value function $V_t(S_t)$ with an approximation of some form. In fact, for many in the research community, approximate dynamic programming is viewed as being equivalent to replacing the value function with an approximation as a way of avoiding “the” curse of dimensionality. If we are approximating the value around the pre-decision state, a value function approximation means that we are making decisions using equations of the form

$$a_t = \arg \max_a (C(S_t, a) + \gamma \mathbb{E} \{\bar{V}(S^M(S_t, a, W_{t+1}))|S_t\}). \quad (6.13)$$

In Chapter 4 we discussed the challenge of computing the expectation, and argued that we can approximate the value around the post-decision state S_t^a , giving us

$$a_t = \arg \max_a (C(S_t, a) + \bar{V}(S^{M,a}(S_t, a))). \quad (6.14)$$

For our presentation here we are not concerned with whether we are approximating around the pre- or post-decision state. We simply want to consider instead different types of approximation strategies. There are three broad classes of approximation strategies that we can use:

Lookup tables. If we are in discrete state s , we have a stored value $\bar{V}(s)$ that gives an estimate of the value of being in state s .

Parametric models. These are analytic functions $\bar{V}(s|\theta)$ parameterized by a vector θ whose dimensionality is much smaller than the number of states.

Nonparametric models. These models draw on the field of nonparametric statistics, and that avoid the problem of designing analytic functions.

We describe each of these briefly below, but defer a more complete presentation to Chapter 8.

6.4.1 Lookup Tables

Lookup tables are the most basic way of representing a function. This strategy is only useful for discrete states, and typically assumes a *flat representation* where the states are numbered $s \in \{1, 2, \dots, |\mathcal{S}|\}$, where \mathcal{S} is the original set of states, where each state might consist of a vector. Using a flat representation, we have one parameter, $\bar{V}(s)$, for each value of s (we refer to the value of being in each state as a parameter, since it is a number that we have to estimate). As we saw in Chapter 4, if \hat{V}^n is a Monte Carlo estimate of the value of being in a state S^n , then we can update the value of being in state S^n using

$$\bar{V}^n(S^n) = (1 - \alpha)\bar{V}^{n-1}(S^n) + \alpha\hat{V}^n.$$

Thus we only learn about $\bar{V}^n(S^n)$ by visiting state S^n . If the state space is large, then this means that we have a very large number of parameters to estimate.

The power of lookup tables is that if we have a discrete state space, then in principle we can eventually approximate $V(S)$ with a high level of precision (if we visit all the states often enough). The down side is that visiting state s tells us nothing about the value of being in state $s' \neq s$.

6.4.2 Parametric Models

One of the simplest ways to avoid the curse of dimensionality is to replace value functions using a lookup table with some sort of regression model. We start by identifying what we feel are important features. If we are in state S , a feature is simply a function $\phi_f(S)$, $f \in \mathcal{F}$ that draws information from S . We would say that \mathcal{F} (or more properly $(\phi_f(S))_{f \in \mathcal{F}}$) is our set of features. In the language of approximate dynamic programming the functions $\phi_f(S)$ are referred to as *basis functions*.

Creating features is an art form, and depends on the problem. Some examples are as follows:

- We are trying to manage blood inventories, where R_{ti} is the number of units of blood type $i \in (1, 2, \dots, 8)$ at time t . The vector R_t is our state variable. Since R_t is a vector, the number of possible values of R_t may be extremely large, making the challenge of approximating $V(R)$ quite difficult. We might create a set of features $\phi_i^1(R) = R_i$, $\phi_i^2(R) = R_i^2$, $\phi_{ij}^3(R) = R_i R_j$.

- Perhaps we wish to design a computer to play tic-tac-toe. We might design features such as (1) the number of X's in corner points, (2) 0/1 to indicate if we have the center point, (3) the number of side points (excluding corners) that we have and (4) the number of rows and columns where we have X's in at least two elements.

Once we have a set of features, we might specify a value function approximation using

$$\bar{V}(S) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(S), \quad (6.15)$$

where θ is a vector of regression parameters to be estimated. Finding θ is the science within the art form. When we use an approximation of the form given in equation (6.15), we say that the function has a *linear architecture* or is a linear model, or simply linear. Statisticians will often describe (6.15) as a linear model, but care has to be used, because the basis functions may be nonlinear functions of the state. Just because an approximation is a linear model does not mean that the function is linear in the state variables.

If we are working with discrete states, it can be analytically convenient to create a matrix Φ with element $\phi_f(s)$ in row s , column f . If \bar{V} is our approximate value function, expressed as a column vector with an element for every state, then we can write the approximation using matrix algebra as

$$\bar{V} = \Phi\theta.$$

There are, of course, problems where a linear architecture will not work. For example, perhaps our value function is expressing a probability that we will win at backgammon, and we have designed a set of features $\phi_f(S)$ that capture important elements of the board, where S is the precise state of the board. We propose that we can create a utility function $U(S|\theta) = \sum_f \theta_f \phi_f(S)$ that provides a measure of the quality of our position. However, we do not observe utility; we do observe if we eventually win or lose a game. If $V = 1$, then we win, and we might write the probability of this using

$$P(V = 1|\theta) = \frac{e^{U(S|\theta)}}{1 + e^{U(S|\theta)}}. \quad (6.16)$$

We would describe this model as one with a *nonlinear architecture*, or nonlinear in the parameters. Our challenge is estimating θ from observations of whether we win or lose the game.

Parametric models are exceptionally powerful when they work. Their real value is that we can estimate a parameter vector θ using a relatively small number of observations. For example, we may feel that we can do a good job if we have 10 observations per feature. Thus, with a very small number of observations, we obtain a model that offers the potential of approximating an entire function, even if S is

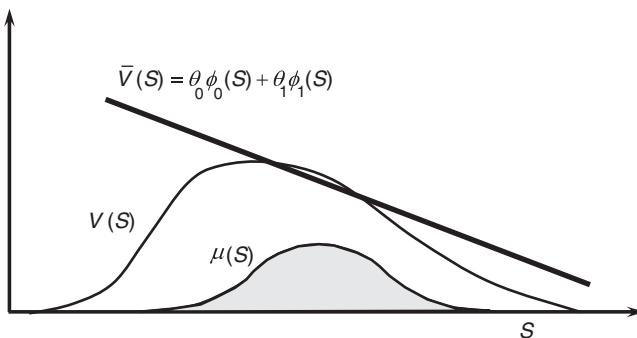


Figure 6.2 Illustration of linear (in the state) value function approximation of a nonlinear value function $\bar{V}(S)$, given the density $\mu(S)$ for visiting states.

multidimensional and continuous. Needless to say, this idea has attracted considerable interest. In fact many people even equate “approximate dynamic programming” with “approximating value functions using basis functions.”

The downside of parametric models is that we may not have the right basis functions to produce a good approximation. As an example consider the problem of approximating the value of storing natural gas in a storage facility, shown in Figure 6.2. Here $V(S)$ is the true value function as a function of the state S , which gives the amount of gas in storage, and $\mu(S)$ is the probability distribution describing the frequency with which we visit different states. Now assume that we have chosen basis functions where $\phi_0(S) = 1$ (the constant term) and $\phi_1(S) = S$ (linear in the state variable). With this choice, we have to fit a value function that is nonlinear in S , with a function that is linear in S . Figure 6.2 depicts the best linear approximation given $\mu(S)$, but obviously this would change quite a bit if the distribution describing what states we visited changed.

6.4.3 Nonparametric Models

Nonparametric models can be viewed as a hybrid of lookup tables and parametric models, but in some key respects are actually closer to lookup tables. There is a wide range of nonparametric models, but they all share the fundamental feature of using simple models (typically constants or linear models) to represent small regions of a function. They avoid the need to specify a specific structure such as the linear architecture in (6.15) or a nonlinear architecture such as that illustrated in (6.16). At the same time, most nonparametric models offer the additional feature of providing very accurate approximations of very general functions, as long as we have enough observations.

There are a variety of nonparametric methods, and the serious reader should consult a reference such as Hastie et al. (2009). The central idea is to estimate a function by using a weighted estimate of local observations of the function (see Figure 6.3). Say we want to estimate the value function $\bar{V}(s)$ at a particular state (called the query state) s . We have made a series of observations $\hat{v}^1, \hat{v}^2, \dots, \hat{v}^n$

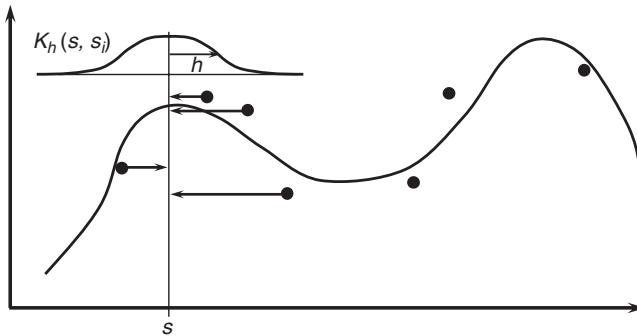


Figure 6.3 Kernel with bandwidth h being used to estimate the value function at a query state s .

at states S^1, S^2, \dots, S^n . We can form an estimate $\bar{V}(s)$ using

$$\bar{V}(s) = \frac{\sum_{i=1}^n K_h(s, s^i) \hat{v}^i}{\sum_{i=1}^n K_h(s, s^i)}, \quad (6.17)$$

where $K_h(s, s^i)$ is a weighting function parameterized by a *bandwidth* h . A common choice of weighting function is the Gaussian density, given by

$$K_h(s, s^i) = \exp - \left(\frac{s - s^i}{h} \right)^2.$$

Here the bandwidth h serves as the standard deviation. This form of weighting is often referred to as a *radial basis function* in the approximate dynamic programming literature, which has the unfortunate effect of placing it in the same family of approximation strategies as linear regression. However, parametric and nonparametric approximation strategies are fundamentally different, with very different implications in terms of the design of algorithms.

Nonparametric methods offer significant flexibility. Unlike parametric models, where we are finding the parameters of a fixed functional form, nonparametric methods do not have a formal model, since the data forms the model. Any observation of the function at a query state s requires summing over all prior observations, which introduces additional computational requirements. For approximate dynamic programming, we encounter the same issue we saw in Chapter 4 of exploration. With lookup tables, we have to visit a particular state in order to estimate the value of a state. With kernel regression, we relax this requirement, but we still have to visit points close to a state in order to estimate the value of a state.

6.5 HYBRID STRATEGIES

The concept of (possibly tunable) myopic policies, lookahead policies, policies based on value function approximations, and policy function approximation represent the core tools in the arsenal for approximating solving dynamic programs.

Given the richness of applications, it perhaps should not be surprising that we often turn to mixtures of these strategies.

6.5.1 Myopic Policies with Tunable Parameters

Myopic policies are so effective for some problem classes that all we want to do is to make them a little smarter, which is to say make decisions that do a better job capturing the impact of a decision now on the future. We can create a family of myopic policies by introducing tunable parameters that might help achieve good behaviors over time. Imagine that task $j \in \mathcal{J}_t$ needs to be handled before time τ_j . If a task is not handled within the time window, it vanishes from the system and we lose the reward that we might have earned. Now define a modified reward given by

$$c_{tij}^\pi(\theta) = c_{ij} + \theta_0 e^{-\theta_1(\tau_j - t)}. \quad (6.18)$$

This modified contribution increases the reward (provided that θ_0 and θ_1 are both positive) for covering a task as it gets closer to its due date τ_j . The reward is controlled by the parameter vector $\theta = (\theta_0, \theta_1)$. We can now write our myopic policy in the form

$$X^\pi(S_t) = \arg \max_{x \in \mathcal{X}_t} \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}_t} c_{tij}^\pi(\theta) x_{tij}. \quad (6.19)$$

The policy π is characterized by the function in (6.19), which includes the choice of the vector θ . We note that by tuning θ , we might be able to get our myopic policy to make decisions that produce better decisions in the future. But the policy does not explicitly use any type of forecast of future information or future decisions.

6.5.2 Rolling Horizon Procedures with Value Function Approximations

Deterministic rolling horizon procedures offer the advantage that we can solve them optimally, and if we have vector-valued decisions, we can use commercial solvers. Limitations of this approach are (1) they require that we use a deterministic view of the future and (2) they can be computationally expensive to solve (pushing us to use shorter horizons). By contrast, a major limitation of value function approximations is that we may not be able to capture the complex interactions that are taking place within our optimization of the future.

An obvious strategy is to combine the two approaches. For low-dimensional action spaces, we can use tree search or a roll-out heuristics for H periods, and then use a value function approximation. If we are using a rolling horizon procedure for vector-valued decisions, we might solve

$$X^\pi(S_t) = \arg \max_{x_t, \dots, x_{t+H}} \sum_{t'=t}^{t+H-1} \gamma^{t'-t} C(S_{t'}, x_{t'}) + \gamma^H \bar{V}_{t+H}(S_{t+H}),$$

where S_{t+H} is determined by X_{t+H} . In this setting, $\bar{V}_{t+H}(S_{t+H})$ would have to be some convenient analytical form (linear, piecewise linear, nonlinear) in order to

be used in an appropriate solver. If we use the algorithmic discount factor λ , the policy would look like

$$X^\pi(S_t) = \arg \max_{x_t, \dots, x_{t+H}} \sum_{t'=t}^{t+H-1} \gamma^{t'-t} \lambda^{t'-t} C(S_{t'}, x_{t'}) + \gamma^H \lambda^H \bar{V}_{t+H}(S_{t+H}).$$

6.5.3 Rolling Horizon Procedures with Tunable Policies

A rolling horizon procedure using a deterministic forecast is, of course, vulnerable to the use of a point forecast of the future. For example, in the energy application we used in Section 6.2.4 to introduce rolling horizon procedures, we might use a deterministic forecast of prices, energy from the solar panels, and demand. A problem with this is that the model will optimize provided that these forecasts are known perfectly, potentially leaving no buffer in the battery to handle surges in demand or drops in solar energy.

This limitation will not be solved by introducing value function approximations at the end of the horizon. It is possible, however, to perturb our deterministic model to make the solution more robust with respect to uncertainty. For example, we might reduce the amount of energy we expect to receive from the solar panel, or we might inflate the demand. As we step forward in time, if we have overestimated demand or underestimated the energy from the solar panel, we can put the excess energy in the battery. Suppose that we factor demand by $\beta^D > 1$, and factor solar energy output by $\beta^S < 1$. Now we have a rolling horizon policy parameterized by (β^D, β^S) . We can tune these parameters just as we would tune the parameters of a policy function approximation.

Just as designing a policy function approximation is an art, designing these adjustments to a rolling horizon policy (or any other lookahead strategy) is also an art. Imagine optimizing the movement of a robot arm to take into consideration variations in the mechanics. We can optimize over a horizon ignoring these variations, but we might plan a trajectory that puts the arm too close to a barrier. We can modify our rolling horizon procedure by introducing a margin of error (the arm cannot be closer than β to a barrier), which becomes a tunable parameter.

6.5.4 Rollout Heuristics with Policy Function Approximation

Rollout heuristics can be particularly effective when we have access to a reasonable policy to simulate decisions that we might take if an action a takes us to a state s' . This heuristic might exist in the form of a policy function approximation $A^\pi(s|\theta)$ parameterized by a vector θ . We can tune θ , recognizing that we are still choosing a decision based on the one period contribution $C(s, a)$ plus the approximate value of being in state $s' = S^M(s, a, W)$.

6.5.5 Tree Search with Roll-out Heuristic and a Lookup Table Policy

A surprisingly powerful heuristic algorithm that has received considerable success in the context of designing computer algorithms to play games uses a limited tree

search, which is then augmented by a roll-out heuristic assisted by a user-defined lookup table policy. For example, a computer might evaluate all the options for a chess game for the next four moves, at which point the tree grows explosively. After four moves, the algorithm might resort to a roll-out heuristic, assisted by rules derived from thousands of chess games. These rules are encapsulated in an aggregated form of lookup table policy that guides the search for a number of additional moves into the future.

6.5.6 Value Function Approximation with Lookup Table or Policy Function Approximation

Suppose that we are given a policy $\bar{A}(S_t)$, which might be in the form of a lookup table or a parameterized policy function approximation. This policy might reflect the experience of a domain expert, or it might be derived from a large database of past decisions. For example, we might have access to the decisions of people playing online poker, or it might be the historical patterns of a company. We can think of $\bar{A}(S_t)$ as the decision of the domain expert or the decision made in the field. If the action is continuous, we could incorporate it into our decision function using

$$A^\pi(S_t) = \arg \max_a (C(S_t, a) + \bar{V}(S^{M,a}(S_t, a)) + \beta(\bar{A}(S_t) - a)^2).$$

The term $\beta(\bar{A}(S_t) - a)^2$ can be viewed as a penalty for choosing actions that deviate from the external domain expert. β controls how important this term is. We note that this penalty term can be set up to handle decisions at some level of aggregation.

6.6 RANDOMIZED POLICIES

In Section 3.10.5 we introduced the idea of randomized policies, and showed that under certain conditions a deterministic policy (one where the state S and a decision rule uniquely determine an action) will always do at least as well or better than a policy that chooses an action partly at random. In the setting of approximate dynamic programming, randomized policies take on a new role. We may, for example, wish to ensure that we are testing all actions (and reachable states) infinitely often. This arises because in ADP, we need to *explore* states and actions to learn how well they do.

The common notation in approximate dynamic programming for representing randomized policies is to let $\pi(a|s)$ be the probability of choosing action a given we are in state s . This is known as notational overloading, since the policy in this case is both the rule for choosing an action, and the probability that the rule produces for choosing the action. We separate these roles by allowing $\pi(a|s)$ to be the probability of choosing action a if we are in state s (or, more often, the parameters of the distribution that determines this probability), and let $A^\pi(s)$ be the

decision function, which might sample an action a at random using the probability distribution $\pi(a|s)$.

There are different reasons in approximate dynamic programming for using randomized policies. We list three such:

- We often have to strike a balance between exploring states and actions, and choosing actions that appear to be best (exploiting). The exploration versus exploitation issue is addressed in depth in Chapter 12.
- In Chapter 9 (Section 9.4) we discuss the need for policies where the probability of selection for an action may be strictly positive.
- In Chapter 10 (Section 10.7) we introduce a class of algorithms that requires that the policy be parameterized so that the probability of choosing an action is differentiable in a parameter, which means that it also needs to be continuous.

The most widely used policy that mixes exploration and exploitation is called ϵ -greedy. In this policy we choose an action $a \in \mathcal{A}$ at random with probability ϵ , and with probability $1 - \epsilon$:

$$a^n = \arg \max_{a \in \mathcal{A}} \left(C(S^n, a) + \gamma \mathbb{E} \bar{V}^{n-1}(S^M(S^n, a, W^{n+1})) \right),$$

which is to say we exploit our estimate of $\bar{V}^{n-1}(s')$. This policy guarantees that we will try all actions (and therefore all reachable states) infinitely often, which helps with convergence proofs. Of course, this policy may be of limited value for problems where \mathcal{A} is large.

One limitation of ϵ -greedy is that it keeps exploring at a constant rate. As the algorithm progresses, we may wish to spend more time evaluating actions that we think are best, rather than exploring actions at random. A strategy that addresses this is to use a declining exploration probability, such as

$$\epsilon^n = \frac{1}{n}.$$

This strategy not only reduces the exploration probability, but it does so at a rate that is sufficiently slow (in theory) that it ensures that we will still try all actions infinitely often. In practice, it is likely to be better to use a probability such as $1/n^\beta$ where $\beta \in (0.5, 1]$.

The ϵ -greedy policy exhibits a property known in the literature as “greedy in the limit with infinite exploration,” or GLIE. Such a property is important in convergence proofs. A limitation of ϵ -greedy is that when we explore, we choose actions at random without regard to their estimated value. In applications with larger action spaces, this can mean that we are spending a lot of time evaluating actions that are quite poor. An alternative is to compute a probability of choosing an action based on its estimated value. The most popular of these rules uses

$$P(a|s) = \frac{e^{\beta \bar{Q}^l(s,a)}}{\sum_{a' \in \mathcal{A}} e^{\beta \bar{Q}^l(s,a')}},$$

where $\bar{Q}^n(s, a)$ is an estimate of the value of being in state s and taking action a . β is a tunable parameter, where $\beta = 0$ produces a pure exploration policy, while as $\beta \rightarrow \infty$, the policy becomes greedy (choosing the action that appears to be best).

This policy is known under different names such as Boltzmann exploration, Gibbs sampling and a soft-max policy. Boltzmann exploration is also known as a “restricted rank-based randomized” policy (or RRR policy). The term “soft-max” refers to the fact that it is maximizing over the actions based on the factors $Q(s, a)$ but produces an outcome where actions other than the best have a positive probability of being chosen.

6.7 HOW TO CHOOSE A POLICY?

Given the choice of policies, the question naturally arises, how do we choose which policy is best for a particular problem? Not surprisingly, it depends on both the characteristics of the problem and constraints on computation time and the complexity of the algorithm. Below we summarize different types of problems, and provide a sample of a policy that appears to be well suited to the application, largely based on our own experiences with real problems.

6.7.1 A Myopic Policy

A business jet company has to assign pilots to aircraft to serve customers. In making decisions about which jet and which pilot to use, it might make sense to think about whether a certain type of jet is better suited given the destination of the customer (are there other customers nearby who prefer that same type of jet)? We might also want to think about whether we want to use a younger pilot who needs more experience. However, the here-and-now costs of finding the pilot and aircraft that can serve the customer at least cost dominate what *might* happen in the future. Also the problem of finding the best pilot and aircraft, across a fleet of hundreds of aircraft and thousands of pilots, produces a fairly difficult integer programming problem. For this situation a myopic policy works extremely well given the financial objectives and algorithmic constraints.

6.7.2 A Lookahead Policy—Tree Search

You are trying to program a computer to operate a robot that is described by several dozen different parameters describing the location, velocity, and acceleration of both the robot and its arms. The number of actions is not too large, but choosing an action requires considering not only what might happen next (exogenous events) but also decisions we might make in the future. The state space is very large with relatively little structure, which makes it very hard to approximate the value of being in a state. Instead, it is much easier to simply step into the future, enumerating future potential decisions with an approximation (e.g., with Monte Carlo sampling) of random events.

6.7.3 A Lookahead Policy—Deterministic Rolling Horizon Procedures

Now assume that we are managing an energy storage device for a building that has to balance energy from the grid (where an advance commitment has been made the previous day), an hourly forecast of energy usage (which is fairly predictable), and energy generation from a set of solar panels on the roof. If the demand of the building exceeds its commitment from the grid, it can turn to the solar panels and battery, using a diesel generator as a backup. With a perfect forecast of the day, it is possible to formulate the decision of how much to store and when, when to use energy in the battery, and when to use the diesel generator as a single, deterministic mixed-integer program. Of course, this ignores the uncertainty in both demand and the sun, but most of the quantities can be predicted with some degree of accuracy. The optimization model makes it possible to balance dynamics of hourly demand and supply to produce an energy plan over the course of the day.

6.7.4 A Lookahead Policy—Stochastic Rolling Horizon Procedures

A utility has to operate a series of water reservoirs to meet demands for electricity, while also observing a host of physical and legal constraints that govern how much water is released and when. For example, during dry periods there are laws that specify that the discomfort of low water flows has to be shared. Of course, reservoirs have maximum and minimum limits on the amount of water they can hold. There can be tremendous uncertainty about the rainfall over the course of a season, and as a result the utility would like to make decisions now while accounting for the different possible outcomes. Stochastic programming enumerates all the decisions over the entire year, for each possible scenario (while ensuring that only one decision is made now). The stochastic program ensures that all constraints will be satisfied for each scenario.

6.7.5 Policy Function Approximations

A utility would like to know the value of a battery that can store electricity when prices are low and release them when prices are high. The price process is highly volatile, with a modest daily cycle. The utility needs a simple policy that is easy to implement in software. The utility chose a policy where we fix two prices, and store when prices are below the lower level and release when prices are above the higher level. This requires optimizing these two price points. A different policy might involve storing at a certain time of day, and releasing at another time of day, to capture the daily cycle.

6.7.6 Value Function Approximations—Transportation

A truckload carrier needs to determine the best driver to assign to a load, which may take several days to complete. The carrier has to think about whether it wants to accept the load (if the load goes to a particular city, will there be too many drivers in that city?). It also has to decide if it wants to use a particular driver, since the driver

eventually needs to get home, and the load may take it to a destination from which it is difficult for the driver to eventually get home. Loads are highly dynamic, so the carrier will not know with any precision which loads will be available when the driver arrives. What has worked very well is to estimate the value of having a particular type of driver in a particular location. With this value, the carrier has a relatively simple optimization problem to determine which driver to assign to each load.

6.7.7 Value Function Approximations—Energy

There is considerable interest in the interaction of energy from wind and hydroelectric storage. Hydroelectric power is fairly easy to manipulate, making it an attractive source of energy that can be paired with energy from wind. However, this requires modeling the problem in hourly increments (to capture the variation in wind) over an entire year (to capture seasonal variations in rainfall and the water being held in the reservoir). It is also critical to capture the uncertainty in both the wind and rainfall. The problem is relatively simple, and decisions can be made using a value function approximation that captures only the value of storing a certain amount of water in the reservoir. However, these value functions have to depend on the time of year. Deterministic approximations provide highly distorted results, and simple rules require parameters that depend on the time (hour) of year. Optimizing a simple policy requires tuning thousands of parameters. Value function approximations make it relatively easy to obtain time-dependent decisions that capture uncertainty.

6.7.8 Discussion

The examples of this section raise a series of questions that should be asked when choosing the structure of a policy:

- Will a myopic policy solve your problem? If not, is it at least a good starting point?
- Does the problem have structure that suggests a simple and natural decision rule? If there is an “obvious” policy (e.g., replenish inventory when it gets too low), then more sophisticated algorithms based on value function approximations are likely to struggle. Exploiting structure always helps.
- Is the problem fairly stationary, or highly nonstationary? Nonstationary problems (e.g., responding to hourly demand or daily water levels) mean that you need a policy that depends on time. Rolling horizon problems can work well if the level of uncertainty is low relative to the predictable variability. It is hard to produce policy function approximations where the parameters vary by time period.
- If you think about approximating the value of being in a state, does this appear to be a relatively simple problem? If the value function is going to be very complex, it will be hard to approximate, making value function approximations hard to use. But if it is not too complex, value function approximations may be a very effective strategy.

Unless you are pursuing an algorithm as an intellectual exercise, it is best to focus on your problem and choose the method that is best suited to the application. For

more complex problems, be prepared to use a hybrid strategy. For example, rolling horizon procedures may be combined with adjustments that depend on tunable parameters (a form of policy function approximation). You might use a tree search combined with a simple value function approximation to help reduce the size of the tree.

6.8 BIBLIOGRAPHIC NOTES

The goal of this chapter is to organize the diverse policies that have been suggested in the ADP and RL communities into a more compact framework. In the process we are challenging commonly held assumptions, for example, that “approximate dynamic programming” always means that we are approximating value functions, even if this is one of the most popular strategies. Lookahead policies and policy function approximations are effective strategies for certain problem classes.

Section 6.2 Lookahead policies have been widely used in engineering practice in operations research under the name of rolling horizon procedure, and in computer science as a receding horizon procedure, and in engineering under the name model predictive control (Camacho and Bordons, 2004). Decision trees are similarly a widely used strategy for which there are many references. Roll-out heuristics were introduced by Wu (1997) and Bertsekas and Castanon (1999). Stochastic programming, which combines uncertainty with multidimensional decision vectors, is reviewed in Birge and Louveaux (1997) and Kall and Wallace (1994), among others. Secomandi (2008) studies the effect of reoptimization on rolling horizon procedures as they adapt to new information.

Section 6.3 While there are over 1000 papers that refer to “value function approximation” in the literature (as of this writing), there were only a few dozen papers using “policy function approximation.” However, this is a term that we feel deserves more widespread use as it highlights the symmetry between the two strategies.

Section 6.4 Making decisions that depend on an approximation of the value of being in a state has defined approximation dynamic programming since Bellman and Kalaba (1959).

PROBLEMS

6.1 Following is a list of how decisions are made in specific situations. For each, classify the decision function in terms of which of the four fundamental classes of policies are being used. If a policy function approximation or value function approximation is used, identify which functional class is being used:

- If the temperature is below 40 degrees F when I wake up, I put on a winter coat. If it is above 40 but less than 55, I will wear a light jacket. Above 55, I do not wear any jacket.
- When I get in my car, I use the navigation system to compute the path I should use to get to my destination.

- To determine which coal plants, natural gas plants and nuclear power plants to use tomorrow, a grid operator solves an integer program that plans over the next 24 hours which generators should be turned on or off, and when. This plan is then used to notify the plants who will be in operation tomorrow.
- A chess player makes a move based on her prior experience of the probability of winning from a particular board position.
- A stock broker is watching a stock rise from \$22 per share up to \$36 per share. After hitting \$36, the broker decides to hold on to the stock for a few more days because of the feeling that the stock might still go up.
- A utility has to plan water flows from one reservoir to the next, while ensuring that a host of legal restrictions will be satisfied. The problem can be formulated as a linear program which enforces these constraints. The utility uses a forecast of rainfalls over the next 12 months to determine what it should do right now.
- The utility now decides to capture uncertainties in the rainfall by modeling 20 different scenarios of what the rainfall might be on a month-by-month basis over the next year.
- A mutual fund has to decide how much cash to keep on hand. The mutual fund uses the rule of keeping enough cash to cover total redemptions over the last 5 days.
- A company is planning sales of TVs over the Christmas season. It produces a projection of the demand on a week-by-week basis, but does not want to end the season with zero inventories. So the company adds a function that provides positive value for up to 20 TVs.
- A wind farm has to make commitments of how much energy it can provide tomorrow. The wind farm creates a forecast, including an estimate of the expected amount of wind and the standard deviation of the error. The operator then makes an energy commitment so that there is an 80 percent probability that he will be able to make the commitment.

6.2 Earlier we considered the problem of assigning a resource i to a task j . If the task is not covered at time t , we hold it in the hope that we can complete it in the future. We would like to give tasks that have been delayed more higher priority, so instead of just maximizing the contribution c_{ij} , we add in a bonus that increases with how long the task has been delayed, giving us the modified contribution

$$c_{ij}^\pi(\theta) = c_{ij} + \theta_0 e^{-\theta_1(\tau_j - t)}.$$

Now imagine using this contribution function, but optimizing over a time horizon T using forecasts of tasks that might arrive in the future. Would solving this problem, using $c_{ij}^\pi(\theta)$ as the contribution for covering task j using resource i at time t , give you the behavior that you want?

Policy Search

In Chapter 6 we described four classes of policies: myopic policies, lookahead policies, policies that depend on value function approximations, and policy function approximations. Each of these policies may be tunable in some way.

Myopic policies. We choose the best action to maximize the contribution now, without regard to the impact of the decision on the future. We may add bonuses to serve a customer that has been delayed, or our myopic policy may depend on design decisions such as the size of buffers or the location of ambulance bases. These bonuses and constraints represent tunable parameters.

Lookahead policies. We optimize over a horizon H , possibly using a deterministic representation of future events such as random customer demands or deviations in trajectories of vehicles or robots. We might introduce buffer stocks of resources to handle surges in demands, or boundaries to keep a vehicle away from a cliff or barrier. The size of these buffer stocks and boundaries represents tunable parameters, as would the horizon H .

Value function approximations. We make a decision now by maximizing

$$A^\pi(S|\theta) = \arg \max_a \left(C(S, a) + \gamma \sum_f \theta_f \phi_f(S^{M,a}(S, a)) \right). \quad (7.1)$$

In this setting we can view the regression variables θ as tunable parameters.

Policy function approximations. Policy function approximations might be simple functions such as a rule to order $Q - S$ units of inventory when the inventory $S < q$, where Q and q are tunable parameters. Or, we may decide that an action a can be expressed using

$$a = \theta_0 + \theta_1 S + \theta_2 S^2.$$

In this setting the vector θ would be our tunable vector. Finally, it is popular to use neural networks to represent a function that computes an action given

a state. The neural network is parameterized by a vector of weights w , which would represent the tunable parameters.

As we can see, there is a wide range of ways that a policy can be tuned, with different notations for the parameter vector being tuned. All of this literature falls under the general umbrella of stochastic search, which is typically written

$$\min_x \mathbb{E} F(x, W).$$

Here x might be discrete, a continuous scalar, or a vector of binary, continuous, or discrete variables, each introducing its own computational complexities. We use x for consistency with the literature on stochastic search where x is a decision. In our dynamic programming problems, x is a parameter that controls a policy that is used to choose an action a . In a specific dynamic programming application, we might use θ or ρ as our tunable parameter, which will help to avoid confusion when our action is the vector x (so many things to do with so few variables!).

This chapter pursues a specific path for designing policies that is distinctly different from the strategy that depends on approximating value functions. For example, if our policy depends on a value function approximation (as shown in equation (7.1)), in this chapter we are going to tune θ to produce the highest contribution. By contrast, in Chapters 8 through 9 we are going to try to find θ so that the value function approximation closely approximates the value of being in a state, with the hope that if we find a good approximation, it will produce a good policy. In this chapter we only care if θ produces a good policy, and not if the resulting value function approximation approximates the value of being in a state.

Readers will (or should) reasonably ask, if the goal of this chapter is to find parameters to produce policies that work the best, why would we devote Chapters 8, 9, and 10 to finding value functions that predict the value of being in a state, with the hope that this produces a good policy? One answer is that it is not always easy to design a policy that, if properly tuned, produces good results. Another is that the techniques in this chapter will generally work best if the parameter vector x is fairly low dimensional. For example, it can be difficult using policy search to solve time-dependent problems, since you effectively have to optimize a policy for each time period. This is much easier when we are using methods based on approximating value functions. At the same time there are many problems where the structure of a good policy is fairly self-evident. Examples include ordering inventory when it falls below some level (we need to tune the right level), or sell a stock when the price exceeds a limit (we need to find the right limit). If we ignore this structure, then we may find that we are producing policies that underperform simple rules.

7.1 BACKGROUND

The stochastic optimization problems that we consider in this volume can all be written in the general form

$$\max_{\pi} \mathbb{E} \sum_{t=0}^T \gamma^t C(S_t, A^\pi(S_t)). \quad (7.2)$$

This mathematical statement only makes sense when we understand the nature of a policy, and then what we mean by finding the best policy. In this chapter we assume we have some function (the policy) characterized by a vector of parameters x . Our choice of the notation “ x ” reflects the roots of this theory to the communities that work on stochastic search and stochastic optimization, where x is the decision vector. Here x is a set of parameters that governs how the policy works. For example, if a policy is given by

$$A^\pi(S_t | \theta) = \arg \max_a \left(C(S_t, a) + \gamma \mathbb{E} \sum_f \theta_f \phi_f(S_t) \right),$$

then this policy is parameterized by θ , and we would be searching for θ instead of x . However, we might have an inventory system where S_t is the inventory, and we order $Q - S_t$ whenever $S_t < q$, and 0 otherwise. In this setting, x would be the parameters (Q, q) .

Now let

$$F(x) = \mathbb{E} F^\pi(x, W),$$

where

$$F^\pi(x, W) = \sum_{t=0}^T \gamma^t C(S_t, A^\pi(S_t)). \quad (7.3)$$

Here we let W represent all the random variables that arise during our simulation. For example, this could represent machine failures, customer demands, fluctuations in wind and prices, and deviations in robotic trajectories. We remind the reader that W may depend on the policy or, equivalently, the parameter x , although we are not going to make a point of this.

We note that there are many problems that can be written in the general form

$$\min_x \mathbb{E} F(x, W), \quad (7.4)$$

where $F(x, W)$ is some function of a random variable where the expectation is hard or impossible to compute, but where we can compute $F(x, W)$, obtaining a noisy (but unbiased) estimate of $\mathbb{E} F(x, W)$. If $\mathbb{E} F(x, W)$ were easy to compute, we would have a deterministic optimization problem $\min_x F(x)$, and we would not need this chapter (or even this book).

Problems that can be written in the form given in equation (7.4) are often described under a variety of names, including

Stochastic search. Stochastic search is probably the closest we can come to a general name that covers the entire field. Stochastic search often refers

to optimization of noisy functions, where x may be vector-valued and continuous.

Ranking and selection. This represents a special case where we are searching for x among a finite set $\{x_1, x_2, \dots, x_M\}$, where M is “not too large.” The size of M in this setting depends on the complexity of computing $F(x, W)$.

Simulation optimization. Assume that we really do have a multiperiod simulation problem, such as might arise in discrete-event simulators for manufacturing, simulations of call centers or simulations of the spread of a disease. Typically a single run of a simulator can be moderately to very time-consuming. We also have the ability to obtain more accurate estimates by running a simulation for a longer period of time, with costs for stopping and starting the simulator.

Optimization of expensive, black-box functions. There are applications where a model may take hours, days, or even weeks to run. The challenge is to tune parameters to obtain the best performance. A single run may be noisy (stochastic), but not necessarily, and the behavior of the function is often nonconvex.

All of these settings are mathematically equivalent, but introduce different computational issues. Our interest is primarily in multiperiod problems, but it is impossible to ignore the mathematical equivalence between choosing a parameter to optimize a sequential decision problem, and choosing a parameter to optimize a single-stage problem (where you choose x , observe W and then stop).

For our purposes it is important to characterize the nature of the decision variable x , which we can divide into four important classes:

- *Scalar continuous.* What is the right price of a product? What is the right temperature of a chemical process?
- *Discrete and finite.* By this we mean a “not too large” number of discrete alternatives (as described under ranking and selection above).
- *Vector-valued and continuous.* This means the search space is multidimensional, requiring us to draw on a more general class of search algorithms that are stochastic analogs of nonlinear programming algorithms.
- *Vector-valued and discrete.* These often arise in complex design problems, such as finding the sequence of jobs to move through a machine, the best set of proposals that should be funded, or the best locations for a set of facilities (rather than just one facility).

We also need to separate three classes of functions:

- The function $F(x, W)$, given W , is differentiable with respect to x . We refer to $\nabla_x F(x, W)$ as a *stochastic gradient*. Note that even if we can compute derivatives, the function may be nondifferentiable. For example, we might have $F(x, W) = p \min\{x, W\} - cx$. For a given value of W ,

$$\frac{\partial F(x, W)}{\partial x} = \begin{cases} p - c & \text{if } x < W, \\ -c & \text{if } x > W. \end{cases}$$

If $x = W$, the derivative is not defined, but we can generally compute a subgradient (for the example above, this is like choosing between $p - c$ or $-c$ randomly), and still get provable convergence.

- We cannot compute the derivative (possibly because it is a complex simulation and we simply do not know how to find a derivative), but we can obtain noisy observations of the function fairly quickly.
- We can obtain noisy observations of the function, but these are expensive, putting special emphasis on intelligent search.

This discussion provides a peek into the rich fields that fall under the broad umbrella of stochastic search. In this chapter we highlight some of the tools that can be applied to approximate dynamic programming. However, we are unable to provide a thorough introduction to the field. For this purpose we refer readers to the excellent survey by Span (2003), which is written at a similar mathematical level to this book.

7.2 GRADIENT SEARCH

We can model our search for a policy parameterized by x in the following generic form

$$\min_x \mathbb{E}F(x, W).$$

For example, assume that we are trying to solve a newsvendor problem, where we wish to allocate a quantity x of resources (“newspapers”) before we know the demand W . The optimization problem is given by

$$\max_x F(x) = \mathbb{E}p \min\{x, W\} - cx. \quad (7.5)$$

If we could compute $F(x)$ exactly (i.e., analytically), and its derivative, then we could find x^* by taking its derivative and setting it equal to zero. If this is not possible, we could still use a classical steepest ascent algorithm

$$x^n = x^{n-1} - \alpha_{n-1} \nabla_x F(x^{n-1}), \quad (7.6)$$

where α_{n-1} is a stepsize. For deterministic problems we typically choose the best stepsize by solving the one-dimensional optimization problem

$$\alpha^* = \arg \min_{\alpha} F(x^{n-1} - \alpha \nabla F(x^{n-1})). \quad (7.7)$$

We would then update x^n using $\alpha_{n-1} = \alpha^*$.

We assume in our work that we cannot compute the expectation exactly. We resort instead to an algorithmic strategy known as stochastic gradients, but also known as stochastic approximation procedures.

7.2.1 A Stochastic Gradient Algorithm

For our stochastic problem we assume that we either cannot compute $F(x)$ or cannot compute the gradient exactly. However, there are many problems where, if we fix $W = W(\omega)$, we can find the derivative of $F(x, W(\omega))$ with respect to x . Then, instead of using the deterministic updating formula in (7.6), we would use

$$x^n = x^{n-1} - \alpha_{n-1} \nabla_x F(x^{n-1}, W^n). \quad (7.8)$$

Here $\nabla_x F(x^{n-1}, W^n)$ is called a *stochastic gradient* because it depends on a sample realization of W^n . It is important to note our indexing. A variable such as x^{n-1} or α_{n-1} that is indexed by $n - 1$ is assumed to be a function of the observations W^1, W^2, \dots, W^{n-1} , but not W^n . Thus our stochastic gradient $\nabla_x F(x^{n-1}, W^n)$ depends on our previous solution x^{n-1} and our most recent observation W^n . To illustrate, consider the simple newsvendor problem with the objective that

$$F(x, W) = p \min\{x, W\} - cx.$$

In this problem we order a quantity x , and then observe a random demand W . We earn a revenue given by $p \min\{x, W\}$ (we cannot sell more than we bought, or more than the demand), but we had to pay for our order, producing a negative cost cx . Let $\nabla F(x, W(\omega))$ be the sample gradient, taken when $W = W(\omega)$. In our example, clearly

$$\frac{\partial F(x, W(\omega))}{\partial x} = \begin{cases} p - c & \text{if } x < W, \\ -c & \text{if } x > W. \end{cases} \quad (7.9)$$

We call $\nabla F(x, \omega)$ a *stochastic gradient* because it is a gradient that depends on a random variable. x^{n-1} is the estimate of x computed from the previous iteration (using the sample realization ω^{n-1}), while ω^n is the sample realization in iteration n (the indexing tells us that x^{n-1} was computed without ω^n). When the function is deterministic, we would choose the stepsize by solving the one-dimensional optimization problem

$$\min_{\alpha} F(x^{n-1} - \alpha_{n-1} \nabla F(x^{n-1}, \omega^n)). \quad (7.10)$$

Now we face the problem of finding the stepsize α_{n-1} . Unlike our deterministic algorithm, we cannot consider solving a one-dimensional search to find the best stepsize. Part of the problem is that a stochastic gradient can even point away from the optimal solution such that any positive stepsize actually makes the solution worse. For example, the right order quantity might be 15. However, even if we order $x = 20$, it is possible that the demand is greater 20 on a particular day. Although our optimal solution is less than our current estimate, the algorithm tells us to increase x .

Remark Many authors will write equation (7.6) in the form

$$x^{n+1} = x^n - \alpha_n \nabla F(x^n, \omega^n). \quad (7.11)$$

With this style, we would say that x^{n+1} is the estimate of x to be used in iteration $n + 1$ (although it was computed with the information from iteration n). We use the form in (7.6) because we will later allow the stepsizes to depend on the data, and the indexing tells us the information content of the stepsize. For theoretical reasons it is important that the stepsize be computed using information up through $n - 1$, hence our use of α_{n-1} . We index x^n on the left-hand side of (7.6) using n because the right-hand side has information from iteration n . It is often the case that time t is also our iteration counter, and so it helps to be consistent with our time indexing notation.

There are many applications where the units of the gradient, and the units of the decision variable, are different. This happens with our newsvendor example, where the gradient is in units of dollars, while the decision variable x is in units of newspapers. This is a significant problem that causes headaches in practice.

Returning to our original problem of estimating the mean, we assume, when running a stochastic gradient algorithm, that x^0 is an initial guess, and that $W(\omega^1)$ is our first observation. If our stepsize sequence uses an initial stepsize $\alpha_0 = 1$, then

$$\begin{aligned} x^1 &= (1 - \alpha_0)x^0 + \alpha_0 W(\omega^1) \\ &= W(\omega^1), \end{aligned}$$

which means that we do not need the initial estimate for x^0 . Smaller initial stepsizes would only make sense if we had access to a reliable initial guess, and in this case the stepsize should reflect the confidence in our original estimate (e.g., we might be warm-starting an algorithm from a previous iteration).

We can evaluate our performance using a mean squared statistical measure. If we have an initial estimate x^0 , then we would use

$$\text{MSE} = \frac{1}{n} \sum_{m=1}^n (x^{m-1} - W(\omega^m))^2. \quad (7.12)$$

However, it is often the case that the sequence of random variables $W(\omega^n)$ is non-stationary, which means that they are coming from a distribution that is changing over time. (Recall that in Chapter 4 we would make random observations of the value of being in a state, which we referred to as \hat{v}_t^n , but these depended on a value function approximation for future events which was changing over time.) In the present case, estimating the mean squared error is similar to our problem of estimating the mean of the random variable W , for which we should use a standard stochastic gradient (smoothing) expression of the form

$$\text{MSE}^n = (1 - \beta_{n-1})\text{MSE}^{n-1} + \beta_{n-1}(x^{n-1} - W(\omega^n))^2,$$

where β_{n-1} is another stepsize sequence (which could be the same as α_{n-1}).

7.2.2 Derivatives of Simulations

In the previous section we illustrated a stochastic gradient algorithm in the context of a fairly simple stochastic function. But imagine that we have a multiperiod simulation, such as we might encounter when simulating flows of jobs around a manufacturing center. Perhaps we use a simple rule to govern how jobs are assigned to machines once they have finished a particular step (e.g., being drilled or painted). However, these rules have to reflect physical constraints such as the size of buffers for holding jobs before a machine can start working on them. If the buffer for a downstream machine is full, the rule might specify that a job be routed to a different machine or to a special holding queue.

This is an example of a policy that is governed by static variables such as the size of the buffer. We would let x be the vector of buffer sizes. It would be helpful, then, if we could do more than simply run a simulation for a fixed vector x . What if we could compute the derivative with respect to each element of x , so that after running a simulation, we obtain all the derivatives?

Computing these derivatives from simulations is the focus of an entire branch of the simulation community. A class of algorithms called *infinitesimal perturbation analysis* was developed specifically for this purpose. It is beyond the scope of our presentation to describe these methods in any detail, but it is important for readers to be aware that the field exists. A good introduction is the book Ho (1992), which effectively invented the field. A short tutorial is given by Fu (2008). For a mathematically more mature treatment, see Glasserman (1991).

7.3 DIRECT POLICY SEARCH FOR FINITE ALTERNATIVES

While stochastic gradient algorithms are simple and elegant, there are many problems where derivatives are simply not available. When this is the case, we have to depend on sophisticated hunt-and-peck algorithms that successively guess at values of x to try while searching for the best value. A major part of the challenge, as before, is that if we try a value x^n , all we get is a noisy measurement $F(x^n, W^{n+1})$.

We are going to guide our search by creating a belief about the function $F(x) = \mathbb{E}F(x, W)$. There are three major strategies we can use for approximating a function:

Lookup table. We store an estimate \bar{F}_x for each discrete value of x .

Parametric model. We represent \bar{F}_x using some sort of parametric model. The parameter x may be discrete or continuous, scalar or vector-valued.

Nonparametric model. We represent \bar{F}_x using nonparametric statistics.

We begin our presentation assuming that x can only take on values from a set $\{x_1, x_2, \dots, x_M\}$, where M is not too large, and using a lookup table model for $\bar{F}(x)$. The problem of finding the best value of x from this set, using noisy measurements, is known as the *ranking and selection* problem.

We are going to use this basic but important problem class to establish some useful concepts in learning. These are directly applicable in the context of policy search (which is why we introduce the concepts here), but they also have a big role in the context of what is famously known as the exploration versus exploitation problem in dynamic programming, a topic that is covered in more depth in Chapter 12.

7.3.1 The Ranking and Selection Problem

We assume our parameter choices consist of a finite set of alternatives chosen from the set $\mathcal{X} = \{1, 2, \dots, M\}$. Let μ_x be the true value of our function at x that is unknown to us. At each iteration $n = 0, 1, \dots$, we have to choose the parameter vector x^n to measure, which may involve simulating a policy parameterized by x^n . We then make a noisy observation of μ_{x^n} , which we can express as

$$W_{x^n}^{n+1} = \mu_{x^n} + \varepsilon^{n+1}.$$

We emphasize that our indexing reflects the information content. Thus at iteration n we have been allowed to see $W_{x^0}^1, W_{x^1}^2, \dots, W_{x^{n-1}}^n$. We choose x^n before we are allowed to see $W_{x^n}^{n+1}$. We use our observations to create estimates θ_x^n .

The goal of the ranking and selection problem is to collect information $W_{x^0}^1, W_{x^1}^2, \dots, W_{x^{N-1}}^N$ within a budget of N measurements to produce the best possible design. The challenge is to design a measurement policy π that determines x^0, x^1, \dots, x^{N-1} and gives us the best information about each set of parameters. Note that we are using π to guide the policy of collecting information. At the end we view the parameter vector x as determining a policy for controlling the underlying dynamic program. Since the parameter vector x effectively determines the policy that controls our dynamic program, we feel that we can temporarily adopt the notation π to specify our policy for collecting information.

After we have made our measurements, we obtain the estimates θ_x^N for each choice x . We may write the problem of finding the best measurement policy as

$$\max_{\pi} \max_x \theta_x^N. \quad (7.13)$$

Equation (7.13) evaluates our success based on the actual estimates θ_x^N , which we can view as our best estimate of the value of x . A more ideal objective would be to solve

$$\max_{\pi} \mu_{x^\pi}, \quad (7.14)$$

where $x^\pi = \arg \max \theta_x^N$ is the choice that we believe is best based on our current estimates. Note that the objective (7.14) assumes that we know μ_x .

Below we present two views of our belief about μ_x , known as the frequentist view and the Bayesian view. The Bayesian view shows us a way of computing (7.14).

7.3.2 The Frequentist Approach

The frequentist view is arguably the approach that is most familiar to people with an introductory course in statistics. Suppose that we are trying to estimate the mean μ of a random variable W , which might be the performance of a device or policy. Let W^n be the n th sample observation. Also let θ^n be our estimate of μ , and $\hat{\sigma}^{2,n}$ be our estimate of the variance of W . We know from elementary statistics that we can write θ^n and $\hat{\sigma}^{2,n}$ using

$$\theta^n = \frac{1}{n} \sum_{i=1}^n W^i, \quad (7.15)$$

$$\hat{\sigma}^{2,n} = \frac{1}{n-1} \sum_{i=1}^n (W^i - \theta^n)^2. \quad (7.16)$$

Since the measurements W^i are random, the estimate θ^n is also a random variable. $\hat{\sigma}^{2,n}$ is our estimate, after n observations, of the variance of the random variable W . The variance of the estimator θ^n is given by

$$\bar{\sigma}^{2,n} = \frac{1}{n} \hat{\sigma}^{2,n}.$$

We can write these expressions recursively using

$$\theta^n = \left(1 - \frac{1}{n}\right) \theta^{n-1} + \frac{1}{n} W^n, \quad (7.17)$$

$$\hat{\sigma}^{2,n} = \begin{cases} \frac{1}{n} (W^n - \theta^{n-1})^2, & n = 2, \\ \frac{n-2}{n-1} \hat{\sigma}^{2,n-1} + \frac{1}{n} (W^n - \theta^{n-1})^2, & n > 2. \end{cases} \quad (7.18)$$

When we are collecting information about individual measurements, we would let N_x^n be the number of times we have measured alternative x after n measurements. The updating of θ_x^n would then be given by

$$\theta_x^{n+1} = \left(1 - \frac{1}{N_x^n}\right) \theta_x^n + \frac{1}{N_x^n} W_x^{n+1}.$$

The updating of $\hat{\sigma}_x^{2,n}$ would be modified similarly.

The process of collecting information about μ_x can be viewed as a dynamic program, since we are making decisions over time, just as we have been doing with the other dynamic programs in this book. However, there is one important difference. In most of our dynamic programming problems, our decisions modify some sort of physical state (the position of a robot, the positions on a game board, the allocation of resources). In the ranking and selection problem we do not have a physical state, but we do have a state of knowledge (often referred to as the belief state), which we represent using K^n . The state of knowledge captures what we

know about the parameters μ_x after n measurements. If we use a frequentist view, the state of knowledge is given by

$$K_{freq}^n = (\theta^n, \hat{\sigma}^{2,n}, n).$$

Typically we also associate a probability distribution around the estimate θ^n . If the errors in our observations W^n are normally distributed, then it is easy to verify that θ^n will be normally distributed. However, by the central limit theorem, θ^n will be closely approximated by a normal distribution even if the observations are not normally distributed.

If we are using the frequentist view, it is most natural to evaluate the success of our sequence of observations using the objective function in (7.13).

7.3.3 The Bayesian View

The Bayesian view is somewhat more subtle. When we adopt a Bayesian perspective, we start by assuming that we have a state of belief (known as the prior) about μ_x before we have made a single measurement. For our presentation we are going to always assume that our belief about μ_x is normally distributed, with mean μ_x and (we assume) a known variance σ_x^2 . To simplify our notation, we are going to introduce the notion of the *precision* of our belief, represented by β_x . The precision is defined simply as

$$\beta_x = \frac{1}{\sigma_x^2}.$$

It is important to recognize that in the Bayesian view, μ_x is viewed as a random variable described by our prior distribution. Before we have collected any information, we would say that $\mu_x \propto N(\theta_x^0, \beta_x^0)$, which means that μ_x is normally distributed with initial mean θ_x^0 and initial precision β_x^0 . Thus (θ_x^0, β_x^0) is our prior distribution of belief about μ_x before we have started taking measurements. After n measurements, we would say that (θ_x^n, β_x^n) is our prior distribution that we use to choose x^n , while $N(\theta_x^{n+1}, \beta_x^{n+1})$ would be the posterior distribution. Throughout our presentation, μ_x represents the true value of x , while θ_x^n represents our best estimate of the true value of μ_x .

Assume that we have chosen to measure x^n , allowing us to observe $W_{x^n}^{n+1}$. We assume that the measurement has a known precision $\beta_{x^n}^W$ (the reciprocal of the variance of W). If the measurements are also normally distributed, it is possible to show (although this takes a little development) that the *posterior* distribution of belief about μ_x after $n + 1$ measurements is normally distributed with mean and precision given by

$$\theta_x^{n+1} = \begin{cases} \frac{\beta_x^n \theta_x^n + \beta_x^W W_{x^n}^{n+1}}{\beta_x^n + \beta_x^W} & \text{if } x^n = x, \\ \theta_x^n & \text{otherwise;} \end{cases} \quad (7.19)$$

$$\beta_x^{n+1} = \begin{cases} \beta_x^n + \beta_x^W & \text{if } x^n = x, \\ \beta_x^n & \text{otherwise.} \end{cases} \quad (7.20)$$

As with the frequentist view, the problem of determining what measurements to observe can be viewed as a dynamic program with the knowledge state (or belief state) given by

$$K^n = (\theta_x^n, \beta_x^n)_{x=1}^M.$$

With the Bayesian view, we can see a way of computing the objective function in equation (7.14). This equation, however, does not recognize that μ_x is viewed as a random variable. For this reason we revise the objective to

$$\max_{\pi} F^\pi = \mathbb{E}_\mu \mathbb{E}_W \mu_{x^\pi}. \quad (7.21)$$

Here \mathbb{E}_W is the expectation over all the potential outcomes of the measurements W , given a particular truth μ . Then \mathbb{E}_μ is the expectation over all potential truths μ given the prior (θ^0, β^0) .

A good way to envision the expectations in F^π is to think about approximating it using Monte Carlo simulation. Imagine an experiment where, for each trial, you randomly sample a truth $\mu(\omega)$ from the prior $N(\theta^0, \beta^0)$ (here $\mu(\omega)$ represents a sample realization of what the truth might be). Then follow some policy π that specifies which alternatives you should measure (more on this below). When you want to measure alternative x , you draw a random realization

$$W_x^n = \mu_x(\omega) + \epsilon^n(\omega),$$

where $\epsilon^n(\omega)$ is a random perturbation around our assumed truth $\mu(\omega)$. If we follow policy π and sample path ω (which determines both the truth $\mu(\omega)$ and the sample observations $W(\omega)$), this will produce estimates $\theta^{\pi, N}(\omega)$ and a decision $x^\pi(\omega) = \arg \max_x \theta^{\pi, N}(\omega)$. This in turn allows us to compute a sample realization of $F^\pi(\omega)$ using

$$F^\pi(\omega) = \mu(\omega)_{x^\pi(\omega)}.$$

Notice how we use our method of sampling an assumed truth to evaluate how well our measurement policy has identified the best set of parameters.

If we do K trials (i.e., k different values of ω), then we can approximate F^π using

$$F^\pi \approx \frac{1}{K} \sum_{k=1}^K F^\pi(\omega^k).$$

A common strategy for evaluating measurement policies is to estimate the *expected opportunity cost*, EOC^π . For a sample realization ω this is given by

$$\text{EOC}^\pi(\omega) = \mu(\omega)_{x*} - \mu(\omega)_{x^\pi(\omega)},$$

where $x* = \arg \max_x \mu(\omega)_x$ is the true best alternative. We then average these using

$$F^\pi \approx \frac{1}{K} \sum_{k=1}^K \text{EOC}^\pi(\omega).$$

The challenge now is to find good measurement policies.

7.3.4 Bayesian Updating with Correlated Beliefs

There are problems where our belief about μ_x is correlated with our belief about $\mu_{x'}$. For example, x might be a price, speed, or temperature, representing a discretization of a continuous variable. Imagine that we measure x^n and obtain an observation W_x^{n+1} that suggests that we should increase our belief about μ_{x^n} . Perhaps this single measurement suggests that we should increase our belief about $\mu_{x'}$ for values x' that are close to x^n . We can incorporate these covariances into our updating strategies, a powerful feature that allows us to update beliefs where the number of measurements is much smaller than the number of alternatives we have to evaluate.

Let

$$\text{Cov}^n(\mu_x, \mu_{x'}) = \text{covariance between } \mu_x \text{ and } \mu_{x'} \text{ after } n \text{ measurements have been made,}$$

$$\Sigma^n = \text{covariance matrix, with element } \Sigma_{xx'}^n = \text{Cov}^n(\mu_x, \mu_{x'}).$$

Just as we defined the precision β_x^n to be the reciprocal of the variance, we are going to define the precision matrix B^n to be

$$B^n = (\Sigma^n)^{-1}.$$

Let e_x be a column vector of 0's with a 1 for element x , and as before we let W^{n+1} be the (scalar) observation when we decide to choose x . If we choose to measure x^n , we obtain a column vector of observations given by $W_x^{n+1}e_{x^n}$. Keeping in mind that θ^n is a column vector of our beliefs about the expectation of μ , the Bayesian equation for updating this vector in the presence of correlated beliefs is given by

$$\theta^{n+1} = (B^{n+1})^{-1}(B^n\theta^n + \beta^W W_x^{n+1}e_{x^n}), \quad (7.22)$$

where B^{n+1} is given by

$$B^{n+1} = (B^n + \beta^W e_{x^n}(e_{x^n})^T). \quad (7.23)$$

Note that $e_x(e_x)^T$ is a matrix of zeroes with a one in row x , column x . β^W is a scalar giving the precision of our measurement W .

7.3.5 Some Learning Policies

We have now set up the foundation for representing our beliefs given a set of observations, and for evaluating different policies for taking measurements. We next face the challenge of deciding how to design policies that determine what we should measure next. Below we list some of the most popular heuristics for determining how to collect information.

Pure Exploitation

Pure exploitation refers to making the decision that appears to be the best given what we know. In this case we would use

$$x^n = \arg \max_{x \in \mathcal{X}} \theta_x^n.$$

Pure Exploration

Pure exploration means simply choosing values of x to try at random.

Epsilon-Greedy Exploration

We can use a mixed strategy where we explore with probability ϵ (known as the exploration rate) and we exploit with probability $1 - \epsilon$. This encourages a policy where we spend more time evaluating what appears to be the best choice.

Boltzmann Exploration

Boltzmann exploration, also known as Gibbs's sampling or soft-max, uses a more nuanced strategy for testing alternatives. Epsilon-greedy will focus a certain amount of attention on the best choice, but the second best is treated no differently than the worst. With Boltzmann exploration we sample measurement x with probability p_x^n given by

$$p_x^n = \frac{\exp(\rho\theta_x^n)}{\sum_{x' \in \mathcal{X}} \exp(\rho\theta_{x'}^n)}.$$

This strategy samples the choices that appear to be better more often, but continues to sample all choices, with a probability that declines with our belief about their attractiveness.

Interval Estimation

Boltzmann exploration improved on the previous heuristics by paying attention to our belief about each alternative. Interval estimation takes this a step further, by also taking into account our confidence in each estimate. Interval estimation proceeds by computing an index v_x^n for each alternative x given by

$$v_x^n = \theta_x^n + z_\alpha \sigma_x^n,$$

where σ_x^n is our estimate of the standard deviation of θ_x^n . When we use an interval estimation policy, we choose the measurement x^n with the highest value of v_x^n .

Interval estimation bases the decision to try an alternative by striking a balance between how well we think an alternative is likely to perform (θ_x^n), and our level of uncertainty about this estimate (σ_x^n). It is unlikely to try an alternative that we think is genuinely poor, but is likely to try alternatives that do not appear to be the best, but about which we have considerable uncertainty.

7.4 THE KNOWLEDGE GRADIENT ALGORITHM FOR DISCRETE ALTERNATIVES

The knowledge gradient is a simple policy that chooses the alternative that produces the greatest value from a single measurement. This can be viewed as a myopic or greedy policy, but there is a growing body of theoretical and empirical evidence that shows that the policy works well, across a wide range of problems. Furthermore it is relatively easy to implement.

7.4.1 The Basic Idea

The idea is simple to explain. If we were to stop now (at iteration n), we would make our choice using

$$x^n = \max_{x' \in \mathcal{X}} \theta_{x'}^n.$$

The value of being in knowledge state $S^n = (\theta^n, \beta^n)$ is then given by

$$V^n(S^n) = \theta_{x^n}^n.$$

If we choose to next measure $x^n = x$ allowing us to observe W_x^{n+1} , this would give us knowledge state $S^{n+1}(x)$, which is a random variable since we do not know W_x^{n+1} . $S^{n+1}(x) = (\theta^{n+1}, \beta^{n+1})$ is computed using equations (12.4) and (12.5). The value of being in state $S^{n+1}(x)$ is given by

$$V^{n+1}(S^{n+1}(x)) = \max_{x' \in \mathcal{X}} \theta_{x'}^{n+1}.$$

We would like to choose x at iteration n that maximizes the expected value of $V^{n+1}(S^{n+1}(x))$, given by

$$\nu_x^{KG,n} = \mathbb{E}[V^{n+1}(S^{n+1}(x)) - V^n(S^n)|S^n]. \quad (7.24)$$

The index $\nu_x^{KG,n}$ is the marginal value of information due to the measurement x , which can be viewed as the gradient with respect to x . The knowledge gradient policy chooses to measure the alternative x that maximizes $\nu_x^{KG,n}$. Alternatively, we may write the knowledge gradient policy as

$$X^{KG,n} = \arg \max_{x \in \mathcal{X}} \mathbb{E}[V^{n+1}(S^{n+1}(x)) - V^n(S^n)|S^n]. \quad (7.25)$$

7.4.2 Computation

If we have finite alternatives with normally distributed and independent beliefs, the knowledge gradient is especially easy to compute. We first present the updating formulas in terms of variances (rather than the precision), given by

$$\begin{aligned} \sigma_x^{2,n} &= ((\sigma_x^{2,n-1})^{-1} + (\sigma_W^2)^{-1})^{-1} \\ &= \frac{(\sigma_x^{2,n-1})}{1 + \sigma_x^{2,n-1}/\sigma_W^2}. \end{aligned} \quad (7.26)$$

To illustrate our problem, let $\sigma_x^{2,n} = 16$ and $\sigma_W = 64$. Then

$$\begin{aligned} \sigma_x^{2,n+1} &= \left(\frac{1}{16} + \frac{1}{64} \right)^{-1} \\ &= (0.078125)^{-1} \\ &= 12.8. \end{aligned}$$

Let $\text{Var}^n(\cdot)$ be the variance of a random variable given what we know about the first n measurements. For example, $\text{Var}^n\theta_x^n = 0$ since, given the first n measurements, θ_x^n is deterministic. Next we need the variance in the *change* in our estimate of θ_x^n as a result of measuring x :

$$\tilde{\sigma}_x^{2,n} = \text{Var}^n[\theta_x^{n+1} - \theta_x^n].$$

Keep in mind that θ_x^{n+1} is random because, after n measurements, we have not yet observed W^{n+1} . We can write $\tilde{\sigma}_x^{2,n}$ in the following ways:

$$\tilde{\sigma}_x^{2,n} = \sigma_x^{2,n} - \sigma_x^{2,n+1} \quad (7.27)$$

$$= \frac{(\sigma_x^{2,n})}{1 + \sigma_W^2/\sigma_x^{2,n}} \quad (7.28)$$

$$= (\beta_x^n)^{-1} - (\beta_x^n + \beta^W)^{-1}. \quad (7.29)$$

For our example this would produce

$$\begin{aligned} \tilde{\sigma}_x^{2,n} &= 16 - 12.8 \\ &= 3.2. \end{aligned}$$

We then compute ζ_x^n , which is given by

$$\zeta_x^n = - \left| \frac{\theta_x^n - \max_{x' \neq x} \theta_{x'}^n}{\tilde{\sigma}_x^n} \right|. \quad (7.30)$$

ζ_x^n is the number of standard deviations from the current estimate of the value of decision x , given by θ_x^n , and the best alternative other than decision x . It is important to recognize that the value of information depends on its ability to change a decision. ζ_x^n captures the distance between the choice we are thinking of measuring and the next best alternative. For our example, assume that $\theta_x^n = 24$ and $\max_{x' \neq x} \theta_{x'}^n = 24.5$. In this case

$$\begin{aligned} \zeta_x^n &= - \left| \frac{24 - 24.5}{\sqrt{3.2}} \right| \\ &= -|-0.27951| \\ &= -0.27951. \end{aligned}$$

We next define

$$f(\zeta) = \zeta \Phi(\zeta) + \phi(\zeta), \quad (7.31)$$

where $\Phi(\zeta)$ and $\phi(\zeta)$ are, respectively, the cumulative standard normal distribution and the standard normal density. That is,

$$\phi(\zeta) = \frac{1}{\sqrt{2\pi}} e^{-\zeta^2}$$

and

$$\Phi(\zeta) = \int_{-\infty}^{\zeta} \phi(x)dx.$$

$\phi(\zeta)$ is, of course, quite easy to compute, while very accurate approximations of $\Phi(\zeta)$ are available in most computer libraries. For our numerical example

$$\begin{aligned} f(\zeta) &= -(0.27951)(0.38993) + 0.38366 \\ &= 0.27467. \end{aligned}$$

Finally, the knowledge gradient is given by

$$v_x^{KG,n} = \tilde{\sigma}_x^n f(\zeta_x^n). \quad (7.32)$$

$$\begin{aligned} &= \sqrt{(3.2)}(0.27467) \\ &= 0.491345. \end{aligned} \quad (7.33)$$

Table 7.1 illustrates the calculations for a problem with five choices. The priors θ^n are shown in the second column, followed by the prior precision. The precision of the measurement is $\beta^W = 1$.

7.4.3 Knowledge Gradient for Correlated Beliefs

A particularly important feature of the knowledge gradient is that it can be adapted to handle the important problem of correlated beliefs. Correlated beliefs arise when we are maximizing a continuous surface (nearby points will be correlated) or choosing subsets (e.g., the location of a set of facilities) that produce correlations when subsets share common elements. If we are trying to estimate a continuous function, we might assume that the covariance matrix satisfies

$$\text{Cov}(x, x') \propto e^{-\rho \|x-x'\|},$$

where ρ captures the relationship between neighboring points. If x is a vector of 0's and 1's indicating elements in a subset, the covariance might be proportional to the number of 1's that are in common between two choices.

Table 7.1 Calculations illustrating the knowledge gradient index

Choice	μ^n	β^n	β^{n+1}	$\tilde{\sigma}$	$\max_{x' \neq x} x'$	ζ	$f(\zeta)$	v_x^{KG}
1	20.0	0.0625	1.0625	3.8806	28	-2.0616	0.0072	0.0279
2	22.0	0.1111	1.1111	2.8460	28	-2.1082	0.0063	0.0180
3	24.0	0.0400	1.0400	4.9029	28	-0.8158	0.1169	0.5731
4	26.0	0.1111	1.1111	2.8460	28	-0.7027	0.1422	0.4048
5	28.0	0.0625	1.0625	3.8806	26	-0.5154	0.1931	0.7493

The covariance function (or matrix) is of central importance to the successful use of the knowledge gradient. Not surprisingly, the procedure works the best when the problem has structure that allows us to determine the covariance function in advance. However, it is possible to estimate the covariance function from data as the algorithm progresses.

There is a more compact way of updating our estimate of θ^n in the presence of correlated beliefs. Let $\lambda^W = \sigma_W^2 = 1/\beta^W$. Let $\Sigma^{n+1}(x)$ be the updated covariance matrix given that we have chosen to measure alternative x , and let $\tilde{\Sigma}^n(x)$ be the change in the covariance matrix due to evaluating x , which is given by

$$\tilde{\Sigma}^n = \Sigma^n - \Sigma^{n+1}, \quad (7.34)$$

$$\tilde{\Sigma}(x) = \frac{\Sigma^n e_x (e_x)^T \Sigma^n}{\sqrt{\Sigma_{xx}^n + \lambda^W}}. \quad (7.35)$$

Now define the vector $\tilde{\sigma}^n(x)$, which gives the standard deviation of the change in our belief due to measuring x and is given by

$$\tilde{\sigma}^n(x) = \frac{\Sigma^n e_x}{\sqrt{\Sigma_{xx}^n + \lambda^W}}. \quad (7.36)$$

Let $\tilde{\sigma}_i(\Sigma, x)$ be the component $(e_i)^T \tilde{\sigma}(x)$ of the vector $\tilde{\sigma}(x)$, and let $\text{Var}^n(\cdot)$ be the variance given what we know after n measurements. We note that if we measure alternative x^n , then

$$\begin{aligned} \text{Var}^n [W^{n+1} - \theta^n] &= \text{Var}^n [\theta_{x^n} + \varepsilon^{n+1}] \\ &= \Sigma_{x^n x^n}^n + \lambda^W. \end{aligned} \quad (7.37)$$

Next define the random variable

$$Z^{n+1} = \frac{W^{n+1} - \theta^n}{\sqrt{\text{Var}^n [W^{n+1} - \theta^n]}}.$$

We can now rewrite (12.4) for updating our beliefs about the mean as

$$\theta^{n+1} = \theta^n + \tilde{\sigma}(x^n) Z^{n+1}. \quad (7.38)$$

The knowledge gradient policy for correlated beliefs is computed using

$$\begin{aligned} X^{KG}(s) &= \arg \max_x \mathbf{E} \left[\max_i \theta_i^{n+1} \mid S^n = s, x^n = x \right] \\ &= \arg \max_x \mathbf{E} \left[\max_i \theta_i^n + \tilde{\sigma}_i(x^n) Z^{n+1} \mid S^n, x^n = x \right], \end{aligned} \quad (7.39)$$

where Z is a scalar, standard normal random variable. The problem with this expression is that the expectation is harder to compute, but a simple algorithm can be used to compute the expectation exactly. We start by defining

$$h(\theta^n, \tilde{\sigma}(x)) = \mathbf{E} \left[\max_i \theta_i^n + \tilde{\sigma}_i(x^n) Z^{n+1} \mid S^n, x^n = x \right]. \quad (7.40)$$

Substituting (7.40) into (7.39) gives us

$$X^{KG}(s) = \arg \max_x h(\theta^n, \tilde{\sigma}(x)). \quad (7.41)$$

Let $h(a, b) = \mathbf{E} \max_i a_i + b_i Z$, where $a_i = \theta_i^n$, $b_i = \tilde{\sigma}_i(x^n)$, and Z is our standard normal deviate. Both a and b are M -dimensional vectors. Sort the elements b_i so that $b_1 \leq b_2 \leq \dots$ so that we get a sequence of lines with increasing slopes, as depicted in Figure 7.1. There are ranges for z over which a particular line may dominate the other lines, and some lines may be dominated all the time (e.g., alternative 3).

We need to identify and eliminate the dominated alternatives. To do this, we start by finding the points where the lines intersect. The lines $a_i + b_i z$ and $a_{i+1} + b_{i+1} z$ intersect at

$$z = c_i = \frac{a_i - a_{i+1}}{b_{i+1} - b_i}.$$

For the moment we are going to assume that $b_{i+1} > b_i$. If $c_{i-1} < c_i < c_{i+1}$, then we can find a range for z over which a particular choice dominates, as depicted in Figure 7.1. A line is dominated when $c_{i+1} < c_i$, at which point the line corresponding to $i+1$ is dropped from the set. Once the sequence c_i has been found, we can compute (7.39) using

$$h(a, b) = \sum_{i=1}^M (b_{i+1} - b_i) f(-|c_i|),$$

where as before, $f(z) = z\Phi(z) + \phi(z)$. Of course, the summation has to be adjusted to skip any choices i that were found to be dominated.

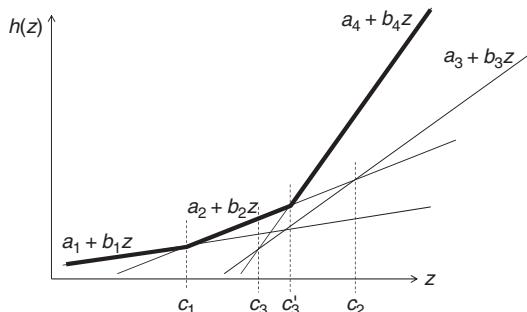


Figure 7.1 Regions of z over which different choices dominate. Choice 3 is always dominated.

It is important to recognize that there is more to incorporating correlated beliefs than simply using the covariances when we update our beliefs after a measurement. With this procedure, we anticipate the updating before we even make a measurement.

The ability to handle correlated beliefs in the choice of what measurement to make is an important feature that has been overlooked in other procedures. It makes it possible to make sensible choices when our measurement budget is much smaller than the number of potential choices we have to evaluate. There are, of course, computational implications. It is relatively easy to handle dozens or hundreds of measurement alternatives, but as a result of the matrix calculations, it becomes expensive to handle problems where the number of potential choices is in the thousands. If this is the case, then it is likely the problem has special structure. For example, we might be discretizing a p -dimensional parameter surface. If this is the case, then it may make sense to consider the adaptation of the knowledge gradient for problems where the belief structure can be represented using a parametric model.

7.4.4 The Knowledge Gradient for Parametric Belief Models

There are many applications (especially in approximate dynamic programming) where we are trying to learn a function that we are approximating using linear regression (the simplest form of parametric model). We can do this fairly easily from our calculation of the knowledge gradient for correlated beliefs using a lookup table belief structure. A limitation of this strategy is that it works with the covariance matrix, which increases with the square of the number of points to measure. Using linear regression, we reduce this problem dealing with a matrix that is $|\mathcal{F}| \times |\mathcal{F}|$, where \mathcal{F} is the number of features in our linear model.

We start by recognizing that the knowledge gradient requires computing the function $h(a, b(j))$, where a is a vector with element $a_i = \theta_i^n$ giving the estimate of the value of the i th alternative, and $b(j)$ is a vector with element $b_i(j) = \tilde{\sigma}_i^n(j)$, which is the conditional variance of the change in θ_i^{n+1} from measuring alternative j . The function $h(a, b)$ is given by

$$h(a, b(j)) = \sum_{i=1}^{M-1} (b_{i+1}(j) - b_i(j)) f(-|c_i(j)|) \quad (7.42)$$

where

$$c_i(j) = \frac{a_i - a_{i+1}}{b_{i+1}(j) - b_i(j)}.$$

The major computational challenges arises when computing $\tilde{\sigma}^n(j)$, which is a vector giving the change in the variance of each alternative i if we choose to measure alternative j . From our presentation of the knowledge gradient for correlated beliefs, we found that

$$\tilde{\Sigma}^n(j) = \frac{\Sigma^n e_j (e_j)^T \Sigma^n}{\sqrt{\Sigma_{jj}^n + \lambda_\epsilon^W}},$$

which gives us the change in the covariance matrix from measuring an alternative j . The matrix $\tilde{\Sigma}^n(j)$ is $M \times M$, which is quite large, but we only need the j th row. We compute the standard deviation of the elements of the j th row using

$$\tilde{\sigma}^n(j) = \frac{\Sigma^n e_j}{\sqrt{\Sigma_{jj}^n + \sigma_\epsilon^2}}. \quad (7.43)$$

Now is where we use the structure of our linear regression model. If there are a large number of choices, the matrix $\tilde{\Sigma}^n$ will be too expensive to compute. However, it is possible to work with the covariance matrix Σ^θ of the regression vector θ , which is dimensioned by the number of parameters in our regression function. To compute Σ^θ , we need to set up some variables. First let

$$x^n = \begin{pmatrix} x_1^n \\ x_2^n \\ \vdots \\ x_K^n \end{pmatrix}$$

be a vector of K independent variables corresponding to the n th observation. Next create the matrix X^n that consists of all of our measurements:

$$X^n = \begin{pmatrix} x_1^1 & x_2^1 & \cdots & x_K^1 \\ x_1^2 & x_2^2 & \cdots & x_K^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^n & x_2^n & \cdots & x_K^n \end{pmatrix}.$$

We can use X^n to compute our regression vector by computing

$$\theta^n = [(X^n)^T X^n]^{-1} (X^n)^T Y^n. \quad (7.44)$$

We do not actually use this equation directly, but we defer to Section 9.3.1 for a more in-depth discussion of linear regression models, and methods for recursively computing θ^n that avoid the use of matrix inversions. We have yet a different purpose for the moment. We can use X^n to compute a compact form for the covariance matrix $\Sigma^{\theta,n}$ of θ . With a bit of algebra, it is possible to show that after n observations

$$\begin{aligned} \Sigma^{\theta,n} &= [(X^n)^T X^n]^{-1} (X^n)^T X^n [(X^n)^T X^n]^{-1} \sigma_\epsilon^2 \\ &= [(X^n)^T X^n]^{-1} \sigma_\epsilon^2, \end{aligned}$$

where σ_ϵ^2 is the variance of our measurements. Of course, this seems to imply that we still have to invert $(X^n)^T X^n$. In Chapter 9 (Section 9.3.1) we provide equations for computing this update recursively.

Knowing that we can compute $\Sigma^{\theta,n}$ in an efficient way, we can quickly compute $\tilde{\sigma}^n$. Note that we do not need the entire matrix. We can show that

$$\Sigma^n = X \Sigma^{\theta,n} X^T,$$

Let X_j be the j th row of X . Then

$$\tilde{\sigma}^{2,n}(j) = X_j \Sigma^{\theta,n} X^T.$$

We still have to multiply the $K \times K$ -dimensional matrix $\Sigma^{\theta,n}$ times the $K \times M$ -dimensional matrix X^T , after which we have to compute equation (7.42) to find the knowledge gradient for each alternative. Even for problems with tens of thousands of alternatives, this can be executed in a few seconds. Now that we have an efficient way of computing $\tilde{\sigma}^n(j)$, we can apply the knowledge gradient for correlated beliefs described in Section 7.4.3.

7.5 SIMULATION OPTIMIZATION

A subcommunity within the larger stochastic search community goes by the name *simulation optimization*. This community also works on problems that can be described in the form of $\min_x \mathbb{E} F(x, W)$, but the context typically arises when x represents the design of a physical system, which is then evaluated (noisily) using discrete-event simulation. The number of potential designs \mathcal{X} is typically in the range of 5 to perhaps 100. We can evaluate a design x more accurately by increasing the run length n_x of the simulation, where n_x might be the number of time periods, the CPU time, or the number of discrete events (e.g., customer arrivals). We assume that we have a global budget N , and we need to find n_x for each x so that

$$\sum_{x \in \mathcal{X}} n_x = N.$$

For our purposes there is no difference between a potential design of a physical system and a policy. Searching for the best design and searching for the best policy is, algorithmically speaking, identical as long as the set of policies is not too large.

We can tackle this problem using the strategies described above (e.g., the knowledge gradient) if we break up the problem into a series of short simulations (e.g., 1 time step or 1 unit of CPU time). Then at each iteration we have to decide which design x to measure, contributing to our estimate θ_x^n for design x . The problem with this strategy is that it ignores the startup time for a simulation. It is much easier to set a run length n_x for each design x , and then run the entire simulation to obtain an estimate of θ_x .

The simulation optimization problem is traditionally formulated in a frequentist framework, reflecting the lack of prior information about the alternatives. A standard strategy is to run the experiments in two stages. In the first stage a sample

n^0 is collected for each design. The information from this first stage is used to develop an estimate of the value of each design. We might learn, for example, that certain designs seem to lack any promise at all, while other designs may seem more interesting. Rather than spreading our budget across all the designs, we can use this information to focus our computing budget across the designs that offer the greatest potential.

From the information at the initial stage, the challenge is to find an allocation n_x that optimizes some objective subject to the budget constraint N . There are several objectives that we might consider:

- *Expected opportunity cost.* Above we described the stochastic search problem in the form of solving $\min_x \mathbb{E}F(x, W)$. If x^* is the optimal solution, consider the equivalent optimization problem $\min_x \mathbb{E}(F(x, W) - F(x^*, W))$, which gives us the cost over the optimal solution. This is called the expected opportunity cost.
- *Subset selection.* Ultimately our goal is to pick the best design. Imagine that we are willing to choose a subset of designs S , and we would like to ensure that $P(x^* \in S) \geq 1 - \alpha$, where $1/|\mathcal{X}| < 1 - \alpha < 1$. Of course, it would be ideal if $|S| = 1$ or, failing this, as small as possible. Let θ_x^n be our estimate of the value of x after n measurements, and assume that all measurements have a constant and known variance σ . We include x in the subset if

$$\theta_x^n \geq \max_{x' \neq x} \theta_{x'}^n - h\sigma \sqrt{\frac{2}{n}}.$$

The parameter h is the $1 - \alpha$ quantile of the random variable $\max_i Z_i^n$, where Z_i^n is given by

$$Z_i^n = \frac{(\theta_i^n - \theta_x^n) - (\mu_i - \mu_x)}{\sigma \sqrt{2/n}}.$$

- *Indifference zone selection.* The goal is to find with probability $1 - \alpha$ the system x such that $\mu_x \geq \mu_{x^*} - \delta$. In other words, we are indifferent about selecting systems that are within δ of the best. The procedure samples each alternative n times when

$$n = \frac{\lceil 2h^2\sigma^2 \rceil}{\delta^2},$$

where $\lceil z \rceil$ means to round up, and h is chosen as it was for the subset selection objective. Use these samples to compute θ_x^n , and then choose the best x . Note that the goal here is to ensure that we meet our goal with a specified probability, rather than to work within a specific budget constraint.

The expected opportunity cost objective focuses on estimating the value of each alternative, while the other two options focus on finding the best or near-best, which is a form of ordinal optimization.

7.5.1 An Indifference Zone Algorithm

There are a number of algorithms that have been suggested to search for the best design using the indifference zone criterion, which is one of the most popular in the simulation optimization community. The algorithm in Figure 7.2 summarizes a method that successively reduces a set of candidates at each iteration, focusing the evaluation effort on a smaller and smaller set of alternatives. The method (under some assumptions) uses a user-specified indifference zone of δ . Of course, as δ is decreased, the computational requirements increase.

Step 0. Initialization

Step 0a. Select the probability of correct selection $1 - \alpha$, indifference zone parameter δ and initial sample size $n_0 \geq 2$. Set $r = n_0$.

Step 0b. Compute

$$\eta = \frac{1}{2} \left[\left(\frac{2\alpha}{k-1} \right)^{-2/(n_0-1)} - 1 \right].$$

Step 0c. Set $h^2 = 2\eta(n_0 - 1)$.

Step 0d. Set $\mathcal{X}^0 = \mathcal{X}$ as the set of systems in contention.

Step 0e. Obtain samples W_x^m , $m = 1, \dots, n_0$ of each $x \in \mathcal{X}^0$, and let θ_x^0 be the resulting sample means for each alternative computed using

$$\theta_x^0 = \frac{1}{n_0} \sum_{m=1}^{n_0} W_x^m.$$

Compute the sample variances for each pair using

$$\hat{\sigma}_{xx'}^2 = \frac{1}{n_0 - 1} \sum_{m=1}^{n_0} [W_x^m - W_{x'}^m - (\theta_x^0 - \theta_{x'}^0)]^2.$$

Step 0f. Set $n = 1$.

Step 1. Compute

$$W_{xx'}(r) = \max \left\{ 0, \frac{\delta}{2r} \left(\frac{h^2 \hat{\sigma}_{xx'}^2}{\delta^2} - r \right) \right\}.$$

Step 2. Refine the eligible set using

$$\mathcal{X}^n = \{x : x \in \mathcal{X}^{n-1} \text{ and } \theta_x^n \geq \theta_{x'}^n - W_{xx'}(r), x' \neq x\}.$$

Step 3. If $|\mathcal{X}^n| = 1$, stop and select the best element in \mathcal{X}^n . Otherwise, perform an additional sample W_x^{n+1} of each $x \in \mathcal{X}^n$, set $r = r + 1$, and return to step 1.

Figure 7.2 Policy search algorithm using the indifference zone criterion, due to Kim and Nelson (2001).

7.5.2 Optimal Computing Budget Allocation

The value of the indifference zone strategy is that it focuses on achieving a specific level of solution quality while being constrained by a specific budget. However, it is often the case that we are trying to do the best we can within a specific computing budget. For this purpose a line of research has evolved under the name *optimal computing budget allocation*, or OCBA.

Figure 7.3 illustrates a typical version of an OCBA algorithm. The algorithm proceeds by taking an initial sample $N_x^0 = n_0$ of each alternative $x \in \mathcal{X}$, which means we use $B^0 = Mn_0$ measurements from our budget B . Letting $M = |\mathcal{X}|$, we divide the remaining budget of measurements $B - B^0$ into equal increments of size Δ so that we do $N = (B - Mn_0)\Delta$ iterations.

Step 0. Initialization.

Step 0a. Given a computing budget B , let n^0 be the initial sample size for each of the $M = |\mathcal{X}|$ alternatives. Divide the remaining budget $B - Mn_0$ into increments so that $N = (B - Mn_0)/\delta$ is an integer.

Step 0b. Obtain samples W_x^m , $m = 1, \dots, n_0$ samples of each $x \in \mathcal{X}$.

Step 0c. Initialize $N_x^1 = n_0$ for all $x \in \mathcal{X}$.

Step 0d. Initialize $n = 1$.

Step 1. Compute

$$\theta_x^n = \frac{1}{N_x^n} \sum_{m=1}^{N_x^n} W_x^m.$$

Compute the sample variances for each alternative x using

$$\hat{\sigma}_x^{2,n} = \frac{1}{N_x^n - 1} \sum_{m=1}^{N_x^n} (W_x^m - \theta_x^n)^2.$$

Step 2. Let $x^n = \arg \max_{x \in \mathcal{X}} \theta_x^n$.

Step 3. Increase the computing budget by Δ and calculate the new allocation $N_1^{n+1}, \dots, N_M^{n+1}$ so that

$$\frac{N_x^{n+1}}{N_{x'}^{n+1}} = \frac{\hat{\sigma}_x^{2,n}/(\theta_{x^n}^n - \theta_x^n)^2}{\hat{\sigma}_{x'}^{2,n}/(\theta_{x^n}^n - \theta_{x'}^n)^2}, \quad x \neq x' \neq x^n,$$

$$N_{x^n}^{n+1} = \hat{\sigma}_{x^n}^n \sqrt{\sum_{i=1, i \neq x^n}^M \left(\frac{N_{x^n}^{n+1}}{\hat{\sigma}_i^n} \right)^2}.$$

Step 4. Perform $\max(N_x^{n+1} - N_x^n, 0)$ additional simulations for each alternative x .

Step 5. Set $n = n + 1$. If $\sum_{x \in \mathcal{X}} N_x^n < B$, go to step 1.

Step 6. Return $x^n = \operatorname{argmax}_{x \in \mathcal{X}} \theta_x^n$.

Figure 7.3 Optimal computing budget allocation procedure.

After n iterations, we assume that we have measured alternative x N_x^n times, and let W_x^m be the m th observation of x , for $m = 1, \dots, N_x^n$. The updated estimate of the value of each alternative x is given by

$$\theta_x^n = \frac{1}{N_x^n} \sum_{m=1}^{N_x^n} W_x^m.$$

Let $x^n = \arg \max \theta_x^n$ be the current best option.

After using Mn_0 observations from our budget, at each iteration we increase our allowed budget by $B^n = B^{n-1} + \Delta$ until we reach $B^N = B$. After each increment the allocation N_x^n , $x \in \mathcal{X}$ is recomputed using

$$\frac{N_x^{n+1}}{N_{x'}^{n+1}} = \frac{\hat{\sigma}_{x^n}^{2,n}/(\theta_{x^n}^n - \theta_x^n)^2}{\hat{\sigma}_{x'}^{2,n}/(\theta_{x^n}^n - \theta_{x'}^n)^2}, \quad x \neq x' \neq x^n, \quad (7.45)$$

$$N_{x^n}^{n+1} = \hat{\sigma}_{x^n}^n \sqrt{\sum_{i=1, i \neq x^n}^M \left(\frac{N_i^{n+1}}{\hat{\sigma}_i^n} \right)^2}. \quad (7.46)$$

We use equations (7.45)–(7.46) to produce an allocation N_x^n such that $\sum_x N_x^n = B^n$. Note that after increasing the budget, it is not guaranteed that $N_x^n \geq N_x^{n-1}$ for some x . So, if this is the case, we do not measure these alternatives at all in the next iteration. We can solve these equations by writing each N_x^n in terms of some fixed alternative (other than x^n), such as N_1^n (assuming $x^n \neq 1$). After writing N_x^n as a function of N_1^n for all x , we determine N_1^n so that $\sum N_x^n \approx B^n$ (within rounding).

The complete algorithm is summarized in Figure 7.3.

7.6 WHY DOES IT WORK?**

Stochastic approximation methods have a rich history starting with the seminal paper Robbins and Monro (1951) and followed by Blum (1954b) and Dvoretzky (1956). The serious reader should see Kushner and Yin (1997) for a modern treatment of the subject. Wasan (1969) is also a useful reference for fundamental results on stochastic convergence theory. A separate line of investigation was undertaken by researchers in eastern European community focusing on constrained stochastic optimization problems (Gaivoronski, 1988; Ermoliev, 1988; Ruszczynski 1980, 1987). This work is critical to our fundamental understanding of Monte Carlo based stochastic learning methods.

The theory behind these proofs is fairly deep and requires some mathematical maturity. For pedagogical reasons we start in Section 7.6.1 with some probabilistic preliminaries, after which Section 7.6.2 presents one of the original proofs that is relatively more accessible and that provides the basis for the universal requirements that stepsizes must satisfy for theoretical proofs. Section 7.6.3 provides a more modern proof based on the theory of martingales.

7.6.1 Some Probabilistic Preliminaries

The goal in this section is to prove that stochastic algorithms work. But what does this mean? The solution \bar{x}^n at iteration n is a random variable. Its value depends on the sequence of sample realizations of the random variables over iterations 1 to n . If $\omega = (\omega^1, \omega^2, \dots, \omega^n, \dots)$ represents the sample path that we are following, we can ask what is happening to the limit $\lim_{n \rightarrow \infty} \bar{x}^n(\omega)$. If the limit is x^* , does x^* depend on the sample path ω ?

In the proofs below we show that the algorithms converge *almost surely*. What this means is that

$$\lim_{n \rightarrow \infty} \bar{x}^n(\omega) = x^*$$

for all $\omega \in \Omega$ that can occur with positive measure. This is the same as saying that we reach x^* with probability 1. Here x^* is a deterministic quantity that does not depend on the sample path. Because of the restriction $p(\omega) > 0$, we accept that in theory there could exist a sample outcome that can never occur and would produce a path that converges to some other point. As a result we say that the convergence is “almost sure,” which is universally abbreviated as “a.s.” Almost sure convergence establishes the core theoretical property that the algorithm will eventually settle in on a single point. This is an important property for an algorithm, but it says nothing about the rate of convergence (an important issue in approximate dynamic programming).

Let $x \in \Re^n$. At each iteration n , we sample some random variables to compute the function (and its gradient). The sample realizations are denoted by ω^n . We let $\omega = (\omega^1, \omega^2, \dots)$ be a realization of all the random variables over all iterations. Let Ω be the set of all possible realizations of ω , and let \mathfrak{F} be the σ -algebra on Ω (i.e., the set of all possible events that can be defined using Ω). We need the concept of the history up through iteration n . Let

H^n = a random variable giving the history of all random variables up through iteration n .

A sample realization of H^n would be

$$\begin{aligned} h^n &= H^n(\omega) \\ &= (\omega^1, \omega^2, \dots, \omega^n). \end{aligned}$$

We could then let Ω^n be the set of all outcomes of the history (that is, $h^n \in H^n$) and let \mathcal{H}^n be the σ -algebra on Ω^n (which is the set of all events, including their complements and unions, defined using the outcomes in Ω^n). Although we could do this, this is not the convention followed in the probability community. So we define instead a sequence of σ -algebras $\mathfrak{F}^1, \mathfrak{F}^2, \dots, \mathfrak{F}^n$ as the sequence of σ -algebras on Ω that can be generated as we have access to the information through the first 1, 2, ..., n iterations, respectively. What does this mean? Consider two outcomes $\omega \neq \omega'$ for which $H^n(\omega) = H^n(\omega')$. If this is the case, then any event in

\mathfrak{F}^n that includes ω must also include ω' . If we say that a function is \mathfrak{F}^n -measurable, then this means that it must be defined in terms of the events in \mathfrak{F}^n , which is in turn equivalent to saying that we cannot be using any information from iterations $n+1, n+2, \dots$.

We would say, then, that we have a standard probability space $(\Omega, \mathfrak{F}, \mathcal{P})$, where $\omega \in \Omega$ represents an elementary outcome, \mathfrak{F} is the σ -algebra on Ω , and \mathcal{P} is a probability measure on (Ω, \mathfrak{F}) . Since our information is revealed iteration by iteration, we would also then say that we have an increasing set of σ -algebras $\mathfrak{F}^1 \subseteq \mathfrak{F}^2 \subseteq \dots \subseteq \mathfrak{F}^n$ (which is the same as saying that \mathfrak{F}^n is a filtration).

7.6.2 An Older Proof

Enough with probabilistic preliminaries. Let $F(x, \omega)$ be a \mathfrak{F} -measurable function. We wish to solve the unconstrained problem

$$\max_x \mathbb{E}\{F(x, \omega)\} \quad (7.47)$$

with x^* being the optimal solution. Let $g(x, \omega)$ be a stochastic ascent vector that satisfies

$$g(x, \omega)^T \nabla F(x, \omega) \geq 0. \quad (7.48)$$

For many problems, the most natural ascent vector is the gradient itself

$$g(x, \omega) = \nabla F(x, \omega), \quad (7.49)$$

which clearly satisfies (7.48).

We assume that $F(x) = \mathbb{E}\{F(x, \omega)\}$ is continuously differentiable and concave, with bounded first and second derivatives so that for finite M

$$-M \leq g(x, \omega)^T \nabla^2 F(x) g(x, \omega) \leq M. \quad (7.50)$$

For notational simplicity, we are letting $F(x)$ be the expectation, while $F(x, \omega)$ is a sample realization. A stochastic gradient algorithm (sometimes called a stochastic approximation method) is given by

$$\bar{x}^n = \bar{x}^{n-1} + \alpha_{n-1} g(\bar{x}^{n-1}, \omega^n). \quad (7.51)$$

We first prove our result using the proof technique of Blum (1954b) that generalized the original stochastic approximation procedure proposed by Robbins and Monro (1951) to multidimensional problems. This approach does not depend on more advanced concepts such as martingales and, as a result, is accessible to a broader audience. This proof helps the reader understand the basis for the conditions $\sum_{n=0}^{\infty} \alpha_n = \infty$ and $\sum_{n=0}^{\infty} (\alpha_n)^2 < \infty$ that are required of all stochastic approximation algorithms.

We make the following (standard) assumptions on stepsizes

$$\alpha_n \geq 0, \quad n \geq 0, \quad (7.52)$$

$$\sum_{n=0}^{\infty} \alpha_n = \infty, \quad (7.53)$$

$$\sum_{n=0}^{\infty} (\alpha_n)^2 < \infty. \quad (7.54)$$

We want to show that under suitable assumptions, the sequence generated by (7.51) converges to an optimal solution. That is, we want to show that

$$\lim_{n \rightarrow \infty} \bar{x}^n = x^* \quad \text{a.s.} \quad (7.55)$$

We now use the mean value theorem, which says that for any continuously differentiable function $F(x)$, there exists a parameter $0 \leq \eta \leq 1$ that satisfies

$$F(x) = F(\bar{x}^0) + \nabla F(\bar{x}^0 + \eta(x - \bar{x}^0))^T(x - \bar{x}^0). \quad (7.56)$$

This is the first-order Taylor series approximation. The second-order version takes the form

$$F(x) = F(\bar{x}^0) + \nabla F(\bar{x}^0)^T(x - \bar{x}^0) + \frac{1}{2}(x - \bar{x}^0)\nabla^2 F(\bar{x}^0 + \eta(x - \bar{x}^0))(x - \bar{x}^0) \quad (7.57)$$

for some $0 \leq \eta \leq 1$. We use the second-order version. Replace \bar{x}^0 with \bar{x}^{n-1} , and replace x with \bar{x}^n . Also we can simplify our notation by using

$$g^n = g(\bar{x}^{n-1}, \omega^n). \quad (7.58)$$

This means that

$$\begin{aligned} x - \bar{x}^0 &= \bar{x}^n - \bar{x}^{n-1} \\ &= (\bar{x}^{n-1} + \alpha_{n-1}g^n) - \bar{x}^{n-1} \\ &= \alpha_{n-1}g^n. \end{aligned}$$

From our stochastic gradient algorithm (7.51), we may write

$$\begin{aligned} F(\bar{x}^n, \omega^n) &= F(\bar{x}^{n-1} + \alpha_{n-1}g^n, \omega^n) \\ &= F(\bar{x}^{n-1}, \omega^n) + \nabla F(\bar{x}^{n-1}, \omega^n)^T(\alpha_{n-1}g^n) \\ &\quad + \frac{1}{2}(\alpha_{n-1}g^n)^T \nabla^2 F(\bar{x}^{n-1} + \gamma(\alpha_{n-1}g^n))(\alpha_{n-1}g^n). \end{aligned} \quad (7.59)$$

It is now time to use a *standard mathematician's trick*. We sum both sides of (7.59) to get

$$\begin{aligned} \sum_{n=1}^N F(\bar{x}^n, \omega^n) &= \sum_{n=1}^N F(\bar{x}^{n-1}, \omega^n) + \sum_{n=1}^N \nabla F(\bar{x}^{n-1}, \omega^n)^T (\alpha_{n-1} g^n) \\ &\quad + \frac{1}{2} \sum_{n=1}^N (\alpha_{n-1} g^n)^T \nabla^2 F(\bar{x}^{n-1} + x(\alpha_{n-1} g^n)) (\alpha_{n-1} g^n). \end{aligned} \quad (7.60)$$

Note that the terms $F(\bar{x}^n), n = 2, 3, \dots, N$ appear on both sides of (7.60). We can cancel these. We then use our lower bound on the quadratic term (7.50) to write

$$F(\bar{x}^N, \omega^N) \geq F(\bar{x}^0, \omega^1) + \sum_{n=1}^N \nabla F(\bar{x}^{n-1}, \omega^n)^T (\alpha_{n-1} g^n) + \frac{1}{2} \sum_{n=1}^N (\alpha_{n-1})^2 (-M). \quad (7.61)$$

We now want to take the limit of both sides of (7.61) as $N \rightarrow \infty$. In doing so, we want to show that everything must be bounded. We know that $F(\bar{x}^N)$ is bounded (*almost surely*) because we assumed that the original function was bounded. We next use the assumption 7.54 that the infinite sum of the squares of the stepsizes is also bounded to conclude that the rightmost term in (7.61) is bounded. Finally, we use (7.48) to claim that all the terms in the remaining summation ($\sum_{n=1}^N \nabla F(\bar{x}^{n-1}, \omega^n)(\alpha_{n-1} g^n)$) are positive. That means that this term is also bounded (from both above and below).

What do we get with all this boundedness? Well, if

$$\sum_{n=1}^{\infty} \alpha_{n-1} \nabla F(\bar{x}^{n-1}, \omega^n)^T g^n < \infty \quad \text{a.s.} \quad (7.62)$$

and (from (11.15))

$$\sum_{n=1}^{\infty} \alpha_{n-1} = \infty. \quad (7.63)$$

We can conclude that

$$\sum_{n=1}^{\infty} \nabla F(\bar{x}^{n-1}, \omega^n)^T g^n < \infty. \quad (7.64)$$

Since all the terms in (7.64) are positive, they must go to zero. (Remember, everything here is true *almost surely*; after a while, it gets a little boring to keep saying *almost surely* every time. It is a little like reading Chinese fortune cookies and adding the automatic phrase "under the sheets" at the end of every fortune.)

We are basically done except for some relatively difficult (albeit important if you are ever going to do your own proofs) technical points to really prove convergence.

At this point we would use technical conditions on the properties of our ascent vector g^n to argue that if $\nabla F(\bar{x}^{n-1}, \omega^n)^T g^n \rightarrow 0$, then $\nabla F(\bar{x}^{n-1}, \omega^n) \rightarrow 0$ (it is okay if g^n goes to zero as $F(\bar{x}^{n-1}, \omega^n)$ goes to zero, but it cannot go to zero too quickly).

This proof was first proposed in the early 1950s by Robbins and Monro and became the basis of a large area of investigation under the heading of stochastic approximation methods. A separate community, growing out of the Soviet literature in the 1960s, addressed these problems under the name of stochastic gradient (or stochastic quasi-gradient) methods. More modern proofs are based on the use of martingale processes, which do not start with the mean value theorem and do not (always) need the differentiability conditions that this approach needs.

Our presentation does, however, help present several key ideas in most proofs of this type. First, concepts of almost sure convergence are virtually standard. Second, it is common to set up equations such as (7.59) and then take a finite sum as in (7.60) using the alternating terms in the sum to cancel all but the first and last elements of the sequence of some function (in our case, $F(\bar{x}^{n-1}, \omega^n)$). We then establish the boundedness of this expression as $N \rightarrow \infty$, which will require the assumption that $\sum_{n=1}^{\infty} (\alpha_{n-1})^2 < \infty$. Then the assumption $\sum_{n=1}^{\infty} \alpha_{n-1} = \infty$ is used to show that if the remaining sum is bounded, then its terms must go to zero.

More modern proofs will use functions other than $F(\bar{x})$. Popular is the introduction of so-called Lyapunov functions, which are artificial functions that provide a measure of optimality. These functions are constructed for the purpose of the proof and play no role in the algorithm itself. For example, we might let $T^n = \|\bar{x}^n - x^*\|$ be the distance between our current solution \bar{x}^n and the optimal solution. We will then try to show that T^n is suitably reduced to prove convergence. Since we do not know x^* , this is not a function we can actually measure, but it can be a useful device for proving that the algorithm actually converges.

It is important to realize that stochastic gradient algorithms of all forms do not guarantee an improvement in the objective function from one iteration to the next. First, a sample gradient g^n may represent an appropriate ascent vector for a sample of the function $F(\bar{x}^{n-1}, \omega^n)$ but not for its expectation. In other words, randomness means that we may go in the wrong direction at any point in time. Second, our use of a nonoptimizing stepsize, such as $\alpha_{n-1} = 1/n$, means that even with a good ascent vector, we may step too far and actually end up with a lower value.

7.6.3 A More Modern Proof

Since the original work by Robbins and Monro, more powerful proof techniques have evolved. Below we illustrate a basic martingale proof of convergence. The concepts are somewhat more advanced, but the proof is more elegant and requires milder conditions. A significant generalization is that we no longer require that our function be differentiable (which our first proof required). For large classes of resource allocation problems, this is a significant improvement.

First, just what is a martingale? Let $\omega_1, \omega_2, \dots, \omega_t$ be a set of exogenous random outcomes, and let $h_t = H_t(\omega) = (\omega_1, \omega_2, \dots, \omega_t)$ represent the history of the process up to time t . We also let \mathfrak{F}_t be the σ -algebra on Ω generated by

H_t . Further let U_t be a function that depends on h_t (we would say that U_t is a \mathfrak{F}_t -measurable function), and bounded ($\mathbb{E}|U_t| < \infty$, $\forall t \geq 0$). This means that if we know h_t , then we know U_t deterministically (needless to say, if we only know h_t , then U_{t+1} is still a random variable). We additionally assume that our function satisfies

$$\mathbb{E}[U_{t+1}|\mathfrak{F}_t] = U_t.$$

If this is the case, then we say that U_t is a *martingale*. Alternatively, if

$$\mathbb{E}[U_{t+1}|\mathfrak{F}_t] \leq U_t \quad (7.65)$$

then we say that U_t is a *supermartingale*. If U_t is a supermartingale, then it has the property that it drifts downward, usually to some limit point U^* . What is important is that it only drifts downward in expectation. That is, it could easily be the case that $U_{t+1} > U_t$ for specific outcomes. This captures the behavior of stochastic approximation algorithms. Properly designed, they provide solutions that improve on average, but where from one iteration to another the results can actually get worse.

Finally, we assume that $U_t \geq 0$. If this is the case, we have a sequence U_t that drifts downward but that cannot go below zero. Not surprisingly, we obtain the following key result:

Theorem 7.6.1 *Let U_t be a positive supermartingale. Then U_t converges to a finite random variable U^* a.s.*

So what does this mean for us? We assume that we are still solving a problem of the form

$$\min_x \mathbb{E}\{F(x, \omega)\}, \quad (7.66)$$

where we assume that $F(x, \omega)$ is continuous and concave (but we do not require differentiability. (Note that we have switched to solving a minimization problem, which better suits the proof technique here.) Let \bar{x}^n be our estimate of x at iteration n (remember that \bar{x}^n is a random variable). Instead of watching the evolution of a process of time, we are studying the behavior of an algorithm over iterations. Let $F^n = \mathbb{E}F(\bar{x}^n)$ be our objective function at iteration n , and let F^* be the optimal value of the objective function. If we are maximizing, we know that $F^n \leq F^*$. If we let $U^n = F^* - F^n$, then we know that $U^n \geq 0$ (this assumes that we can find the true expectation rather than some approximation of it). A stochastic algorithm will not guarantee that $F^n \geq F^{n-1}$, but if we have a good algorithm, then we may be able to show that U^n is a supermartingale, which at least tells us that in the limit, U^n will approach some limit \bar{U} . With additional work we might be able to show that $\bar{U} = 0$, which means that we have found the optimal solution.

A common strategy is to define U^n as the distance between \bar{x}^n and the optimal solution, which is to say

$$U^n = (\bar{x}^n - x^*)^2. \quad (7.67)$$

Of course, we do not know x^* . As a result, we cannot actually compute U^n , but that is not really a problem for us (we are just trying to prove convergence). Note that we immediately get $U^n \geq 0$ (without an expectation). If we can show that U^n is a supermartingale, then we get the result that U^n converges to a random variable U^* (which means that the algorithm converges). Showing that $U^* = 0$ means that our algorithm will (eventually) produce the optimal solution.

We are solving this problem using a stochastic gradient algorithm

$$\bar{x}^n = \bar{x}^{n-1} - \alpha_{n-1} g^n, \quad (7.68)$$

where g^n is our stochastic gradient. If F is differentiable, we would write

$$g^n = \nabla_x F(\bar{x}^{n-1}, \omega^n).$$

But, in general, F may be nondifferentiable, in which case we may have multiple subgradients at a point \bar{x}^{n-1} (for a single sample realization). In this case we write

$$g^n \in \partial_x F(\bar{x}^{n-1}, \omega^n),$$

where $\partial_x F(\bar{x}^{n-1}, \omega^n)$ refers to the set of subgradients at \bar{x}^{n-1} . We assume our problem is unconstrained, so $\nabla_x \mathbb{E}F(x^*, \omega^n) = 0$ if F is differentiable. If it is nondifferentiable, we would assume that $0 \in \partial_x \mathbb{E}F(x^*, \omega^n)$.

Throughout our presentation, we assume that x (and hence g^n) is a scalar (exercise 11.10 provides an opportunity to redo this section using vector notation). In contrast with the previous section, we are now going to allow our stepsizes to be stochastic. For this reason we need to slightly revise our original assumptions about stepsizes (equations (7.52) to (7.54)) by assuming

$$\alpha_n \geq 0 \quad \text{a.s.,} \quad (7.69)$$

$$\sum_{n=0}^{\infty} \alpha_n = \infty \quad \text{a.s.,} \quad (7.70)$$

$$\mathbb{E} \left[\sum_{n=0}^{\infty} (\alpha_n)^2 \right] < \infty. \quad (7.71)$$

The requirement that α_n be nonnegative “almost surely” (a.s.) recognizes that α_n is a random variable. We can write $\alpha_n(\omega)$ as a sample realization of the stepsize (i.e., this is the stepsize at iteration n if we are following sample path ω). When we require that $\alpha_n \geq 0$ “almost surely” we mean that $\alpha_n(\omega) \geq 0$ for all ω where the probability (more precisely, probability measure) of ω , $p(\omega)$, is greater than zero; said differently, this means that the probability that $\mathbb{P}[\alpha_n \geq 0] = 1$. The same reasoning applies to the sum of the stepsizes given in equation (7.70). As the proof unfolds, we will see the reason for needing the conditions (and why they are stated as they are).

We next need to assume some properties of the stochastic gradient g^n . Specifically, we need to assume the following:

Assumption 1 $\mathbb{E}[g^{n+1}(\bar{x}^n - x^*)|\mathfrak{F}^n] \geq 0$.

Assumption 2 $|g^n| \leq B_g$.

Assumption 3 For any x where $|x - x^*| > \delta$, $\delta > 0$, there exists $\epsilon > 0$ such that $\mathbb{E}[g^{n+1}|\mathfrak{F}^n] > \epsilon$.

Assumption 1 implies that on average, the gradient g^n points toward the optimal solution x^* . This is easy to prove for deterministic, differentiable functions, but it may be harder to establish for stochastic problems or problems where $F(x)$ is nondifferentiable. This is because we do not have to assume that $F(x)$ is differentiable. Nor do we assume that a particular gradient g^{n+1} moves toward the optimal solution (for a particular sample realization, it is entirely possible that we are going to move away from the optimal solution). Assumption 2 implies that the gradient is bounded. Assumption 3 requires that the expected gradient cannot vanish at a nonoptimal value of x . This assumption will be satisfied for any strictly concave function.

To show that U^n is a supermartingale, we start with

$$\begin{aligned} U^{n+1} - U^n &= (\bar{x}^{n+1} - x^*)^2 - (\bar{x}^n - x^*)^2 \\ &= ((\bar{x}^n - \alpha_n g^{n+1}) - x^*)^2 - (\bar{x}^n - x^*)^2 \\ &= ((\bar{x}^n - x^*)^2 - 2\alpha_n g^{n+1}(\bar{x}^n - x^*) + (\alpha_n g^{n+1})^2) - (\bar{x}^n - x^*)^2 \\ &= (\alpha_n g^{n+1})^2 - 2\alpha_n g^{n+1}(\bar{x}^n - x^*). \end{aligned} \quad (7.72)$$

Taking conditional expectations on both sides gives

$$\mathbb{E}[U^{n+1}|\mathfrak{F}^n] - \mathbb{E}[U^n|\mathfrak{F}^n] = \mathbb{E}[(\alpha_n g^{n+1})^2|\mathfrak{F}^n] - 2\mathbb{E}[\alpha_n g^{n+1}(\bar{x}^n - x^*)|\mathfrak{F}^n]. \quad (7.73)$$

We note that

$$\mathbb{E}[\alpha_n g^{n+1}(\bar{x}^n - x^*)|\mathfrak{F}^n] = \alpha_n \mathbb{E}[g^{n+1}(\bar{x}^n - x^*)|\mathfrak{F}^n] \quad (7.74)$$

$$\geq 0. \quad (7.75)$$

Equation (7.74) is subtle but important, as it explains a critical piece of notation in this book. Keep in mind that we may be using a stochastic stepsize formula, which means that α_n is a random variable. We assume that α_n is \mathfrak{F}^n -measurable, which means that we are not allowed to use information from iteration $n + 1$ to compute it. This is why we use α_{n-1} in updating equations such as equation (7.6) instead of α_n . When we condition on \mathfrak{F}^n in equation (7.74), α_n is deterministic, allowing us to take it outside the expectation. This allows us to write the conditional expectation of the product of α_n and g^{n+1} as the product of the expectations. Equation (7.75) comes from assumption 1 and the nonnegativity of the stepsizes.

Recognizing that $\mathbb{E}[U^n|\mathfrak{F}^n] = U^n$ (given \mathfrak{F}^n), we may rewrite (7.73) as

$$\begin{aligned} \mathbb{E}[U^{n+1}|\mathfrak{F}^n] &= U^n + \mathbb{E}[(\alpha_n g^{n+1})^2|\mathfrak{F}^n] - 2\mathbb{E}[\alpha_n g^{n+1}(\bar{x}^n - x^*)|\mathfrak{F}^n] \\ &\leq U^n + \mathbb{E}[(\alpha_n g^{n+1})^2|\mathfrak{F}^n]. \end{aligned} \quad (7.76)$$

Because of the positive term on the right-hand side of (7.76), we cannot directly get the result that U^n is a supermartingale. But hope is not lost. We appeal to a neat little trick that works as follows. Let

$$W^n = U^n + \sum_{m=n}^{\infty} (\alpha_m g^{m+1})^2. \quad (7.77)$$

We are going to show that W^n is a supermartingale. From its definition, we obtain

$$W^n = W^{n+1} + U^n - U^{n+1} + (\alpha_n g^{n+1})^2. \quad (7.78)$$

Taking conditional expectations of both sides gives

$$W^n = \mathbb{E}[W^{n+1}|\mathcal{F}^n] + U^n - \mathbb{E}[U^{n+1}|\mathcal{F}^n] + \mathbb{E}[(\alpha_n g^{n+1})^2|\mathcal{F}^n],$$

which is the same as

$$\mathbb{E}[W^{n+1}|\mathcal{F}^n] = W^n - \underbrace{\left(U^n + \mathbb{E}[(\alpha_n g^{n+1})^2|\mathcal{F}^n] - \mathbb{E}[U^{n+1}|\mathcal{F}^n] \right)}_I.$$

We see from equation (7.76) that $I \geq 0$. Removing this term gives us the inequality

$$\mathbb{E}[W^{n+1}|\mathcal{F}^n] \leq W^n. \quad (7.79)$$

This means that W^n is a supermartingale. It turns out that this is all we really need because $\lim_{n \rightarrow \infty} W^n = \lim_{n \rightarrow \infty} U^n$. This means that

$$\lim_{n \rightarrow \infty} U^n \rightarrow U^* \quad \text{a.s.} \quad (7.80)$$

Now that we have the basic convergence of our algorithm, we have to ask: but what is it converging to? For this result we return to equation (7.72) and sum it over the values $n = 0$ up to some number N , giving us

$$\sum_{n=0}^N (U^{n+1} - U^n) = \sum_{n=0}^N (\alpha_n g^{n+1})^2 - 2 \sum_{n=0}^N \alpha_n g^{n+1} (\bar{x}^n - x^*). \quad (7.81)$$

The left-hand side of (7.81) is an alternating sum (sometimes referred to as a telescoping sum), which means that every element cancels out except the first and the last, giving us

$$U^{N+1} - U^0 = \sum_{n=0}^N (\alpha_n g^{n+1})^2 - 2 \sum_{n=0}^N \alpha_n g^{n+1} (\bar{x}^n - x^*).$$

Taking expectations of both sides gives

$$\mathbb{E}[U^{N+1} - U^0] = \mathbb{E}\left[\sum_{n=0}^N (\alpha_n g^{n+1})^2\right] - 2\mathbb{E}\left[\sum_{n=0}^N \alpha_n g^{n+1} (\bar{x}^n - x^*)\right]. \quad (7.82)$$

We want to take the limit of both sides as N goes to infinity. To do this, we have to appeal to the *Dominated Convergence Theorem* (DCT), which tells us that

$$\lim_{N \rightarrow \infty} \int_x f^n(x) dx = \int_x \left(\lim_{N \rightarrow \infty} f^n(x) \right) dx$$

if $|f^n(x)| \leq g(x)$ for some function $g(x)$, where

$$\int_x g(x) dx < \infty.$$

For our application the integral represents the expectation (we would use a summation instead of the integral if x were discrete), which means that the DCT gives us the conditions needed to exchange the limit and the expectation. Above we showed that $\mathbb{E}[U^{n+1} | \mathfrak{F}^n]$ is bounded (from (7.76) and the boundedness of U^0 and the gradient). This means that the right-hand side of (7.82) is also bounded for all n . The DCT then allows us to take the limit as N goes to infinity inside the expectations, giving us

$$U^* - U^0 = \mathbb{E} \left[\sum_{n=0}^{\infty} (\alpha_n g^{n+1})^2 \right] - 2\mathbb{E} \left[\sum_{n=0}^{\infty} \alpha_n g^{n+1} (\bar{x}^n - x^*) \right].$$

We can rewrite the first term on the right-hand side as

$$\mathbb{E} \left[\sum_{n=0}^{\infty} (\alpha_n g^{n+1})^2 \right] \leq \mathbb{E} \left[\sum_{n=0}^{\infty} (\alpha_n)^2 (B)^2 \right] \quad (7.83)$$

$$= B^2 \mathbb{E} \left[\sum_{n=0}^{\infty} (\alpha_n)^2 \right] \quad (7.84)$$

$$< \infty. \quad (7.85)$$

Equation (7.83) comes from assumption 2, which requires that $|g^n|$ be bounded by B , and this immediately gives us equation (7.84). The requirement that $\mathbb{E} \sum_{n=0}^{\infty} (\alpha_n)^2 < \infty$ (equation (7.54)) gives us (7.85), which means that the first summation on the right-hand side of (7.82) is bounded. Since the left-hand side of (7.82) is bounded, we can conclude that the second term on the right-hand side of (7.82) is also bounded.

Now let

$$\begin{aligned} \beta^n &= \mathbb{E} [g^{n+1} (\bar{x}^n - x^*)] \\ &= \mathbb{E} [\mathbb{E} [g^{n+1} (\bar{x}^n - x^*) | \mathfrak{F}^n]] \\ &\geq 0, \end{aligned}$$

since $\mathbb{E}[g^{n+1} (\bar{x}^n - x^*) | \mathfrak{F}^n] \geq 0$ from Assumption 1. This means that

$$\sum_{n=0}^{\infty} \alpha_n \beta^n < \infty \quad \text{a.s.} \quad (7.86)$$

But we have required that $\sum_{n=0}^{\infty} \alpha_n = \infty$ a.s. (equation (7.70)). Since $\alpha_n \geq 0$ and $\beta^n \geq 0$ (a.s.), we conclude that

$$\lim_{n \rightarrow \infty} \beta^n \rightarrow 0 \quad \text{a.s.} \quad (7.87)$$

If $\beta^n \rightarrow 0$, then $\mathbb{E}[g^{n+1}(\bar{x}^n - x^*)] \rightarrow 0$, which allows us to conclude that $\mathbb{E}[g^{n+1}(\bar{x}^n - x^*)|\mathcal{F}^n] \rightarrow 0$ (the expectation of a nonnegative random variable cannot be zero unless the random variable is always zero). But what does this tell us about the behavior of \bar{x}^n ? Knowing that $\beta^n \rightarrow 0$ does not necessarily imply that $g^{n+1} \rightarrow 0$ or $\bar{x}^n \rightarrow x^*$. There are three scenarios:

1. $\bar{x}^n \rightarrow x^*$ for all n , and of course all sample paths ω . If this were the case, we are done.
2. $\bar{x}^{n_k} \rightarrow x^*$ for a subsequence $n_1, n_2, \dots, n_k, \dots$. For example, it might be that the sequence $\bar{x}^1, \bar{x}^3, \bar{x}^5, \dots \rightarrow x^*$, while $\mathbb{E}[g^2|\mathcal{F}^1], \mathbb{E}[g^4|\mathcal{F}^3], \dots \rightarrow 0$. This would mean that for the subsequence n_k , $U^{n_k} \rightarrow 0$. But we already know that $U^n \rightarrow U^*$, where U^* is the unique limit point, which means that $U^* = 0$. But, if this is the case, then this is the limit point for every sequence of \bar{x}^n .
3. There is no subsequence \bar{x}^{n_k} that has \bar{x}^* as its limit point. This means that $\mathbb{E}[g^{n+1}|\mathcal{F}^n] \rightarrow 0$. However, assumption 3 tells us that the expected gradient cannot vanish at a nonoptimal value of x . This means that this case cannot happen.

This completes the proof. □

7.7 BIBLIOGRAPHIC NOTES

Section 7.2 The theoretical foundation for estimating value functions from Monte Carlo estimates has its roots in stochastic approximation theory, originated by Robbins and Monro (1951), with important early contributions made by Kiefer and Wolfowitz (1952), Blum (1954a) and Dvoretzky (1956). For thorough theoretical treatments of stochastic approximation theory, see Wasan (1969), Kushner and Clark (1978), and Kushner and Yin (1997). Very readable treatments of stochastic optimization can be found in Pflug (1996) and Spall (2003).

Section 7.3 A nice introduction to various learning strategies is contained in Kaelbling (1993) and Sutton and Barto (1998). Thrun (1992) contains a good discussion of exploration in the learning process. The discussion of Boltzmann exploration and epsilon-greedy exploration is based on Singh et al. (2000).

Section 7.4 The knowledge gradient policy for normally distributed rewards and independent beliefs was introduced by Gupta and Miescke (1996), and subsequently analyzed in greater depth by Frazier et al. (2008). The

knowledge gradient for correlated beliefs was introduced by Frazier et al. (2009). The adaptation of the knowledge gradient for online problems is due to Ryzhov and Powell (2009). The knowledge gradient for correlated beliefs is due to Negoescu et al. (2010).

Section 7.5 There is an advanced field of research within the simulation community that has addressed the problem of using simulation (in particular, discrete event simulation) to find the best setting of a set of parameters that controls the behavior of the simulation. An early survey is given by Bechhofer et al. (1995); a more recent survey can be found in Fu et al. (2007). Kim and Nelson (2006) provides a nice tutorial overview of methods based on ordinal optimization. Other important contributions in this line include Hong and Nelson (2006, 2007). Most of this literature considers problems where the number of potential alternatives is not too large. Nelson et al. (2001) consider the case where the number of designs is large. Ankenman et al. (2009) discusses the use of a technique called kriging, which is useful when the parameter vector x is continuous. The literature on optimal computing budget allocation is based on a series of articles originating with Chen (1995), and including Yucesan et al. (1997), Chen et al. (1997, 2000) and Chick et al. (2000). Chick and Inoue (2001) introduce the $LL(B)$ strategy, which maximizes the linear loss with measurement budget B . He et al. (2007) introduce an OCBA procedure for optimizing the expected value of a chosen design, using the Bonferroni inequality to approximate the objective function for a single stage. A common strategy in simulation is to test different parameters using the same set of random numbers to reduce the variance of the comparisons. Fu et al. (2007) apply the OCBA concept to measurements using common random numbers.

Section 7.6.2 This proof is based on Blum (1954b), which generalized the original paper by Robbins and Monro (1951).

Section 7.6.3 The proof in Section 7.6.3 uses standard techniques drawn from several sources, notably Wasan (1969), Chong (1991), Kushner and Yin (1997) and, for this author, Powell and Cheung (2000).

PROBLEMS

- 7.1** We are going to optimize a policy for selling an asset using a spreadsheet. Let column A be the time period t starting at $t = 0$, and generate a series of prices in column B using $p_t = p_{t-1} + 2U$, where U is a random variable uniformly distributed between -0.5 and $+0.5$, generated in Excel using $U = (\text{RAND}) - 0.5$. Set $p_0 = 8$. In column C, let $\bar{p}_t = 0.9\bar{p}_{t-1} + 0.1p_t$. In column D, create a variable $R_t = R_{t-1} - x_{t-1}$ with $R_0 = 1$. $R_t = 1$ means that we still own the asset. In column E, program the policy $X_t = R_t * \text{IF}(p_t - \bar{p}_t > \beta, 1, 0)$, which means that we will sell the asset if its current price exceeds the smoothed price by more than β . Finally, compute the return

if the asset is sold in column F using $C_t = x_t p_t$. Note that if $x_t = 1$, $R_{t+1} = 0$ and will then remain zero. Repeat the calculations for 100 rows, by which time the asset should have sold.

- (a) Program the spreadsheet and perform 10 repetitions of the simulation using $\beta = 1.0$. Compute the average, the sample variance and the variance of the average (by dividing the sample variance by 10).
- (b) Repeat (a) using $\beta = 2.0$. Compare the confidence intervals for the performance of each policy, and determine if they are statistically different at the 95 percent confidence level.
- (c) Create a parallel simulation to the first one that simulates the policy using $\beta + \delta$, where δ is a perturbation parameter that you specify. The second simulation should be run using the same prices as the first simulation. If $\bar{F}(\beta)$ is the performance of the policy using β , compute a sample gradient using

$$g^n = \frac{\bar{F}(\beta^n + \delta) - \bar{F}(\beta^n)}{\delta}.$$

Using $\delta = 1$, perform 10 iterations of a stochastic gradient algorithm

$$\beta^{n+1} = \beta^n + \alpha_n g^n$$

Let $\alpha_n = 5/(10 + n)$, and perform 10 iterations of this algorithm.

- (d) Perform three iterations each for $\beta = 0.5, 1.0, 1.5, 2.0, 3.0$. Plot the points and use the LINEST routine in Excel (or any other statistical package) to fit a curve to identify the best value of β .

- 7.2** We are going to optimize a policy for storing electricity in a battery when prices are low, and selling the power when prices are high. Let column A be the time period t starting at $t = 0$, and generate a series of prices in column B using $p_t = p_{t-1} + 0.7(\mu - p_{t-1}) + 40U$ where U is a random variable uniformly distributed between -0.5 and $+0.5$, generated in Excel using $U = (\text{RAND}() - 0.5)$. Set $p_0 = 40$. In column C, create a variable $R_t = \max\{0, R_{t-1} + x_{t-1}\}$ with $R_0 = 0$, where R_t represents the amount of energy in megawatt-hours stored in the battery. In column D, program the policy

$$X_t = \begin{cases} 1, & p_t \leq p^L, \\ -1, & p_t \geq p^U, \\ 0, & \text{otherwise.} \end{cases}$$

In column E, program the contribution $C_t = p^t x^t$. Sum the contributions over 200 time periods and compute the average $\bar{F}(P^U, P^L)$.

- (a) Program the spreadsheet and perform 10 repetitions of the simulation using $p^U = 45$ and $P^L = 25$. Compute the average, the sample variance and the variance of the average (by diving the sample variance by 10).

- (b) Repeat (a) using $p^U = 50$ and $P^L = 25$. Compare the confidence intervals for the performance of each policy and determine if they are statistically different at the 95 percent confidence level.
- (c) Create a parallel simulation to the first one that simulates the policy using $P^U + \delta$ and $P^L + \delta$ for the upper and lower prices, where δ is a perturbation parameter that you specify. The second simulation should be run using the same prices as the first simulation. If $\bar{F}(\beta)$ is the performance of the policy using β , compute a sample gradient using

$$g^n = \frac{\bar{F}(\beta^n + \delta) - \bar{F}(\beta^n)}{\delta}.$$

Using $\delta = 5$, perform 10 iterations of a stochastic gradient algorithm

$$\beta^{n+1} = \beta^n + \alpha_n g^n.$$

Let $\alpha_n = 5/(10 + n)$, and perform 10 iterations of this algorithm.

- (d) Perform one iteration each while varying P^L over the grid from 20 to 35 in increments of 5, and varying P^U from 45 to 60 in increments of 5. Plot the points using a three-dimensional plot using Excel, and from this estimate what appear to be the best values of P^U and P^L .

- 7.3** You are trying to plan how much energy you can use from your solar panels, and how much should be stored so that it can be used later when energy may be expensive. At time t , let p_t be the price of electricity from the grid, and assume it evolves according to $p_{t+1} = p_t + \hat{p}_{t+1}$. Let E_t be the energy generated by the solar panels, where $E_{t+1} = E_t + \hat{E}_{t+1}$, and let $D_t = \mu_t^D + \epsilon_t d$ be the demand for energy which follows a predictable pattern μ_t^D with some variation. Let R_t be the energy stored in the battery, and let x_t be the amount of solar energy that is stored (if $x_t > 0$) or withdrawn (if $x_t < 0$).

- (a) What is the state variable for this problem?
 (b) What types of policies would you consider using for a problem such this. Discuss the strengths and weaknesses of each.
 (c) Now introduce the dimension that at time t , you are given a forecast $\bar{E}_{tt'}$ for the energy from the solar panels that will be produced at time t' . What is the state variable for this problem?
 (d) How would your answer to (b) change with this new information?

C H A P T E R 8

Approximating Value Functions

One of the most powerful, yet difficult, tools in approximate dynamic programming is the ability to create a policy by approximating the value of being in a state, famously known as value function approximations. If $\bar{V}_t(S_t)$ is our approximation of the value of being in state S_t , we can make decisions using the policy

$$a_t = \arg \max_{a_t \in \mathcal{A}_t} (C(S_t, a_t) + \gamma \mathbb{E} \bar{V}_{t+1}(S_{t+1})),$$

where $S_{t+1} = S^M(S_t, a_t, W_{t+1})$. The challenge is finding a good approximation $\bar{V}_t(S_t)$.

We break the problem of finding this approximation into three specific tasks, covered in this and the following two chapters:

Approximating value functions. This first chapter addresses the different ways we can approximate value functions, divided along three lines: lookup tables (with aggregation), parametric models, and nonparametric models.

Learning value function approximations. Chapter 9 describes different methods for computing \hat{v}^n , a sampled estimate of the value of being in a state S^n , and then using this information to update the value function approximation *for a fixed policy*.

Learning while optimizing. Chapter 10 introduces the challenge of searching for better policies while simultaneously estimating the value of a policy. Approximate value iteration and policy iteration are illustrated using a mixture of lookup tables, parametric models, and nonparametric models.

Readers should view this chapter as a peek into the vast field of statistical learning (also known as machine learning). All the methods we present have been used in the approximate dynamic programming community. However, it is important to realize that ADP imposes special demands on statistical algorithms. One of these

Approximate Dynamic Programming: Solving the Curses of Dimensionality, Second Edition.

Warren B. Powell.

© 2011 John Wiley & Sons, Inc. Published 2011 by John Wiley & Sons, Inc.

is the need to update statistical models recursively, since estimates of the value of being in a state arrive in a continuous stream, and not as a single batch.

8.1 LOOKUP TABLES AND AGGREGATION

For many years researchers addressed the “curse of dimensionality” by using aggregation to reduce the size of the state space. The idea was to aggregate the original problem, solve it exactly (using the techniques of Chapter 3), and then disaggregate it back to obtain an approximate solution to the original problem. Not only would the value function be defined over this smaller state space, so would the one-step transition matrix. In fact, simplifying the transition matrix (since its size is given by the square of the number of states) was even more important than simplifying the states over which the value function was defined.

In our presentation we only use aggregation in the value function approximation. Once we have chosen an action a_t , we simulate our way to the next state as before using $S_{t+1} = S^M(S_t, a_t, W_{t+1})$ using the original, disaggregate state variable. We feel that this is a more effective modeling strategy than methods that aggregate the dynamics of the process itself.

8.1.1 ADP and Aggregation

Perhaps one of the most powerful features of approximate dynamic programming is that even if we use aggregation, we do not have to simplify the state of the system. The transition function $S_{t+1} = S^M(S_t, a_t, W_{t+1})$ always uses the original, disaggregate (possibly continuous) state vector. Aggregation is only used to approximate the value function. For example, in our nomadic trucker problem it is necessary to capture location, domicile, fleet type, equipment type, number of hours he has driven that day, how many hours he has driven on each of the past seven days, and the number of days he has been driving since he was last at home. All of this information is needed to simulate the driver forward in time. But we might estimate the value function using only the location of the driver, his domicile, and fleet type. We are not trying to simplify how we represent the problem; rather, we only want to simplify how we approximate the value function for the purpose of making decisions.

If our nomadic trucker is described by the state vector S_t , the transition function $S_{t+1} = S^M(S_t, a_t, W_{t+1})$ may represent the state vector at a high level of detail (some values may be continuous). But the decision problem

$$\max_{a_t \in \mathcal{A}} (C(S_t, a_t) + \gamma \mathbb{E} \bar{V}_{t+1}(G(S_{t+1}))) \quad (8.1)$$

uses a value function $\bar{V}_{t+1}(G(S_{t+1}))$, where $G(\cdot)$ is an aggregation function that maps the original (and very detailed) state S into something much simpler. The aggregation function G may ignore a dimension, discretize it, or use any of a variety of ways to reduce the number of possible values of a state vector. This also reduces the number of parameters we have to estimate. In what follows we drop

the explicit reference of the aggregation function G and simply use $\bar{V}_{t+1}(S_{t+1})$. The aggregation is implicit in the value function approximation.

Some examples of aggregation include:

Spatial. A transportation company is interested in estimating the value of truck drivers at a particular location. Locations may be calculated at the level of a five-digit zip code (there are about 55,000 in the United States), three-digit zip code (about 1000), or the state level (48 contiguous states).

Temporal. A bank may be interested in estimating the value of holding an asset at a point in time. Time may be measured by the day, week, month, or quarter.

Continuous parameters. The state of an aircraft may be its fuel level; the state of a traveling salesman may be how long he has been away from home; the state of a water reservoir may be the depth of the water; the state of the cash reserve of a mutual fund is the amount of cash on hand at the end of the day. These are examples of systems with at least one dimension of the state that is at least approximately continuous. The variables may all be discretized into intervals.

Hierarchical classification. A portfolio problem may need to estimate the value of investing money in the stock of a particular company. It may be useful to aggregate companies by industry segment (e.g., a particular company might be in the chemical industry), and it might be further aggregated based on whether it is viewed as a domestic or multinational company. Similarly, problems of managing large inventories of parts (e.g., for cars) may benefit by organizing parts into part families (transmission parts, engine parts, dashboard parts).

The examples below provide additional illustrations.

■ EXAMPLE 8.1

The state of a jet aircraft may be characterized by multiple attributes that include spatial and temporal dimensions (location and flying time since the last maintenance check) as well other attributes. A continuous parameter could be the fuel level; an attribute that lends itself to hierarchical aggregation might be the specific type of aircraft. We can reduce the number of states (attributes) of this resource by aggregating each dimension into a smaller number of potential outcomes. ■

■ EXAMPLE 8.2

The state of a portfolio might consist of the number of bonds that are characterized by the source of the bond (a company, a municipality, or the federal government), the maturity (six months, 12 months, 24 months), when it was

purchased, and its rating by bond agencies. Companies can be aggregated up by industry segment. ■

■ EXAMPLE 8.3

Blood stored in blood banks can be characterized by type, the source (which might indicate risks for diseases), age (it can be stored for up to 42 days), and the current location where it is being stored. A national blood management agency might want to aggregate the state space by ignoring the source (ignoring a dimension is a form of aggregation), discretizing the age from days into weeks, and aggregating locations into more aggregate regions. ■

■ EXAMPLE 8.4

The value of an asset is determined by its current price, which is continuous. We can estimate the asset using a price discretized to the nearest dollar. ■

There are many applications where aggregation is naturally hierarchical. For example, in our nomadic trucker problem we might want to estimate the value of a truck based on three attributes: location, home domicile, and fleet type. The first two represent geographical locations, which can be represented (for this example) at three levels of aggregation: 400 sub-regions, 100 regions, and 10 zones. Table 8.1 illustrates five levels of aggregation that might be used. In this example each higher level can be represented as an aggregation of the previous level.

Aggregation is also useful for continuous variables. Suppose that our state variable is the amount of cash we have on hand, a number that might be as large as \$10 million dollars. We might discretize our state space in units of \$1 million, \$100 thousand, \$10 thousand, \$1000, \$100, and \$10. This discretization produces a natural hierarchy since 10 segments at one level of aggregation naturally group into one segment at the next level of aggregation.

Hierarchical aggregation is often the simplest to work with, but in most cases there is no reason to assume that the structure is hierarchical. In fact we may even use overlapping aggregations (sometimes known as “soft” aggregation), where the same state s aggregates into multiple elements in \mathcal{S}^g . For example, let s represent

Table 8.1 Examples of aggregations of the state space for the nomadic trucker problem

Aggregation		Fleet Type	Domicile	Size of State Space
Level	Location			
0	Sub-region	Fleet	Region	$400 \times 5 \times 100 = 200,000$
1	Region	Fleet	Region	$100 \times 5 \times 100 = 50,000$
2	Region	Fleet	Zone	$100 \times 5 \times 10 = 5,000$
3	Region	Fleet	—	$100 \times 5 \times 1 = 500$
4	Zone	—	—	$10 \times 1 \times 1 = 10$

Note: The dash indicates that the particular dimension is ignored.

an (x, y) coordinate in a continuous space that has been discretized into the set of points $(x_i, y_i)_{i \in \mathcal{I}}$. Assume that we have a distance metric $\rho((x, y), (x_i, y_i))$ that measures the distance from any point (x, y) to every aggregated point $(x_i, y_i), i \in \mathcal{I}$. We might use an observation at the point (x, y) to update estimates at each (x_i, y_i) with a weight that declines with $\rho((x, y), (x_i, y_i))$.

8.1.2 Computing Bias and Variance

Before we present our methods for hierarchical aggregation, we are going to need some basic results on bias and variance in statistical estimation. These results are needed both here and later in the volume. To develop these results, we assume that we are trying to estimate a time-varying series that we denote θ^n (think of this as the true value of being in a state after n iterations, just as the value of being in a state changes as we execute the value iteration algorithm). Let $\bar{\theta}^n$ be our statistical estimate of θ^n , and let $\hat{\theta}^n$ be a sampled estimate that we are going to use to update $\bar{\theta}^n$ as

$$\bar{\theta}^n = (1 - \alpha_{n-1})\bar{\theta}^{n-1} + \alpha_{n-1}\hat{\theta}^n,$$

where $\hat{\theta}^n$ is an unbiased observation (i.e., $\mathbb{E}\hat{\theta}^n = \theta^n$), which is assumed to be independent of $\bar{\theta}^{n-1}$. We emphasize that the parameter we are trying to estimate, θ^n , varies with n just as expected value functions vary with n (recall the behavior of the value function when using value iteration from Chapter 3). We are interested in estimating the variance of $\bar{\theta}^n$ and its bias, which is defined by $\bar{\theta}^{n-1} - \theta^n$.

We start by computing the variance of $\bar{\theta}^n$. We can represent our observations of θ^n using

$$\hat{\theta}^n = \theta^n + \varepsilon^n,$$

where $\mathbb{E}\varepsilon^n = 0$ and $\text{Var}[\varepsilon^n] = \sigma^2$. Previously ε^n was the error between our earlier estimate and our later observation. Here we treat this as an exogenous measurement error. With this model we can compute the variance of $\bar{\theta}^n$ using

$$\text{Var}[\bar{\theta}^n] = \lambda^n \sigma^2, \quad (8.2)$$

where λ^n can be computed from the simple recursion

$$\lambda^n = \begin{cases} (\alpha_{n-1})^2, & n = 1, \\ (1 - \alpha_{n-1})^2 \lambda^{n-1} + (\alpha_{n-1})^2, & n > 1. \end{cases} \quad (8.3)$$

To see this, we start with $n = 1$. For a given (deterministic) initial estimate $\bar{\theta}^0$, we first observe that the variance of $\bar{\theta}^1$ is given by

$$\begin{aligned} \text{Var}[\bar{\theta}^1] &= \text{Var}[(1 - \alpha_0)\bar{\theta}^0 + \alpha_0\hat{\theta}^1] \\ &= (\alpha_0)^2 \text{Var}[\hat{\theta}^1] \\ &= (\alpha_0)^2 \sigma^2. \end{aligned}$$

For general $\bar{\theta}^n$, we use a proof by induction. Assume that $\text{Var}[\bar{\theta}^{n-1}] = \lambda^{n-1}\sigma^2$. Then, since $\bar{\theta}^{n-1}$ and $\hat{\theta}^n$ are independent, we find

$$\begin{aligned}\text{Var}[\bar{\theta}^n] &= \text{Var}\left[(1 - \alpha_{n-1})\bar{\theta}^{n-1} + \alpha_{n-1}\hat{\theta}^n\right] \\ &= (1 - \alpha_{n-1})^2\text{Var}\left[\bar{\theta}^{n-1}\right] + (\alpha_{n-1})^2\text{Var}[\hat{\theta}^n] \\ &= (1 - \alpha_{n-1})^2\lambda^{n-1}\sigma^2 + (\alpha_{n-1})^2\sigma^2 \tag{8.4}\end{aligned}$$

$$= \lambda^n\sigma^2. \tag{8.5}$$

Equation (8.4) is true by assumption (in our induction proof), while equation (8.5) establishes the recursion in equation (8.3). This gives us the variance, assuming of course that σ^2 is known.

The bias of our estimate is the difference between our current estimate and the true value, given by

$$\beta^n = \mathbb{E}[\bar{\theta}^{n-1}] - \theta^n.$$

We note that $\bar{\theta}^{n-1}$ is our estimate of θ^n computed using the information up through iteration $n - 1$. Of course, our formula for the bias assumes that θ^n is known. These two results for the variance and bias are called the *parameters-known* formulas.

We next require the mean squared error, which can be computed using

$$\mathbb{E}\left[\left(\bar{\theta}^{n-1} - \theta^n\right)^2\right] = \lambda^{n-1}\sigma^2 + (\beta^n)^2. \tag{8.6}$$

See exercise 8.3 to prove this. This formula gives the variance around the known mean, θ^n . For our purposes it is also useful to have the variance around the observations $\hat{\theta}^n$. Let

$$v^n = \mathbb{E}\left[\left(\bar{\theta}^{n-1} - \hat{\theta}^n\right)^2\right]$$

be the mean squared error (including noise and bias) between the current estimate $\bar{\theta}^{n-1}$ and the observation $\hat{\theta}^n$. It is possible to show that (see exercise 8.4)

$$v^n = (1 + \lambda^{n-1})\sigma^2 + (\beta^n)^2, \tag{8.7}$$

where λ^n is computed using equation (8.3).

In practice, we do not know σ^2 , and we certainly do not know θ^n . As a result we have to estimate both parameters from our data. We begin by providing an estimate of the bias using

$$\bar{\beta}^n = (1 - \eta_{n-1})\bar{\beta}^{n-1} + \eta_{n-1}(\bar{\theta}^{n-1} - \hat{\theta}^n),$$

where η_{n-1} is a (typically simple) stepsize rule used for estimating the bias and variance. As a general rule, we should pick a stepsize for η_{n-1} , which produces

larger stepsizes than α_{n-1} , because we are more interested in tracking the true signal than producing an estimate with a low variance. We have found that a constant stepsize such as 0.10 works quite well on a wide range of problems, but if precise convergence is needed, it is necessary to use a rule where the stepsize goes to zero such as the harmonic stepsize rule (equation (11.19)).

To estimate the variance, we begin by finding an estimate of the total variation v^n . Let \bar{v}^n be the estimate of the total variance which we might compute using

$$\bar{v}^n = (1 - \eta_{n-1})\bar{v}^{n-1} + \eta_{n-1}(\bar{\theta}^{n-1} - \hat{\theta}^n)^2.$$

Using \bar{v}^n as our estimate of the total variance, we can compute an estimate of σ^2 by way of

$$(\bar{\sigma}^n)^2 = \frac{\bar{v}^n - (\bar{\beta}^n)^2}{1 + \lambda^{n-1}}.$$

We can use $(\bar{\sigma}^n)^2$ in equation (8.2) to obtain an estimate of the variance of $\bar{\theta}^n$.

If we are doing true averaging (as would occur if we use a stepsize of $1/n$), we can get a more precise estimate of the variance for small samples by using the recursive form of the small sample formula for the variance

$$(\hat{\sigma}^2)^n = \frac{n-2}{n-1}(\hat{\sigma}^2)^{n-1} + \frac{1}{n}(\bar{\theta}^{n-1} - \hat{\theta}^n)^2. \quad (8.8)$$

The quantity $(\hat{\sigma}^2)^n$ is an estimate of the variance of $\hat{\theta}^n$. The variance of our estimate $\bar{\theta}^{(n)}$ is computed using

$$(\bar{\sigma}^2)^n = \frac{1}{n}(\hat{\sigma}^2)^n.$$

8.1.3 Modeling Aggregation

We begin by defining a family of aggregation functions

$$G^g : \mathcal{S} \rightarrow \mathcal{S}^{(g)}.$$

Here $\mathcal{S}^{(g)}$ represents the g th level of aggregation of the state space \mathcal{S} . Let

$$S^{(g)} = G^g(s), \text{ the } g\text{th level aggregation of the state vector } s.$$

$$\mathcal{G} = \text{set of indexes corresponding to the levels of aggregation.}$$

In this section we assume that we have a single aggregation function G that maps the disaggregate state $s \in \mathcal{S} = \mathcal{S}^{(0)}$ into an aggregated space $\mathcal{S}^{(g)}$. In Section 8.1.4, we let $g \in \mathcal{G} = \{0, 1, 2, \dots\}$ and we work with all levels of aggregation at the same time.

To begin our study of aggregation, we first need to characterize how we sample different states (at the disaggregate level). For this discussion, we assume that we

have two exogenous processes: at iteration n , the first process chooses a state to sample (which we denote by \hat{s}^n), and the second produces an observation of the value of being in state $\hat{s}^n \in \mathcal{S}$, which we denote by \hat{v}^n (or \hat{v}_s^n). There are different ways of choosing a state \hat{s} ; we could choose it at random, or by finding a_t^n by solving (8.1) and then choosing a state $\hat{s}_t^n = S^M(S_t^n, a_t^n, W_{t+1})$.

We need to characterize the errors that arise in our estimate of the value function. Let

$$v_s^{(g)} = \text{true value of being in state } s \text{ at aggregation level } g.$$

Here s is the original, disaggregated state vector. We let $v_s = v_s^{(0)}$ be the true (expected) value of being in state s . $v_s^{(g)}$ is the expected value of being in aggregated state $G(s)$. We can think of $v_s^{(g)}$ as an average over all the values of state s such that $G(s) = \bar{s}$. In fact there are different ways to weight all these disaggregate values, and we have to specify the weighting in a specific application. We might weight by how many times we actually visit a state (easy and practical, but it means the aggregate measure depends on how you are visiting states), or we might weight all states equally (this takes away the dependence on the policy, but we might include states we would never visit).

We need a model of how we sample and measure values. Assume that at iteration n , we sample a (disaggregated) state \hat{s}^n , and we then observe the value of being in this state with the noisy observation

$$\hat{v}^n = v_{\hat{s}^n} + \varepsilon^n.$$

We let ω be a sample realization of the sequence of states and values, given by

$$\omega = (\hat{s}^1, \hat{v}^1, \hat{s}^2, \hat{v}^2, \dots).$$

Let

$$\bar{v}_s^{(g,n)} = \text{estimate of the value associated with the state vector } s \text{ at the } g\text{th level of aggregation after } n \text{ observations.}$$

Throughout our discussion, a bar over a variable means it was computed from sample observations. A hat means the variable was an exogenous observation. If there is nothing (e.g., v or β), then it means this is the true value (which is not known to us).

When we are working at the most disaggregate level ($g = 0$), the state s that we measure is the observed state $s = \hat{s}^n$. For $g > 0$, the subscript s in $\bar{v}_s^{(g,n)}$ refers to $G^g(\hat{s}^n)$, or the g th level of aggregation of \hat{s}^n . Given an observation (\hat{s}^n, \hat{v}^n) , we would update our estimate of being in state $s = \hat{s}^n$ using

$$\bar{v}_s^{(g,n)} = (1 - \alpha_{s,n-1}^{(g)})\bar{v}_s^{(g,n-1)} + \alpha_{s,n-1}^{(g)}\hat{v}^n.$$

Here we have written the stepsize $\alpha_{s,n-1}^{(g)}$ to explicitly represent the dependence on the state and level of aggregation. Implicit is that this is also a function of the

number of times that we have updated $\bar{v}_s^{(g,n)}$ by iteration n , rather than a function of n itself.

To illustrate, imagine that our nomadic trucker is described by the state vector $s = (\text{Loc}, \text{Equip}, \text{Home}, \text{DOThrs}, \text{Days})$, where “Loc” is location, “Equip” denotes the type of trailer (long, short, refrigerated), “Home” is the location of where the driver lives, “DOThrs” is a vector giving the number of hours the driver has worked on each of the last eight days, and “Days” is the number of days the driver has been away from home. We are going to estimate the value $\bar{V}(s)$ for different levels of aggregation of s , where we aggregate purely by ignoring certain dimensions of s . We start with our original disaggregate observation $\hat{v}(s)$, which we are going to write as

$$\hat{v} \begin{pmatrix} \text{Loc} \\ \text{Equip} \\ \text{Home} \\ \text{DOThrs} \\ \text{Days} \end{pmatrix} = \max_a (C(s, a) + \bar{V}(S^M(s, a))).$$

We now wish to use this estimate of the value of a driver in state s to produce value functions at different levels of aggregation. We can do this by simply smoothing this disaggregate estimate in with estimates at different levels of aggregation, as in

$$\begin{aligned} \bar{v}^{(1,n)} \begin{pmatrix} \text{Loc} \\ \text{Equip} \\ \text{Home} \end{pmatrix} &= (1 - \alpha_{s,n-1}^{(1)}) \bar{v}^{(1,n-1)} \begin{pmatrix} \text{Loc} \\ \text{Equip} \\ \text{Home} \end{pmatrix} + \alpha_{s,n-1}^{(1)} \hat{v} \begin{pmatrix} \text{Loc} \\ \text{Equip} \\ \text{Home} \\ \text{DOThrs} \\ \text{Days} \end{pmatrix}, \\ \bar{v}^{(2,n)} \begin{pmatrix} \text{Loc} \\ \text{Equip} \end{pmatrix} &= (1 - \alpha_{s,n-1}^{(2)}) \bar{v}^{(2,n-1)} \begin{pmatrix} \text{Loc} \\ \text{Equip} \end{pmatrix} + \alpha_{s,n-1}^{(2)} \hat{v} \begin{pmatrix} \text{Loc} \\ \text{Equip} \\ \text{Home} \\ \text{DOThrs} \\ \text{Days} \end{pmatrix}, \\ \bar{v}^{(3,n)} \begin{pmatrix} \text{Loc} \end{pmatrix} &= (1 - \alpha_{s,n-1}^{(3)}) \bar{v}^{(3,n-1)} \begin{pmatrix} \text{Loc} \end{pmatrix} + \alpha_{s,n-1}^{(3)} \hat{v} \begin{pmatrix} \text{Loc} \\ \text{Equip} \\ \text{Home} \\ \text{DOThrs} \\ \text{Days} \end{pmatrix}. \end{aligned}$$

In the first equation we are smoothing the value of a driver based on a five-dimensional state vector, given by $\hat{v}(s)$, in with an approximation indexed by a

three-dimensional state vector. The second equation does the same using value function approximation indexed by a two-dimensional state vector, while the third equation does the same with a one-dimensional state vector. It is important to keep in mind that the stepsize must reflect the number of times a state has been updated.

We can estimate the variance of $\bar{v}_s^{(g,n)}$ using the techniques described in Section 8.1.2. Let

$$(s_s^2)^{(g,n)} = \text{estimate of the variance of observations made of state } s, \\ \text{using data from aggregation level } g, \text{ after } n \text{ observations.}$$

$(s_s^2)^{(g,n)}$ is the estimate of the variance of the observations \hat{v} when we observe a state \hat{s}^n which aggregates to state s (i.e., $G^g(\hat{s}^n) = s$). We are really interested in the variance of our estimate of the mean, $\bar{v}_s^{(g,n)}$. In Section 8.1.2 we showed that

$$\begin{aligned} (\bar{\sigma}_s^2)^{(g,n)} &= \text{Var}[\bar{v}_s^{(g,n)}] \\ &= \lambda_s^{(g,n)} (s_s^2)^{(g,n)}, \end{aligned} \quad (8.9)$$

where $(s_s^2)^{(g,n)}$ is an estimate of the variance of the observations \hat{v}_t^n at the g th level of aggregation (computed below), and $\lambda_s^{(g,n)}$ can be computed from the recursion

$$\lambda_s^{(g,n)} = \begin{cases} (\alpha_{s,n-1}^{(g)})^2, & n = 1, \\ (1 - \alpha_{s,n-1}^{(g)})^2 \lambda_s^{(g,n-1)} + (\alpha_{s,n-1}^{(g)})^2, & n > 1. \end{cases}$$

Note that if the stepsize $\alpha_{s,n-1}^{(g)}$ goes to zero, then $\lambda_s^{(g,n)}$ will also go to zero, as will $(\bar{\sigma}_s^2)^{(g,n)}$. We now need to compute $(s_s^2)^{(g,n)}$, which is the estimate of the variance of observations \hat{v}^n for states \hat{s}^n for which $G^g(\hat{s}^n) = s$ (the observations of states that aggregate up to s). Let $\bar{v}_s^{(g,n)}$ be the total variation, given by

$$\bar{v}_s^{(g,n)} = (1 - \eta_{n-1}) \bar{v}_s^{(g,n-1)} + \eta_{n-1} (\bar{v}_s^{(g,n-1)} - \hat{v}_s^n)^2,$$

where η_{n-1} follows some stepsize rule (which may be just a constant). We refer to $\bar{v}_s^{(g,n)}$ as the total variation because it captures deviations that arise due to both measurement noise (the randomness when we compute \hat{v}_s^n) and bias (since $\bar{v}_s^{(g,n-1)}$ is a biased estimate of the mean of \hat{v}_s^n).

We finally need an estimate of the bias. There are two types of bias. In Section 8.1.2 we measured the transient bias that arose from the use of smoothing applied to a nonstationary time series using

$$\bar{\beta}_s^{(g,n)} = (1 - \eta_{n-1}) \bar{\beta}_s^{(g,n-1)} + \eta_{n-1} (\hat{v}^n - \bar{v}_s^{(g,n-1)}). \quad (8.10)$$

This bias arises because the observations \hat{v}^n may be steadily increasing (or decreasing) with the iterations (the usual evolution of the value function that we first saw with value iteration). When we smooth on past observations, we obtain an estimate $\bar{v}_s^{(g,n-1)}$ that tends to underestimate (overestimate if \hat{v}^n tends to decrease) the true mean of \hat{v}^n .

The second form of bias is the aggregation bias given by the difference between the estimate at an aggregate level and the disaggregate level. We compute the aggregation bias using

$$\bar{\mu}_s^{(g,n)} = \bar{v}_s^{(g,n)} - \bar{v}_s^{(0,n)}. \quad (8.11)$$

By the same reasoning presented in Section 8.1.2, we can separate out the effect of bias to obtain an estimate of the variance of the error using

$$(s_s^2)^{(g,n)} = \frac{\bar{v}_s^{(g,n)} - (\bar{\beta}_s^{(g,n)})^2}{1 + \lambda^{n-1}}. \quad (8.12)$$

In the next section we put the estimate of aggregation bias, $\bar{\mu}_s^{(g,n)}$, to work.

The relationships are illustrated in Figure 8.1, which shows a simple function defined over a single continuous state (e.g., the price of an asset). If we select a particular state s , we find we have only two observations for that state, versus seven for that section of the function. If we use an aggregate approximation, we would produce a single number over that range of the function, creating a bias between the true function and the aggregated estimate. As the illustration shows, the size of the bias depends on the shape of the function in that region.

One method for choosing the best level of aggregation is to choose the level that minimizes $(\bar{\sigma}_s^2)^{(g,n)} + (\bar{\mu}_s^{(g,n)})^2$, which captures both bias and variance. In the next section we use the bias and variance to develop a method that uses estimates at all levels of aggregation at the same time.

8.1.4 Combining Multiple Levels of Aggregation

Rather than try to pick the best level of aggregation, it is intuitively appealing to use a weighted sum of estimates at different levels of aggregation. The simplest

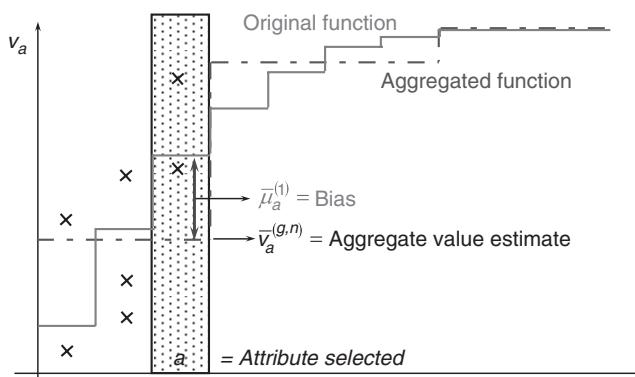


Figure 8.1 Illustration of a disaggregate function, an aggregated approximation, and a set of samples. For a particular state s , we show the estimate and the bias.

strategy is to use

$$\bar{v}_s^n = \sum_{g \in \mathcal{G}} w^{(g)} \bar{v}_s^{(g)}, \quad (8.13)$$

where $w^{(g)}$ is the weight applied to the g th level of aggregation. We would expect the weights to be positive and sum to one, but we can also view these simply as coefficients in a regression function. In such a setting we would normally write the regression as

$$\bar{V}(s|\theta) = \theta_0 + \sum_{g \in \mathcal{G}} \theta_g \bar{v}_s^{(g)}$$

(see Section 8.2.2 for a discussion of general regression methods). The problem with this strategy is that the weight does not depend on the state s . Intuitively it makes sense to put a higher weight on the more disaggregate estimates of the value of states s that have more observations, or where the estimated variance is lower. This behavior is lost if the weight does not depend on s .

In practice, we will generally observe some states much more frequently than others, suggesting that the weights should depend on s . To accomplish this, we need to use

$$\bar{v}_s^n = \sum_{g \in \mathcal{G}} w_s^{(g)} \bar{v}_s^{(g,n)}.$$

Now the weight depends on the state, allowing us to put a higher weight on the disaggregate estimates when we have a lot of observations. This is clearly the most natural, but when the state space is large, we face the challenge of computing thousands (perhaps hundreds of thousands) of weights. If we are going to go this route, we need a fairly simple method to compute the weights.

We can view the estimates $(\bar{v}^{(g,n)})_{g \in \mathcal{G}}$ as different ways of estimating the same quantity. There is an extensive statistics literature on this problem. For example, it is well known that the weights that minimize the variance of \bar{v}_s^n in equation (8.13) are given by

$$w_s^{(g)} \propto ((\bar{\sigma}_s^2)^{(g,n)})^{-1}.$$

Since the weights should sum to one, we obtain

$$w_s^{(g)} = \left(\frac{1}{(\bar{\sigma}_s^2)^{(g,n)}} \right) \left(\sum_{g \in \mathcal{G}} \frac{1}{(\bar{\sigma}_s^2)^{(g,n)}} \right)^{-1}. \quad (8.14)$$

These weights work if the estimates are unbiased, which is clearly not the case. This is easily fixed by using the total variation (variance plus the square of the

bias), producing the weights

$$w_s^{(g,n)} = \frac{1}{(\bar{\sigma}_s^2)^{(g,n)} + (\bar{\mu}_s^{(g,n)})^2} \left[\sum_{g' \in \mathcal{G}} \frac{1}{(\bar{\sigma}_{s'}^2)^{(g',n)} + (\bar{\mu}_{s'}^{(g',n)})^2} \right]^{-1}. \quad (8.15)$$

These are computed for each level of aggregation $g \in \mathcal{G}$. Furthermore we compute a different set of weights for each state s . $(\bar{\sigma}_s^2)^{(g,n)}$ and $\bar{\mu}_s^{(g,n)}$ are easily computed recursively using equations (8.9) and (8.11), which makes the approach well suited to large scale applications. Note that if the stepsize used to smooth \hat{v}^n goes to zero, then the variance $(\bar{\sigma}_s^2)^{(g,n)}$ will also go to zero as $n \rightarrow \infty$. However, the bias $\bar{\beta}_s^{(g,n)}$ will in general not go to zero.

Figure 8.2 shows the average weight put on each level of aggregation (when averaged over all the states s) for a particular application. The behavior of the weights as the number of observation grows illustrates the intuitive property that the weights on the aggregate level are highest when there are only a few observations, with a shift to the more disaggregate level as the algorithm progresses. This is an important behavior to consider when approximating value functions. It is simply not possible to produce good value function approximations with only a few data points, so simple functions (with only a few parameters) should be used.

The weights computed using (8.2) minimize the variance in the estimate $\bar{v}_s^{(g)}$ if the estimates at different levels of aggregation are independent, but this is simply not going to be the case. $\bar{v}_s^{(0)}$ (an estimate of s at the most disaggregate level) and

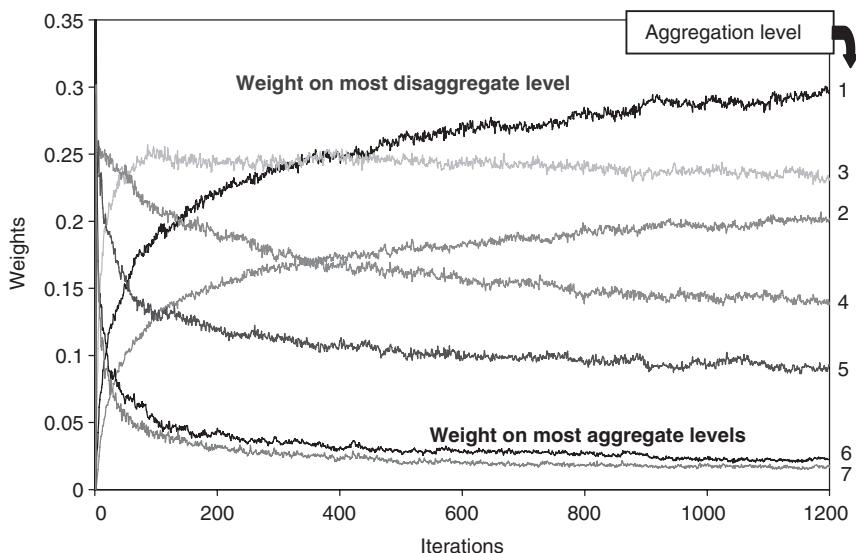


Figure 8.2 Average weight (across all states) for each level of aggregation using equation (8.15).

$\bar{v}_s^{(1)}$ will be correlated since $\bar{v}_s^{(1)}$ is estimated using some of the same observations used to estimate $\bar{v}_s^{(0)}$. So it is fair to ask if the weights produce accurate estimates.

To get a handle on this question, consider the scalar function in Figure 8.3a. At the disaggregate level the function is defined for 10 discrete values. This range is then divided into three larger intervals, and an aggregated function is created by estimating the function over each of these three larger ranges. Instead of using the

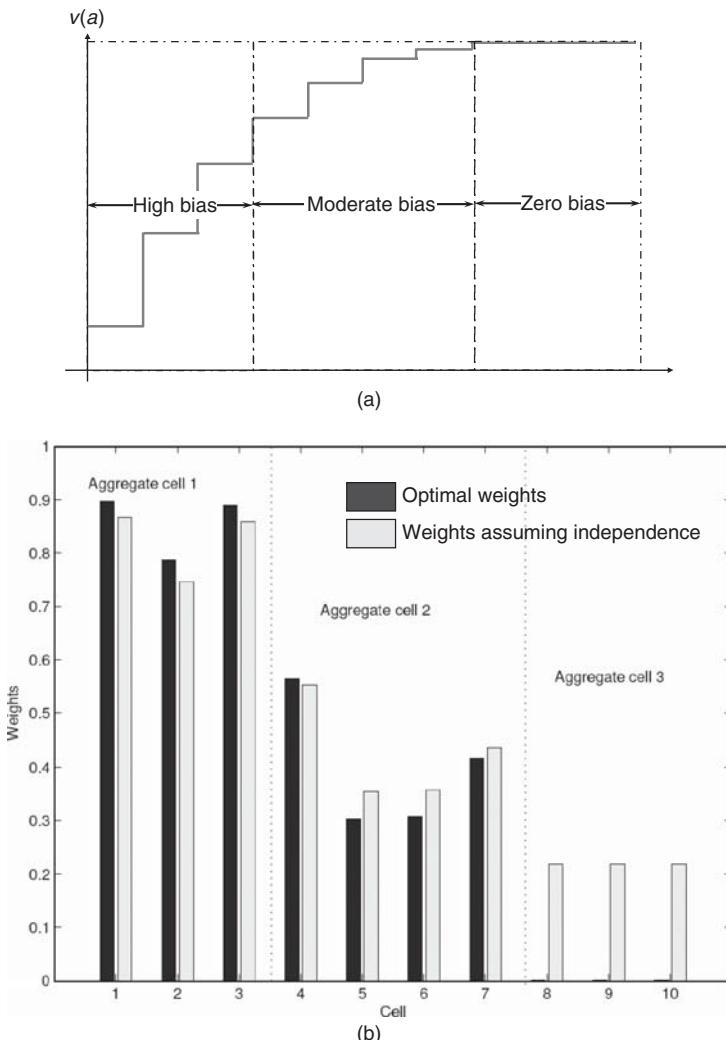


Figure 8.3 Weight given to the disaggregate level for a two-level problem at each of 10 points, with and without the independence assumption. (a) Scalar, nonlinear function; (b) weight given to disaggregate level (from George et al., 2008).

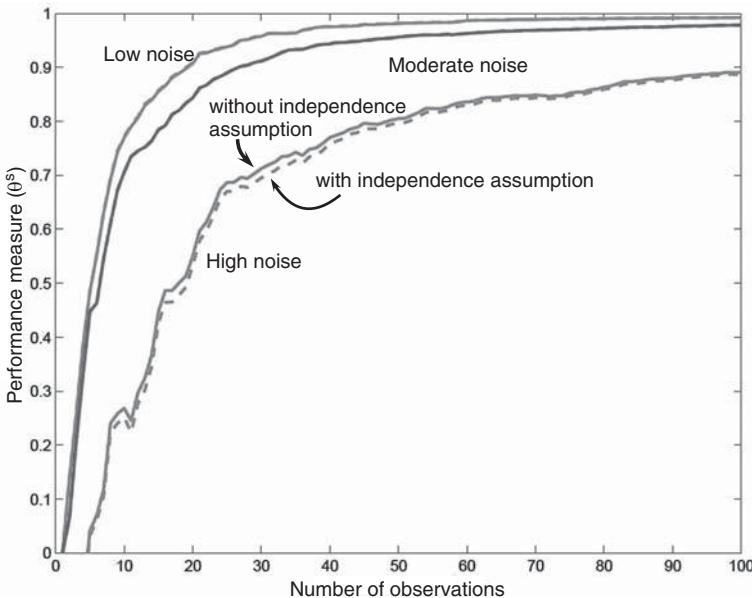


Figure 8.4 Effect of ignoring the correlation between estimates at different levels of aggregation.

weights computed using (8.15), we can fit a regression of the form

$$\hat{v}^n = \theta_0 \bar{v}_s^{(0,n)} + \theta_1 \bar{v}_s^{(1,n)}. \quad (8.16)$$

The parameters θ_0 and θ_1 can be fit using regression techniques. Note that while we would expect $\theta_0 + \theta_1$ to be approximately 1, there is no formal guarantee of this. If we use only two levels of aggregation, then we can find (θ_0, θ_1) using linear regression and can compare these weights to those computed using equation (8.15) where we assume independence.

For this example the weights are shown in Figure 8.3b. The figure illustrates that the weight on the disaggregate level is highest when the function has the greatest slope, which produces the highest biases. When we compute the optimal weights (by which we capture the correlation), the weight on the disaggregate level for the portion of the curve that is flat is zero, as we would expect. Note that when we assume independence, the weight on the disaggregate level (when the slope is zero) is no longer zero. Clearly, a weight of zero is best because it means that we are aggregating all the points over the interval into a single estimate, which is going to be better than trying to produce three individual estimates.

One would expect that using the optimal weights, which captures the correlations between estimates at different levels of aggregation, would also produce better estimates of the function itself. This does not appear to be the case. We compared the errors between the estimated function and the actual function using both methods for computing weights and three levels of noise around the function. The results are shown in Figure 8.4, which indicates that there is virtually no difference in the

accuracy of the estimates produced by the two methods. This observation has held up under a number of experiments.

8.1.5 State Aliasing and Aggregation

An issue that arises when we use aggregation is that two different states (call them S_1 and S_2) may have the same behavior (as a result of aggregation), despite the fact that the states are different, and perhaps should exhibit different behaviors. When this happens, we refer to S_1 as an alias of S_2 (and vice versa). We refer to this behavior as *aliasing*.

In approximate dynamic programming we do not (as a rule) aggregate the state variable as we step forward through time. In most applications, transition functions can handle state variables at a high level of detail. However, we may aggregate the state for the purpose of computing the value function. In this case, if we are in state S_i and take action x_i , we might expect to end up in state S'_i , where $S'_1 \neq S'_2$. But we might have $\bar{V}(S'_1) = \bar{V}(S'_2)$ as a result of aggregation. So it might happen that we make the same decision even though the difference in the state S_1 and S_2 might call for different decisions. Welcome to approximate dynamic programming.

8.2 PARAMETRIC MODELS

Up to now we have focused on lookup table representations of value functions, where if we are in a (discrete) state s , we compute an approximation $\bar{v}(s)$ that is an estimate of the value of being in state s . Using aggregation (even mixtures of estimates at different levels of aggregation) is still a form of lookup table (we are just using a simpler lookup table). An advantage of this strategy is that it avoids the need to exploit specialized structure in the state variable (other than the definition of levels of aggregation). A disadvantage is that it does not allow you to take advantage of structure in the state variable.

There has been considerable interest in estimating value functions using regression methods. A classical presentation of linear regression poses the problem of estimating a parameter vector θ to fit a model that predicts a variable y using a set of observations (known as covariates in the machine learning community) $(x_i)_{i \in \mathcal{I}}$, where we assume a model of the form

$$y = \theta_0 + \sum_{i=1}^I \theta_i x_i + \varepsilon. \quad (8.17)$$

In the language of approximate dynamic programming, the independent variables x_i are created using *basis functions* that reduce potentially large state variables into a small number of *features*, a term widely used in the artificial intelligence community. If we are playing the game of tic-tac-toe, we might want to know if our opponent has captured the center square, or we might want to know the number of corner squares we own. If we are managing a fleet of taxis, we might define for

each region of the city a feature that gives the number of taxis in the region, plus 0.5 times the number of taxis in each of the neighboring regions.

Using this language, instead of an independent variable x_i , we would have a basis function $\phi_f(S)$, where $f \in \mathcal{F}$ is a *feature*. $\phi_f(S)$ might be an indicator variable (e.g., 1 if we have an “X” in the center square of our tic-tac-toe board), a discrete number (the number of X’s in the corners of our tic-tac-toe board), or a continuous quantity (the price of an asset, the amount of oil in our inventories, the amount of AB-negative blood on hand at the hospital). Some problems might have fewer than 10 features, others may have dozens, and some may have hundreds of thousands. In general, however, we would write our value function in the form

$$\bar{V}(S|\theta) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(S).$$

In a time-dependent model the parameter vector θ and basis functions ϕ may also be indexed by time (but not necessarily).

In the remainder of this section, we provide a brief review of linear regression, followed by some examples of regression models. We close with a more advanced presentation that provides insights into the geometry of basis functions (including a better understanding of why they are called “basis functions”). Given the tremendous amount of attention this class of approximations has received in the literature, we defer to Chapter 9 a full description of how to fit linear regression models recursively for an ADP setting.

8.2.1 Linear Regression Review

Let y^n be the n th observation of our dependent variable (what we are trying to predict) based on the observation $(x_1^n, x_2^n, \dots, x_I^n)$ of our independent (or explanatory) variables (the x_i are equivalent to the basis functions we used earlier). Our goal is to estimate a parameter vector θ that solves

$$\min_{\theta} \sum_{m=1}^n \left(y^m - (\theta_0 + \sum_{i=1}^I \theta_i x_i^m) \right)^2. \quad (8.18)$$

This is the standard linear regression problem. Let $\bar{\theta}^n$ be the optimal solution for this problem. Throughout this section we assume that the underlying process from which the observations y^n are drawn is stationary (an assumption that is almost never satisfied in approximate dynamic programming).

If we define $x_0 = 1$, we let

$$x^n = \begin{pmatrix} x_0^n \\ x_1^n \\ \vdots \\ x_I^n \end{pmatrix}$$

be an $I + 1$ -dimensional column vector of observations. Throughout this section, and unlike the rest of the book, we use traditional vector operations, where $x^T x$ is an inner product (producing a scalar) while xx^T is an outer product, producing a matrix of cross terms.

Letting θ be the column vector of parameters, we can write our model as

$$y = \theta^T x + \varepsilon.$$

We assume that the errors $(\varepsilon^1, \dots, \varepsilon^n)$ are independent and identically distributed. We do not know the parameter vector θ , so we replace it with an estimate $\bar{\theta}$ that gives us the predictive formula

$$\bar{y}^n = (\bar{\theta})^T x^n,$$

where \bar{y}^n is our predictor of y^{n+1} . Our prediction error is

$$\hat{\varepsilon}^n = y^n - (\bar{\theta})^T x^n.$$

Our goal is to choose θ to minimize the mean squared error

$$\min_{\theta} \sum_{m=1}^n (y^m - \theta^T x^m)^2. \quad (8.19)$$

It is well known that this can be solved very simply. Let X^n be the n by $I + 1$ matrix

$$X^n = \begin{pmatrix} x_0^1 & x_1^1 & \dots & x_I^1 \\ x_0^2 & x_1^2 & \dots & x_I^2 \\ \vdots & \vdots & \dots & \vdots \\ x_0^n & x_1^n & \dots & x_I^n \end{pmatrix}.$$

Next denote the vector of observations of the dependent variable as

$$Y^n = \begin{pmatrix} y^1 \\ y^2 \\ \vdots \\ y^n \end{pmatrix}.$$

The optimal parameter vector $\bar{\theta}$ (after n observations) is given by

$$\bar{\theta} = [(X^n)^T X^n]^{-1} (X^n)^T Y^n. \quad (8.20)$$

Solving a static optimization problem such as (8.19), which produces the elegant equations for the optimal parameter vector in (8.20), is the most common approach taken by the statistics community. It has little direct application in approximate dynamic programming since our problems tend to be recursive in nature, reflecting

the fact that at each iteration we obtain new observations, which require updates to the parameter vector. In addition our observations tend to be notoriously non-stationary. Later we show how to overcome this problem using the methods of recursive statistics.

8.2.2 Illustrations Using Regression Models

There are many problems where we can exploit structure in the state variable, allowing us to propose functions characterized by a small number of parameters that have to be estimated statistically. Section 8.1.4 represented one version where we had a parameter for each (possibly aggregated) state. The only structure we assumed was implicit in the ability to specify a series of one or more aggregation functions.

The remainder of this section illustrates the use of regression models in specific applications. The examples use a specific method for estimating the parameter vector θ that will typically prove to be somewhat clumsy in practice.

Pricing an American Option

Consider the problem of determining the value of an American-style put option, which gives us the right to sell an asset (or contract) at a specified price at any of a set of discrete time periods. For example, we might be able to exercise the option on the last day of the month over the next 12 months.

Suppose that we have an option that allows us to sell an asset at \$1.20 at any of four time periods. We assume a discount factor of 0.95 to capture the time value of money. If we wait until time period 4, we must exercise the option, receiving zero if the price is over \$1.20. At intermediate periods, however, we may choose to hold the option even if the price is below \$1.20 (of course, exercising it if the price is above \$1.20 does not make sense). Our problem is to determine whether to hold or exercise the option at the intermediate points.

From history we have found 10 samples of price trajectories, which are shown in Table 8.2. If we wait until time period 4, our payoff is shown in Table 8.3, which is zero if the price is above 1.20, and $1.20 - p_4$ for prices below \$1.20.

At time $t = 3$, we have access to the price history (p_1, p_2, p_3) . Since we may not be able to assume that the prices are independent or even Markovian (where p_3 depends only on p_2), the entire price history represents our state variable, along with an indicator that tells us if we are still holding the asset. We wish to predict the value of holding the option at time $t = 4$. Let $V_4(S_4)$ be the value of the option if we are holding it at time 4, given the state (which includes the price p_4) at time 4. Now let the conditional expectation at time 3 be

$$\bar{V}_3(S_3) = \mathbb{E}\{V_4(S_4)|S_3\}.$$

Our goal is to approximate $\bar{V}_3(S_3)$ using information we know at time 3. We propose a linear regression of the form

$$Y = \theta_0 + \theta_1 X_1 + \theta_2 X_2 + \theta_3 X_3,$$

Table 8.2 Ten sample realizations of prices over four time periods

Outcome	Stock Prices			
	Time Period			
	1	2	3	4
1	1.21	1.08	1.17	1.15
2	1.09	1.12	1.17	1.13
3	1.15	1.08	1.22	1.35
4	1.17	1.12	1.18	1.15
5	1.08	1.15	1.10	1.27
6	1.12	1.22	1.23	1.17
7	1.16	1.14	1.13	1.19
8	1.22	1.18	1.21	1.28
9	1.08	1.11	1.09	1.10
10	1.15	1.14	1.18	1.22

Table 8.3 Payout at time 4 if we are still holding the option

Outcome	Option Value at $t = 4$			
	Time Period			
	1	2	3	4
1	—	—	—	0.05
2	—	—	—	0.07
3	—	—	—	0.00
4	—	—	—	0.05
5	—	—	—	0.00
6	—	—	—	0.03
7	—	—	—	0.01
8	—	—	—	0.00
9	—	—	—	0.10
10	—	—	—	0.00

where

$$Y = V_4,$$

$$X_1 = p_2,$$

$$X_2 = p_3,$$

$$X_3 = (p_3)^2.$$

The variables X_1 , X_2 , and X_3 are our basis functions. Keep in mind that it is important that our explanatory variables X_i be a function of information we have at time $t = 3$, although we are trying to predict what will happen at time $t = 4$ (the payoff). We would then set up the data matrix given in Table 8.4.

Table 8.4 Data table for our regression at time 3

Outcome	Regression Data			Dependent Variable Y
	X_1	X_2	X_3	
1	1.08	1.17	1.3689	0.05
2	1.12	1.17	1.3689	0.07
3	1.08	1.22	1.4884	0.00
4	1.12	1.18	1.3924	0.05
5	1.15	1.10	1.2100	0.00
6	1.22	1.23	1.5129	0.03
7	1.44	1.13	1.2769	0.01
8	1.18	1.21	1.4641	0.00
9	1.11	1.09	1.1881	0.10
10	1.14	1.18	1.3924	0.00

We can now run a regression on these data to determine the parameters $(\theta_i)_{i=0}^3$. It makes sense to consider only the paths that produce a positive value in the fourth time period, since these represent the sample paths where we are most likely to still be holding the asset at the end. The linear regression is only an approximation, and it is best to fit the approximation in the region of prices that are the most interesting (we could use the same reasoning to include some “near misses”). We only use the value function to estimate the value of holding the asset, so it is this part of the function we wish to estimate. For our illustration, however, we use all 10 observations, which produces the equation

$$\bar{V}_3 \approx 0.0056 - 0.1234 p_2 + 0.6011 p_3 - 0.3903(p_3)^2.$$

\bar{V}_3 is an approximation of the expected value of the price that we would receive if we hold the option until time period 4. We can now use this approximation to help us decide what to do at time $t = 3$. Table 8.5 compares the value of exercising the option at time 3 against holding the option until time 4, computed as $\gamma \bar{V}_3(S_3)$. Taking the larger of the two payouts, we find, for example, that we would hold the option given samples 1–4, 6, 8, and 10 but would sell given samples 5, 7, and 9.

We can repeat the exercise to estimate $\bar{V}_2(S_t)$. This time our dependent variable Y can be calculated two different ways. The simplest is to take the larger of the two columns from Table 8.5 (marked in bold). So for sample path 1 we would have $Y_1 = \max\{0.03, 0.03947\} = 0.03947$. This means that our observed value is actually based on our approximate value function $\bar{V}_3(S_3)$.

An alternative way of computing the observed value of holding the option in time 3 is to use the approximate value function to determine the decision, but then use the actual price we receive when we eventually exercise the option. Using this method, we receive 0.05 for the first sample path because we decide to hold the asset at time 3 (based on our approximate value function) after which the price of

Table 8.5 Payout if we exercise at time 3, and the expected value of holding based on our approximation

Outcome	Rewards	
	Exercise	Decision
	Exercise	Hold
1	0.03	$0.04155 \times .95 = \mathbf{0.03947}$
2	0.03	$0.03662 \times .95 = \mathbf{0.03479}$
3	0.00	$0.02397 \times .95 = \mathbf{0.02372}$
4	0.02	$0.03346 \times .95 = \mathbf{0.03178}$
5	0.10	$0.05285 \times .95 = 0.05021$
6	0.00	$0.00414 \times .95 = \mathbf{0.00394}$
7	0.07	$0.00899 \times .95 = 0.00854$
8	0.00	$0.01610 \times .95 = \mathbf{0.01530}$
9	0.11	$0.06032 \times .95 = 0.05731$
10	0.02	$0.03099 \times .95 = \mathbf{0.02944}$

^aThe best decision is indicated in bold.

the option turns out to be worth 0.05. Discounted, this is worth 0.0475. For sample path 2 the option proves to be worth 0.07, which discounts back to 0.0665 (we decided to hold at time 3, and the option was worth 0.07 at time 4). For sample path 5 the option is worth 0.10 because we decided to exercise at time 3.

Regardless of which way we compute the value of the problem at time 3, the remainder of the procedure is the same. We have to construct the independent variables Y and regress them against our observations of the value of the option at time 3 using the price history (p_1, p_2). Our only change in methodology would occur at time 1 where we would have to use a different model (because we do not have a price at time 0).

Playing “Lose Tic-Tac-Toe”

The game of “lose tic-tac-toe” is the same as the familiar game of tic-tac-toe, with the exception that now you are trying to make the other person get three in a row. This nice twist on the popular children’s game provides the setting for our next use of regression methods in approximate dynamic programming.

Unlike our exercise in pricing options, representing a tic-tac-toe board requires capturing a discrete state. Assume the cells in the board are numbered left to right, top to bottom as shown in Figure 8.5a. Now consider the board in Figure 8.5b. We can represent the state of the board after the t th play using

$$S_{ti} = \begin{cases} 1 & \text{if cell } i \text{ contains an “X,”} \\ 0 & \text{if cell } i \text{ is blank,} \\ -1 & \text{if cell } i \text{ contains an “O,”} \end{cases}$$

$$S_t = (S_{ti})_{i=1}^9.$$

1	2	3
4	5	6
7	8	9

		X
X	O	O
O	X	O

Figure 8.5 Some tic-tac-toe boards. (8.5a) Our indexing scheme; (8.5b) sample board.

We see that this simple problem has up to $3^9 = 19,683$ states. While many of these states will never be visited, the number of possibilities is still quite large, and seems to overstate the complexity of the game.

We quickly realize that what is important about a game board is not the status of every cell as we have represented it. For example, rotating the board does not change a thing, but it does represent a different state. Also we tend to focus on strategies (early in the game when it is more interesting) such as winning the center of the board or a corner. We might start defining variables (basis functions) such as

- $\phi_1(S_t) = 1$ if there is an “X” in the center of the board, 0 otherwise,
- $\phi_2(S_t) =$ number of corner cells with an “X,”
- $\phi_3(S_t) =$ number of instances of adjacent cells with an “X” (horizontally, vertically, or diagonally).

There are, of course, numerous such functions we can devise, but it is unlikely that we could come up with more than a few dozen (if that) that appear to be useful. It is important to realize that we do not need a value function to tell us to make obvious moves.

Once we form our basis functions, our value function approximation is given by

$$\bar{V}_t(S_t) = \sum_{f \in \mathcal{F}} \theta_{tf} \phi_f(S_t).$$

We note that we have indexed the parameters by time (the number of plays) since this might play a role in determining the value of the feature being measured by a basis function, but it is reasonable to try fitting a model where $\theta_{tf} = \theta_f$. We estimate the parameters θ by playing the game (and following some policy) after which we see if we won or lost. We let $Y^n = 1$ if we won the n th game, 0 otherwise. This also means that the value function is trying to approximate the probability of winning if we are in a particular state.

We may play the game by using our value functions to help determine a policy. Another strategy, however, is simply to allow two people (ideally, experts) to play the game and use this to collect observations of states and game outcomes. This is an example of supervisory learning. If we lack a “supervisor” then we have to depend on simple strategies combined with the use of slowly learned value function approximations. In this case we also have to recognize that in the early iterations

we are not going to have enough information to reliably estimate the coefficients for a large number of basis functions.

8.2.3 A Geometric View of Basis Functions*

For readers comfortable with linear algebra, we can obtain an elegant perspective on the geometry of basis functions. In Section 8.2.1 we found the parameter vector θ for a regression model by minimizing the expected square of the errors between our model and a set of observations. Assume now that we have a “true” value function $V(s)$ that gives the value of being in state s , and let $p(s)$ be the probability of visiting state s . We wish to find the approximate value function that best fits $V(s)$ using a given set of basis functions $(\phi_f(s))_{f \in \mathcal{F}}$. If we minimize the expected square of the errors between our approximate model and the true value function, we would want to solve

$$\min_{\theta} F(\theta) = \sum_{s \in \mathcal{S}} p(s) \left(V(s) - \sum_{f \in \mathcal{F}} \theta_f \phi_f(s) \right)^2, \quad (8.21)$$

where we have weighted the error for state s by the probability of actually being in state s . Our parameter vector θ is unconstrained, so we can find the optimal value by taking the derivative and setting this equal to zero. Differentiating with respect to $\theta_{f'}$ gives

$$\frac{\partial F(\theta)}{\partial \theta_{f'}} = -2 \sum_{s \in \mathcal{S}} p(s) \left(V(s) - \sum_{f \in \mathcal{F}} \theta_f \phi_f(s) \right) \phi_{f'}(s).$$

Setting the derivative equal to zero and rearranging gives

$$\sum_{s \in \mathcal{S}} p(s) V(s) \phi_{f'}(s) = \sum_{s \in \mathcal{S}} p(s) \sum_{f \in \mathcal{F}} \theta_f \phi_f(s) \phi_{f'}(s). \quad (8.22)$$

At this point it is much more elegant to revert to matrix notation. Define an $|\mathcal{S}| \times |\mathcal{S}|$ diagonal matrix D where the diagonal elements are the state probabilities $p(s)$, as follows:

$$D = \begin{pmatrix} p(1) & 0 & 0 \\ 0 & p(2) & 0 \\ \vdots & 0 & \ddots & \vdots \\ 0 & \vdots & & p(|\mathcal{S}|) \end{pmatrix}.$$

Let V be the column vector giving the value of being in each state

$$V = \begin{pmatrix} V(1) \\ V(2) \\ \vdots \\ V(|\mathcal{S}|) \end{pmatrix}.$$

Finally, let Φ be an $|\mathcal{S}| \times |\mathcal{F}|$ matrix of the basis functions given by

$$\Phi = \begin{pmatrix} \phi_1(1) & \phi_2(1) & \cdots & \phi_{|\mathcal{F}|}(1) \\ \phi_1(2) & \phi_2(2) & \cdots & \phi_{|\mathcal{F}|}(2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(|\mathcal{S}|) & \phi_2(|\mathcal{S}|) & \cdots & \phi_{|\mathcal{F}|}(|\mathcal{S}|) \end{pmatrix}.$$

Recognizing that equation (8.22) is for a particular feature f' , with some care it is possible to see that equation (8.22) for all features is given by the matrix equation

$$\Phi^T D V = \Phi^T D \Phi \theta. \quad (8.23)$$

It helps to keep in mind that Φ is an $|\mathcal{S}| \times |\mathcal{F}|$ matrix, D is an $|\mathcal{S}| \times |\mathcal{S}|$ diagonal matrix, V is an $|\mathcal{S}| \times 1$ column vector, and θ is an $|\mathcal{F}| \times 1$ column vector. The reader should carefully verify that (8.23) is the same as (8.22).

Now pre-multiply both sides of (8.23) by $(\Phi^T D \Phi)^{-1}$. This gives us the optimal value of θ as

$$\theta = (\Phi^T D \Phi)^{-1} \Phi^T D V. \quad (8.24)$$

This equation is closely analogous to the normal equations of linear regression, given by equation (8.20), with the only difference being the introduction of the scaling matrix D that captures the probability that we are going to visit a state.

Now pre-multiply both sides of (8.24) by Φ , which gives

$$\Phi \theta = \bar{V} = \Phi (\Phi^T D \Phi)^{-1} \Phi^T D V.$$

$\Phi \theta$ is, of course, our approximation of the value function, which we have denoted by \bar{V} . This, however, is the best possible value function given the set of functions $\phi = (\phi_f)_{f \in \mathcal{F}}$. If the vector ϕ formed a complete basis over the space formed by the value function $V(s)$ and the state space \mathcal{S} , then we would obtain $\Phi \theta = \bar{V} = V$. Since this is generally not the case, we can view \bar{V} as the nearest point projection (where “nearest” is defined as a weighted measure using the state probabilities $p(s)$) onto the space formed by the basis functions. In fact, we can form a projection operator Π defined by

$$\Pi = \Phi (\Phi^T D \Phi)^{-1} \Phi^T D$$

so that $\bar{V} = \Pi V$ is the value function closest to V that can be produced by the set of basis functions.

This discussion brings out the geometric view of basis functions (and at the same time, the reason why we use the term “basis function”). There is an extensive literature on basis functions that has evolved in the approximation literature.

8.3 REGRESSION VARIATIONS

Regression comes in many flavors. This section serves primarily as a reminder that there is more to regression than classical linear regression.

8.3.1 Shrinkage Methods

A common problem in regression is estimates of the regression parameters that are nonzero, but where the corresponding explanatory variable does not provide any predictive value. An effective strategy is to add a term to the objective function that penalizes nonzero regression parameters in some way. Ridge regression does this by penalizing the sum of squares of the regression coefficients, giving us the objective function ridge regression

$$\bar{Y}(x) = \min_{\theta} \left(\sum_{i=1}^n (y^i - \theta^T x^i)^2 + \eta \sum_{i=1}^p \theta_i^2 \right). \quad (8.25)$$

A limitation of ridge regression is that it can still produce small but nonzero regression parameters. A variant of the LASSO method minimizes the weighted sum of regression coefficients, lasso regression:

$$\bar{Y}(x) = \min_{\theta} \left(\sum_{i=1}^n (y^i - \theta^T x^i)^2 + \eta \sum_{i=1}^p |\theta_i| \right). \quad (8.26)$$

These methods can play a role in feature selection. Using the language of basis functions, we can write an independent variable $x_f = \phi_f(S)$. Now imagine that we create a large number of basis functions (independent variables), but we are not sure if all of them are useful. Shrinkage methods can be used to help identify the basis functions that are most important.

8.3.2 Support Vector Regression

Support vector machines (for classification) and support vector regression (for continuous problems) have attracted considerable interest in the machine learning community. For the purpose of fitting value function approximations, we are primarily interested in support vector regression, but we can also use regression to fit policy function approximations, and if we have discrete actions, we may be interested in classification. For the moment we focus on fitting continuous functions.

Support vector regression, in its most basic form, is linear regression with a different objective than simply minimizing the sum of the squares of the errors. With support vector regression, we consider two goals. First, we wish to minimize the absolute sum of deviations that are larger than a set amount ξ . Second, we wish to minimize the regression parameters, to push as many as possible close to zero.

As before, we let our predictive model be given by

$$y = \theta x + \epsilon.$$

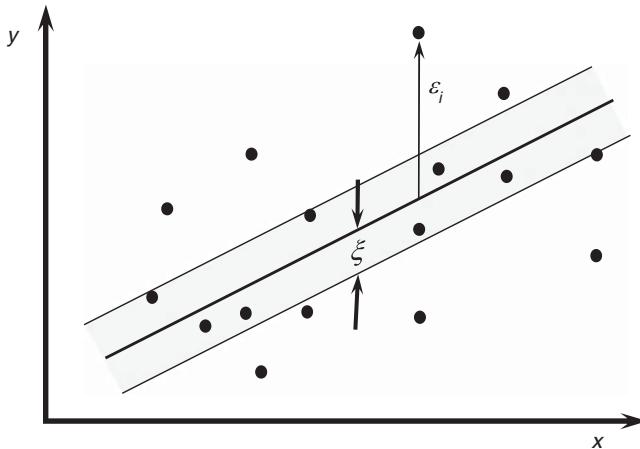


Figure 8.6 Penalty structure for support vector regression. Deviations within the gray area are assessed a value of zero. Deviations outside the gray area are measured based on their distance to the gray area.

Let $\epsilon^i = y^i - \theta x^i$ be the error. We then choose θ by solving the following optimization problem:

$$\min_{\theta} \left(\frac{\eta}{2} \|\theta\|^2 + \sum_{i=1}^n \max\{0, |\epsilon^i| - \xi\} \right). \quad (8.27)$$

The first term penalizes positive values of θ , encouraging the model to minimize values of θ unless they contribute in a significant way to producing a better model. The second term penalizes errors that are greater than ξ . The parameters η and ξ are both tunable parameters. The error ϵ^i and error margin ξ are illustrated in Figure 8.6.

It can be shown by solving the dual that the optimal value of θ and the best fit $\bar{Y}(x)$ have the form

$$\begin{aligned} \theta &= \sum_{i=1}^n (\bar{\beta}^i - \bar{\alpha}^i) x^i, \\ \bar{Y}(x) &= \sum_{i=1}^n (\bar{\beta}^i - \bar{\alpha}^i) (x^i)^T x^i. \end{aligned}$$

Here $\bar{\beta}^i$ and $\bar{\alpha}^i$ are scalars found by solving

$$\min_{\bar{\beta}^i, \bar{\alpha}^i} \xi \sum_{i=1}^n (\bar{\beta}^i + \bar{\alpha}^i) - \sum_{i=1}^n y^i (\bar{\beta}^i + \bar{\alpha}^i) + \frac{1}{2} \sum_{i=1}^n \sum_{i'=1}^n (\bar{\beta}^i + \bar{\alpha}^i)(\bar{\beta}^{i'} + \bar{\alpha}^{i'}) (x^i)^T x^{i'}$$

subject to the constraints

$$\begin{aligned} 0 \leq \bar{\alpha}^i, \bar{\beta}^i &\leq \frac{1}{\eta}, \\ \sum_{i=1}^n (\bar{\beta}^i - \bar{\alpha}^i) &= 0, \\ \bar{\alpha}^i \bar{\beta}^i &= 0. \end{aligned}$$

A very rich field of study has evolved around support vector machines and support vector regression. For a thorough tutorial, see Smola and Schölkopf (2004). A shorter and more readable introduction is contained in chapter 12 of Hastie et al. (2009). Note that SVR does not lend itself readily to recursive updating, which we suspect will limit its usefulness in approximate dynamic programming.

8.4 NONPARAMETRIC MODELS

The power of parametric models is matched by their fundamental weakness: they are only effective if you can design an effective parametric model, and this remains a frustrating art. For this reason nonparametric statistics have attracted recent attention. They avoid the art of specifying a parametric model but introduce other complications. Nonparametric methods work primarily by building local approximations to functions using observations rather than depending on functional approximations.

There is an extensive literature on the use of approximation methods for continuous functions. These problems, which arise in many applications in engineering and economics, require the use of approximation methods that can adapt to a wide range of functions. Interpolation techniques, orthogonal polynomials, Fourier approximations and splines are just some of the most popular techniques. Often these methods are used to closely approximate the expectation without the use of Monte Carlo methods. We do not cover these techniques in this book, but instead refer readers to Chapter 6 of Judd (1998) for a very nice review of approximation methods (see also chapter 12 of the same volume for their use in dynamic programming).

In this section we review some of the nonparametric methods that have received the most attention within the approximate dynamic programming community. This is an active area of research that offers tremendous potential, but significant hurdles remain before this approach can be widely adopted.

8.4.1 k -Nearest Neighbor

Perhaps the simplest form of nonparametric regression forms estimates of functions by using a weighted average of the k -nearest neighbors. As above, we assume we have a response y^n corresponding to a measurement $x^n = (x_1^n, x_2^n, \dots, x_I^n)$. Let $\rho(x, x^n)$ be a distance metric between a query point x (in dynamic programming,

this would be a state), and an observation x^n . Then let $\mathcal{N}^n(x)$ be the set of the k -nearest points to the query point x , where clearly we require $k \leq n$. Finally, let $\bar{Y}^n(x)$ be the response function, which is our best estimate of the true function $Y(x)$ given the observations x^1, \dots, x^n . When we use a k -nearest neighbor model, this is given by

$$\bar{Y}^n(x) = \frac{1}{k} \sum_{n \in \mathcal{N}^n(x)} y^n. \quad (8.28)$$

Thus our best estimate of the function $Y(x)$ is made by averaging the k points nearest to the query point x .

Using a k -nearest neighbor model requires, of course, choosing k . Not surprisingly, we obtain a perfect fit of the data by using $k = 1$ if we base our error on the training dataset.

A weakness of this logic is that the estimate $\bar{Y}^n(x)$ can change abruptly as x changes continuously, as the set of nearest neighbors changes. An effective way of avoiding this behavior is using kernel regression, which uses a weighted sum of all data points.

8.4.2 Kernel Regression

Kernel regression has attracted considerable attention in the statistical learning literature. As with k -nearest neighbor, kernel regression forms an estimate $\bar{Y}(x)$ by using a weighted sum of prior observations, which we can write generally as

$$\bar{Y}^n(x) = \frac{\sum_{m=1}^n K_h(x, x^m) y^m}{\sum_{m=1}^n K_h(x, x^m)} \quad (8.29)$$

where $K_h(x, x^m)$ is a weighting function that declines with the distance between the query point x and the measurement x^m . h is referred to as the *bandwidth* which plays an important scaling role. There are many possible choices for the weighting function $K_h(x, x^m)$. One of the most popular is the Gaussian kernel, given by

$$K_h(x, x^m) = e^{-(\|x - x^m\|/h)^2}.$$

where $\|\cdot\|$ is the Euclidean norm. Here h plays the role of the standard deviation. Note that the bandwidth h is a tunable parameter that captures the range of influence of a measurement x^m . The Gaussian kernel, often referred to as *radial basis functions* in the ADP literature, provide a smooth, continuous estimate $\bar{Y}^n(x)$. Another popular choice of kernel function is the symmetric beta family, given by

$$K_h(x, x^m) = \max(0, (1 - \|x - x^m\|)^2)^h.$$

Here h is a nonnegative integer. $h = 1$ gives the uniform kernel, $h = 2$ gives the Epanechnikov kernel, and $h = 3$ gives the biweight kernel. Figure 8.7 illustrates each of these four kernel functions.

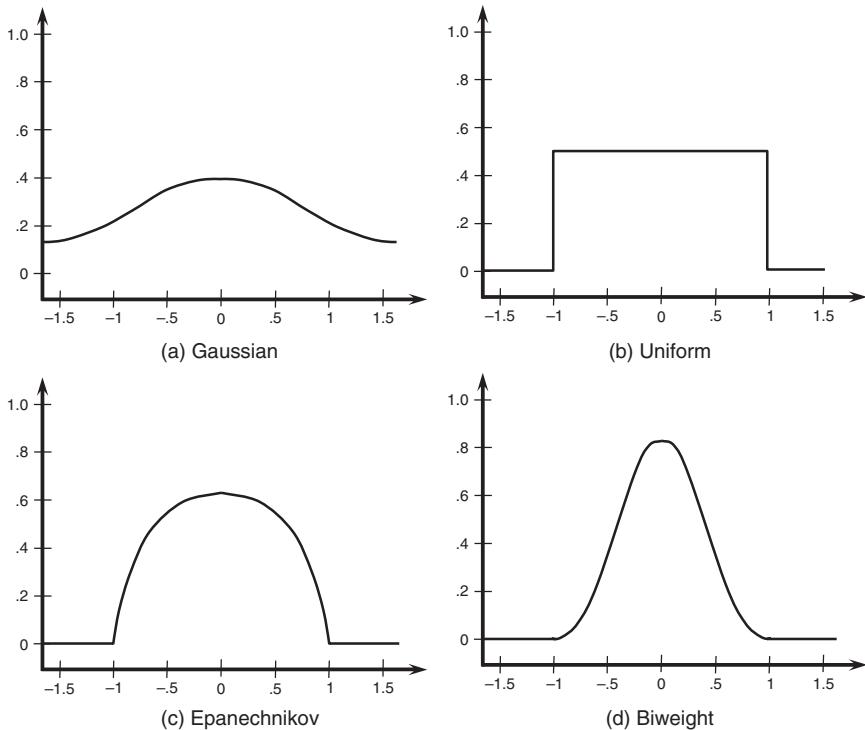


Figure 8.7 Gaussian, uniform, Epanechnikov, and biweight kernel weighting functions.

We pause to briefly discuss some issues surrounding k -nearest neighbors and kernel regression. First, it is fairly common in the ADP literature to see k -nearest neighbors and kernel regression being treated as a form of aggregation. The process of giving a set of states that are aggregated together has a certain commonality with k -nearest neighbor and kernel regression, where points near each other will produce estimates of $Y(x)$ that are similar. But this is where the resemblance ends. Simple aggregation is actually a form of parametric regression using dummy variables, and it offers neither the continuous approximations nor the asymptotic unbiasedness of kernel regression.

Kernel regression is a method of approximation that is fundamentally different from linear regression and other parametric models. Parametric models use an explicit estimation step, where each observation results in an update to a vector of parameters. At any point in time, our approximation consists of the pre-specified parametric model, along with the current estimates of the regression parameters. With kernel regression all we do is store data until we need an estimate of the function at a query point. Only then do we trigger the approximation method, which requires looping over all previous observations, a step that clearly can become expensive as the number of observations grow.

Kernel regression enjoys an important property from an old result known as Mercer's theorem. The result states that there exists a set of basis functions $\phi_f(S)$, $f \in \mathcal{F}$, possibly of very high dimensionality, where

$$K_h(S, S') = \phi(S)^T \phi(S'),$$

as long as the kernel function $K_h(S, S')$ satisfies some basic properties (satisfied by the kernels listed above). In effect this means that using appropriately designed kernels is equivalent to finding potentially very high dimensional basis functions, without having to actually create them.

Unfortunately, the news is not all good. First, there is the annoying dimension of bandwidth selection, although this can be mediated partially by scaling the explanatory variables. More seriously, kernel regression (and this includes k -nearest neighbors), cannot be immediately applied to problems with more than about five dimensions (and even this can be a stretch). The problem is that these methods are basically trying to aggregate points in a multidimensional space. As the number of dimensions grows, the density of points in the d -dimensional space becomes quite sparse, making it very difficult to use "nearby" points to form an estimate of the function. A strategy for high-dimensional applications is to use separable approximations. These methods have received considerable attention in the broader machine learning community but have not been widely tested in an ADP setting.

8.4.3 Local Polynomial Regression

Classical kernel regression uses a weighted sum of responses y^n to form an estimate of $Y(x)$. An obvious generalization is to estimate locally linear regression models around each point x^n by solving a least squares problem that minimizes a weighted sum of least squares. Let $\bar{Y}^n(x|x^k)$ be a linear model around the point x^k , formed by minimizing the weighted sum of squares given by

$$\min_{\theta} \left(\sum_{m=1}^n K_h(x^k, x^m) (y^m - (\sum_{i=1}^I \theta_i x_i^m))^2 \right). \quad (8.30)$$

Thus we are solving a classical linear regression problem, but we do this for each point x^k , and we fit the regression using all the other points (y^m, x^m) , $m = 1, \dots, n$. However, we weight deviations between the fitted model and each observation y^m by the kernel weighting factor $K_h(x^k, x^m)$, which is centered on the point x^k .

Local polynomial regression offers significant advantages in modeling accuracy, but with a significant increase in complexity.

8.4.4 Neural Networks

Neural networks represent an unusually powerful and general class of approximation strategies that have been widely used in approximate dynamic programming,

primarily in classic engineering applications. There are a number of excellent textbooks on the topic, so our presentation is designed only to introduce the basic idea and encourage readers to experiment with this technology if simpler models are not effective.

Up to now we have considered approximation functions of the form

$$\bar{V}(S) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(S),$$

where \mathcal{F} is our set of features, and $(\phi_f(S))_{f \in \mathcal{F}}$ are the basis functions that extract what are felt to be the important characteristics of the state variable that explain the value of being in a state. We have seen that when we use an approximation that is linear in the parameters, we can estimate the parameters θ recursively using standard methods from linear regression. For example, if R_i is the number of resources of type i , our approximation might look like

$$\bar{V}(R|\theta) = \sum_{i \in \mathcal{I}} (\theta_{1i} R_i + \theta_{2i} R_i^2).$$

Now assume that we feel that the best function might not be quadratic in R_i , but we are not sure of the precise form. We might want to estimate a function of the form

$$\bar{V}(R|\theta) = \sum_{i \in \mathcal{I}} (\theta_{1i} R_i + \theta_{2i} R_i^{\theta_3}).$$

So we have a function that is nonlinear in the parameter vector $(\theta_1, \theta_2, \theta_3)$, where θ_1 and θ_2 are vectors and θ_3 is a scalar. If we have a training dataset of state-value observations, $(\hat{v}_n, R_n)_{n=1}^N$, we can find θ by solving

$$\min_{\theta} \sum_{n=1}^N (\hat{v}_n - \bar{V}(R_n|\theta))^2,$$

which generally requires the use of nonlinear programming algorithms. One challenge is that nonlinear optimization problems do not lend themselves to the simple recursive updating equations that we obtained for linear (in the parameters) functions. But more problematic is that we have to experiment with various functional forms to find the one that fits best.

Neural networks offer a much more flexible set of architectures, and at the same time can be updated recursively. The technology has matured to the point that there are a number of commercial packages available that implement the algorithms. However, applying the technology to specific dynamic programming problems can be a nontrivial challenge. In addition it is not possible to know in advance which problem classes will benefit most from the additional generality, in contrast with the simpler strategies that we have covered in this chapter.

Neural networks are ultimately a form of statistical model which, for our application, predicts the value of being in a state as a function of the state, using a

series of observations of states and values. The simplest neural network is nothing more than a linear regression model. If we are in post-decision state S_t^a , we are going to make the random transition $S_{t+1} = S^M(S_t, a_t, W_{t+1}(\omega))$ and then observe a random value \hat{v}_{t+1} from being in state S_{t+1} . We would like to estimate a statistical function $f_t(S_t^a)$ (the same as $\bar{V}_t(S_t^a)$) that predicts \hat{v}_{t+1} . To write this in the traditional notation of regression, let $X_t = S_t^a$ where $X_t = (X_{t1}, X_{t2}, \dots, X_{tI})$ (we assume that all the components X_{ti} are numerical). If we use a linear model, we might write

$$f_t(X_t) = \theta_0 + \sum_{i=1}^I \theta_i X_{ti}.$$

In the language of neural networks, we have I inputs (we have $I + 1$ parameters since we also include a constant term), which we wish to use to estimate a single output \hat{v}_{t+1} (a random observation of being in a state). The relationships are illustrated in Figure 8.8 where we show the I inputs that are then “flowed” along the links to produce $f_t(X_t)$. After this we learn the sample realization \hat{v}_{t+1} that we were trying to predict, which allows us to compute the error $\epsilon_{t+1} = \hat{v}_{t+1} - f_t(X_t)$. We would like to find a vector θ that solves

$$\min_{\theta} \mathbb{E} \frac{1}{2} (f_t(X_t) - \hat{v}_{t+1})^2.$$

Let $F(\theta) = \mathbb{E}(0.5(f_t(X_t) - \hat{v}_{t+1})^2)$, and let $F(\theta, \hat{v}_{t+1}(\omega)) = 0.5(f_t(X_t) - \hat{v}_{t+1}(\omega))^2$, where $\hat{v}_{t+1}(\omega)$ is a sample realization of the random variable $\hat{v}_{t+1}(\omega)$.

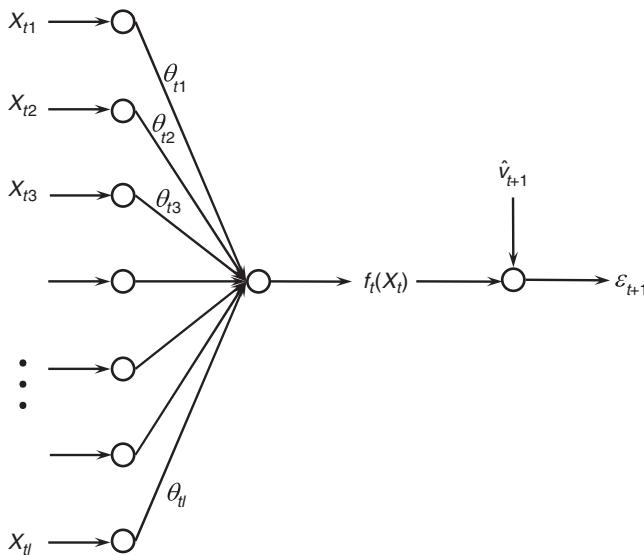


Figure 8.8 Neural networks with a single layer.

As before, we can solve this iteratively using a stochastic gradient algorithm

$$\theta_{t+1} = \theta_t - \alpha_t \nabla_\theta F(\theta_t, \hat{v}_{t+1}(\omega)),$$

where $\nabla_\theta F(\theta_t, \hat{v}_{t+1}(\omega)) = \epsilon_{t+1}$.

We illustrated our linear model by assuming that the inputs were the individual dimensions of the state variable which we denoted X_{ti} . We may not feel that this is the best way to represent the state of the system (imagine representing the states of a Connect-4 game board). We may feel it is more effective (and certainly more compact) if we have access to a set of basis functions $\phi_f(S_t)$, $f \in \mathcal{F}$, where $\phi_f(S_t)$ captures a relevant feature of our system. In this case we would be using our standard basis function representation, where each basis function provides one of the inputs to our neural network.

This was a simple illustration, but it shows that if we have a linear model, we get the same basic class of algorithms that we have already used. A richer model, given in Figure 8.9, illustrates a more classical neural network. Here the “input signal” X_t (this can be the state variable or the set of basis functions) is communicated through several layers. Let $X_t^{(1)} = X_t$ be the input to the first layer (recall that X_{ti} might be the i th dimension of the state variable itself, or a basis function). Let $\mathcal{I}^{(1)}$ be the set of inputs to the first layer (e.g., the set of basis functions).

Here the first linear layer produces J outputs given by

$$Y_{tj}^{(2)} = \sum_{i \in \mathcal{I}^{(1)}} \theta_{ij}^{(1)} X_{ti}^{(1)}, \quad j \in \mathcal{I}^{(2)}.$$

$Y_{tj}^{(2)}$ becomes the input to a nonlinear *perceptron* node that is characterized by a nonlinear function that may dampen or magnify the input. A typical functional

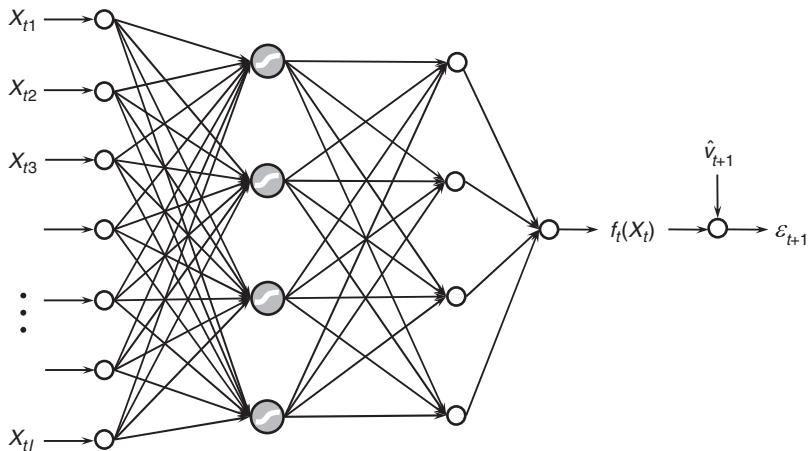


Figure 8.9 Three-layer neural network.

form for a perceptron node is the logistics function given by

$$\sigma(y) = \frac{1}{1 + e^{-\beta y}},$$

where β is a scaling coefficient. The function $\sigma(y)$ is illustrated in Figure 8.10. $\sigma(x)$ introduces nonlinear behavior into the communication of the “signal” X_t .

We next calculate

$$X_{ti}^{(2)} = \sigma(Y_{ti}^{(2)}), \quad i \in \mathcal{I}^{(2)}$$

and use $X_{ti}^{(2)}$ as the input to the second linear layer. We then compute

$$Y_{tj}^{(3)} = \sum_{i \in \mathcal{I}^{(2)}} \theta_{ij}^{(2)} X_{ti}^{(2)}, \quad j \in \mathcal{I}^{(3)}.$$

Finally, we compute the single output using

$$f_t = \sum_{i \in \mathcal{I}^{(3)}} \theta_i^{(3)} X_{ti}^{(3)}.$$

As before, f_t is our estimate of the value of the input X_t . This is effectively our value function approximation $\bar{V}_t(S_t^x)$ that we update using the observation \hat{v}_{t+1} . We update the parameter vector $\theta = (\theta^{(1)}, \theta^{(2)}, \theta^{(3)})$ using the same stochastic gradient algorithms we used for a single layer network. The only difference is that the derivatives have to capture the fact that changing $\theta^{(1)}$, for example, impacts the “flows” through the rest of the network. The derivatives are slightly more difficult to compute, but the logic is basically the same.

Our presentation above assumes that there is a single output, which is to say that we are trying to match a scalar quantity \hat{v}_{t+1} , the observed value of being in

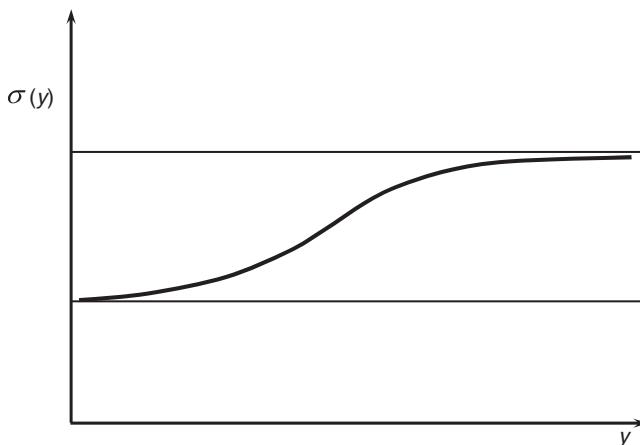


Figure 8.10 Illustrative logistics function for introducing nonlinear behavior into neural networks.

a state. In some settings, \hat{v}_{t+1} might be a vector. For example, in Chapter 13 (see, in particular, Section 13.1), we describe problems where \hat{v}_{t+1} is the gradient of a value function, which of course would be multidimensional. In this case, f_t would also be a vector which would be estimated using

$$f_{tj} = \sum_{i \in \mathcal{I}^{(3)}} \theta_{ij}^{(3)} X_{ti}^{(3)}, \quad j \in \mathcal{I}^{(4)},$$

where $|\mathcal{I}^{(4)}|$ is the dimensionality of the output layer (that is, the dimensionality of \hat{v}_{t+1}).

This presentation should be viewed as nothing more than a very simple illustration of an extremely rich field. The advantage of neural networks is that they offer a much richer class of nonlinear functions (“nonlinear architectures” in the language of neural networks) that can be trained in an iterative way, consistent with the needs of approximate dynamic programming. This said, neural networks are no panacea (a statement that can be made about almost anything). As with our simple linear models, the updating mechanisms struggle with scaling (the units of X_t and the units of \hat{v}_{t+1} may be completely different) that has to be handled by the parameter vectors $\theta^{(\ell)}$. Neural networks typically introduce significantly more parameters that can also introduce problems with stability. There are significant opportunities for taking advantage of problem structure, which can be reflected both in the design of the inputs (as with the choice of basis functions) as well as the density of the internal networks (these do not have to be dense).

8.4.5 Indexed Functions, Tree Structures, and Clustering

There are many problems where we feel comfortable specifying a simple set of basis functions for some of the parameters, but we do not have a good feel for the nature of the contribution of other parameters. For example, we may wish to plan how much energy to hold in storage over the course of the day. Let R_t be the amount of energy stored at time t , and let H_t be the hour of the day. Our state variable might be $S_t = (R_t, H_t)$. We feel that the value of energy in storage is a concave function in R_t , but this value depends in a complex way on the hour of day. It would not make sense, for example, to specify a value function approximation using

$$\bar{V}(S_t) = \theta_0 + \theta_1 R_t + \theta_2 R_T^2 + \theta_3 H_t + \theta_4 H_t^2.$$

There is no reason to believe that the hour of day will be related to the value of energy storage in any convenient way. Instead, we can estimate a function $\bar{V}(S_t | H_t)$ given by

$$\bar{V}(S_t | h) = \theta_0(h) + \theta_1(h) R_t + \theta_2(h) R_T^2.$$

What we are doing here is estimating a linear regression model for each value of $h = H_t$. This is simply a form of lookup table using regression given a particular

value of the complex variables. Imagine that we can divide our state variable S_t into two sets: the first set, f_t , contains variables where we feel comfortable capturing the relationship using linear regression. The second set, g_t , includes more complex variables whose contribution is not as easily approximated. If g_t is a discrete scalar (e.g., as hour of day), we can consider estimating a regression model for each value of g_t . However, if g_t is a vector (possibly with continuous dimensions), then there will be too many possible values.

When the vector g_t cannot be enumerated, we can resort to various clustering strategies. These fall under names such as regression trees and local polynomial regression (a form of kernel regression). These methods cluster g_t (or possibly the entire state S_t) and then fit simple regression models over subsets of data. In this case we would create a set of clusters \mathcal{C}^n based on n observations of states and values. We then fit a regression function $\bar{V}(S_t|c)$ for each cluster $c \in \mathcal{C}^n$. In traditional batch statistics this process proceeds in two stages: clustering and then fitting. In approximate dynamic programming we have to deal with the fact that we may change our clusters as we collect additional data.

A much more sophisticated strategy is based on a concept known as Dirichlet process mixtures. This is a fairly sophisticated technique, but the essential idea is that you form clusters that produce good fits around local polynomial regressions. However, unlike traditional cluster-then-fit methods, the idea with Dirichlet process mixtures is that membership in a cluster is probabilistic, where the probabilities depend on the query point (e.g., the state whose value we are trying to estimate).

8.5 APPROXIMATIONS AND THE CURSE OF DIMENSIONALITY

There are many applications where state variables have multiple, possibly continuous dimensions. In some applications the number of dimensions can number in the thousands.

■ EXAMPLE 8.5

An unmanned aerial vehicle may be described by location (three dimensions), velocity (three dimensions), and acceleration (three dimensions), in addition to fuel level. All 10 dimensions are continuous. ■

■ EXAMPLE 8.6

A utility is trying to plan the amount of energy that should be put in storage as a function of the wind history (six hourly measurements), the history of electricity spot prices (six measurements), and the demand history (six measurements). ■

■ EXAMPLE 8.7

A trader is designing a policy for selling an asset that is priced against a basket of 20 securities, creating a 20-dimensional state variable. ■

■ EXAMPLE 8.8

A car rental company has to manage its inventory of 12 different types of cars spread among 500 car rental locations, creating an inventory vector with 6000 dimensions. ■

Each of these problems has a multidimensional state vector, and in all but the last example the dimensions are continuous. In the car rental example, the inventories will be discrete, but potentially fairly large (a major rental lot may have dozens of each type of car).

If we have 10 dimensions, and discretize each dimension into 100 elements, our state space is $100^{10} = 10^{20}$, which is clearly a very large number. A reasonable strategy might be to aggregate. Instead of discretizing each dimension into 100 elements, what if we discretize into 5 elements? Now our state space is $5^{10} = 9.76 \times 10^6$, or almost 10 million states. Much smaller, but still quite large. Figure 8.11 illustrates the growth in the state space with the number of dimensions.

Approximating high-dimensional functions is fundamentally intractable, and is not related to the specific approximation strategy. If we use a second-order parametric representation, we might approximate a two-dimensional function using

$$V(S) \approx \theta_0 + \theta_1 S_1 + \theta_2 S_2 + \theta_{11} S_1^2 + \theta_{22} S_2^2 + \theta_{12} S_1 S_2.$$

If we have N dimensions, the approximation would look like

$$V(S) \approx \theta_0 + \sum_{i=1}^N \theta_i S_i + \sum_{i_1=1}^N \sum_{i_2=1}^N \theta_{ij} S_i S_j,$$

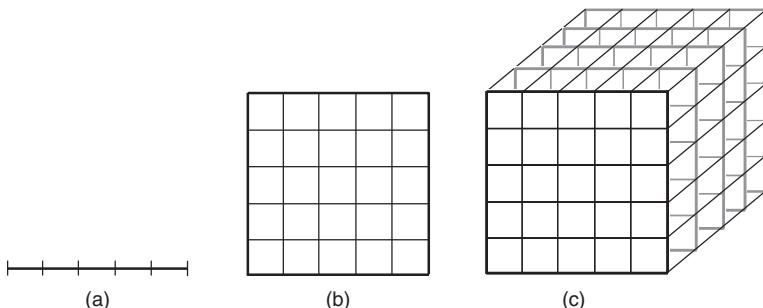


Figure 8.11 Effect of higher dimensions on the number of grids in an aggregated state space.

which means we have to estimate $1 + N + N^2$ parameters. As N grows, this grows very quickly, and this is only a second-order approximation. If we allow N th-order interactions, the approximation would look like

$$\begin{aligned} V(S) &\approx \theta_0 + \sum_{i=1}^N \theta_i S_i + \sum_{i_1=1}^N \sum_{i_2=1}^N \theta_{i_1 i_2} S_{i_1} S_{i_2} \\ &+ \sum_{i_1=1}^N \sum_{i_2=1}^N \cdots \sum_{i_N=1}^N \theta_{i_1, i_2, \dots, i_N} S_{i_1} S_{i_2} \cdots S_{i_N}. \end{aligned}$$

The number of parameters we now have to estimate is given by $1 + N + N^2 + N^3 + \cdots + N^N$. Not surprisingly, this becomes intractable even for relatively small values of N .

The problem follows us if we were to use kernel regression, where an estimate of a function at a point s can be estimated from a series of observations $(\hat{v}^i, s^i)_{i=1}^N$ using

$$V(s) \approx \frac{\sum_{i=1}^N \hat{v}^i k(s, s^i)}{\sum_{i=1}^N k(s, s^i)}$$

where $k(s, s^i)$ might be the Gaussian kernel

$$k(s, s^i) = e^{-\|s - s^i\|^2/b}$$

where b is a bandwidth. Kernel regression is effectively a soft form of the aggregation depicted in Figure 8.11c. The problem is that we would have to choose a bandwidth that covers most of the data to get a statistically reliable estimate of a single point.

To see this, imagine that our observations are uniformly distributed in an N -dimensional cube that measures 1.0 on each side, which means it has a volume of 1.0. If we carve out an N -dimensional cube that measures 0.5 on a side, then this would capture 12.5 percent of the observations in a three-dimensional cube, and 0.1 percent of the observations in a 10-dimensional cube. If we would like to choose a cube that captures $\eta = 0.1$ of our cube, we would need a cube that measures $r = \eta^{1/N} = 0.1^{1/10} = 0.794$, which means that our cube is covering almost 80 percent of the range of each input dimension.

The problem is that we have a multidimensional function, and we are trying to capture the joint behavior of all N dimensions. If we are willing to live with separable approximations, then we can scale to very large number of dimensions. For example, the approximation

$$V(S) \approx \theta_0 + \sum_{i=1}^N \theta_{1i} S_i + \sum_{i=1}^N \theta_{2i} S_i^2,$$

captures quadratic behavior but without any cross terms. The number of parameters is $1 + 2N$, which means we may be able to handle very high-dimensional

problems. However, we lose the ability to handle interactions among different dimensions.

8.6 WHY DOES IT WORK?**

8.6.1 Correlations in Hierarchical Estimation

It is possible to derive the optimal weights for the case where the statistics $\bar{v}_s^{(g)}$ are not independent. In general, if we are using a hierarchical strategy and have $g' > g$ (which means that aggregation g' is more aggregate than g), then the statistic $\bar{v}_s^{(g',n)}$ is computed using observations \hat{v}_s^n that are also used to compute $\bar{v}_s^{(g,n)}$.

We begin by defining

$\mathcal{N}_s^{(g,n)}$ = set of iterations n where $G^g(\hat{s}^n) = G^g(s)$ (i.e., \hat{s}^n aggregates to the same state as s),

$N_s^{(g,n)} = |\mathcal{N}_s^{(g,n)}|$,

$\bar{\varepsilon}_s^{(g,n)}$ = estimate of the average error when observing state $s = G(\hat{s}^n)$

$$= \sum_{n \in \mathcal{N}_s^{(g,n)}} \hat{\varepsilon}_s^{(g,n)} / 1/N_s^{(g,n)}.$$

The average error $\bar{\varepsilon}_s^{(g,n)}$ can be written

$$\begin{aligned} \bar{\varepsilon}_s^{(g,n)} &= \frac{1}{N_s^{(g,n)}} \left(\sum_{n \in \mathcal{N}_s^{(0,n)}} \varepsilon^n + \sum_{n \in \mathcal{N}_s^{(g,n)} \setminus \mathcal{N}_s^{(0,n)}} \varepsilon^n \right) \\ &= \frac{N_s^{(0,n)}}{N_s^{(g,n)}} \bar{\varepsilon}_s^{(0)} + \frac{1}{N_s^{(g,n)}} \sum_{n \in \mathcal{N}_s^{(g,n)} \setminus \mathcal{N}_s^{(0,n)}} \varepsilon^n. \end{aligned} \quad (8.31)$$

This relationship shows us that we can write the error term at the higher level of aggregation g' as a sum of a term involving the errors at the lower level of aggregation g (for the same state s) and a term involving errors from other states s'' where $G^{g'}(s'') = G^{g'}(s)$, given by

$$\begin{aligned} \bar{\varepsilon}_s^{(g',n)} &= \frac{1}{N_s^{(g',n)}} \left(\sum_{n \in \mathcal{N}_s^{(g,n)}} \varepsilon^n + \sum_{n \in \mathcal{N}_s^{(g',n)} \setminus \mathcal{N}_s^{(g,n)}} \varepsilon^n \right) \\ &= \frac{1}{N_s^{(g',n)}} \left(N_s^{(g,n)} \frac{\sum_{n \in \mathcal{N}_s^{(g,n)}} \varepsilon^n}{N_s^{(g,n)}} + \sum_{n \in \mathcal{N}_s^{(g',n)} \setminus \mathcal{N}_s^{(g,n)}} \varepsilon^n \right) \\ &= \frac{N_s^{(g,n)}}{N_s^{(g',n)}} \bar{\varepsilon}_s^{(g,n)} + \frac{1}{N_s^{(g',n)}} \sum_{n \in \mathcal{N}_s^{(g',n)} \setminus \mathcal{N}_s^{(g,n)}} \varepsilon^n. \end{aligned} \quad (8.32)$$

We can overcome this problem by rederiving the expression for the optimal weights. For a given (disaggregate) state s , the problem of finding the optimal weights $(w_s^{(g,n)})_{g \in \mathcal{G}}$ is stated by

$$\min_{w_s^{(g,n)}, g \in \mathcal{G}} \mathbb{E} \left[\frac{1}{2} \left(\sum_{g \in \mathcal{G}} w_s^{(g,n)} \cdot \bar{v}_s^{(g,n)} - v_s^{(g,n)} \right)^2 \right] \quad (8.33)$$

subject to

$$\sum_{g \in \mathcal{G}} w_s^{(g,n)} = 1, \quad (8.34)$$

$$w_s^{(g,n)} \geq 0, \quad g \in \mathcal{G}. \quad (8.35)$$

Let

$$\begin{aligned} \bar{\delta}_s^{(g,n)} &= \text{error in the estimate } \bar{v}_s^{(g,n)} \text{ from the true value associated with attribute vector } s, \\ &= \bar{v}_s^{(g,n)} - v_s. \end{aligned}$$

The optimal weights are computed using the following theorem:

Theorem 8.6.1 *For a given state vector, s , the optimal weights, $w_s^{(g,n)}$, $g \in \mathcal{G}$, where the individual estimates are correlated by way of a tree structure, are given by solving the following system of linear equations in (w, λ) :*

$$\sum_{g \in \mathcal{G}} w_s^{(g,n)} \mathbb{E} \left[\bar{\delta}_s^{(g,n)} \bar{\delta}_s^{(g',n)} \right] - \lambda = 0 \quad \forall g' \in \mathcal{G}, \quad (8.36)$$

$$\sum_{g \in \mathcal{G}} w_s^{(g,n)} = 1, \quad (8.37)$$

$$w_s^{(g,n)} \geq 0 \quad \forall g \in \mathcal{G}. \quad (8.38)$$

Proof The proof is not too difficult, and it illustrates how we obtain the optimal weights. We start by formulating the Lagrangian for the problem formulated in (8.33)–(8.35), which gives us

$$\begin{aligned} L(w, \lambda) &= \mathbb{E} \left[\frac{1}{2} \left(\sum_{g \in \mathcal{G}} w_s^{(g,n)} \cdot \bar{v}_s^{(g,n)} - v_s^{(g,n)} \right)^2 \right] + \lambda \left(1 - \sum_{g \in \mathcal{G}} w_s^{(g,n)} \right) \\ &= \mathbb{E} \left[\frac{1}{2} \left(\sum_{g \in \mathcal{G}} w_s^{(g,n)} (\bar{v}_s^{(g,n)} - v_s^{(g,n)}) \right)^2 \right] + \lambda \left(1 - \sum_{g \in \mathcal{G}} w_s^{(g,n)} \right). \end{aligned}$$

The first order optimality conditions are

$$\mathbb{E} \left[\sum_{g \in \mathcal{G}} w_s^{(g,n)} (\bar{v}_s^{(g,n)} - v_s^{(g,n)}) (\bar{v}_s^{(g',n)} - v_s^{(g',n)}) \right] - \lambda = 0 \quad \forall g' \in \mathcal{G}, \quad (8.39)$$

$$\sum_{g \in \mathcal{G}} w_s^{(g,n)} - 1 = 0. \quad (8.40)$$

To simplify equation (8.39), we note that

$$\begin{aligned} \mathbb{E} \left[\sum_{g \in \mathcal{G}} w_s^{(g,n)} (\bar{v}_s^{(g,n)} - v_s^{(g,n)}) (\bar{v}_s^{(g',n)} - v_s^{(g',n)}) \right] &= \mathbb{E} \left[\sum_{g \in \mathcal{G}} w_s^{(g,n)} \bar{\delta}_s^{(g,n)} \bar{\delta}_s^{(g',n)} \right] \\ &= \sum_{g \in \mathcal{G}} w_s^{(g,n)} \mathbb{E} \left[\bar{\delta}_s^{(g,n)} \bar{\delta}_s^{(g',n)} \right]. \end{aligned} \quad (8.41)$$

Combining equations (8.39) and (8.41) gives us equation (8.36) which completes the proof. \square

Finding the optimal weights that handle the correlations among the statistics at different levels of aggregation requires finding $\mathbb{E} \left[\bar{\delta}_s^{(g,n)} \bar{\delta}_s^{(g',n)} \right]$. We are going to compute this expectation by conditioning on the set of attributes \hat{s}^n that are sampled. This means that our expectation is defined over the outcome space Ω^ε . Let $N_s^{(g,n)}$ be the number of observations of state s at aggregation level g . The expectation is computed using:

Proposition 8.6.1 *The coefficients of the weights in equation (8.37) can be expressed as follows:*

$$\mathbb{E} \left[\bar{\delta}_s^{(g,n)} \bar{\delta}_s^{(g',n)} \right] = \mathbb{E} \left[\bar{\beta}_s^{(g,n)} \bar{\beta}_s^{(g',n)} \right] + \frac{N_s^{(g,n)}}{N_s^{(g',n)}} \mathbb{E} \left[\bar{\varepsilon}_s^{(g,n)} \right]^2 \quad \forall g \leq g'; \quad g, g' \in \mathcal{G}. \quad (8.42)$$

The proof is given in Section 8.6.2.

Now consider what happens when we make the assumption that the measurement error ε^n is independent of the attribute being sampled, \hat{s}^n . We do this by assuming that the variance of the measurement error is a constant given by σ_ε^2 . This gives us the following result:

Corollary 8.6.1 *For the special case where the statistical noise in the measurement of the values is independent of the attribute vector sampled, equation (8.42)*

reduces to

$$\mathbb{E} \left[\bar{\delta}_s^{(g,n)} \bar{\delta}_s^{(g',n)} \right] = \mathbb{E} \left[\bar{\beta}_s^{(g,n)} \bar{\beta}_s^{(g',n)} \right] + \frac{\sigma_{\varepsilon}^2}{N_s^{(g',n)}}. \quad (8.43)$$

For the case where $g = 0$ (the most disaggregate level), we assume that $\beta_s^{(0)} = 0$, which gives us

$$\mathbb{E} \left[\bar{\beta}_s^{(0,n)} \bar{\beta}_s^{(g',n)} \right] = 0.$$

This allows us to further simplify (8.43) to obtain

$$\mathbb{E} \left[\bar{\delta}_s^{(0,n)} \bar{\delta}_s^{(g',n)} \right] = \frac{\sigma_{\varepsilon}^2}{N_s^{(g',n)}}. \quad (8.44)$$

8.6.2 Proof of Proposition 8.6.1

We start by defining

$$\bar{\delta}_s^{(g,n)} = \bar{\beta}_s^{(g,n)} + \bar{\varepsilon}_s^{(g,n)}. \quad (8.45)$$

Equation (8.45) gives us

$$\begin{aligned} \mathbb{E} \left[\bar{\delta}_s^{(g,n)} \bar{\delta}_s^{(g',n)} \right] &= \mathbb{E} \left[(\bar{\beta}_s^{(g,n)} + \bar{\varepsilon}_s^{(g,n)}) (\bar{\beta}_s^{(g',n)} + \bar{\varepsilon}_s^{(g',n)}) \right] \\ &= \mathbb{E} \left[\bar{\beta}_s^{(g,n)} \bar{\beta}_s^{(g',n)} + \bar{\beta}_s^{(g',n)} \bar{\varepsilon}_s^{(g,n)} + \bar{\beta}_s^{(g,n)} \bar{\varepsilon}_s^{(g',n)} + \bar{\varepsilon}_s^{(g,n)} \bar{\varepsilon}_s^{(g',n)} \right] \\ &= \mathbb{E} \left[\bar{\beta}_s^{(g,n)} \bar{\beta}_s^{(g',n)} \right] + \mathbb{E} \left[\bar{\beta}_s^{(g',n)} \bar{\varepsilon}_s^{(g,n)} \right] + \mathbb{E} \left[\bar{\beta}_s^{(g,n)} \bar{\varepsilon}_s^{(g',n)} \right] \\ &\quad + \mathbb{E} \left[\bar{\varepsilon}_s^{(g,n)} \bar{\varepsilon}_s^{(g',n)} \right]. \end{aligned} \quad (8.46)$$

We note that

$$\mathbb{E} \left[\bar{\beta}_s^{(g',n)} \bar{\varepsilon}_s^{(g,n)} \right] = \bar{\beta}_s^{(g',n)} \mathbb{E} \left[\bar{\varepsilon}_s^{(g,n)} \right] = 0.$$

Similarly

$$\mathbb{E} \left[\bar{\beta}_s^{(g,n)} \bar{\varepsilon}_s^{(g',n)} \right] = 0.$$

This allows us to write equation (8.46) as

$$\mathbb{E} \left[\bar{\delta}_s^{(g,n)} \bar{\delta}_s^{(g',n)} \right] = \mathbb{E} \left[\bar{\beta}_s^{(g,n)} \bar{\beta}_s^{(g',n)} \right] + \mathbb{E} \left[\bar{\varepsilon}_s^{(g,n)} \bar{\varepsilon}_s^{(g',n)} \right]. \quad (8.47)$$

We start with the second term on the right-hand side of equation (8.47). This term can be written as

$$\begin{aligned} \mathbb{E} \left[\bar{\varepsilon}_s^{(g,n)} \bar{\varepsilon}_s^{(g',n)} \right] &= \mathbb{E} \left[\bar{\varepsilon}_s^{(g,n)} \cdot \frac{N_s^{(g,n)}}{N_s^{(g')}} \bar{\varepsilon}_s^{(g,n)} \right] + \mathbb{E} \left[\bar{\varepsilon}_s^{(g,n)} \cdot \frac{1}{N_s^{(g')}} \sum_{n \in \mathcal{N}_s^{(g',n)} \setminus \mathcal{W}_s^{(g,n)}} \varepsilon^n \right] \\ &= \frac{N_s^{(g,n)}}{N_s^{(g')}} \mathbb{E} \left[\bar{\varepsilon}_s^{(g,n)} \bar{\varepsilon}_s^{(g,n)} \right] + \underbrace{\frac{1}{N_s^{(g')}} \mathbb{E} \left[\bar{\varepsilon}_s^{(g,n)} \cdot \sum_{n \in \mathcal{N}_s^{(g',n)} \setminus \mathcal{W}_s^{(g,n)}} \varepsilon^n \right]}_I \dots \end{aligned}$$

The term I can be rewritten using

$$\begin{aligned} \mathbb{E} \left[\bar{\varepsilon}_s^{(g,n)} \cdot \sum_{n \in \mathcal{N}_s^{(g',n)} \setminus \mathcal{W}_s^{(g,n)}} \varepsilon^n \right] &= \mathbb{E} \left[\bar{\varepsilon}_s^{(g,n)} \right] \mathbb{E} \left[\sum_{n \in \mathcal{N}_s^{(g',n)} \setminus \mathcal{W}_s^{(g,n)}} \varepsilon^n \right] \\ &= 0, \end{aligned}$$

which means that

$$\mathbb{E} \left[\bar{\varepsilon}_s^{(g,n)} \bar{\varepsilon}_s^{(g',n)} \right] = \frac{N_s^{(g,n)}}{N_s^{(g')}} \mathbb{E} \left[\bar{\varepsilon}_s^{(g)} \right]^2. \quad (8.48)$$

Combining (8.47) and (8.48) proves the proposition. \square

The second term on the right-hand side of equation (8.48) can be further simplified using

$$\begin{aligned} \mathbb{E} \left[\bar{\varepsilon}_s^{(g)} \right]^2 &= \mathbb{E} \left[\left(\frac{1}{N_s^{(g,n)}} \sum_{n \in \mathcal{N}_s^{(g,n)}} \varepsilon^n \right)^2 \right] \quad \forall g' \in \mathcal{G} \\ &= \frac{1}{\left(N_s^{(g,n)} \right)^2} \sum_{m \in \mathcal{N}_s^{(g,n)}} \sum_{n \in \mathcal{N}_s^{(g,n)}} \mathbb{E} [\varepsilon^m \varepsilon^n] \\ &= \frac{1}{\left(N_s^{(g,n)} \right)^2} \sum_{n \in \mathcal{N}_s^{(g,n)}} \mathbb{E} [(\varepsilon^n)^2] \\ &= \frac{1}{\left(N_s^{(g,n)} \right)^2} N_s^{(g,n)} \sigma_\varepsilon^2 \\ &= \frac{\sigma_\varepsilon^2}{N_s^{(g,n)}}. \end{aligned} \quad (8.49)$$

Combining equations (8.42), (8.48), and (8.49) gives us the result in equation (8.43). \square

8.7 BIBLIOGRAPHIC NOTES

This chapter is primarily a brief tutorial into statistical learning. Readers interested in pursuing approximate dynamic programming should obtain a good statistical reference such as Bishop (2006) and Hastie et al. (2009). The second reference can be downloaded from <http://www-stat.stanford.edu/~tibs/ElemStatLearn/>.

Sections 8.1 Aggregation has been a widely used technique in dynamic programming as a method to overcome the curse of dimensionality. Early work focused on picking a fixed level of aggregation (Whitt, 1978; Bean et al., 1987), or using adaptive techniques that change the level of aggregation as the sampling process progresses (Bertsekas and Castanon, 1989, Mendelsohn, 1982; Bertsekas and Tsitsiklis, 1996), but that still use a fixed level of aggregation at any given time. Much of the literature on aggregation has focused on deriving error bounds (Zipkin 1980a,b). A recent discussion of aggregation in dynamic programming can be found in Limbert et al. (2002). For a good discussion of aggregation as a general technique in modeling, see Rogers et al. (1991). The material in Section 8.1.4 is based on George et al. (2008) and George and Powell (2006). LeBlanc and Tibshirani (1996) and Yang (2001) provide excellent discussions of mixing estimates from different sources. For a discussion of soft state aggregation, see Singh et al. (1995). Section 8.1.2 on bias and variance is based on George and Powell (2006).

Section 8.2 Basis functions have their roots in the modeling of physical processes. A good introduction to the field from this setting is Heuberger et al. (2005). Schweitzer and Seidmann (1985) describe generalized polynomial approximations for Markov decision processes for use in value iteration, policy iteration, and the linear programming method. Menache et al. (2005) discuss basis function adaptations in the context of reinforcement learning. For a very nice discussion of the use of basis functions in approximate dynamic programming, see Tsitsiklis and Roy (1996b) and Van Roy (2001). Tsitsiklis and Van Roy (1997) prove convergence of iterative stochastic algorithms for fitting the parameters of a regression model when the policy is held fixed. For Section 8.2.2, the first use of approximate dynamic programming for evaluating an American call option is given in Longstaff and Schwartz (2001), but the topic has been studied for decades (see Taylor (1967)). Tsitsiklis and Van Roy (2001) also provide an alternative ADP algorithm for American call options. Clement et al. (2002) provide formal convergence results for regression models used to price American options. This presentation on the geometric view of basis functions is based on Tsitsiklis and Van Roy (1997).

Section 8.3 There is, of course, an extensive literature on different statistical methods. This section provides only a sampling. For a much more thorough treatment, see Bishop (2006) and Hastie et al. (2009).

Section 8.4 An excellent introduction to continuous approximation techniques is given in Judd (1998) in the context of economic systems and computational dynamic programming. Ormoneit and Sen (2002) and Ormoneit and Glynn (2002) discuss the use of kernel-based regression methods in an approximate dynamic programming setting, providing convergence proofs for specific algorithmic strategies. For a thorough introduction to locally polynomial regression methods, see Fan and Gijbels (1996). An excellent discussion of a broad range of statistical learning methods can be found in Hastie et al. (2009). Bertsekas and Tsitsiklis (1996) provides an excellent discussion of neural networks in the context of approximate dynamic programming. Haykin (1999) presents a much more in-depth presentation of neural networks, including a chapter on approximate dynamic programming using neural networks.

Section 8.5 See Hastie et al. (2009, section 2.5), for a very nice discussion of the challenges of approximating high-dimensional functions.

PROBLEMS

- 8.1** In a spreadsheet, create a 4×4 grid where the cells are numbered 1, 2, ..., 16 starting with the upper left-hand corner and moving left to right, as shown in the lists below. We are going to treat each number in the cell as the mean of the observations drawn from that cell. Now assume that if we observe a cell, we observe the mean plus a random variable that is uniformly distributed between -1 and $+1$. Next define a series of aggregations where aggregation 0 is the disaggregate level, aggregation 1 divides the grid into four 2×2 cells, and aggregation 2 aggregates everything into a single cell. After n iterations, let $\bar{v}_s^{(g,n)}$ be the estimate of cell “ s ” at the n th level of aggregation, and let

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

$$\bar{v}_s^n = \sum_{g \in \mathcal{G}} w_s^{(g)} \bar{v}_s^{(g,n)}$$

be your best estimate of cell s using a weighted aggregation scheme. Compute an overall error measure using

$$(\bar{\sigma}^2)^n = \sum_{s \in \mathcal{S}} (\bar{v}_s^n - v_s)^2,$$

where v_s is the true value (taken from your grid) of being in cell s . Also let $w^{(g,n)}$ be the average weight after n iterations given to the aggregation level g when averaged over all cells at that level of aggregation (e.g., there is only one cell for $w^{(2,n)}$). Perform 1000 iterations where at each iteration you randomly sample a cell and measure it with noise. Update your estimates at each level of aggregation, and compute the variance of your estimate with and without the bias correction.

- (a) Plot $w^{(g,n)}$ for each of the three levels of aggregation at each iteration. Do the weights behave as you would expect? Explain.
- (b) For each level of aggregation, set the weight given to that level equal to one (in other words, use a single level of aggregation), and plot the overall error as a function of the number of iterations.
- (c) Add to your plot the average error when you use a weighted average, where the weights are determined by equation (8.14) without the bias correction.
- (d) Finally add to your plot the average error when you used a weighted average, but now determine the weights by equation (8.15), which uses the bias correction.
- (e) Repeat the above assuming that the noise is uniformly distributed between -5 and $+5$.

8.2 Show that

$$\sigma_s^2 = (\sigma_s^2)^{(g)} + (\beta_s^{(g)})^2, \quad (8.50)$$

which breaks down the total variation in an estimate at a level of aggregation is the sum of the variation of the observation error plus the bias squared.

- 8.3 Show that $\mathbb{E} \left[(\bar{\theta}^{n-1} - \theta^n)^2 \right] = \lambda^{n-1} \sigma^2 + (\beta^n)^2$. [Hint: Add and subtract $\mathbb{E} \bar{\theta}^{n-1}$ inside the expectation and expand.]
- 8.4 Show that $\mathbb{E} \left[(\bar{\theta}^{n-1} - \hat{\theta}^n)^2 \right] = (1 + \lambda^{n-1}) \sigma^2 + (\beta^n)^2$ (which proves equation 8.7). [Hint: See previous exercise.]
- 8.5 Derive the small sample form of the recursive equation for the variance given in (8.8). Recall that if

$$\bar{\theta}^n = \frac{1}{n} \sum_{m=1}^n \hat{\theta}^m,$$

then an estimate of the variance of $\hat{\theta}$ is

$$\text{Var}[\hat{\theta}] = \frac{1}{n-1} \sum_{m=1}^n (\hat{\theta}^m - \bar{\theta}^n)^2.$$

- 8.6** Show that the matrix H^n in the recursive updating formula from equation (9.80),

$$\overline{\theta}^n = \overline{\theta}^{n-1} - H^n x^n \hat{\varepsilon}^n,$$

reduces to $H^n = 1/n$ for the case of a single parameter (which means we are using $Y = \text{constant}$, with no independent variables).

- 8.7** A general aging and replenishment problem arises as follows: Let s_t be the “age” of our process at time t . At time t , we may choose between a decision $d = C$ to continue the process, incurring a cost $g(s_t)$ or a decision $d = R$ to replenish the process, which incurs a cost $K + g(0)$. Assume that $g(s_t)$ is convex and increasing. The state of the system evolves according to

$$s_{t+1} = \begin{cases} s_t + D_t & \text{if } d = C, \\ 0 & \text{if } d = R, \end{cases}$$

where D_t is a nonnegative random variable giving the degree of deterioration from one epoch to another (also called the “drift”).

- (a) Prove that the structure of this policy is monotone. Clearly state the conditions necessary for your proof.
- (b) How does your answer to part (1) change if the random variable D_t is allowed to take on negative outcomes? Give the weakest possible conditions on the distribution of required to ensure the existence of a monotone policy.
- (c) Now assume that the action is to reduce the state variable by an amount $q \leq s_t$ at a cost of cq (instead of K). Further assume that $g(s) = as^2$. Show that this policy is also monotone. Say as much as you can about the specific structure of this policy.

Learning Value Function Approximations

In Chapter 6 we described a number of different ways of constructing a policy. One of the most important ways, and the way that is most widely associated with the term “approximate dynamic programming,” requires approximating the value of being in a state. Chapter 8 described a number of different approximation strategies, including lookup table, aggregated functions, parametric models and nonparametric models. All of these statistical models are created by generating a state S^n , next computing some observation of a value \hat{v}^n of being in state S^n , and finally using the pair (S^n, \hat{v}^n) to estimate (or update the estimate) of the value of being in a state.

In this chapter we focus primarily on the different ways of calculating \hat{v}^n and then use this information to estimate a value function approximation, *for a fixed policy*. To emphasize that we are computing values for a fixed policy, we index parameters such as the value function V^π by the policy π . After we establish the fundamentals for estimating the value of a policy, we address in Chapter 10 the last step of searching for good policies.

9.1 SAMPLING THE VALUE OF A POLICY

At first glance the problem of statistically estimating the value of a fixed policy should not be any different than estimating a function from noisy observations. We start by showing that from one perspective this is correct. However, the context of dynamic programming, even when we fix a policy, introduces opportunities and challenges as we realize that we can take advantage of the dynamics of information that may arise in both finite and infinite horizon settings.

9.1.1 Direct Policy Evaluation for Finite Horizon Problems

Imagine that we have a fixed policy $A^\pi(s)$ (or $X^\pi(s)$ if we are working with vector-valued decisions). The policy may take any of the forms described in Chapter 6. For iteration n , if we are in state S_t^n at time t , we then choose action $a_t^n = A^\pi(S_t^n)$, after which we sample the exogenous information W_{t+1}^n . We sometimes say that we are following sample path ω^n from which we observe $W_{t+1}^n = W_{t+1}(\omega^n)$. The exogenous information W_{t+1}^n may depend on both S_t^n and the action a_t^n . From this we may compute our contribution (cost if we are minimizing) from

$$\hat{C}_t^n = C(S_t^n, a_t^n, W_{t+1}^n).$$

Finally, we compute our next state from our transition function

$$S_{t+1}^n = S^M(S_t^n, a_t^n, W_{t+1}^n).$$

This process continues until we reach the end of our horizon T . The basic algorithm is described in Figure 9.1. In step 6 we use a batch routine to fit a statistical model. It is often more natural to use some sort of recursive procedure and embed the updating of the value function within the iterative loop. The type of recursive procedure depends on the nature of the value function approximation. Later in this chapter we describe several recursive procedures if we are using linear regression.

Finite horizon problems are sometimes referred to as *episodic*, where an episode refers to a simulation of a policy until the end of the horizon (also known as trials). However, the term “episodic” can also be interpreted more broadly. For example, an emergency vehicle may repeatedly return to base where the system then restarts.

Step 0. Initialization.

Step 0a. Initialize \bar{V}^0 .

Step 0b. Initialize S^1 .

Step 0c. Set $n = 1$.

Step 1. Choose a sample path ω^n .

Step 2. Choose a starting state S_0^n .

Step 3. Do for $t = 0, 1, \dots, T$:

Step 3a. $a_t^n = A^\pi(S_t^n)$.

Step 3b. $\hat{C}_t^n = C(S_t^n, a_t^n)$.

Step 3c. $W_{t+1}^n = W_{t+1}(\omega^n)$.

Step 3d. $S_{t+1}^n = S^M(S_t^n, a_t^n, W_{t+1}^n)$.

Step 4. Compute $\hat{v}_0^n = \sum_{t=0}^T \gamma^t \hat{C}_t^n$.

Step 5. Increment n . If $n \leq N$, go to step 1.

Step 6. Use the sequence of state-value pairs $(S^i, \hat{v}^i)_{i=1}^N$ to fit a value function approximation $\bar{V}^\pi(s)$.

Figure 9.1 Basic policy approximation method.

Each cycle of starting from a home base and then returning to the home base can be viewed as an episode. As a result, if we are working with a finite horizon problem, we prefer to refer to these specifically as such.

Evaluating a fixed policy is mathematically equivalent to making unbiased observations of a noisy function. Fitting a functional approximation is precisely what the entire field of statistical learning has been trying to do for decades. If we are fitting a linear model, then there are some powerful recursive procedures that can be used. These are discussed below.

9.1.2 Policy Evaluation for Infinite Horizon Problems

Not surprisingly, infinite horizon problems introduce a special complication, since we cannot obtain an unbiased observation in a finite number of measurements. Below we present some methods that have been used for infinite horizon applications.

Recurrent Visits

There are many infinite horizon problems where the system resets itself periodically. A simple example of this is a finite horizon problem where hitting the end of the horizon and starting over (as would occur in a game) can be viewed as an episode. A different example is a queueing system where the admission of patients to an emergency room must be managed. From time to time the queue may become empty, at which point the system resets and starts over. For such systems it makes sense to estimate the value of following a policy π when starting from the base state.

Even if we do not have such a renewal system, imagine that we find ourselves in a state s . Now follow a policy π until we re-enter state s again. Let $R^n(s)$ be the reward earned, and let $\tau^n(s)$ be the number of time periods required before re-entering state s . Here n is counting the number of times we visit state s . An observation of the average reward earned when in state s and following policy π would be given by

$$\hat{v}^n(s) = \frac{R^n(s)}{\tau^n(s)}.$$

$\hat{v}^n(s)$ would be computed when we return to state s . We might then update the *average* value of being in state s using

$$\bar{v}^n(s) = (1 - \alpha_{n-1})\bar{v}^{n-1}(s) + \alpha_{n-1}\hat{v}^n(s).$$

Note that as we make each transition from some state s' to some state s'' , we are accumulating rewards in $R(s)$ for every state s that we have visited prior to reaching state s' . Each time we arrive at some state s'' , we stop accumulating rewards for s'' and compute $\hat{v}^n(s'')$, and then we smooth this into the current estimate of $\bar{v}(s'')$. Note that we have presented this only for the case of computing the average reward per time period.

Partial Simulations

While we may not be able to simulate an infinite trajectory, we may simulate a long trajectory T , long enough to ensure that we are producing an estimate that is “good enough.” When we are using discounting, we realize that eventually γ^t becomes small enough that a longer simulation does not really matter. This idea can be implemented in a relatively simple way.

Consider the algorithm in Figure 9.1, and insert the calculation in step 3:

$$\bar{c}_t = \frac{t-1}{t} \bar{c}_{t-1} + \frac{1}{t} \hat{C}_t^n.$$

\bar{c}_t is an average over the time periods of the contribution per time period. As we follow our policy over progressively more time periods, \bar{c}_t approaches an average contribution per time period. Over an infinite horizon we would expect to find

$$\hat{v}_0^n = \lim_{t \rightarrow \infty} \sum_{t=0}^{\infty} \gamma^t \hat{C}_t^n = \frac{1}{1-\gamma} \bar{c}_{\infty}.$$

Now suppose that we only progress T time periods, and let \bar{c}_T be our estimate of \bar{c}_{∞} at this point. We would expect that

$$\begin{aligned} \hat{v}_0^n(T) &= \sum_{t=0}^T \gamma^t \hat{C}_t^n \\ &\approx \frac{1-\gamma^{T+1}}{1-\gamma} \bar{c}_T. \end{aligned} \tag{9.1}$$

The error between our T -period estimate $\hat{v}_0^n(T)$ and the infinite horizon estimate \hat{v}_0^n is given by

$$\begin{aligned} \delta_T^n &= \frac{1}{1-\gamma} \bar{c}_{\infty} - \frac{1-\gamma^{T+1}}{1-\gamma} \bar{c}_T \\ &\approx \frac{1}{1-\gamma} \bar{c}_T - \frac{1-\gamma^{T+1}}{1-\gamma} \bar{c}_T \\ &= \frac{\gamma^{T+1}}{1-\gamma} \bar{c}_T. \end{aligned}$$

Thus we just have to find T to make δ_T small enough. This strategy is embedded in some optimal algorithms that only require $\delta_T^n \rightarrow 0$ as $n \rightarrow \infty$ (meaning that we have to steadily allow T to grow).

Infinite Horizon Projection

The analysis above leads to another idea that has received considerable attention in the approximate dynamic programming community under the name *least squares temporal differencing* (LSTD), although we present it here with a somewhat different development. We can easily see from (9.1) that if we stop after

T time periods, we will underestimate the infinite horizon contribution by a factor $1 - \gamma^{T+1}$. Assuming that T is reasonably large (e.g., $\gamma^{T+1} < 0.1$), we might introduce the correction

$$\hat{v}_0^n = \frac{1}{1 - \gamma^{T+1}} \hat{v}_0^n(T).$$

In essence we are taking a sample estimate of a T -period path, and projecting it out over an infinite horizon.

9.1.3 Temporal Difference Updates

Suppose that we are in state S_t^n and we make decision a_t^n (using policy π), after which we observe the information W_{t+1} that puts us in state $S_{t+1}^n = S^M(S_t^n, a_t^n, W_{t+1}^n)$. The contribution from this transition is given by $C(S_t^n, a_t^n)$ (or $C(S_t^n, a_t^n, W_{t+1}^n)$ if the contribution depends on the outcome W_{t+1}). Imagine now that we continue this until the end of our horizon T . For simplicity, we are going to drop discounting. In this case the contribution along this path would be

$$\hat{v}_t^n = C(S_t^n, a_t^n) + C(S_{t+1}^n, a_{t+1}^n) + \cdots + C(S_T^n, a_T^n). \quad (9.2)$$

This is the contribution from following the path produced by a combination of the information from outcome ω^n (which determines $W_{t+1}^n, W_{t+2}^n, \dots, W_T^n$) and policy π . \hat{v}_t^n is an unbiased sample estimate of the value of being in state S_t and following policy π over sample path ω^n . We can use a stochastic gradient algorithm to estimate the value of being in state S_t :

$$\bar{V}_t^n(S_t^n) = \bar{V}_t^{n-1}(S_t^n) - \alpha_n \left(\bar{V}_t^{n-1}(S_t^n) - \hat{v}_t^n \right). \quad (9.3)$$

We can obtain a richer class of algorithms by breaking down our path cost in (9.2) by using

$$\begin{aligned} \hat{v}_t^n &= \sum_{\tau=t}^T C(S_\tau^n, a_\tau^n, W_{\tau+1}^n) \\ &\quad - \underbrace{\left\{ \sum_{\tau=t}^T \left(\bar{V}_\tau^{n-1}(S_\tau) - \bar{V}_{\tau+1}^{n-1}(S_{\tau+1}) \right) \right\}}_{=0} + \left(\bar{V}_t^{n-1}(S_t) - \bar{V}_{T+1}^{n-1}(S_{T+1}) \right). \end{aligned}$$

We now use the fact that $\bar{V}_{T+1}^{n-1}(S_{T+1}) = 0$ (this is where our finite horizon model is useful). Rearranging gives

$$\hat{v}_t^n = \bar{V}_t^{n-1}(S_t) + \sum_{\tau=t}^T \left(C(S_\tau^n, a_\tau^n, W_{\tau+1}^n) + \bar{V}_{\tau+1}^{n-1}(S_{\tau+1}) - \bar{V}_\tau^{n-1}(S_\tau) \right).$$

Let

$$\delta_t^\pi = C(S_t^n, a_t^n, W_{t+1}^n) + \bar{V}_{t+1}^{n-1}(S_{t+1}^n) - \bar{V}_t^{n-1}(S_t^n). \quad (9.4)$$

The terms δ_t^π are called *temporal differences*. They are indexed by π because they depend on the particular policy for choosing actions. Let $\hat{v}_t^n = C(S_t^n, a_t^n, W_{t+1}^n) + \bar{V}_{t+1}^{n-1}(S_{t+1}^n)$ be our sample observation of being in state S_t , while $\bar{V}_t^{n-1}(S_t)$ is our current estimate of the value of being in state S_t . This means that the temporal difference at time t , $\delta_t^\pi = \hat{v}_t^n - \bar{V}_t^{n-1}(S_t)$, is the difference in our estimate of the value of being in state S_t between our current estimate and the updated estimate. The temporal difference is a sample realization of what is also known as the *Bellman error*.

Using equation (9.4), we can write \hat{v}_t^n in the more compact form:

$$\hat{v}_t^n = \bar{V}_t^{n-1}(S_t) + \sum_{\tau=t}^T \delta_\tau^\pi. \quad (9.5)$$

Substituting (9.5) into (9.3) gives

$$\begin{aligned} \bar{V}_t^n(S_t) &= \bar{V}_t^{n-1}(S_t) - \alpha_{n-1} \left[\bar{V}_t^{n-1}(S_t) - \left(\bar{V}_t^{n-1}(S_t) + \sum_{\tau=t}^T \delta_\tau^\pi \right) \right] \\ &= \bar{V}_t^{n-1}(S_t) + \alpha_{n-1} \sum_{\tau=t}^{T-1} \delta_\tau^\pi. \end{aligned} \quad (9.6)$$

We next use this bit of algebra to build an important class of updating mechanisms for estimating value functions.

9.1.4 TD(λ)

The temporal differences δ_τ^π are the errors in our estimates of the value of being in state S_τ . We can think of each term in equation (9.6) as a correction to the estimate of the value function. It makes sense that updates farther along the path should not be given as much weight as those earlier in the path. As a result it is common to introduce an artificial discount factor λ , producing updates of the form

$$\bar{V}_t^n(S_t) = \bar{V}_t^{n-1}(S_t) + \alpha_{n-1} \sum_{\tau=t}^T \lambda^{\tau-t} \delta_\tau^\pi. \quad (9.7)$$

We derived this formula without a time discount factor. We leave as an exercise to the reader to show that if we have a time discount factor γ , then the temporal-difference update becomes

$$\bar{V}_t^n(S_t) = \bar{V}_t^{n-1}(S_t) + \alpha_{n-1} \sum_{\tau=t}^T (\gamma \lambda)^{\tau-t} \delta_\tau^\pi. \quad (9.8)$$

Equation (9.8) shows that the discount factor γ , which is typically viewed as capturing the time value of money, and the algorithmic discount λ , which is a purely algorithmic device, have exactly the same effect. Not surprisingly, modelers in operations research have often used a discount factor γ set to a much smaller number than would be required to capture the time value of money. Artificial discounting allows us to look into the future, but then we discount the results when we feel that the results are not perfectly accurate. Note that our use of λ in this setting is precisely equivalent to our discounting (also using λ) when we presented discounted rolling horizon policies in Section 6.2.4.

Updates of the form given in equation (9.7) produce an updating procedure that is known as TD(λ) (or, temporal-difference learning with discount λ). We have seen this form of discounting in Section 6.2.3, where we introduced λ as a form of algorithmic discounting.

The updating formula in equation (9.7) requires that we step all the way to the end of the horizon before updating our estimates of the value. There is, however, another way of implementing the updates. The temporal differences δ_t^π are computed as the algorithm steps forward in time. As a result our updating formula can be implemented recursively. Say we are at time t' in our simulation. We would simply execute

$$\bar{V}_t^n(S_t^n) := \bar{V}_t^{n-1}(S_t) + \alpha_{n-1} \lambda^{t'-t} \delta_{t'}^\pi \quad \text{for all } t \leq t'. \quad (9.9)$$

Here our notation “ $=$ ” means that we take the current value of $\bar{V}_t^{n-1}(S_t)$, add $\alpha_{n-1} \lambda^{t'-t} \delta_{t'}^\pi$ to it to obtain an updated value of $\bar{V}_t^n(S_t)$. When we reach time $t' = T$, our value functions would have undergone a complete update. We note that at time t' , we need to update the value function for every $t \leq t'$.

9.1.5 TD(0) and Approximate Value Iteration

An important special case of TD(λ) occurs when we use $\lambda = 0$. In this case

$$\begin{aligned} \bar{V}_t^n(S_t^n) &= \bar{V}_t^{n-1}(S_t^n) + \alpha_{n-1} (C(S_t^n, a_t^n) + \gamma \bar{V}^{n-1}(S_t^M, a_t^n, W_{t+1}^n)) \\ &\quad - \bar{V}_t^{n-1}(S_t^n). \end{aligned} \quad (9.10)$$

Now consider value iteration. In Chapter 3, where we did not have to deal with Monte Carlo samples and statistical noise, value iteration (for a fixed policy) looked like

$$V_t^n(s) = C(s, A^\pi(s)) + \gamma \sum_{s' \in \mathcal{S}} p^\pi(s'|s) V_{t+1}^n(s').$$

In steady state we would write it as

$$V^n(s) = C(s, A^\pi(s)) + \gamma \sum_{s' \in \mathcal{S}} p^\pi(s'|s) V^{n-1}(s').$$

When we use classical temporal difference learning (for a fixed policy), we are following a sample path that puts us in state S_t^n , where we observe a sample realization of a contribution \hat{C}_t^n , after which we observe a sample realization of the next downstream state S_{t+1}^n (the action is determined by our fixed policy). A sample observation of the value of being in state S_t^n would be computed using

$$\hat{v}_t^n = C(S_t^n, a_t^n) + \gamma \bar{V}_{t+1}^{n-1}(S_{t+1}^n),$$

where $a_t^n = A^\pi(S_t^n)$. We can then use this to update our estimate of the value of being in state S_t^n using

$$\begin{aligned}\bar{V}_t^n(S_t^n) &= (1 - \alpha_{n-1})\bar{V}_t^{n-1}(S_t^n) + \alpha_{n-1}\hat{v}_t^n \\ &= (1 - \alpha_{n-1})\bar{V}_t^{n-1}(S_t^n) \\ &\quad + \alpha_{n-1}(C(S_t^n, a_t^n) + \gamma \bar{V}^{n-1}(S^M(S_t^n, a_t^n, W_{t+1}))).\end{aligned}\quad (9.11)$$

It is not hard to see that equations (9.10) and (9.11) are the same. The idea is popular because it is particularly easy to implement. It is also well suited to high-dimensional decision vectors x , as we illustrate in Chapters 13 and 14.

Temporal difference learning derives its name because $\bar{V}^{n-1}(S)$ is viewed as the “current” value of being in state S , while $C(S, a) + \bar{V}^{n-1}(S^M(S, a, W))$ is viewed as the updated value of being in state S . The difference $\bar{V}^{n-1}(S) - (C(S, a) + \bar{V}^{n-1}(S^M(S, a, W)))$ is the difference in these estimates across iterations (or time), hence the name. TD(0) is a form of statistical bootstrapping because, rather than simulate the full trajectory, it depends on the current estimate of the value $\bar{V}^{n-1}(S^M(S, a, W))$ of being in the downstream state $S^M(S, a, W)$. TD learning, Q -learning, and approximate value iteration all use statistical bootstrapping.

While TD(0) can be very easy to implement, it can also produce very slow convergence. The effect is illustrated in Table 9.1, where there are five steps before earning a reward of 1 (which we always earn). In this illustration there are no decisions and the contribution is zero for every other time period. A stepsize of $1/n$ was used throughout.

Table 9.1 illustrates that the rate of convergence for \bar{V}_0 is dramatically slower than for \bar{V}_4 . The reason is that as we smooth \hat{v}_t into \bar{V}_{t-1} , the stepsize has a discounting effect. The problem is most pronounced when the value of being in a state at time t depends on contributions that are a number of steps into the future (imagine the challenge of training a value function to play the game of chess). For problems with long horizons, and in particular those where it takes many steps before receiving a reward, this bias can be so serious that it can appear that temporal differencing (and algorithms that use it) simply does not work. We can partially overcome the slow convergence by carefully choosing a stepsize rule. Stepsizes are discussed in depth in Chapter 11.

Table 9.1 Effect of stepsize on backward learning

Iteration	\bar{V}_0	\hat{v}_1	\bar{V}_1	\hat{v}_2	\bar{V}_2	\hat{v}_3	\bar{V}_3	\hat{v}_4	\bar{V}_4	\hat{v}_5
0	0.000		0.000		0.000		0.000		0.000	1
1	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	1
2	0.000	0.000	0.000	0.000	0.000	0.000	0.500	1.000	1.000	1
3	0.000	0.000	0.000	0.000	0.167	0.500	0.667	1.000	1.000	1
4	0.000	0.000	0.042	0.167	0.292	0.667	0.750	1.000	1.000	1
5	0.008	0.042	0.092	0.292	0.383	0.750	0.800	1.000	1.000	1
6	0.022	0.092	0.140	0.383	0.453	0.800	0.833	1.000	1.000	1
7	0.039	0.140	0.185	0.453	0.507	0.833	0.857	1.000	1.000	1
8	0.057	0.185	0.225	0.507	0.551	0.857	0.875	1.000	1.000	1
9	0.076	0.225	0.261	0.551	0.587	0.875	0.889	1.000	1.000	1
10	0.095	0.261	0.294	0.587	0.617	0.889	0.900	1.000	1.000	1

9.1.6 TD Learning for Infinite Horizon Problems

We can perform updates using a general $\text{TD}(\lambda)$ strategy as we did for finite horizon problems. However, there are some subtle differences. With finite horizon problems, it is common to assume that we are estimating a different function \bar{V}_t for each time period t . As we step through time, we obtain information that can be used for a value function at a *specific* point in time. With stationary problems, each transition produces information that can be used to update the value function, which is then used in all future updates. By contrast, if we update \bar{V}_t for a finite horizon problem, then this update is not used until the next forward pass through the states.

When we move to infinite horizon problems, we drop the indexing by t . Instead of stepping forward in time, we step through iterations, where at each iteration we generate a temporal difference

$$\delta^{\pi,n} = C(s^n, a^n) + \gamma \bar{V}^{n-1}(S^{M,a}(s^n, a^n)) - \bar{V}^{n-1}(s^n).$$

To do a proper update of the value function at each state, we would have to use an infinite series of the form

$$\bar{V}^n(s) = \bar{V}^{n-1}(s) + \alpha_n \sum_{m=0}^{\infty} (\gamma \lambda)^m \delta^{\pi,n+m}, \quad (9.12)$$

where we can use any initial starting state $s^0 = s$. Of course, we would use the same update for each state s^m that we visit, so we might write

$$\bar{V}^n(s^m) = \bar{V}^{n-1}(s^m) + \alpha_n \sum_{n=m}^{\infty} (\gamma \lambda)^{(n-m)} \delta^{\pi,n}. \quad (9.13)$$

Equations (9.12) and (9.13) both imply stepping forward in time (presumably a “large” number of iterations) and computing temporal differences before performing an update. A more natural way to run the algorithm is to do the updates

incrementally. After we compute $\delta^{\pi,n}$, we can update the value function at each of the previous states we visited. So, at iteration n , we would execute

$$\bar{V}^n(s^m) := \bar{V}^n(s^m) + \alpha_n(\gamma\lambda)^{n-m}\delta^{\pi,n}, \quad m = n, n-1, \dots, 1. \quad (9.14)$$

We can now use the temporal difference $\delta^{\pi,n}$ to update the estimate of the value function for every state we have visited up to iteration n .

Step 0. Initialization.

Step 0a. Initialize $\bar{V}^0(S)$ for all S .

Step 0b. Initialize the state S^0 .

Step 0c. Set $n = 1$.

Step 1. Choose ω^n .

Step 2. Solve

$$a^n = \arg \max_{a \in \mathcal{A}^n} (C(S^n, a) + \gamma \bar{V}^{n-1}(S^{M,a}(S^n, a))). \quad (9.15)$$

Step 3. Compute the temporal difference for this step:

$$\delta^{\pi,n} = C(S^n, a^n) + \gamma (\bar{V}^{n-1}(S^{M,a}(S^n, a^n)) - \bar{V}^{n-1}(S^n)).$$

Step 4. Update \bar{V} for $m = n, n-1, \dots, 1$:

$$\bar{V}^n(S^m) = \bar{V}^{n-1}(S^m) + (\gamma\lambda)^{n-m}\delta^{\pi,n}. \quad (9.16)$$

Step 5. Compute $S^{n+1} = S^M(S^n, a^n, W(\omega^n))$.

Step 6. Let $n = n + 1$. If $n < N$, go to step 1.

Figure 9.2 TD(λ) algorithm for infinite horizon problems.

Figure 9.2 outlines the basic structure of a TD(λ) algorithm for an infinite horizon problem. Step 1 begins by computing the first post-decision state, after which step 2 makes a single step forward. After computing the temporal difference in step 3, we traverse previous states we have visited in step 4 to update their value functions. The sequence of states that we traverse and the associated discount factor $(\gamma\lambda)^{n-m}$ is known in the reinforcement learning community as an *eligibility trace*.

In step 4 we update all the states $(S^m)_{m=1}^n$ that we have visited up to then. Thus, at iteration n , we would have simulated the partial update

$$\bar{V}^n(S^0) = \bar{V}^{n-1}(S^0) + \alpha_{n-1} \sum_{m=0}^n (\gamma\lambda)^m \delta^{\pi,m}. \quad (9.17)$$

This means that at any iteration n , we have updated our values using biased sample observations (as is generally the case in value iteration). We avoided this problem for finite horizon problems by extending out to the end of the horizon. We can

obtain unbiased updates for infinite horizon problems by assuming that all policies eventually put the system into an “absorbing state.” For example, if we are modeling the process of holding or selling an asset, we might be able to guarantee that we eventually sell the asset.

One subtle difference between temporal difference learning for finite horizon and infinite horizon problems is that in the infinite horizon case, we may be visiting the same state two or more times on the same sample path. For the finite horizon case the states and value functions are all indexed by the time that we visit them. Since we step forward through time, we can never visit the same state at the same point in time twice in the same sample path. By contrast, it is quite easy in a steady-state problem to revisit the same state over and over again. For example, we could trace the path of our nomadic trucker, who might go back and forth between the same pair of locations in the same sample path. As a result we are using the value function to determine what state to visit, but at the same time we are updating the value of being in these states.

9.2 STOCHASTIC APPROXIMATION METHODS

A central idea in recursive estimation is the use of stochastic approximation methods and stochastic gradients. We have already seen this in one setting in Section 7.2.1. We review the idea again here, but in a different context. We begin with the same stochastic optimization problem, which we originally introduced as the problem

$$\min_x \mathbb{E} F(x, W).$$

Now assume that we are choosing a scalar value v to solve the problem

$$\min_v \mathbb{E} F(v, \hat{V}), \quad (9.18)$$

where

$$F(v, \hat{V}) = \frac{1}{2}(v - \hat{V})^2,$$

and where \hat{V} is a random variable with unknown mean. We would like to use a series of sample realizations \hat{v}^n to guide an algorithm that generates a sequence v^n that converges to the optimal solution v^* that solves (9.18). We use the same basic strategy as we introduced in Section 7.2.1 where we update v^n using

$$\begin{aligned} v^n &= v^{n-1} - \alpha_{n-1} \nabla F(v^{n-1}, \hat{v}^n) \\ &= v^{n-1} - \alpha_{n-1} (v^{n-1} - \hat{v}^n). \end{aligned} \quad (9.19)$$

Now, if we make the transition that instead of updating a scalar v^n , we are updating $\bar{V}_t^n(S_t^n)$, we obtain the updating equation

$$\bar{V}_t^n(S_t^n) = \bar{V}_t^{n-1}(S_t^n) - \alpha_{n-1} (\bar{V}_t^{n-1}(S_t^n) - \hat{v}^n). \quad (9.20)$$

If we use $\hat{v}^n = C(S_t^n, a_t^n) + \gamma \bar{V}^{n-1}(S_{t+1}^n)$, we quickly see that the updating equation produced using our stochastic gradient algorithm (9.20) gives us the same update that we obtained using temporal difference learning (equation (9.10)) and approximate value iteration (equation (9.11)). In equation (9.19), α_n is called a stepsize because it controls how far we go in the direction of $\nabla F(v^{n-1}, \hat{v}^n)$, and for this reason this is the term that we adopt for α_n throughout this book. In contrast to our first use of this idea in Section 7.2, where the stepsize had to serve a scaling function, in this setting the units of the variable being optimized, v^n , and the units of the gradient are the same. Indeed we can expect that $0 < \alpha_n \leq 1$, which is a major simplification.

Remark Stochastic gradient methods pervade approximate dynamic programming and reinforcement learning, but it is important to recognize an odd problem with signs. Recall that we have defined the temporal difference (also known as the Bellman error) using $\delta_t^n = \hat{v}_t^n - \bar{V}_t(S_t^n)$. If we substitute the temporal difference into (9.20), we would obtain

$$\begin{aligned}\bar{V}_t^n(S_t^n) &= \bar{V}_t^{n-1}(S_t^n) - \alpha_{n-1}(-\delta_t^n) \\ &= \bar{V}_t^{n-1}(S_t^n) + \alpha_{n-1}\delta_t^n.\end{aligned}$$

This type of equation can be found throughout the ADP/RL literature. It looks like a stochastic gradient updating equation where we are trying to minimize the error. But instead of seeing a minus sign before the stepsize, you will see a plus sign. This arises purely because of the disconnect between the traditional definition of the temporal difference and the standard strategy in statistics to define errors as the “actual” (\hat{v}^n) minus the “predicted” ($\bar{V}_t(S_t^{n-1})$).

Let us consider what happens when we replace the lookup table representation $\bar{V}(s)$ that we used above with a linear regression $\bar{V}(s|\theta) = \theta^T \phi$. Now we want to find the best value of θ , which we can do by solving

$$\min_{\theta} \mathbb{E} \frac{1}{2} (\bar{V}(s|\theta) - \hat{v})^2.$$

Applying a stochastic gradient algorithm, we obtain the updating step

$$\theta^n = \theta^{n-1} - \alpha_{n-1}(\bar{V}(s|\theta^{n-1}) - \hat{v}^n) \nabla_{\theta} \bar{V}(s|\theta^n). \quad (9.21)$$

Since $\bar{V}(s|\theta^n) = \sum_{f \in \mathcal{F}} \theta_f^n \phi_f(s) = (\theta^n)^T \phi(s)$, the gradient with respect to θ is given by

$$\nabla_{\theta} \bar{V}(s|\theta^n) = \left(\begin{array}{c} \frac{\partial \bar{V}(s|\theta^n)}{\partial \theta_1} \\ \frac{\partial \bar{V}(s|\theta^n)}{\partial \theta_2} \\ \vdots \\ \frac{\partial \bar{V}(s|\theta^n)}{\partial \theta_F} \end{array} \right) = \left(\begin{array}{c} \phi_1(s^n) \\ \phi_2(s^n) \\ \vdots \\ \phi_F(s^n) \end{array} \right) = \phi(s^n).$$

Thus the updating equation (9.21) is given by

$$\begin{aligned}\theta^n &= \theta^{n-1} - \alpha_{n-1}(\bar{V}(s|\theta^{n-1}) - \hat{v}^n)\phi(s^n) \\ &= \theta^{n-1} - \alpha_{n-1}(\bar{V}(s|\theta^{n-1}) - \hat{v}^n) \begin{pmatrix} \phi_1(s^n) \\ \phi_2(s^n) \\ \vdots \\ \phi_F(s^n) \end{pmatrix}. \quad (9.22)\end{aligned}$$

Using a stochastic gradient algorithm requires that we have some starting estimate θ^0 for the parameter vector, although $\theta^0 = 0$ is a common choice.

While this is a simple and elegant algorithm, we have reintroduced the problem of scaling. Just as we encountered in Section 7.2, the units of θ^{n-1} and the units of $(\bar{V}(s|\theta^{n-1}) - \hat{v}^n)\phi(s^n)$ may be completely different. What we have learned about stepsizes still applies, except that we may need an initial stepsize that is quite different than 1.0 (our common starting point). Our experimental work has suggested that the following policy works well: when you choose a stepsize formula, scale the first value of the stepsize so that the change in θ^n in the early iterations of the algorithm is approximately 20 to 50 percent (you will typically need to observe several iterations). You want to see individual elements of θ^n moving consistently in the same direction during the early iterations. If the stepsize is too large, the values can swing wildly, and the algorithm may not converge at all. If the changes are too small, the algorithm may simply stall out. It is very tempting to run the algorithm for a period of time and then conclude that it appears to have converged (presumably to a good solution). While it is important to see the individual elements moving in the same direction (consistently increasing or decreasing) in the early iterations, it is also important to see oscillatory behavior toward the end.

9.3 RECURSIVE LEAST SQUARES FOR LINEAR MODELS

Linear regression plays a major role in approximate dynamic programming, since it is a powerful and effective way for approximating value functions for certain problem classes. Perhaps one of the most appealing features of linear regression is the ease with which models can be updated recursively. Recursive methods are well known in the statistics and machine learning communities, but these communities often focus on batch methods. Recursive statistics is especially valuable in approximate dynamic programming because we are often updating functional approximations (value functions or policy functions) iteratively as the algorithm progresses.

In Chapter 8 we reviewed linear statistical models using the classical notation of statistics,

$$y = \theta^T x + \varepsilon,$$

where $\theta = (\theta_1, \dots, \theta_I)^T$ was the vector of regression coefficients. We let X^n be the $n \times I$ matrix of observations (where n is the number of observations). Using

batch statistics, we can estimate θ from the normal equation

$$\theta = [(X^n)^T X^n]^{-1} (X^n)^T Y^n. \quad (9.23)$$

We now make the conversion to the notation of approximate dynamic programming and reinforcement learning. Instead of x_i we use a basis function $\phi_f(S)$, where $f \in \mathcal{F}$ is a feature. We let $\phi(s)$ be a column vector of the features, and so $\phi^n = \phi(S^n)$ replaces x^n . We also write our value function approximation using

$$\bar{V}(S) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(S) = \phi(S)^T \theta.$$

Throughout our presentation we assume that we have access to an observation \hat{v}_t^n of the value of being in state S_t^n . As we have shown, there are different ways of computing \hat{v} in dynamic programming applications, but at this point we are going to simply take these estimates as data. The estimate \hat{v}^n may be the simulation of a policy as in

$$\hat{v}^n = \sum_{t=0}^T \gamma^t C(S_t, A^\pi(S_t)),$$

or a simple bootstrapped estimate

$$\hat{v}_t^n = C(S_t, A^\pi(S_t)) + \gamma \mathbb{E} \bar{V}_{t+1}^{n-1}(S_{t+1}).$$

Ultimately the choice of how we compute \hat{v} will have other algorithmic implications in terms of convergence proofs and updating strategies. But for now, we are simply trying to use the estimates \hat{v}^n to estimate a value function approximation.

9.3.1 Recursive Least Squares for Stationary Data

In the setting of approximate dynamic programming, estimating the coefficient vector θ using batch methods such as equation (9.23) would be very expensive. Fortunately, it is possible to compute these formulas recursively. The updating equation for θ is

$$\theta^n = \theta^{n-1} - H^n \phi^n \hat{\varepsilon}^n, \quad (9.24)$$

where H^n is a matrix computed using

$$H^n = \frac{1}{\gamma^n} B^{n-1}. \quad (9.25)$$

The error $\hat{\varepsilon}^n$ is computed using

$$\hat{\varepsilon}^n = \bar{V}_s(\theta^{n-1}) - \hat{v}^n. \quad (9.26)$$

Note that it is common in statistics to compute the error in a regression using “actual minus predicted” while we are using “predicted minus actual” (see also

equation (9.21) above). Our sign convention is motivated by the derivation from first principles of optimization. B^{n-1} is an $|\mathcal{F}|$ by $|\mathcal{F}|$ matrix, which is updated recursively using

$$B^n = B^{n-1} - \frac{1}{\gamma^n} (B^{n-1} \phi^n (\phi^n)^T B^{n-1}). \quad (9.27)$$

γ^n is a scalar computed using

$$\gamma^n = 1 + (\phi^n)^T B^{n-1} \phi^n. \quad (9.28)$$

The derivation of equations (9.24) through (9.28) is given in Section 9.10.1. Equation (9.24) has the feel of a stochastic gradient algorithm, but it has one significant difference. Instead of using a typical stepsize, we have the matrix H^n to serve as a scaling matrix.

It is possible in any regression problem that the matrix $(X^n)^T X^n$ (in equation (9.23)) is noninvertible. If this is the case, then our recursive formulas are not going to overcome this problem. When this happens, we will observe $\gamma^n = 0$. Alternatively, the matrix may be invertible, but unstable, which occurs when γ^n is very small (e.g., $\gamma^n < \epsilon$ for some small ϵ). When this occurs, the problem can be circumvented by using

$$\bar{\gamma}^n = \gamma^n + \delta,$$

where δ is a suitably chosen small perturbation that is large enough to avoid instabilities. Some experimentation is likely to be necessary, since the right value depends on the scale of the parameters being estimated.

The only missing step in our algorithm is initializing B^n . One strategy is to collect a sample of m observations where m is large enough to compute B^m using full inversion. Once we have B^m , we can then proceed to update it using the formula above. A second strategy is to use $B^0 = \epsilon I$, where I is the identity matrix and ϵ is a “small constant.” This strategy is not guaranteed to give the exact values, but it should work well if the number of observations is relatively large.

In our dynamic programming applications the observations \hat{v}^n will represent estimates of the value of being in a state, and our independent variables will be either the states of our system (if we are estimating the value of being in each state) or the basis functions. So in this case we are estimating the coefficients of the basis functions. The equations assume implicitly that the estimates come from a stationary series.

There are many problems where the number of basis functions can be extremely large. In these cases even the efficient recursive expressions in this section cannot avoid the fact that we are still updating a matrix where the number of rows and columns is the number of states (or basis functions). If we are only estimating a few dozen or a few hundred parameters, this can be fine. If the number of parameters extends into the thousands, even this strategy would probably bog down. It is important to work out the approximate dimensionality of the matrices before using these methods.

9.3.2 Recursive Least Squares for Nonstationary Data

It is generally the case in approximate dynamic programming that our observations \hat{v}^n (typically updates to an estimate of a value function) come from a nonstationary process. This is true even when we are estimating the value of a fixed policy if we use TD learning, but it is always true when we introduce the dimension of optimizing over policies. Recursive least squares puts equal weight on all prior observations, whereas we would prefer to put more weight on more recent observations.

Instead of minimizing total errors (as we do in equation (8.18)), it makes sense to minimize a geometrically weighted sum of errors

$$\min_{\theta} \sum_{m=1}^n \lambda^{n-m} \left(\hat{v}^m - (\theta_0 + \sum_{i=1}^I \theta_i \phi_i^m) \right)^2, \quad (9.29)$$

where λ is a discount factor that we use to discount older observations. If we repeat the derivation in Section 9.3.1, the only changes we have to make are in the updating formula for B^n , which is now given by

$$B^n = \frac{1}{\lambda} \left(B^{n-1} - \frac{1}{\gamma^n} (B^{n-1} \phi^n (\phi^n)^T B^{n-1}) \right), \quad (9.30)$$

and the expression for γ^n , which is now given by

$$\gamma^n = \lambda + (\phi^n)^T B^{n-1} \phi^n. \quad (9.31)$$

λ works in a way similar to a stepsize, although in the opposite direction. Setting $\lambda = 1$ means that we are putting an equal weight on all observations, whereas smaller values of λ puts more weight on more recent observations. This way λ plays a role similar to our use of λ in $\text{TD}(\lambda)$.

We could use this logic and view λ as a tunable parameter. Of course, a constant goal in the design of approximate dynamic programming algorithms is to avoid the need to tune yet another parameter. Ideally we would like to take advantage of the theory we developed for automatically adapting stepsizes to automatically adjust λ . For the special case where our regression model is just a constant (in which case $\phi^n = 1$), we can develop a simple relationship between α_n and the discount factor (which we now compute at each iteration, so we write it as λ_n). Let $G^n = (B^n)^{-1}$, which means that our updating equation is now given by

$$\theta^n = \theta^{n-1} - \frac{1}{\gamma^n} (G^n)^{-1} \phi^n \hat{\epsilon}^n.$$

Recall that we compute the error $\hat{\epsilon}^n$ as predicted minus actual as given in equation (9.26). This is required if we are going to derive our optimization algorithm based on first principles, which means that we are minimizing a stochastic function. The matrix G^n is updated recursively using

$$G^n = \lambda_n G^{n-1} + \phi^n (\phi^n)^T, \quad (9.32)$$

with $G^0 = 0$. For the case where $\phi^n = 1$ (in which case G^n is also a scalar), $(G^n)^{-1}\phi^n = (G^n)^{-1}$ plays the role of our stepsize, so we would like to write $\alpha_n = (G^n)^{-1}$. Assume that $\alpha_{n-1} = (G^{n-1})^{-1}$. Equation (9.32) implies that

$$\begin{aligned}\alpha_n &= (\lambda_n G^{n-1} + 1)^{-1} \\ &= \left(\frac{\lambda_n}{\alpha_{n-1}} + 1 \right)^{-1}.\end{aligned}$$

Solving for λ_n gives

$$\lambda_n = \alpha_{n-1} \left(\frac{1 - \alpha_n}{\alpha_n} \right). \quad (9.33)$$

Note that if $\lambda_n = 1$, then we want to put equal weight on all the observations (which would be optimal if we have stationary data). We know that in this setting, the best stepsize is $\alpha_n = 1/n$. Substituting this stepsize into equation (9.33) verifies this identity.

The value of equation (9.33) is that it allows us to relate the discounting produced by λ_n to the choice of stepsize rule, which has to be chosen to reflect the nonstationarity of the observations. In Chapter 11 we introduce a much broader range of stepsize rules, some of which have tunable parameters. Using equation (9.33) allows us to avoid introducing yet another tunable parameter.

9.3.3 Recursive Estimation Using Multiple Observations

The previous methods assume that we get one observation and use it to update the parameters. Another strategy is to sample several paths and solve a classical least squares problem for estimating the parameters. In the simplest implementation we would choose a set of realizations $\hat{\Omega}^n$ (rather than a single sample ω^n) and follow all of them, producing a set of estimates $(\hat{v}^n(\omega))_{\omega \in \hat{\Omega}^n}$ that we can use to update the value function.

If we have a set of observations, we then face the classical problem of finding a vector of parameters $\hat{\theta}^n$ that best match all of these value function estimates. Thus we want to solve

$$\hat{\theta}^n = \arg \min_{\hat{\theta}} \frac{1}{|\hat{\Omega}^n|} \sum_{\omega \in \hat{\Omega}^n} (\bar{V}(\hat{\theta}) - \hat{v}(\omega))^2.$$

This is the standard parameter estimation problem faced in the statistical estimation community. If $\bar{V}(\theta)$ is linear in θ , then we can use the usual formulas for linear regression. If the function is more general, then we would typically resort to nonlinear programming algorithms to solve the problem. In either case, $\hat{\theta}^n$ is still an update that needs to be smoothed in with the previous estimate θ^{n-1} , which we would do using

$$\theta^n = (1 - \alpha_{n-1})\theta^{n-1} + \alpha_{n-1}\hat{\theta}^n. \quad (9.34)$$

One advantage of this strategy is that in contrast with the updates that depend on the gradient of the value function, updates of the form given in equation (9.34) do not encounter a scaling problem, and therefore we return to our more familiar territory where $0 < \alpha_n \leq 1$. Of course, as the sample size $\hat{\Omega}$ increases, the stepsize should also be increased because there is more information in $\hat{\theta}^n$. Stepsizes based on the Kalman filter (Sections 11.4.2 and 11.4.3) will automatically adjust to the amount of noise in the estimate.

The usefulness of this particular strategy will very much depend on the problem at hand. In many applications the computational burden of producing multiple estimates $\hat{v}^n(\omega)$, $\omega \in \hat{\Omega}^n$ before producing a parameter update will simply be too costly.

9.3.4 Recursive Time-Series Estimation*

Up to now we have used regression models that depend on explanatory variables (which we sometimes refer to as basis functions) that depend on the state of the system. The goal has been to represent the value of being in a state S_t with a small number of parameters by using the structure of the state variable.

Now consider a somewhat different use of linear regression. Let \hat{v}^n be an estimate of the value of being in some state $S_t = s$ (which we suppress to keep the notation simple). By now we are well aware that the random observations \hat{v}^n tend to grow (or shrink) over time. If they are growing, then the standard updating equation

$$\bar{v}^n = (1 - \alpha_{n-1})\bar{v}^{n-1} + \alpha_{n-1}\hat{v}^n \quad (9.35)$$

will generally produce an estimate \bar{v}^n that is less than \hat{v}^{n+1} . We can minimize this by estimating a regression model of the form

$$\hat{v}^{n+1} = \theta_0 \hat{v}^n + \theta_1 \bar{v}^{n-1} + \cdots + \theta_F \bar{v}^{n-F} + \hat{\epsilon}^{n+1}. \quad (9.36)$$

What we are doing is estimating \hat{v}^{n+1} as a function of the recent history of observations, which offers the potential that we will be predicting trends in the estimates that might arise when using algorithms such as value iteration or Q -learning.

After iteration n , our estimate of the value of being in state s is

$$\bar{v}^n = \theta_0^n \hat{v}^n + \theta_1^n \bar{v}^{n-1} + \cdots + \theta_F^n \bar{v}^{n-F}. \quad (9.37)$$

Contrast this update with our simple exponential smoothing (e.g., equation (9.35)). We see that exponential smoothing is simply a form of linear regression where the parameter vector θ is determined by the stepsize.

We can update our parameter vector θ^n using the techniques described earlier in this section. We need to keep in mind that we are estimating the value of being in a state s using a $F+1$ -dimensional parameter vector. There are applications where we are trying to estimate the value of being in hundreds of thousands of states. For such large applications we have to decide if we are going to have a different parameter vector θ for each state s or one for the entire system.

There are many variations on this basic idea that draw on the entire field of time-series analysis. For example, we can use our recursive formulas to estimate a more general time-series model. At iteration n let the elements of our basis function be given by

$$\phi^n = (\phi_f(S^n))_{f \in \mathcal{F}},$$

which is the observed value of each function given the state vector S^n . If we wished to include a constant term, we would define a basis function $\phi_0 = 1$. Let $F = |\mathcal{F}|$ be the number of basis functions used to explain the value function, so ϕ^n is an F -dimensional column vector.

We can combine the information in our basis functions (which depend on the state at time t) with the history of observations of the updated values. This we represent using the column vector

$$\tilde{v}^n = (\hat{v}^n, \bar{v}^{n-1}, \dots, \bar{v}^{n-F})^T$$

just as we did in Sections 9.2 and 9.3. Taken together, ϕ^n and \tilde{v}^n is our population of potential explanatory variables that we can use to help us predict \hat{v}^{n+1} . ϕ^n is an F -dimensional column vector, while \tilde{v}^n is a $F+1$ -dimensional column vector. We can formulate a model using

$$\hat{v}^{n+1} = (\theta^\phi)^T \phi^n + (\theta^v)^T \tilde{v}^n + \hat{\epsilon}^{n+1},$$

where θ^ϕ and θ^v are, respectively, F and $F+1$ -dimensional parameter vectors. This gives us a prediction of the value of being in state S^n using

$$\bar{v}^n = (\theta^{\phi,n})^T \phi^n + (\theta^{v,n})^T \tilde{v}^n.$$

Again, we update our estimate of the parameter vector $\theta^n = (\theta^{\phi,n}, \theta^{v,n})$ using the same techniques we presented above.

It is important to be creative. For example, we could use a time series model based on differences, as in

$$\begin{aligned} \hat{v}^{n+1} = & \theta_0 \hat{v}^n + \theta_1 (\bar{v}^{n-1} - \hat{v}^n) + \theta_2 (\bar{v}^{n-2} - \bar{v}^{n-1}) + \dots \\ & + \theta_F (\bar{v}^{n-F} - \bar{v}^{n-F+1}) + \hat{\epsilon}^{n+1}. \end{aligned}$$

Alternatively, we can use an extensive library of techniques developed under the umbrella of time-series modeling.

The signal processing community has developed a broad range of techniques for estimating models in a dynamic environment. This community refers to models such as (9.36) and (9.38) as *linear filters*. Using these techniques to improve the process of estimating value functions remains an active area of research. Our sense is that if the variance of \hat{v}^n is large relative to the rate at which the mean is changing, then we are unlikely to derive much value from a more sophisticated model which considers the history of prior estimates of the value function. However, there are

problems where the mean of \bar{v}^n is either changing quickly or changes for many iterations (due to a slow learning process). For these problems a time-series model that captures the behavior of the estimates of the value function over time may improve the convergence of the algorithm.

9.4 TEMPORAL DIFFERENCE LEARNING WITH A LINEAR MODEL

We are now ready to introduce one of the most powerful and attractive algorithmic procedures in approximate dynamic programming, which is temporal difference learning using a value function based on a linear model. Section 9.3 has introduced a family of recursive procedures for linear models. It is convenient to represent these using a generic updating function that we can depict using

$$\bar{V}^n(s) \leftarrow U^V(\bar{V}^{n-1}(s), S^n, \hat{v}^n).$$

Here we are saying that we are visiting state S^n where we observe an estimate \hat{v}^n of the value of being in state S^n , which is then used to update our approximate value function $\bar{V}^{n-1}(s)$ to obtain an updated function $\bar{V}^n(s)$. It might be the case that $\bar{V}^{n-1}(s) = \sum_f \theta_f^{n-1} \phi_f(s)$. The new information would allow us to use recursive least squares to obtain θ^n , giving us $\bar{V}^n(s) = \sum_f \theta_f^n \phi_f(s)$. To avoid the usual complications of taking expectations, we are going to assume that we are estimating the value function approximation around the post-decision state S^a .

Figure 9.3 is an illustration of a fairly typical TD(0) learning algorithm using linear models to approximate the value function. Note that the algorithm easily handles complex state variables (which may be multidimensional and continuous). We might also be able to handle multidimensional and continuous actions (which we normally write as x) if we can solve the maximization problem in step 1.

The algorithm seeks to find the best fit of the linear model when we fix the policy using

$$A^\pi(s) = \arg \max_{a \in \mathcal{A}^n} (C(S^n, a) + \gamma \sum_f \theta_f^\pi \phi_f(S^{M,a}(S^n, a))).$$

Note that a fixed parameter vector θ^π defines how actions are chosen, and this is not updated as the algorithm progresses. Also note that we choose the next state to visit using $S^{n+1} = S^M(S^n, a^n, W^{n+1})$, which means that we use the action determined by policy π to choose the next state that we visit. This is known as on-policy learning, a term we first saw in Section 4.3.3.

Define the true value of our policy as

$$V^\pi(s) = \mathbb{E} \sum_{t=0}^{\infty} \gamma^t C(s, A^\pi(s)).$$

Step 0. Initialization.

Step 0a. Initialize $\bar{V}_t^0, t \in \mathcal{T}$.

Step 0b. Set $n = 1$.

Step 0c. Initialize the previous post-decision state $S^{a,0}$.

Step 1. Solve

$$\hat{v}^n = \max_{a \in \mathcal{A}^n} \left(C(S^n, a) + \gamma \sum_f \theta_f^\pi \phi_f(S^{M,a}(S^n, a)) \right), \quad (9.38)$$

and let a^n be the value of a that solves (9.38).

Step 2. Update the value function around the previous post-decision state variable:

$$\bar{V}^n(s) \leftarrow U^V(\bar{V}^{n-1}(s), S^{a,n-1}, \hat{v}^n).$$

Step 3. Sample W^{n+1} and update the state variable:

$$S^{n+1} = S^M(S^n, a^n, W^{n+1}).$$

Step 4. Increment n . If $n \leq N$, go to step 1.

Step 5. Return the value functions $\bar{V}^\pi \approx \bar{V}^N$.

Figure 9.3 Temporal difference learning for a fixed policy.

It turns out that if we use on-policy learning, this algorithm has been proved to produce a value function approximation that minimizes the function

$$\min_\theta \sum_s d_s^\pi \left(\sum_f \theta_f \phi_f(s) - V^\pi(s) \right)^2, \quad (9.39)$$

where d_s^π is the steady-state probability of being in state s while following policy π . The weighting by d_s^π is an important dimension of our objective function. We note that the result still applies even if the state variable is multidimensional and continuous, something that the algorithm handles without modification. The convergence proof requires an approximation that is linear in the parameters; TD-learning using approximations that are nonlinear in the parameters may not converge, and may even diverge.

Using our policy $A^\pi(s)$ to determine the next state to visit seems like a perfectly reasonable assumption, but it can cause problems when we embed this logic in an algorithm that is searching for the best policy. As we discussed in Section 4.3, we might be following a policy that avoids certain states and actions that might lead to even better policies. Sometimes we have to force our algorithms to explore states and actions that may not appear to be the best but that might be very attractive. This is not an issue when we are evaluating a fixed policy, but it becomes critical if we are doing this to find better policies.

For the purpose of determining the next state to visit, we might introduce an exploration step where we choose an action a at random with probability ϵ , or we might use the action determined by our policy $A^\pi(s)$ with probability $1 - \epsilon$. This logic was essential in our Q -learning algorithm, but when we make the transition to using linear models, not only do we lose our guarantee of convergence, the algorithm may actually diverge! Furthermore, divergence is not restricted to pathological cases.

The interaction of exploration policies and linear models is a subtle but critical dimension of approximate dynamic programming, especially when we introduce the dimension of optimizing over policies (which we address in Chapter 10). The reinforcement learning community struggled for years with the issue of whether TD learning converged while using linear models, with a mixture of results that both supported and contradicted convergence. The issue was resolved in the seminal paper Tsitsiklis and Van Roy (1997) that established that online learning was a critical component in a convergent algorithm, along with the presence of the weighting term d_s^π in the objective function (9.39).

The resolution that on-policy learning was critical to the convergence of TD learning left unanswered the challenge of how to combine the need for convergence with the need to explore states. One strategy is to use an ϵ -greedy exploration with ϵ declining with the iterations so that the algorithm is actually a sequence of policies converging on a single greedy policy. Precup et al. (2001) offered the first version of an off-policy TD algorithm using linear models that converged to the optimal solution (i.e., the approximation that solves (9.39)), but the algorithm is restricted to policies such as Boltzmann exploration so that the probability of choosing any action is strictly positive.

Often overlooked in this debate is that if we have a good architecture (i.e., a good set of basis functions), we should not need to explore. That is, the issue is not so much *exploration* as it is *identification*, which addresses the question of whether we are collecting observations from a sufficiently diverse set of states that allows us to produce statistically reliable estimates of the function.

Consider the illustration in Figure 9.4, where we are only sampling a subset of states determined by the density d_s^π (assume that the states are continuous). The true value function $V^\pi(s)$ is depicted by the solid line. If we have a good set of basis functions, all we need is enough variation in the states that we do visit to find an approximation $\bar{V}(s)$ that works well for *all* states. For example, if we could only sample a narrow region of the state space, we would have a difficult time estimating our function. Of course, the assumption that we have a good set of basis functions is critical, and hard to verify. In practice, we generally can only hope to do well in a certain region, which is why the exploration issue is important.

9.5 BELLMAN'S EQUATION USING A LINEAR MODEL

It is possible to solve Bellman's equation for infinite horizon problems by starting with the assumption that the value function is given by a linear model $V(s) =$

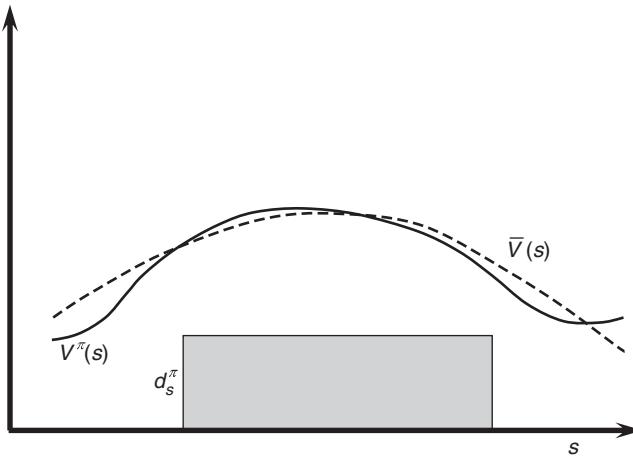


Figure 9.4 True value function $V^\pi(s)$ (solid line), a fitted approximation $\bar{V}(s)$ (dashed line), with observations drawn from a subset of states determined by the density d_s^π .

$\theta^T \phi(s)$ where $\Phi(s)$ is a column vector of basis functions for a particular state s . Of course, we are still working with a single policy, so we are using Bellman's equation only as a method for finding the best linear approximation for the infinite horizon value of a fixed policy π .

We begin with a derivation based on matrix linear algebra, which is more advanced and which does not produce expressions that can be implemented in practice. We follow this discussion with a simulation-based algorithm that can be implemented fairly easily.

9.5.1 A Matrix-Based Derivation*

In Section 8.2.3 we provided a geometric view of basis functions, drawing on the elegance and obscurity of matrix linear algebra. We are going to continue this presentation and present a version of Bellman's equation assuming linear models. However, we are not yet ready to introduce the dimension of optimizing over policies, so we are still simply trying to approximate the value of being in a state. Also we are only considering infinite horizon models, since we have already handled the finite horizon case. This presentation can be viewed as another method for handling infinite horizon models, while using a linear architecture to approximate the value function.

First recall that Bellman's equation (for a fixed policy) is written

$$V^\pi(s) = C(s, A^\pi(s)) + \gamma \sum_{s' \in S} p(s'|s, A^\pi(s)) V^\pi(s').$$

In vector-matrix form, we let V^π be a vector with element $V^\pi(s)$, we let c^π be a vector with element $C(s, A^\pi(s))$, and we let P^π be the one-step transition

matrix with element $p(s'|s, A^\pi(s))$ at row s , column s' . By this notation, Bellman's equation becomes

$$V^\pi = c^\pi + \gamma P^\pi V^\pi,$$

allowing us to solve for V^π using

$$V^\pi = (I - \gamma P^\pi)^{-1} c^\pi.$$

This works with a lookup table representation (a value for each state). Now assume that we replace V^π with an approximation $\bar{V}^\pi = \Phi\theta$, where Φ is a $|\mathcal{S}| \times |\mathcal{F}|$ matrix with element $\Phi_{s,f} = \phi_f(s)$. Also let d_s^π be the steady-state probability of being in state s while following policy π , and let D^π be a $|\mathcal{S}| \times |\mathcal{S}|$ diagonal matrix where the state probabilities $(d_1^\pi, \dots, d_{|\mathcal{S}|}^\pi)$ make up the diagonal. We would like to choose θ to minimize the weighted sum of errors squared, where the error for state s is given by

$$\epsilon^n(s) = \sum_f \theta_f \phi_f(s) - \left(c^\pi(s) + \gamma \sum_{s' \in \mathcal{S}} p^\pi(s'|s, A^\pi) \sum_f \theta_f^n \phi_f(s') \right). \quad (9.40)$$

The first term on the right-hand side of (9.40) is the predicted value of being in each state given θ , while the second term on the right-hand side is the “predicted” value computed using the one-period contribution plus the expected value of the future, which is computed using θ^n . The expected sum of errors squared is then given by

$$\min_{\theta} \sum_{s \in \mathcal{S}} d_s^\pi \left(\sum_f \theta_f \phi_f(s) - \left(c^\pi(s) + \gamma \sum_{s' \in \mathcal{S}} p^\pi(s'|s, A^\pi) \sum_f \theta_f^n \phi_f(s') \right) \right)^2.$$

In matrix form this can be written

$$\min_{\theta} (\Phi\theta - (c^\pi + \gamma P^\pi \Phi\theta^n))^T D^\pi (\Phi\theta - (c^\pi + \gamma P^\pi \Phi\theta^n)), \quad (9.41)$$

where D^π is a $|\mathcal{S}| \times |\mathcal{S}|$ diagonal matrix with elements d_s^π that serves a scaling role (we want to focus our attention on states we visit the most). We can find the optimal value of θ (given θ^n) by taking the derivative of the function being optimized in (9.41) with respect to θ and setting it equal to zero. Let θ^{n+1} be the optimal solution, which means we can write

$$\Phi^T D^\pi (\Phi\theta^{n+1} - (c^\pi + \gamma P^\pi \Phi\theta^n)) = 0. \quad (9.42)$$

We can find a fixed point $\lim_{n \rightarrow \infty} \theta^n = \lim_{n \rightarrow \infty} \theta^{n+1} = \theta^*$, that allows us to write equation (9.42) in the form

$$A\theta^* = b, \quad (9.43)$$

where $A = \Phi^T D^\pi (I - \gamma P^\pi) \Phi$ and $b = \Phi^T D^\pi c^\pi$. This allows us, in theory at least, to solve for θ^* using

$$\theta^* = A^{-1} b, \quad (9.44)$$

which can be viewed as a scaled version of the normal equations (equation (9.23)). Equation (9.44) is very similar to our calculation of the steady-state value of being in each state introduced in Chapter 3, given by

$$V^\pi = (I - \gamma P^\pi)^{-1} c^\pi.$$

Equation (9.44) differs only in the scaling by the probability of being in each state (D^π) and then the transformation to the feature space by Φ .

We note that equation (9.42) can also be written in the form

$$A\theta - b = \Phi^T D^\pi (\Phi\theta - (c^\pi + \gamma P^\pi \Phi\theta)). \quad (9.45)$$

The term $\Phi\theta$ can be viewed as the approximate value of each state. The term $(c^\pi + \gamma P^\pi \Phi\theta)$ can be viewed as the one-period contribution plus the expected value of the state that we would transition to under policy π , again computed for each state. Let δ^π be a column vector containing the temporal difference for each state when we choose an action according to policy π . By tradition, the temporal difference has always been written in the form $C(S_t, a) + \bar{V}(S_{t+1}) - \bar{V}(S_t)$, which can be thought of as “estimated minus predicted.” If we continue to let δ^π be the traditional definition of the temporal difference, it would be written

$$\delta^\pi = -(\Phi\theta - (c^\pi + \gamma P^\pi \Phi\theta)). \quad (9.46)$$

The pre-multiplication of δ^π by D^π in (9.45) has the effect of factoring each temporal difference by the probability that we are in each state. Then pre-multiplying $D^\pi \delta^\pi$ by Φ^T has the effect of transforming this scaled temporal difference for each state into the feature space.

The goal is to find the value θ that produces $A\theta - b = 0$, which means we are trying to find the value θ that produces a scaled version of $\Phi\theta - (c^\pi + \gamma P^\pi \Phi\theta) = 0$ but transformed to the feature space.

Linear algebra offers a compact elegance, but at the same time can be hard to parse, and for this reason we encourage the reader to stop and think about the relationships. One useful exercise is to think of a set of basis functions where we have a “feature” for each state, with $\phi_f(s) = 1$ if feature f corresponds to state s . In this case, Φ is the identity matrix. D^π , the diagonal matrix with diagonal elements d_s^π giving the probability of being in state s , can be viewed as scaling quantities for each state by the probability of being in a state. If Φ is the identity matrix, then $A = D^\pi - \gamma D^\pi P^\pi$ where $D^\pi P^\pi$ is the matrix of *joint* probabilities of being in state s and then transitioning to state s' . The vector b becomes the vector of the cost of being in each state (and then taking the action corresponding to policy π) times the probability of being in the state.

When we have a smaller set of basis functions, then multiplying c^π or $D^\pi(I - \gamma P^\pi)$ times Φ has the effect of scaling quantities that are indexed by the state into the feature space, which also transforms an $|S|$ -dimensional space into an $|\mathcal{F}|$ -dimensional space.

9.5.2 A Simulation-Based Implementation

We start by simulating a trajectory of states, actions and information,

$$(S^0, a^0, W^1, S^1, a^1, W^2, \dots, S^n, a^n, W^{n+1}).$$

Recall that $\phi(s)$ is a column vector with an element $\phi_f(s)$ for each feature $f \in \mathcal{F}$. Using our simulation above, we also obtain a sequence of column vectors $\phi(s^i)$ and contributions $C(S^i, a^i, W^{i+1})$. We can create a sample estimate of the $|\mathcal{F}|$ by $|\mathcal{F}|$ matrix A in the section above using

$$A^n = \frac{1}{n} \sum_{i=0}^{n-1} \phi(S^i)(\phi(S^i) - \gamma \phi(S^{i+1}))^T. \quad (9.47)$$

We can also create a sample estimate of the vector b using

$$b^n = \frac{1}{n} \sum_{i=0}^{n-1} \phi(S^i) C(S^i, a^i, W^{i+1}). \quad (9.48)$$

To gain some intuition, again stop and assume that there is a feature for every state, which means that $\phi(S^i)$ is a vector of 0's with a 1 corresponding to the element for state i , making it an indicator variable telling us what state we are in. The term $(\phi(S^i) - \gamma \phi(S^{i+1}))$ is then a simulated version of $D^\pi(I - \gamma P^\pi)$, weighted by the probability that we are in a particular state, where we replace the probability of being in a state with a sampled realization of actually being in a particular state.

We are going to use this foundation to introduce two important algorithms for infinite horizon problems when using linear models to approximate value function approximations. These are known as *least squares temporal differences* (LSTD) and *least squares policy evaluation* (LSPE).

9.5.3 Least Squares Temporal Differences (LSTD)

As long as A^n is invertible (which is not guaranteed), we can compute a sample estimate of θ using

$$\theta^n = (A^n)^{-1} b^n. \quad (9.49)$$

This algorithm is known in the literature as *least squares temporal differences*. As long as the number of features is not too large (as is typically the case), the inverse is not too hard to compute. LSTD can be viewed as a batch algorithm

that operates by collecting a sample of temporal differences, and then using least squares regression to find the best linear fit.

We can see the role of temporal differences more clearly by doing a little algebra. We use equations (9.47) and (9.48) to write

$$\begin{aligned} A^n \theta^n - b^n &= \frac{1}{n} \sum_{i=0}^{n-1} (\phi(S^i)(\phi(S^i) - \gamma \phi(S^{i+1}))^T \theta^n - \phi(S^i) C(S^i, a^i, W^{i+1})) \\ &= \frac{1}{n} \sum_{i=0}^{n-1} \phi(S^i) (\phi(S^i)^T \theta^n - (C(S^i, a^i, W^{i+1}) + \gamma \phi(S^{i+1})^T \theta^n)) \\ &= \frac{-1}{n} \sum_{i=0}^{n-1} \phi(S^i) \delta^i(\theta^n), \end{aligned}$$

where $\delta^i(\theta^n) = (C(S^i, a^i, W^{i+1}) + \gamma \phi(S^{i+1})^T \theta^n) - \phi(S^i)^T \theta^n$ is the i th temporal difference given the parameter vector θ^n , where we are using the standard definition of temporal difference. Thus we are doing a least squares regression so that the sum of the temporal differences over the simulation (which approximations the expectation) is equal to zero. We would, of course, like it if θ could be chosen so that $\delta^i(\theta) = 0$ for all i . However, when working with sample realizations, the best we can expect is that the average across the observations of $\delta^i(\theta)$ tends to zero.

9.5.4 Least Squares Policy Evaluation (LSPE)

LSTD is basically a batch algorithm, which requires collecting a sample of n observations and then using regression to fit a model. An alternative strategy uses a stochastic gradient algorithm that successively updates estimates of θ . The basic updating equation is

$$\theta^n = \theta^{n-1} - \frac{\alpha}{n} G^n \sum_{i=0}^{n-1} \phi(S^i) \delta^i(n), \quad (9.50)$$

where G^n is a scaling matrix. Although there are different strategies for computing G^n , the most natural is a simulation-based estimate of $(\Phi^T D^\pi \Phi)^{-1}$ that can be computed using

$$G^n = \left(\frac{1}{n+1} \sum_{i=0}^n \phi(S^i) \phi(S^i)^T \right)^{-1}.$$

To visualize G^n , return again to the assumption that there is a feature for every state. In this case $\phi(S^i) \phi(S^i)^T$ is an $|\mathcal{S}|$ by $|\mathcal{S}|$ matrix with a 1 on the diagonal for row S^i and column S^i . As n approaches infinity, the matrix

$$\left(\frac{1}{n+1} \sum_{i=0}^n \phi(S^i) \phi(S^i)^T \right)$$

approaches the matrix D^π of the probability of visiting each state, stored in elements along the diagonal.

9.6 ANALYSIS OF TD(0), LSTD, AND LSPE USING A SINGLE STATE

A useful exercise to understand the behavior of recursive least squares, LSTD, and LSPE is to consider what happens when they are applied to a trivial dynamic program with a single state and a single action. Obviously we are interested in the policy that chooses the single action. This dynamic program is equivalent to computing the sum

$$F = \mathbb{E} \sum_{i=0}^{\infty} \gamma^i \hat{C}^i, \quad (9.51)$$

where \hat{C}^i is a random variable giving the i th contribution. If we let $\bar{c} = \mathbb{E} \hat{C}^i$, then clearly $F = \bar{c}/(1 - \gamma)$. But let us pretend that we do not know this, and we are using these various algorithms to compute the expectation.

9.6.1 Recursive Least Squares and TD(0)

Let \hat{v}^n be an estimate of the value of being in state S^n . We continue to assume that the value function is approximated using

$$\bar{V}(s) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(s).$$

We wish to choose θ by solving

$$\min_{\theta} \sum_{i=1}^n \left(\hat{v}^i - \left(\sum_{f \in \mathcal{F}} \theta_f \phi_f(S^i) \right) \right)^2.$$

Let θ^n be the optimal solution. We can determine this recursively using the techniques presented earlier in this chapter which gives us the updating equation

$$\theta^n = \theta^{n-1} - \frac{1}{1 + (x^n)^T B^{n-1} x^n} B^{n-1} x^n (\bar{V}^{n-1}(S^n) - \hat{v}^n) \quad (9.52)$$

where $x^n = (\phi_1(S^n), \dots, \phi_f(S^n), \dots, \phi_F(S^n))$, and the matrix B^n is computed using

$$B^n = B^{n-1} - \frac{1}{1 + (x^n)^T B^{n-1} x^n} (B^{n-1} x^n (x^n)^T B^{n-1}).$$

If we have only one state and one action, we only have one basis function $\phi(s) = 1$ and one parameter $\theta^n = \bar{V}^n(s)$. Now the matrix B^n is a scalar and equation (9.53) reduces to

$$\begin{aligned} v^n &= v^{n-1} - \frac{B^{n-1}}{1 + B^{n-1}}(v^{n-1} - \hat{v}^n) \\ &= \left(1 - \frac{B^{n-1}}{1 + B^{n-1}}\right)v^{n-1} + \frac{B^{n-1}}{1 + B^{n-1}}\hat{v}^n. \end{aligned}$$

If $B^0 = 1$, $B^{n-1} = 1/n$, giving us

$$v^n = \frac{n-1}{n}v^{n-1} + \frac{1}{n}\hat{v}^n. \quad (9.53)$$

Imagine now that we are using TD(0) where $\hat{v}^n = \hat{C}^n + \gamma v^{n-1}$. In this case we obtain

$$v^n = \left(1 - (1-\gamma)\frac{1}{n}\right)v^{n-1} + \frac{1}{n}\hat{C}^n. \quad (9.54)$$

Equation (9.54) can be viewed as an algorithm for finding

$$v = \sum_{n=0}^{\infty} \gamma^n \hat{C}^n,$$

where the solution is $v^* = \mathbb{E}\hat{C}/(1-\gamma)$.

Equation (9.54) shows us that recursive least squares, when \hat{v}^n is computed using temporal difference learning, has the effect of successively adding sample realizations of costs, with a “discount factor” of $1/n$. The factor $1/n$ arises directly as a result of the need to smooth out the noise in \hat{C}^n . For example, if $\hat{C} = c$ is a known constant, we could use standard value iteration, which would give us

$$v^n = c + \gamma v^{n-1}. \quad (9.55)$$

It is easy to see that v^n in (9.55) will rise much more quickly toward v^* than the algorithm in equation (9.54). We return to this topic in some depth in Chapter 11.

9.6.2 LSPE

LSPE requires that we first generate a sequence of states S^i and contributions \hat{C}^i for $i = 1, \dots, n$. We then compute θ by solving the regression problem

$$\theta^n = \arg \min_{\theta} \sum_{i=1}^n \left(\sum_f \theta_f \phi_f(S^i) - (\hat{C}^i + \gamma \bar{V}^{n-1}(S^{i+1})) \right)^2.$$

For a problem with one state where $\theta^n = v^n$, this reduces to

$$v^n = \arg \min_{\theta} \sum_{i=1}^n \left(\theta - (\hat{C}^i + \gamma v^{n-1}) \right)^2.$$

This problem can be solved in closed form, giving us

$$v^n = \left(\frac{1}{n} \sum_{i=1}^n \hat{C}^i \right) + \gamma v^{n-1}.$$

9.6.3 LSTD

Finally, we showed above that the LSTD procedure finds θ by solving the system of equations

$$\sum_{i=1}^n \phi_f(S^i) (\phi_f(S^i) - \gamma \phi_f(S^{i+1}))^T \theta^n = \sum_{i=1}^n \phi_f(S^i) \hat{C}^i,$$

for each $f \in \mathcal{F}$. Again, since we have only one basis function $\phi(s) = 1$ for our single state problem, this reduces to finding the scalar $\theta^n = v^n$ using

$$v^n = \frac{1}{1 - \gamma} \left(\frac{1}{n} \sum_{i=1}^n \hat{C}^i \right).$$

9.6.4 Discussion

This presentation illustrates three different styles for estimating an infinite horizon sum. In recursive least squares, equation (9.53) demonstrates the successive smoothing of the previous estimate v^n and the latest estimate \hat{v}^n . We are, at the same time, adding contributions over time while trying to smooth out the noise.

LSPE, by contrast, separates the estimation of the mean of the single-period contribution, and the process of summing contributions over time. At each iteration we improve our estimate of $\mathbb{E}\hat{C}$, and then accumulate our latest estimate in a telescoping sum.

LSTD, finally, updates its estimate of $\mathbb{E}\hat{C}$, and then projects this over the infinite horizon by factoring the result by $1/(1 - \gamma)$.

9.7 GRADIENT-BASED METHODS FOR APPROXIMATE VALUE ITERATION*

There has been a strong desire for approximation algorithms with the following features: (1) off-policy learning, (2) temporal-difference learning, (3) linear models for value function approximation, and (4) complexity (in memory and computation) that is linear in the number of features. The last requirement is primarily of interest

in specialized applications that require thousands or even millions of features. Off-policy learning is desirable because it provides an important degree of control over exploration. Temporal-difference learning is useful because it is so simple, as is the use of linear models, that it is possible to provide an estimate of the entire value function with a small number of measurements.

Off-policy, temporal-difference learning was first introduced in the form of Q -learning using a lookup table representation, where it is known to converge. But we lose this property if we introduce value function approximations that are linear in the parameters. In fact Q -learning can be shown to diverge for any positive stepsize. The reason is that there is no guarantee that our linear model is accurate, which can introduce significant instabilities in the learning process.

Q -learning and temporal difference learning can be viewed as forms of stochastic gradient algorithms, but the problem with earlier algorithms when we use linear value function approximations can be traced to the choice of objective function. For example, if we wish to find the best linear approximation $\bar{V}(s|\theta)$, a hypothetical objective function would be to minimize the expected mean squared difference between $\bar{V}(s|\theta)$ and the true value function $V(s)$. If d_s^π is the probability of being in state s , this objective would be written

$$\text{MSE}(\theta) = \frac{1}{2} \sum_s d_s^\pi (\bar{V}(s|\theta) - V(s))^2.$$

If we are using approximate value iteration, a more natural objective function is to minimize the mean squared Bellman error. We use the Bellman operator \mathcal{M}^π (as we did in Chapter 3) for policy π to represent

$$\mathcal{M}^\pi v = c^\pi + \gamma P^\pi v,$$

where v is a column vector giving the value of being in state s , and c^π is the column vector of contributions $C(s, A^\pi(s))$ if we are in state s and choose an action a according to policy π . This allows us to define

$$\begin{aligned} \text{MSBE}(\theta) &= \frac{1}{2} \sum_s d_s^\pi \left(\bar{V}(s|\theta) - (c^\pi(s) + \gamma \sum_{s'} p^\pi(s'|s) \bar{V}(s'|\theta)) \right)^2 \\ &= \frac{1}{2} \|\bar{V}(\theta) - \mathcal{M}\bar{V}(\theta)\|_D^2. \end{aligned}$$

We can minimize $\text{MSBE}(\theta)$ by generating a sequence of states $(S^1, \dots, S^i, S^{i+1}, \dots)$ and then computing a stochastic gradient

$$\nabla_\theta \text{MSBE}(\theta) = -\delta^{\pi,i} (\phi(S^i) - \gamma \phi(S^{i+1})),$$

where $\phi(S^i)$ is a column vector of basis functions evaluated at state S^i . The scalar $\delta^{\pi,i}$ is the temporal difference given by

$$-\delta^{\pi,i} = \bar{V}(S^i|\theta) - (c^\pi(S^i) + \gamma \bar{V}(S^{i+1}|\theta)).$$

We note that $\delta^{\pi,i}$ depends on the policy π that affects both the single-period contribution and the likelihood of transitioning to state S^{i+1} . To emphasize that we are working with a fixed policy, we carry the superscript π throughout.

A stochastic gradient algorithm, then, would seek to optimize θ using

$$\theta^{n+1} = \theta^n - \alpha_n \nabla_{\theta} \text{MSBE}(\theta) \quad (9.56)$$

$$\begin{aligned} &= \theta^n - \alpha_n (-\delta^{\pi,n})(\phi(S^n) - \gamma \phi(S^{n+1})) \\ &= \theta^n + \alpha_n \delta^{\pi,n} (\phi(S^n) - \gamma \phi(S^{n+1})). \end{aligned} \quad (9.57)$$

We note that the sign convention for temporal differences gives us a stochastic updating formula that seems to have the sign in front of the stepsize reversed. This derivation tracks the reason for this sign change, which can be traced to the sign convention for temporal differences.

A variant of this basic algorithm, called the generalized TD(0) (or, GTD(0)) algorithm, is given by

$$\theta^{n+1} = \theta^n + \alpha_n (\phi(S^n) - \gamma \phi(S^{n+1})) \phi(S^n)^T u^n, \quad (9.58)$$

where

$$u^{n+1} = u^n + \beta_n (u^n - \delta^{\pi,n} \phi(S^n)). \quad (9.59)$$

α_n and β_n are both stepsizes. u^n is a smoothed estimate of the product $\delta^{\pi,n} \phi(S^n)$.

Gradient descent methods based on temporal differences will not minimize $\text{MSBE}(\theta)$ because there does not exist a value of θ that would allow $\hat{v}(s) = c^\pi(s) + \gamma \bar{V}(s|\theta)$ to be represented as $\bar{V}(s|\theta)$. We can fix this using the mean-squared projected Bellman error (MSPBE(θ)) which we compute as follows. It is more compact to do this development using matrix-vector notation. We first recall the projection operator Π given by

$$\Pi = \Phi(\Phi^T D^\pi \Phi)^{-1} \Phi^T D^\pi.$$

(See Section 8.2.3 for a derivation of this operator.) If V is a vector giving the value of being in each state, ΠV is the nearest projection of V on the space generated by $\theta \phi(s)$. We are trying to find $\bar{V}(\theta)$ that will match the one-step lookahead given by $\mathcal{M}^\pi \bar{V}(\theta)$, but this produces a column vector that cannot be represented directly as $\Phi \theta$, where Φ is the $|\mathcal{S}| \times |\mathcal{F}|$ matrix of feature vectors ϕ . We accomplish this by pre-multiplying $\mathcal{M}^\pi V(\theta)$ by the projection operator Π . This allows us to form the mean-squared projected Bellman error using

$$\text{MSPBE}(\theta) = \frac{1}{2} \|\bar{V}(\theta) - \Pi \mathcal{M}^\pi \bar{V}(\theta)\|_D^2 \quad (9.60)$$

$$= \frac{1}{2} (\bar{V}(\theta) - \Pi \mathcal{M}^\pi \bar{V}(\theta))^T D (\bar{V}(\theta) - \Pi \mathcal{M}^\pi \bar{V}(\theta)). \quad (9.61)$$

We can now use this new objective function as the basis of an optimization algorithm to find θ . Recall that D^π is a $|\mathcal{S}| \times |\mathcal{S}|$ diagonal matrix with elements d_s^π ,

giving us the probability that we are in state s while following policy π . We use D^π as a scaling matrix to give us the probability that we are in state s . We start by noting the identities

$$\begin{aligned}\mathbb{E}[\phi\phi^T] &= \sum_{s \in \mathcal{S}} d_s^\pi \phi_s \phi_s^T \\ &= \Phi^T D^\pi \Phi, \\ \mathbb{E}[\delta^\pi \phi] &= \sum_{s \in \mathcal{S}} d_s^\pi \phi_s \left(c^\pi(s) + \gamma \sum_{s' \in \mathcal{S}} p^\pi(s'|s) \bar{V}(s'|\theta) - \bar{V}(s|\theta) \right) \\ &= \Phi^T D^\pi (\mathcal{M}^\pi \bar{V}(\theta) - \bar{V}(\theta)).\end{aligned}$$

In writing these identities, we are slightly abusing notation by thinking of ϕ as a column vector that depends on a random state S , where the expectation is being taken over the states. In the second identity, ϕ^π is a scalar depending on a random state S , where again the expectation is over the states.

The derivations here and below make extensive use of matrices, which can be difficult to parse. A useful exercise is to write out the matrices, assuming that there is a feature $\phi_f(s)$ for each state s so that $\phi_f(s) = 1$ if feature f corresponds to state s (see exercise 9.4).

We see that the role of the scaling matrix D^π is to enable us to take the expectation of the quantities $\phi\phi^T$ and $\delta^\pi \phi$. Below we are going to simulate these quantities, where a state will occur with probability d_s^π . We also use

$$\begin{aligned}\Pi^T D^\pi \Pi &= (\Phi(\Phi^T D^\pi \Phi)^{-1} \Phi^T D^\pi)^T D^\pi (\Phi(\Phi^T D^\pi \Phi)^{-1} \Phi^T D^\pi) \\ &= (D^\pi)^T \Phi(\Phi^T D^\pi \Phi)^{-1} \Phi^T D^\pi \Phi(\Phi^T D^\pi \Phi)^{-1} \Phi^T D^\pi \\ &= (D^\pi)^T \Phi(\Phi^T D^\pi \Phi)^{-1} \Phi^T D^\pi.\end{aligned}$$

We have one last painful piece of linear algebra that gives us a more compact form for MSPBE(θ). Pulling the $1/2$ to the left-hand side (this will later vanish when we take the derivative), we can write

$$\begin{aligned}2\text{MSPBE}(\theta) &= \|\bar{V}(\theta) - \Pi \mathcal{M}^\pi \bar{V}(\theta)\|_D^2 \\ &= \|\Pi(\bar{V}(\theta) - \mathcal{M}^\pi \bar{V}(\theta))\|_D^2 \\ &= (\Pi(\bar{V}(\theta) - \mathcal{M}^\pi \bar{V}(\theta)))^T D^\pi (\Pi(\bar{V}(\theta) - \mathcal{M}^\pi \bar{V}(\theta))) \\ &= (\bar{V}(\theta) - \mathcal{M}^\pi \bar{V}(\theta))^T \Pi^T D^\pi \Pi (\bar{V}(\theta) - \mathcal{M}^\pi \bar{V}(\theta)) \\ &= (\bar{V}(\theta) - \mathcal{M}^\pi \bar{V}(\theta))^T (D^\pi)^T \Phi(\Phi^T (D^\pi) \Phi)^{-1} \Phi^T D^\pi (\bar{V}(\theta) \\ &\quad - \mathcal{M}^\pi \bar{V}(\theta))\end{aligned}$$

$$\begin{aligned}
&= (\Phi^T D^\pi (\mathcal{M}^\pi \bar{V}(\theta) - \bar{V}(\theta)))^T (\Phi^T D^\pi \Phi)^{-1} \Phi^T D^\pi (\mathcal{M}^\pi \bar{V}(\theta) \\
&\quad - \bar{V}(\theta)) \\
&= \mathbb{E}[\delta^\pi \phi]^T \mathbb{E}[\phi \phi^T]^{-1} \mathbb{E}[\delta^\pi \phi]. \tag{9.62}
\end{aligned}$$

We next need to estimate the gradient of this error $\nabla_\theta \text{MSPBE}(\theta)$. Keep in mind that $\delta^\pi = c^\pi + \gamma P^\pi \Phi \theta - \Phi \theta$, where here we interpret δ^π as a vector with element δ_s^π . If ϕ is the column vector with element $\phi(s)$, assume that s' occurs with probability $p^\pi(s'|s)$ under policy π , and let ϕ' be the corresponding column vector. Differentiating (9.62) gives

$$\begin{aligned}
\nabla_\theta \text{MSPBE}(\theta) &= \mathbb{E}[(\gamma \phi' - \phi) \phi^T] \mathbb{E}[\phi \phi^T]^{-1} \mathbb{E}[\delta^\pi \phi] \\
&= -\mathbb{E}[(\phi - \gamma \phi') \phi^T] \mathbb{E}[\phi \phi^T]^{-1} \mathbb{E}[\delta^\pi \phi].
\end{aligned}$$

We are going to use a standard stochastic gradient updating algorithm for minimizing the error given by $\text{MSPBE}(\theta)$, which is given by

$$\theta^{n+1} = \theta^n - \alpha_n \nabla_\theta \text{MSPBE}(\theta) \tag{9.63}$$

$$= \theta^n + \alpha_n \mathbb{E}[(\phi - \gamma \phi') \phi^T] \mathbb{E}[\phi \phi^T]^{-1} \mathbb{E}[\delta^\pi \phi]. \tag{9.64}$$

We can create a linear predictor which approximates

$$w \approx \mathbb{E}[\phi \phi^T]^{-1} \mathbb{E}[\delta^\pi \phi],$$

where w is approximated using

$$w^{n+1} = w^n + \beta_n (\delta^{\pi,n} - (\phi^n)^T w^n) \phi^n.$$

This allows us to write the gradient

$$\begin{aligned}
\nabla_\theta \text{MSPBE}(\theta) &= -\mathbb{E}[(\phi - \gamma \phi') \phi^T] \mathbb{E}[\phi \phi^T]^{-1} \mathbb{E}[\delta^\pi \phi] \\
&\approx -\mathbb{E}[(\phi - \gamma \phi') \phi^T] w.
\end{aligned}$$

We have now created the basis for two algorithms. The first is called *generalized temporal difference 2* (GTD2), given by

$$\theta^{n+1} = \theta^n + \alpha_n (\phi^n - \gamma \phi^{n+1}) ((\phi^n)^T w^n). \tag{9.65}$$

Here ϕ^n is the column vector of basis functions when we are in state S^n , while ϕ^{n+1} is the column vector of basis functions for the next state S^{n+1} . Note that if equation (9.65) is executed right to left, all calculations are linear in the number of features F . This linearity property is important for algorithms which use a large number of features. For example, recent research to develop algorithms for the Chinese game of Go might use millions of basis functions.

A variant, called temporal difference with gradient corrector (TDC) is derived by using a slightly modified calculation of the gradient

$$\begin{aligned}\nabla_{\theta} \text{MSPBE}(\theta) &= -\mathbb{E}[(\phi - \gamma\phi')\phi^T] \mathbb{E}[\phi\phi^T]^{-1} \mathbb{E}[\delta^\pi \phi] \\ &= -(\mathbb{E}[\phi\phi^T] - \gamma\mathbb{E}[\phi'\phi^T]) \mathbb{E}[\phi\phi^T]^{-1} \mathbb{E}[\delta^\pi \phi] \\ &= -(\mathbb{E}[\delta^\pi \phi] - \gamma\mathbb{E}[\phi'\phi^T]\mathbb{E}[\phi\phi^T]^{-1}\mathbb{E}[\delta^\pi \phi]) \\ &\approx -(\mathbb{E}[\delta^\pi \phi] - \gamma\mathbb{E}[\phi'\phi^T]w).\end{aligned}$$

This gives us the TDC algorithm

$$\theta^{n+1} = \theta^n + \alpha_n \left(\delta^{\pi,n} \phi^n - \gamma \phi^{n'} ((\phi^n)^T w^n) \right). \quad (9.66)$$

GTD2 and TDC have both been proven to converge to the optimal value of θ for a fixed learning policy $A^\pi(s)$ that may be different than the behavior (sampling) policy. That is, after updating θ^n where the temporal difference $\delta^{\pi,n}$ is computed assuming we are in state S^n and follow policy π , we are allowed to follow a separate sampling policy to determine S^{n+1} . This allows us to directly control the states that we visit, rather than depending on the decisions made by the learning policy. As of this writing, these algorithms are quite new and there is very little empirical work to test stability and rate of convergence.

9.8 LEAST SQUARES TEMPORAL DIFFERENCING WITH KERNEL REGRESSION*

In Section 8.4.2 we introduced the idea of kernel regression, where we can approximate the value of a function by using a weighted sum of nearby observations. If S^i is the i th observation of a state, and we observe a value \hat{v}^i , we can approximate the value of visiting a generic state s by using

$$\bar{V}(s) = \frac{\sum_{i=1}^n K_h(s, S^i) \hat{v}^i}{\sum_{i=1}^n K_h(s, S^i)},$$

where $K_h(s, S^i)$ is a weighting function that declines with the distance between s and S^i . Kernel functions are introduced in Section 8.4.2.

We now use two properties of kernels. One, which we first introduced in section 8.4.2, is that for most kernel functions $K_h(s, s')$, there exists a potentially high-dimensional set of basis functions $\phi(s)$ such that

$$K_h(s, s') = \phi(s)^T \phi(s').$$

There is also a result known as the kernel representer theorem that states that there exists a vector of coefficients β_i , $i = 0, \dots, m$ that allows us to write

$$\theta^m = \sum_{i=0}^m \phi(S^i) \beta^i. \quad (9.67)$$

This allows us to write our value function approximation using

$$\begin{aligned}\bar{V}(s) &= \phi(s)^T \theta^m \\ &= \sum_{i=0}^m \phi(s)^T \phi(S^i) \beta^i \\ &= \sum_{i=0}^m K_h(s, S^i) \beta^i.\end{aligned}$$

Recall from equations (9.43), (9.47), and (9.48) that

$$A^m \theta = b^m,$$

where

$$\left(\frac{1}{m} \sum_{i=1}^m \phi(S^i) (\phi(S^i) - \gamma \phi(S^{i+1}))^T \right) \theta = \frac{1}{m} \sum_{i=1}^m \phi(S^i) \hat{C}^i + \varepsilon^i,$$

with ε^i representing the error in the fit. Substituting in (9.67) gives us

$$\left(\frac{1}{m} \sum_{i=1}^m \phi(S^i) (\phi(S^i) - \gamma \phi(S^{i+1}))^T \right) \sum_{i=1}^m \phi^i \beta^i = \frac{1}{m} \sum_{i=1}^m \phi(S^i) \hat{C}^i + \varepsilon^i.$$

The single step regression function is given by

$$\phi(S^i) (\phi(S^i) - \gamma \phi(S^{i+1}))^T \sum_{i=1}^m \phi^i \beta^i = \frac{1}{m} \sum_{i=1}^m \phi(S^i) \hat{C}^i + \varepsilon^i. \quad (9.68)$$

Let $\Phi^m = [\phi(S^1), \dots, \phi(S^m)]^T$ be a $m \times |\mathcal{F}|$ matrix, where each row is the vector $\phi(S^i)$, and let $k^m(S^i) = [K_h(S^1, S^i), \dots, K_h(S^m, S^i)]^T$ be a column vector of the kernel functions capturing the weight between each observed state S^1, \dots, S^m , and a particular state S^i . Multiplying both sides of (9.68) by Φ^m gives us,

$$\Phi^m \phi(S^i) (\phi(S^i) - \gamma \phi(S^{i+1}))^T \sum_{i=1}^m \phi^i \beta^i = \Phi^m \frac{1}{m} \sum_{i=1}^m \phi(S^i) \hat{C}^i + \Phi^m \varepsilon^i.$$

Keeping in mind that products of basis functions can be replaced with kernel functions, we obtain

$$\sum_{i=1}^m k^m(S^i) (k^m(S^i) - \gamma k^m(S^{i+1})) \beta^m = \sum_{i=1}^m k^m(S^i) \hat{C}^i.$$

Define the $m \times m$ matrix and m -vector b^m

$$M^m = \sum_{i=1}^m k^m(S^i)(k^m(s^i) - \gamma k^m(S^{i+1})),$$

$$b^m = \sum_{i=1}^m k^m(S^i)\hat{C}^i.$$

Then we can solve for β^m recursively using

$$\begin{aligned}\beta^m &= (M^m)^{-1}b^m, \\ M^{m+1} &= M^m + k^{m+1}(S^{m+1})((k^{m+1}(S^{m+1}))^T - \gamma(k^{m+2}(S^{m+2}))^T), \\ b^{m+1} &= b^m + k^{m+1}(S^{m+1})\hat{C}^m.\end{aligned}$$

The power of kernel regression is that it does not require specifying basis functions. However, this flexibility comes with a price. If we are using a parametric model, we have to deal with a vector θ with $|\mathcal{F}|$ elements. Normally we try to specify a relatively small number of features, although there are applications that might use a million features. With kernel regression we have to invert the $m \times m$ matrix M^m , which can become very expensive as m grows. As a result practical algorithms would have to use advanced research on sparsification.

9.9 VALUE FUNCTION APPROXIMATIONS BASED ON BAYESIAN LEARNING*

A different strategy for updating value functions is one based on Bayesian learning. Suppose that we start with a prior $V^0(s)$ of the value of being in state s . We assume that we have a known covariance function $\text{Cov}(s, s')$ that captures the relationship in our belief about $V(s)$ and $V(s')$. A good example where this function would be known might be a function where s is continuous (or a discretization of a continuous surface), where we might use

$$\text{Cov}(s, s') \propto e^{-\|s-s'\|^2/b}, \quad (9.69)$$

where b is a bandwidth. This function captures the intuitive behavior that if two states are close to each other, their covariance is higher. So, if we make an observation that raises our belief about $V(s)$, then our belief about $V(s')$ will increase also, and will increase more if s and s' are close to each other. We also assume that we have a variance function $\lambda(s)$ that captures the noise in a measurement $\hat{v}(s)$ of the function at state s .

Our Bayesian updating model is designed for applications where we have access to observations \hat{v}^n of our true function $V(s)$, which we can view as coming from our prior distribution of belief. This assumption effectively precludes using updating algorithms based on approximate value iteration, Q -learning, and least squares

policy evaluation. We cannot eliminate the bias, but below we describe how to minimize it. We then describe Bayesian updating using lookup tables and parametric models.

9.9.1 Minimizing bias

We would very much like to have observations $\hat{v}^n(s)$ that we can view as an unbiased observation of $V(s)$. One way to do this is to build on the methods described in Section 9.1.

To illustrate, suppose that we have a policy π that determines the action a_t we take when in state S_t , generating a contribution \hat{C}_t^n . Say we simulate this policy for T time periods using

$$\hat{v}^n(T) = \sum_{t=0}^T \gamma^t \hat{C}_t.$$

If we have a finite horizon problem and T is the end of our horizon, then we are done. If our problem has an infinite horizon, we can project the infinite horizon value of our policy by first approximating the one-period contribution using

$$\bar{c}_T^n = \frac{1}{T} \sum_{t=0}^T \hat{C}_t^n.$$

Now suppose that this estimates the average contribution per period starting at time $T+1$. Our infinite horizon estimate would be

$$\hat{v}^n = \hat{v}_0(T) + \gamma^{T+1} \frac{1}{1-\gamma} \bar{c}_T^n.$$

Finally, we use \hat{v}^n to update our value function approximation \bar{V}^{n-1} to obtain \bar{V}^n .

We next illustrate the Bayesian updating formulas for lookup tables and parametric models.

9.9.2 Lookup Tables with Correlated Beliefs

Previously when we have used lookup tables, if we update the value $\bar{V}^n(s)$ for some state s , we do not use this information to update the values of any other states. With our Bayesian model, we can do much more if we have access to a covariance function such as the one we illustrated in equation (9.69).

Suppose that we have discrete states. Assume that we have a covariance function $\text{Cov}(s, s')$ in the form of a covariance matrix Σ , where $\text{Cov}(s, s') = \Sigma(s, s')$. Let V^n be our vector of beliefs about the value $V(s)$ of being in each state (we use V^n to represent our Bayesian beliefs so that \bar{V}^n can represent our frequentist estimates). Also let Σ^n be the covariance matrix of our belief about the vector V . If $\hat{v}^n(S^n)$

is an (approximately) unbiased sample observation of $V(s)$, the Bayesian formula for updating V^n is given by

$$V^{n+1}(s) = V^n(s) + \frac{\hat{v}^n(S^n) - V^n(s)}{\lambda(S^n) + \Sigma^n(S^n, S^n)} \Sigma^n(s, S^n).$$

This has to be computed for each s (or at least each s where $\Sigma^n(s, S^n) > 0$). We update the covariance matrix using

$$\Sigma^{n+1}(s, s') = \Sigma^n(s, s') - \frac{\Sigma^n(s, S^n) \Sigma^n(S^n, s')}{\lambda(S^n) + \Sigma^n(S^n, S^n)}.$$

9.9.3 Parametric Models

For most applications a parametric model (specifically, a linear model) is going to be much more practical. Our frequentist updating equations for our regression vector θ^n were given above as

$$\theta^n = \theta^{n-1} - \frac{1}{\gamma^n} B^{n-1} \phi^n \hat{\varepsilon}^n, \quad (9.70)$$

$$B^n = B^{n-1} - \frac{1}{\gamma^n} (B^{n-1} \phi^n (\phi^n)^T B^{n-1}), \quad (9.71)$$

$$\gamma^n = 1 + (\phi^n)^T B^{n-1} \phi^n, \quad (9.72)$$

where $\hat{\varepsilon}^n = \bar{V}(\theta^{n-1})(S^n) - \hat{v}^n$ is the difference between our current estimate $\bar{V}(\theta^{n-1})(S^n)$ of the value function at our observed state S^n and our most recent observation \hat{v}^n . The adaptation for a Bayesian model is quite minor. The matrix B^n represents

$$B^n = [(X^n)^T X^n]^{-1}.$$

It is possible to show that the covariance matrix Σ^θ (which is dimensioned by the number of basis functions) is given by

$$\Sigma^\theta = B^n \lambda.$$

In our Bayesian model, λ is the variance of the difference between our observation \hat{v}^n and the true value function $v(S^n)$, where we assume λ is known. This variance may depend on the state that we have observed, in which case we would write it as $\lambda(s)$, but in practice, since we do not know the function $V(s)$, it is hard to believe that we would be able to specify $\lambda(s)$. We replace B^n with $\Sigma^{\theta,n}$ and rescale γ^n to create the following set of updating equations:

$$\theta^n = \theta^{n-1} - \frac{1}{\gamma^n} \Sigma^{\theta,n-1} \phi^n \hat{\varepsilon}^n, \quad (9.73)$$

$$\Sigma^{\theta,n} = \Sigma^{\theta,n-1} - \frac{1}{\gamma^n} (\Sigma^{\theta,n-1} \phi^n (\phi^n)^T \Sigma^{\theta,n-1}), \quad (9.74)$$

$$\gamma^n = \lambda + (\phi^n)^T \Sigma^{\theta,n-1} \phi^n. \quad (9.75)$$

9.9.4 Creating the Prior

Approximate dynamic programming has been approached from a Bayesian perspective in the research literature but has apparently received very little attention otherwise. We suspect that while there exist many applications in stochastic search where it is valuable to use a prior distribution of belief, it is much harder to build a prior on a value function.

Lacking any specific structural knowledge of the value function, we anticipate that the easiest strategy will be to start with $V^0(s) = v^0$, which is a constant across all states. There are several strategies we might use to estimate v^0 . We might sample a state S^i at random, and find the best contribution $\hat{C}^i = \max_a C(S^i, a)$. Repeat this n times and compute

$$\bar{v} = \frac{1}{n} \sum_{i=1}^n \hat{C}^i.$$

Finally, let $v^0 = \bar{v}/(1 - \gamma)$ if we have an infinite horizon problem. The hard part is that the variance λ has to capture the variance of the difference between v^0 and the true $V(s)$. This requires having some sense of the degree to which v^0 differs from $V(s)$. We recommend being very conservative, which is to say choose a variance λ such that $v^0 + 2\sqrt{\lambda}$ easily covers what $V(s)$ might be. Of course, this also requires some judgment about the likelihood of visiting different states.

9.10 WHY DOES IT WORK*

9.10.1 Derivation of the Recursive Estimation Equations

Here we derive the recursive estimation equations given by equations (9.24) through (9.28). To begin, we note that the matrix $(X^n)^T X^n$ is an $I+1$ by $I+1$ matrix where the element for row i , column j is given by

$$[(X^n)^T X^n]_{i,j} = \sum_{m=1}^n x_i^m x_j^m.$$

This term can be computed recursively using

$$[(X^n)^T X^n]_{i,j} = \sum_{m=1}^{n-1} (x_i^m x_j^m) + x_i^n x_j^n.$$

In matrix form this can be written

$$[(X^n)^T X^n] = [(X^{n-1})^T X^{n-1}] + x^n (x^n)^T.$$

Keeping in mind that x^n is a column vector, and $x^n (x^n)^T$ is an $I+1$ by $I+1$ matrix formed by the cross products of the elements of x^n . We now use the

Sherman–Morrison formula (see Section 9.10.2 for a derivation) for updating the inverse of a matrix

$$[A + uu^T]^{-1} = A^{-1} - \frac{A^{-1}uu^TA^{-1}}{1 + u^TA^{-1}u}, \quad (9.76)$$

where A is an invertible $n \times n$ matrix, and u is an n -dimensional column vector. Applying this formula to our problem, we obtain

$$\begin{aligned} [(X^n)^T X^n]^{-1} &= [(X^{n-1})^T X^{n-1} + x^n(x^n)^T]^{-1} \\ &= [(X^{n-1})^T X^{n-1}]^{-1} \\ &\quad - \frac{[(X^{n-1})^T X^{n-1}]^{-1} x^n(x^n)^T [(X^{n-1})^T X^{n-1}]^{-1}}{1 + (x^n)^T [(X^{n-1})^T X^{n-1}]^{-1} x^n}. \end{aligned} \quad (9.77)$$

The term $(X^n)^T Y^n$ can also be updated recursively using

$$(X^n)^T Y^n = (X^{n-1})^T Y^{n-1} + x^n(y^n). \quad (9.78)$$

To simplify the notation, let

$$\begin{aligned} B^n &= [(X^n)^T X^n]^{-1}, \\ \gamma^n &= 1 + (x^n)^T [(X^{n-1})^T X^{n-1}]^{-1} x^n. \end{aligned}$$

This simplifies our inverse updating equation (9.77) to

$$B^n = B^{n-1} - \frac{1}{\gamma^n} (B^{n-1} x^n (x^n)^T B^{n-1}).$$

Recall that

$$\bar{\theta}^n = [(X^n)^T X^n]^{-1} (X^n)^T Y^n. \quad (9.79)$$

Combining (9.79) with (9.77) and (9.78) gives

$$\begin{aligned} \bar{\theta}^n &= [(X^n)^T X^n]^{-1} (X^n)^T Y^n \\ &= \left(B^{n-1} - \frac{1}{\gamma^n} (B^{n-1} x^n (x^n)^T B^{n-1}) \right) ((X^{n-1})^T Y^{n-1} + x^n y^n), \\ &= B^{n-1} (X^{n-1})^T Y^{n-1} \\ &\quad - \frac{1}{\gamma^n} B^{n-1} x^n (x^n)^T B^{n-1} [(X^{n-1})^T Y^{n-1} + x^n y^n] + B^{n-1} x^n y^n. \end{aligned}$$

We can start to simplify by using $\bar{\theta}^{n-1} = B^{n-1} (X^{n-1})^T Y^{n-1}$. We are also going to bring the term $x^n B^{n-1}$ inside the square brackets. Finally, we are going to bring the last term $B^{n-1} x^n y^n$ inside the brackets by taking the coefficient $B^{n-1} x^n$ outside the

brackets and multiplying the remaining y^n by the scalar $\gamma^n = 1 + (x^n)^T B^{n-1} x^n$, giving us

$$\begin{aligned}\bar{\theta}^n &= \bar{\theta}^{n-1} - \frac{1}{\gamma^n} B^{n-1} x^n \left[(x^n)^T (B^{n-1} (X^{n-1})^T Y^{n-1}) \right. \\ &\quad \left. + (x^n)^T B^{n-1} x^n y^n - (1 + (x^n)^T B^{n-1} x^n) y^n \right].\end{aligned}$$

Again, we use $\bar{\theta}^{n-1} = B^{n-1} (X^{n-1})^T Y^{n-1}$ and observe that there are two terms $(x^n)^T B^{n-1} x^n y^n$ that cancel, leaving

$$\bar{\theta}^n = \bar{\theta}^{n-1} - \frac{1}{\gamma^n} B^{n-1} x^n \left((x^n)^T \bar{\theta}^{n-1} - y^n \right).$$

We note that $(\bar{\theta}^{n-1})^T x^n$ is our prediction of y^n using the parameter vector from iteration $n - 1$ and the explanatory variables x^n . y^n is, of course, the actual observation, so our error is given by

$$\hat{\varepsilon}^n = y^n - (\bar{\theta}^{n-1})^T x^n.$$

Let

$$H^n = -\frac{1}{\gamma^n} B^{n-1}.$$

We can now write our updating equation using

$$\bar{\theta}^n = \bar{\theta}^{n-1} - H^n x^n \hat{\varepsilon}^n. \quad (9.80)$$

9.10.2 The Sherman–Morrison Updating Formula

The Sherman–Morrison matrix updating formula (also known as the Woodbury formula or the Sherman–Morrison–Woodbury formula) assumes that we have a matrix A and that we are going to update it with the outer product of the column vector u to produce the matrix B , given by

$$B = A + uu^T. \quad (9.81)$$

Pre-multiply by B^{-1} and post-multiply by A^{-1} to obtain

$$A^{-1} = B^{-1} + B^{-1}uu^T A^{-1}. \quad (9.82)$$

Post-multiply by u :

$$\begin{aligned}A^{-1}u &= B^{-1}u + B^{-1}uu^T A^{-1}u \\ &= B^{-1}u (1 + u^T A^{-1}u).\end{aligned}$$

Note that $u^T A^{-1} u$ is a scalar. Divide through by $(1 + u^T A^{-1} u)$:

$$\frac{A^{-1} u}{(1 + u^T A^{-1} u)} = B^{-1} u.$$

Now post-multiply by $u^T A^{-1}$:

$$\frac{A^{-1} u u^T A^{-1}}{(1 + u^T A^{-1} u)} = B^{-1} u u^T A^{-1}. \quad (9.83)$$

Equation (9.82) gives us

$$B^{-1} u u^T A^{-1} = A^{-1} - B^{-1}. \quad (9.84)$$

Substituting (9.84) into (9.83) gives

$$\frac{A^{-1} u u^T A^{-1}}{(1 + u^T A^{-1} u)} = A^{-1} - B^{-1}. \quad (9.85)$$

Solving for B^{-1} gives us

$$\begin{aligned} B^{-1} &= [A + u u^T]^{-1} \\ &= A^{-1} - \frac{A^{-1} u u^T A^{-1}}{(1 + u^T A^{-1} u)}, \end{aligned}$$

which is the desired formula.

9.11 BIBLIOGRAPHIC NOTES

Section 9.1 This section reviews a number of classical methods for estimating the value of a policy drawn from the reinforcement learning community. The best overall reference for this is Sutton and Barto (1998). Least squares temporal differencing is due to Bradtke and Barto (1996).

Section 9.2 Tsitsiklis (1994) and Jaakkola et al. (1994) were the first to make the connection between emerging algorithms in approximate dynamic programming (Q -learning, temporal difference learning) and the field of stochastic approximation theory (Robbins and Monro (1951), Blum (1954a), Kushner and Yin (2003)).

Section 9.3 Ljung and Soderstrom (1983) and Young (1984) provide nice treatments of recursive statistics. Precup et al. (2001) gives the first convergent algorithm for off-policy temporal-difference learning by way of basis functions using an adjustment that is based on the relative probabilities of choosing an action from the target and behavioral policies. Lagoudakis et al. (2002) and Bradtke and Barto (1996) presents least squares methods in the context of reinforcement learning. Choi and Van (2006) uses the Kalman filter

to perform scaling for stochastic gradient updates, avoiding the scaling problems inherent in stochastic gradient updates such as equation (9.22). Bertsekas and Nedić (2003) describes the use of least squares equation with a linear (in the parameters) value function approximation using policy iteration and proves convergence for $\text{TD}(\lambda)$ with general λ . Bertsekas et al. (2004) presents a scaled method for estimating linear value function approximations within a temporal-differencing algorithm. Section 9.3.4 is based on Soderstrom et al. (1978).

Section 9.4 The issue of whether TD learning with a linear approximation architecture was convergent was debated in the literature for some time as a result of a mixture of supporting and contradictory results. The fundamental question was largely resolved in Tsitsiklis and Van Roy (1997); see the references there for more of the history. Precup et al. (2001) provides the first convergent off-policy version of a TD model with a linear approximation, but this is limited to policies that retain a positive probability of selecting any action.

Section 9.5 The development of Bellman's equation using linear models is based on Tsitsiklis and Van Roy (1997), Lagoudakis and Parr (2003), and Bertsekas (2009). Tsitsiklis and Van Roy (1997) highlights the central role of the D -norm used in this section, which also plays a central role in the design of a simulation-based version of the algorithm.

Section 9.6 The analysis of dynamic programs with a single state is based on Ryzhov et al. (2009).

Section 9.7 Baird (1995) provides a nice example showing that approximate value iteration may diverge when using a linear architecture, even when the linear model may fit the true value function perfectly. Tsitsiklis and Van Roy (1997) establishes the importance of using Bellman errors weighted by the probability of being in a state. De Farias and Van Roy (2000) shows that there does not necessarily exist a fixed point to the projected form of Bellman's equation $\Phi\theta = \Pi\mathcal{M}\Phi\theta$, where \mathcal{M} is the max operator. This paper also shows that a fixed point does exist for a projection operator Π_D defined with respect to the norm $\|\cdot\|_D$ that weights a state s with the probability d_s of being in this state. This result is first shown for a fixed policy, and then for a class of randomized policies. GTD2 and TDC are due to Sutton et al. (2009a), with material from Sutton et al. (2009b).

Section 9.8 Our adaptation of least squares temporal differencing using kernel regression was presented in Ma and Powell (2010a).

Section 9.9 Dearden et al. (1998) introduces the idea of using Bayesian updating for Q -learning. Dearden et al. (1999) then considers model-based Bayesian learning. Our presentation is based on Ryzhov and Powell (2010b), which introduces the idea of correlated beliefs.

Section 9.10.2 The Sherman–Morrison updating formulas are given in a number of references, such as Ljung and Soderstrom (1983) and Golub and Loan (1996).

PROBLEMS

- 9.1** Consider a “Markov decision process” with a single state and single policy. Assume that we do not know the expected value of the contribution \hat{C} , but each time it is sampled, we draw a sample realization from the uniform distribution between 0 and 20. Also assume a discount factor of $\gamma = 0.90$. Let $V = \sum_{t=0}^{\infty} \gamma^t \hat{C}_t$. The exercises below can be formed in a spreadsheet.
- (a) Estimate V using LSTD using 100 iterations.
 - (b) Estimate V using LSPE using 100 iterations.
 - (c) Estimate V using recursive least squares, executing the algorithm for 100 iterations.
 - (d) Estimate V using temporal differencing (approximate value iteration) and a stepsize of $1/n^7$.
 - (e) Repeat (d) using a stepsize of $5/(5+n-1)$.
 - (f) Compare the rates of convergence of the different procedures.
- 9.2** Repeat the exercise above using a discount factor of 0.95.
- 9.3** Contrast the objective function used in equation (9.39) with the objective being minimized that we used throughout Chapter 3.
- 9.4** We are going to walk through the derivation of the equations in Section 9.7 assuming that there is a feature for each state, where $\phi_f(s) = 1$ if feature f corresponds to state s , and 0 otherwise. When asked for a sample of a vector or matrix, we assume there are three states and three features. As above, let d_s^π be the probability of being in state s under policy π , and let D^π be the diagonal matrix consisting of the elements d_s^π .
- (a) What is the column vector ϕ if $s = 1$? What does $\phi\phi^T$ look like?
 - (b) If d_s^π is the probability of being in state s under policy π , write out $\mathbb{E}[\phi\phi^T]$.
 - (c) Write out the matrix Φ .
 - (d) What is the projection matrix Π ?
 - (e) Write out equation (9.62) for MSPBE(θ).
- 9.5** Repeat the derivation of (9.7), but this time introduce a discount factor γ . Show that you obtain it by updating equation (9.8).

Optimizing While Learning

We are finally ready to tackle the problem of searching for good policies while simultaneously trying to produce good value function approximations. Our discussion is restricted to problems where the policy is based on the value function approximation, since Chapter 7 has already addressed the strategy of direct policy search. The guiding principle in this chapter is that we can find good policies if we can find good value function approximations.

The statistical tools presented in Chapters 8 and 9 focused on finding the best statistical fit within a particular approximation architecture. We actually did not address whether we had chosen a good architecture. This is particularly true of our linear models, where we used the tools of stochastic optimization and linear regression to ensure that we obtain the best fit given a model, without regard to whether it was a good model.

In this chapter we return to the problem of trying to find the best policy, where we assume throughout that our policies are of the form

$$A^\pi(S) = \arg \max_{a \in \mathcal{A}} (C(S, a) + \gamma \mathbb{E} \bar{V}(S^M(S, a, W))),$$

if we have an infinite horizon problem with discrete actions. Or, we may consider finite horizon problems with vector-valued decisions x_t , where a policy would look like

$$X_t^\pi(S_t) = \arg \max_{x_t \in \mathcal{X}_t} (C(S_t, x_t) + \gamma \mathbb{E} \bar{V}_t(S^M(S_t, x_t, W_{t+1}))).$$

The point is that the policy depends on some sort of value function approximation. When we write our generic optimization problem

$$\max_{\pi} \mathbb{E} \sum_{t=0}^T \gamma^t C(S_t, A_t^\pi(S_t)),$$

the maximization over policies can mean choosing an architecture for $\bar{V}_t(S_t)$, and choosing the parameters that control the architecture. For example, we might be choosing between a myopic policy (see equation (6.4) in Chapter 6), or perhaps a simple linear architecture with one basis function

$$\bar{V}(S) = \theta_0 + \theta_1 S, \quad (10.1)$$

or perhaps a linear architecture with two basis functions,

$$\bar{V}(S) = \theta_0 + \theta_1 S + \theta_2 S^2. \quad (10.2)$$

We might even use a nonlinear architecture such as

$$\bar{V}(S) = \frac{e^{\theta_0 + \theta_1 S}}{1 + e^{\theta_0 + \theta_1 S}}.$$

Optimizing over policies may consist of choosing a value function approximation such as (10.1) or (10.2), but then we still have to choose the best parameter vector within each class.

We begin our presentation with an overview of the basic algorithmic strategies that we cover in this chapter, all of which are based on using value function approximations that are intended to approximate the value of being in a state. The remainder of the chapter is organized around covering the following strategies:

Approximate value iteration. These are policies that iteratively update the value function approximation, and then immediately update the policy. We strive to find a value function approximation that estimates the value of being in each state while following a (near) optimal policy, but only in the limit. We intermingle the treatment of finite and infinite horizon problems. Variations include

Lookup table representations. Here we introduce three major strategies that reflect the use of the pre-decision state, state-action pairs, and the post-decision state:

AVI for pre-decision state. Approximate value iteration using the classical pre-decision state variable.

Q -learning. Estimating the value of state-action pairs.

AVI for the post-decision state. Approximate value iteration where value function approximations are approximated around the post-decision state.

Parametric architectures. We discuss the issues that arise when trying to use linear models in the context of approximate value iteration.

Approximate policy iteration. These are policies that attempt to explicitly approximate the value of a policy to some level of accuracy within an inner loop, within which the policy is held fixed.

API using lookup tables. We use this setting to present the basic idea.

API using linear models. This is perhaps one of the most important areas of research in approximate dynamic programming.

API using nonparametric models. This is a relatively young area of research, and we summarize some recent results.

Linear programming method. The linear programming method, first introduced in Chapter 3, can be adapted to exploit value function approximations.

10.1 OVERVIEW OF ALGORITHMIC STRATEGIES

The algorithmic strategies that we examine in this chapter are based on the principles of value iteration and policy iteration, first introduced in Chapter 3. We continue to adapt our algorithms to finite and infinite horizons. Basic value iteration for finite horizon problems work by solving

$$V_t(S_t) = \max_{a_t} (C(S_t, a_t) + \gamma \mathbb{E}\{V_{t+1}(S_{t+1})|S_t\}). \quad (10.3)$$

Equation (10.3) works by stepping backward in time, where $V_t(S_t)$ is computed for each (presumably discrete) state S_t . This is classical “backward” dynamic programming that suffers from the well-known curse of dimensionality because we typically are unable to “loop over all the states.”

Approximate dynamic programming approaches finite horizon problems by solving problems of the form

$$\hat{v}_t^n = \max_{a_t} (C(S_t^n, a_t) + \gamma \bar{V}_{t+1}^{n-1}(S_t^{M,a}(S_t^n, a_t))). \quad (10.4)$$

Here we have formed the value function approximation around the post-decision state. We execute the equations by stepping forward in time. If a_t^n is the action that optimizes (10.4), then we compute our next state using $S_{t+1}^n = S^M(S_t^n, a_t^n, W_{t+1}^n)$, where W_{t+1}^n is sampled from some distribution. The process runs until we reach the end of our horizon, at which point we return to the beginning of the horizon and repeat the process.

Classical value iteration for infinite horizon problems is centered on the basic iteration

$$V^n(S) = \max_a (C(S, a) + \gamma \mathbb{E}\{V^{n-1}(S')|S\}). \quad (10.5)$$

Again, equation (10.5) has to be executed for each state S . After each iteration the new estimate V^n replaces the old estimate V^{n-1} on the right, after which n is incremented.

When we use approximate methods, we might observe an estimate of the value of being in a state using

$$\hat{v}^n = \max_a (C(S^n, a) + \gamma \bar{V}^{n-1}(S^{M,a}(S^n, a^n))). \quad (10.6)$$

We then use the observed state-value pair (S^n, \hat{v}^n) to update the value function approximation.

When we use approximate value iteration, \hat{v}^n (or \hat{v}_t^n) cannot be viewed as a noisy but unbiased observation of the value of being in a state. These observations are calculated as a function of the value function $\bar{V}^{n-1}(s)$. While we hope the value function approximation converges to something, we generally cannot say anything about the function prior to convergence. This means that \hat{v}^n does not have any particular property. We are simply guided by the basic value iteration update in equation (10.5) (or (10.3)), which suggests that if we repeat this step often enough, we may eventually learn the right value function for the right policy. Unfortunately, we have only limited guarantees that this is the case when we depend on approximations.

Approximate value iteration embeds a policy approximation loop within an outer loop where policies are updated. Suppose that we fix our policy using

$$A^{\pi,n}(S) = \arg \max_{a \in \mathcal{A}} (C(S, a) + \gamma \bar{V}^{n-1}(S^{M,a}(S, a))). \quad (10.7)$$

Now perform the loop over $m = 1, \dots, M$:

$$\hat{v}^{n,m} = \max_{a \in \mathcal{A}} (C(S^{n,m}, a) + \gamma \bar{V}^{n-1}(S^{M,a}(S^{n,m}, a))),$$

where $S^{n,m+1} = S^M(S^{n,m}, a^{n,m}, W^{n,m+1})$. Note that the value function $\bar{V}^{n-1}(s)$ remains constant within this inner loop. After executing this loop, we take the series of observations $\hat{v}^{n,1}, \dots, \hat{v}^{n,M}$ and use them to update $\bar{V}^{n-1}(s)$ to obtain $\bar{V}^n(s)$. Typically $\bar{V}^n(s)$ does not depend on $\bar{V}^{n-1}(s)$, other than to influence the calculation of $\hat{v}^{n,m}$. If M is large enough, $\bar{V}^n(s)$ will represent an accurate approximation of the value of being in state s while following the policy in equation (10.7). In fact it is specifically because of this ability to approximate a policy that approximate policy iteration is emerging as a powerful algorithmic strategy for approximate dynamic programming. However, the cost of using the inner policy evaluation loop can be significant, and for this reason approximate value iteration and its variants remain popular.

10.2 APPROXIMATE VALUE ITERATION AND Q -LEARNING USING LOOKUP TABLES

Arguably the most natural and elementary approach for approximate dynamic programming uses approximate value iteration. In this section we explore variations of approximate value iteration and Q -learning.

10.2.1 Value Iteration Using a Pre-decision State Variable

Classical value iteration (for a finite-horizon problem) estimates the value of being in a specific state S_t^n using

$$\hat{v}_t^n = \max_{a_t} (C(S_t^n, a_t) + \gamma \mathbb{E}\{\bar{V}_{t+1}^{n-1}(S_{t+1})|S_t^n\}), \quad (10.8)$$

where $S_{t+1} = S^M(S_t^n, x_t, W_{t+1}^n)$, and S_t^n is the state that we are in at time t , iteration n . We assume that we are following a sample path ω^n , where we compute $W_{t+1}^n = W_{t+1}(\omega^n)$. After computing \hat{v}_t^n , we update the value function using the standard equation

$$\bar{V}_t^n(S_t^n) = (1 - \alpha_{n-1})\bar{V}_t^{n-1}(S_t^n) + \alpha_{n-1}\hat{v}_t^n. \quad (10.9)$$

If we sample states at random (rather than following the trajectory) and repeat equations (10.8) and (10.9), we will eventually converge to the correct value of being in each state. Note that we are assuming a finite horizon model, and that we can compute the expectation exactly. When we can compute the expectation exactly, this is very close to classical value iteration, with the only exception that we are not looping over all the states at every iteration.

One reason to use the pre-decision state variable is that for some problems, computing the expectation is easy. For example, W_{t+1} might be a binomial random variable (did a customer arrive, did a component fail), which makes the expectation especially easy. If this is not the case, then we have to approximate the expectation. For example, we might use

$$\hat{v}_t^n = \max_{a_t} \left(C(S_t^n, a_t) + \gamma \sum_{\hat{\omega} \in \hat{\Omega}^n} p^n(\hat{\omega}) \bar{V}_{t+1}^{n-1}(S^M(S_t^n, a_t, W_{t+1}(\hat{\omega}))) \right). \quad (10.10)$$

Either way, using a lookup table representation we can update the value of being in state S_t^n using equation (10.9). Keep in mind that if we can compute an expectation (or if we approximate it using a large sample $\hat{\Omega}$), then the stepsize should be much larger than when we are using a single sample realization (as we did with the post-decision formulation). An outline of the overall algorithm is given by Figure 10.1.

At this point a reasonable question to ask is: Does this algorithm work? The answer is possibly, but not in general. Before we get an algorithm that will work (at least in theory), we need to deal with what is known as the exploration-exploitation problem.

10.2.2 On-policy, Off-policy, and the Exploration–Exploitation Problem

The algorithm in Figure 10.1 uses a kind of default logic for determining the next state to visit. Specifically, we solve the optimization problem in equation (10.10), and from this we not only determine \hat{v}_t^n , which we use to update the value of being in a state, we also determine an action a_t^n . Then, in step 2b of the algorithm, we use this action to help determine the next state to visit using the transition function

$$S_{t+1}^n = S^M(S_t^n, a_t^n, W_{t+1}^n).$$

Step 0. Initialization

Step 0a. Initialize \bar{V}_t^0 , $t \in \mathcal{T}$.

Step 0b. Set $n = 1$.

Step 0c. Initialize S_0^1 .

Step 1. Sample ω^n .

Step 2. Do for $t = 0, 1, \dots, T$:

Step 2a: Choose $\hat{\Omega}^n \subseteq \Omega$ and solve

$$\hat{v}_t^n = \max_{a_t} \left(C_t(S_t^{n-1}, a_t) + \gamma \sum_{\hat{\omega} \in \hat{\Omega}^n} p^n(\hat{\omega}) \bar{V}_{t+1}^{n-1}(S^M(S_t^{n-1}, a_t, W_{t+1}(\hat{\omega}))) \right),$$

and let a_t^n be the value of a_t that solves the maximization problem.

Step 2b: Compute

$$S_{t+1}^n = S^M(S_t^n, a_t^n, W_{t+1}(\omega^n)).$$

Step 2c. Update the value function

$$\bar{V}_t^n \leftarrow U^V(\bar{V}_t^{n-1}, S_t^n, \hat{v}_t^n).$$

Step 3. Increment n . If $n \leq N$, go to step 1.

Step 4. Return the value functions $(\bar{V}_t^n)_{t=1}^T$.

Figure 10.1 Approximate dynamic programming using a pre-decision state variable.

Using the action a_t^n , which is the action determined by the policy we are trying to optimize, means that we are using a concept known as *trajectory following*. The policy that determines the action we would like to take is known in the reinforcement learning community as the *target policy*, but we prefer the alternate term *learning policy*. When we are optimizing policies, what we are doing is trying to improve the learning policy.

We can encounter serious problems if we use the learning policy to determine the next state to visit. Consider the two-state dynamic program illustrated in Figure 10.2. Assume that we start in state 1, and further assume that we initialize the value of being in each of the two states to $\bar{V}^0(1) = \bar{V}^0(2) = 0$. We see a negative contribution of $-\$5$ to move from state 1 to 2, but a contribution of $\$0$ to stay in state 1. We do not see the contribution of $\$20$ to move from state 2 back to state 1, so it appears to be best to stay in state 1.

We need some way to force the system to visit state 2 so that we discover the contribution of $\$20$. One way to do this is to adopt logic that forces the system to *explore* by choosing actions at random. For example, we may flip a coin and choose an action with probability ϵ , or choose the action a_t^n determined by the learning policy with probability $1 - \epsilon$. This policy is known in the literature as *epsilon greedy*.

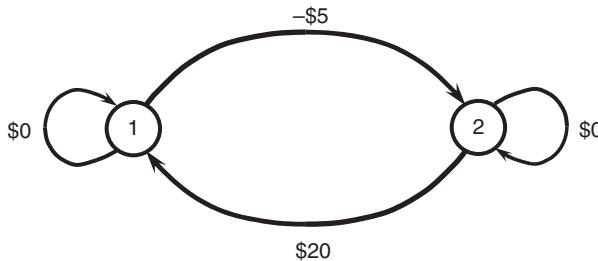


Figure 10.2 Two-state dynamic program, with transition contributions.

The policy that determines which action to use to determine the next state to visit, if it is different than the learning policy, is known as the *behavior policy* or the *sampling policy*. The name “behavior policy” arises when we are modeling a real system such as a human playing a game or a factory assembling components. The behavior policy is, literally, the policy that describes how the system behaves. By contrast, if we are simply designing an algorithm, we feel that the term “sampling policy” more accurately describes the actual function being served by this policy. We also note that while it is common to implement a sampling policy through the choice of action, we may also simply choose a state at random.

If the learning policy also determines the next state we visit, then we say that the algorithm is *on policy*. If the sampling policy is different than the learning policy, then we say that the algorithm is *off policy*, which means that the policy that we use to determine the next state to visit does not follow the policy we are trying to optimize.

In the remainder of this chapter, we are going to distinguish on-policy and off-policy algorithms. However, we defer to Chapter 12 a more complete discussion of the different policies that can be used.

10.2.3 *Q*-Learning

We first introduced *Q*-learning and its cousin SARSA in Chapter 4, and we refer the reader back to this presentation for an introduction to this important algorithmic strategy. For completeness, we present a version of the full algorithm in Figure 10.3 for a time-dependent, finite horizon problem.

10.2.4 Value Iteration Using a Post-decision State Variable

For the many applications that lend themselves to a compact post-decision state variable, it is possible to adapt approximate value iteration to value functions estimated around the post-decision state variable. At the heart of the algorithm we choose actions (and estimate the value of being in state S_t^n) using

$$\hat{v}_t^n = \arg \max_{a_t \in A_t} (C(S_t^n, a_t) + \gamma \bar{V}_t^{n-1}(S_t^{M,a}(S_t^n, a_t))).$$

Step 0. Initialization.

Step 0a. Initialize an approximation for the value function $\bar{Q}_t^0(S_t, a_t)$ for all states S_t and decisions $a_t \in \mathcal{A}_t$, $t = \{0, 1, \dots, T\}$.

Step 0b. Set $n = 1$.

Step 0c. Initialize S_0^n .

Step 1. Choose a sample path ω^n .

Step 2. Do for $t = 0, 1, \dots, T$:

Step 2a: Determine the action using ϵ -greedy. With probability ϵ , choose an action a^n at random from \mathcal{A} . With probability $1 - \epsilon$, choose a^n using

$$a_t^n = \arg \max_{a_t \in \mathcal{A}_t} \bar{Q}_t^{n-1}(S_t^n, a_t).$$

Step 2b. Sample $W_{t+1}^n = W_{t+1}(\omega^n)$ and compute the next state $S_{t+1}^n = S^M(S_t^n, a_t^n, W_{t+1}^n)$.

Step 2c. Compute

$$\hat{q}_t^n = C(S_t^n, a_t^n) + \gamma \max_{a_{t+1} \in \mathcal{A}_{t+1}} \bar{Q}_{t+1}^{n-1}(S_{t+1}^n, a_{t+1}).$$

Step 2d. Update \bar{Q}_t^{n-1} and \bar{V}_t^{n-1} using

$$\bar{Q}_t^n(S_t^n, a_t^n) = (1 - \alpha_{n-1})\bar{Q}_t^{n-1}(S_t^n, a_t^n) + \alpha_{n-1}\hat{q}_t^n.$$

Step 3. Increment n . If $n \leq N$, go to step 1.

Step 4. Return the Q-factors $(\bar{Q}_t^n)_{t=1}^T$.

Figure 10.3 Q -learning algorithm.

The distinguishing feature when we use the post-decision state variable is that the maximization problem is now deterministic. The key step is how we update the value function approximation. Instead of using \hat{v}_t^n to update a pre-decision value function approximation $\bar{V}_t^{n-1}(S_t^n)$, we use \hat{v}_t^n to update a post-decision value function approximation around the *previous* post-decision state $S_{t-1}^{a,n}$. This is done using

$$\bar{V}_{t-1}^n(S_{t-1}^{a,n}) = (1 - \alpha_{n-1})\bar{V}_{t-1}^{n-1}(S_{t-1}^{a,n}) + \alpha_{n-1}\hat{v}_t^n.$$

The post-decision state not only allows us to solve deterministic optimization problems, there are many applications where the post-decision state has either the same dimensionality as the pre-decision state, or, for some applications, a much lower dimensionality. Examples were given in Section 4.6. A complete summary of the algorithm is given in Figure 10.4.

Q -learning shares certain similarities with dynamic programming using a post-decision value function. In particular, both require the solution of a deterministic optimization problem to make a decision. However, Q -learning accomplishes this goal by creating an artificial post-decision state given by the state/action pair (S, a) .

Step 0. Initialization.

Step 0a. Initialize an approximation for the value function $\bar{V}_t^0(S_t^a)$ for all post-decision states S_t^a , $t = \{0, 1, \dots, T\}$.

Step 0b. Set $n = 1$.

Step 0c. Initialize $S_0^{a,1}$.

Step 1. Choose a sample path ω^n .

Step 2. Do for $t = 0, 1, \dots, T$.

Step 2a: Determine the action using ϵ -greedy. With probability ϵ , choose an action a^n at random from \mathcal{A} . With probability $1 - \epsilon$, choose a^n using

$$\hat{v}_t^n = \arg \max_{a_t \in \mathcal{A}_t} (C(S_t^n, a_t) + \gamma \bar{V}_t^{n-1}(S_t^{M,a}(S_t^n, a_t))).$$

Let a_t^n be the action that solves the maximization problem.

Step 2b. Update \bar{V}_{t-1}^{n-1} using

$$\bar{V}_{t-1}^n(S_{t-1}^{a,n}) = (1 - \alpha_{n-1}) \bar{V}_{t-1}^{n-1}(S_{t-1}^{a,n}) + \alpha_{n-1} \hat{v}_t^n.$$

Step 2c. Sample $W_{t+1}^n = W_{t+1}(\omega^n)$ and compute the next state $S_{t+1}^n = S^M(S_t^n, a_t^n, W_{t+1}^n)$.

Step 3. Increment n . If $n \leq N$, go to step 1.

Step 4. Return the value functions $(\bar{V}_t^n)_{t=1}^T$.

Figure 10.4 Approximate value iteration for finite horizon problems using the post-decision state variable.

We then have to learn the value of being in (S, a) , rather than the value of being in state S alone (which is already very hard for most problems).

If we compute the value function approximation $\bar{V}^n(S^a)$ around the post-decision state $S^a = S^{M,a}(S, a)$, we can create Q -factors directly from the contribution function and the post-decision value function using

$$\bar{Q}^n(S, a) = C(S, a) + \gamma \bar{V}_t^n(S^{M,a}(S, a)).$$

Viewed this way, approximate value iteration using value functions estimated around a post-decision state variable is equivalent to Q -learning. However, if the post-decision state is compact, then estimating $\bar{V}(S^a)$ is much easier than estimating $\bar{Q}(S, a)$.

10.2.5 Value Iteration Using a Backward Pass

Classical approximate value iteration, which is equivalent to temporal-difference learning with $\lambda = 0$ (also known as TD(0)), can be implemented using a pure forward pass, which enhances its simplicity. However, there are problems where it is useful to simulate decisions moving forward in time, and then updating value functions moving backward in time. This is also known as temporal-difference

Step 0. Initialization.

Step 0a. Initialize \bar{V}_t^0 , $t \in \mathcal{T}$.

Step 0b. Initialize S_0^1 .

Step 0c. Choose an initial policy $A^{\pi,0}$.

Step 0d. Set $n = 1$.

Step 1. Repeat for $m = 1, 2, \dots, M$.

Step 1. Choose a sample path ω^m .

Step 2. Do for $t = 0, 1, 2, \dots, T$.

Step 2a. Find

$$a_t^{n,m} = A^{\pi,n-1}(S_t^{n,m}).$$

Step 2b. Update the state variable

$$S_{t+1}^{n,m} = S^M(S_t^{n,m}, a_t^{n,m}, W_{t+1}(\omega^m)).$$

Step 3. Set $\hat{v}_{T+1}^{n,m} = 0$, and do for $t = T, T-1, \dots, 1$,

$$\hat{v}_t^{n,m} = C(S_t^{n,m}, a_t^{n,m}) + \gamma \hat{v}_{t+1}^{n,m}.$$

Step 4. Compute the average value from starting in state S_0^1 :

$$\bar{v}_0^n = \frac{1}{M} \sum_{m=1}^M \hat{v}_0^{n,m}.$$

Step 5. Update the value function approximation by using the average values

$$\bar{V}_0^n \leftarrow U^V(\bar{V}_0^{n-1}, S_0^{a,n}, \bar{v}_0^n).$$

Step 6. Update the policy

$$A^{\pi,n}(S) = \arg \max_{a \in \mathcal{A}} (C(S_0^n, a) + \gamma \bar{V}_0^n(S^{M,a}(S_0^n, a))).$$

Step 6. Increment n . If $n \leq N$ go to step 1.

Step 7. Return the value functions $(\bar{V}_t^n)_{t=1}^T$.

Figure 10.5 Double-pass version of the approximate dynamic programming algorithm for a finite horizon problem.

learning with $\lambda = 1$, but we find “backward pass” to be more descriptive. The algorithm is depicted in Figure 10.5.

In this algorithm we step forward through time creating a trajectory of states, actions, and outcomes. We then step backward through time, updating the value of being in a state using information from the same trajectory in the future. We are going to use this algorithm to also illustrate ADP for a time-dependent, finite horizon problem. In addition we are going to illustrate a form of policy evaluation. Pay careful attention to how variables are indexed.

The idea of stepping backward through time to produce an estimate of the value of being in a state was first introduced in the control theory community under the name of *backpropagation through time* (BTT). The result of our backward pass is \hat{v}_t^n , which is the contribution from the sample path ω^n and a particular policy. Our policy is, quite literally, the set of decisions produced by the value function approximation $\bar{V}_t^{n-1}(S_t^a)$. Unlike our forward-pass algorithm (where \hat{v}_t^n depends on the approximation $\bar{V}_t^{n-1}(S_t^a)$), \hat{v}_t^n is a valid, unbiased estimate of the value of being in state S_t^n at time t and following the policy produced by \bar{V}^{n-1} .

We introduce an inner loop so that rather than updating the value function approximation with a single \hat{v}_0^n , we average across a set of samples to create a more stable estimate, \bar{v}_0^n .

These two strategies are easily illustrated using our simple asset-selling problem. For this illustration we are going to slightly simplify the model we provided earlier, where we assumed that the change in price, \hat{p}_t , was the exogenous information. If we use this model, we have to retain the price p_t in our state variable (even the post-decision state variable). For our illustration we are going to assume that the exogenous information is the price itself, so that $p_t = \hat{p}_t$. We further assume that \hat{p}_t is independent of all previous prices (a pretty strong assumption). For this model the pre-decision state is $S_t = (R_t, p_t)$ while the post-decision state variable is simply $S_t^a = R_t^a = R_t - a_t$, which indicates whether we are holding the asset or not. Further $S_{t+1} = S_t^a$, since the resource transition function is deterministic.

With this model, a single-pass algorithm (approximate value iteration) is performed by stepping forward through time, $t = 1, 2, \dots, T$. At time t we first sample \hat{p}_t and we find

$$\hat{v}_t^n = \max_{a_t \in \{0,1\}} (\hat{p}_t^n a_t + (1 - a_t)(-c_t + \bar{v}_t^{n-1})). \quad (10.11)$$

Assume that the holding cost $c_t = 2$ for all time periods.

Table 10.1 illustrates three iterations of a single-pass algorithm for a three-period problem. We initialize $\bar{v}_t^0 = 0$ for $t = 0, 1, 2, 3$. Our first decision is a_1 after we see \hat{p}_1 . The first column shows the iteration counter, while the second shows the stepsize $\alpha_{n-1} = 1/n$. For the first iteration we always choose to sell because $\bar{v}_t^0 = 0$, which means that $\hat{v}_t^1 = \hat{p}_t^1$. Since our stepsize is 1.0, this produces $\bar{v}_{t-1}^1 = \hat{p}_t^1$ for each time period.

Table 10.1 Illustration of a single-pass algorithm

Iteration	α_{n-1}	$t = 0$		$t = 1$				$t = 2$				$t = 3$		
		\bar{v}_0	\hat{v}_1	\hat{p}_1	a_1	\bar{v}_1	\hat{v}_2	\hat{p}_2	a_2	\bar{v}_2	\hat{v}_3	\hat{p}_3	a_3	\bar{v}_3
0		0				0				0				0
1	1	30	30	30	1	34	34	34	1	31	31	31	1	0
2	0.50	31	32	24	0	31.5	29	21	0	29.5	30	30	1	0
3	0.3	32.3	35	35	1	30.2	27.5	24	0	30.7	33	33	1	0

In the second iteration our first decision problem is

$$\begin{aligned}\hat{v}_1^2 &= \max\{\hat{p}_1^2, -c_1 + \bar{v}_1^1\} \\ &= \max\{24, -2 + 34\} \\ &= 32,\end{aligned}$$

which means that $a_1^2 = 0$ (since we are holding the asset). We then use \hat{v}_1^2 to update \bar{v}_0^2 :

$$\begin{aligned}\bar{v}_0^2 &= (1 - \alpha_1)\bar{v}_0^1 + \alpha_1\hat{v}_1^1 \\ &= (0.5)30.0 + (0.5)32.0 \\ &= 31.0\end{aligned}$$

Repeating this logic, we hold again for $t = 2$ but we always sell at $t = 3$, since this is the last time period. In the third pass we again sell in the first time period, but hold for the second time period.

It is important to recognize that this problem is quite simple, and we do not have to deal with exploration issues. If we sell, we are no longer holding the asset, and the forward pass should stop (more precisely, we should continue to simulate the process given that we have sold the asset). Instead, even if we sell the asset, we step forward in time and continue to evaluate the state that we are holding the asset (the value of the state where we are not holding the asset is, of course, zero). Normally we evaluate only the states that we transition to (see step 2b), but for this problem we are actually visiting all the states (since there is in fact only one state that we really need to evaluate).

Now consider a double-pass algorithm. Table 10.2 illustrates the forward pass, followed by the backward pass, where for simplicity we are going to use only a single inner iteration ($M = 1$). Each line of the table only shows the numbers determined during the forward or backward pass. In the first pass, we always sell (since the value of the future is zero), which means that at each time period the value of holding the asset is the price in that period.

In the second pass, it is optimal to hold for two periods until we sell in the last period. The value \hat{v}_t^2 for each time period is the contribution of the rest of the

Table 10.2 Illustration of a double-pass algorithm

Iteration	Pass	$t = 0$		$t = 1$				$t = 2$				$t = 3$		
		\bar{v}_0	\hat{v}_1	\hat{p}_1	a_1	\bar{v}_1	\hat{v}_2	\hat{p}_2	a_2	\bar{v}_2	\hat{v}_3	\hat{p}_3	a_3	\bar{v}_3
0		0				0				0				0
1	Forward	\rightarrow	\rightarrow	30	1	\rightarrow	\rightarrow	34	1	\rightarrow	\rightarrow	31	1	
1	Back	30	30	\leftarrow	\leftarrow	34	34	\leftarrow	\leftarrow	31	31	\leftarrow	\leftarrow	0
2	Forward	\rightarrow	\rightarrow	24	0	\rightarrow	\rightarrow	21	0	\rightarrow	\rightarrow	27	1	
2	Back	26.5	23	\leftarrow	\leftarrow	29.5	25	\leftarrow	\leftarrow	29	27	\leftarrow	\leftarrow	0

trajectory, which in this case is the price we receive in the last time period. So, since $a_1 = a_2 = 0$ followed by $a_3 = 1$, the value of holding the asset at time 3 is the \$27 price we receive for selling in that time period. The value of holding the asset at time $t = 2$ is the holding cost of -2 plus \hat{v}_3^2 , giving $\hat{v}_2^2 = -2 + \hat{v}_3^2 = -2 + 27 = 25$. Similarly, holding the asset at time 1 means that $\hat{v}_1^2 = -2 + \hat{v}_2^2 = -2 + 25 = 23$. The smoothing of \hat{v}_t^n with \bar{v}_{t-1}^{n-1} to produce \bar{v}_{t-1}^n is the same as for the single-pass algorithm.

The value of implementing the double-pass algorithm depends on the problem. For example, imagine that our asset is an expensive piece of replacement equipment for a jet aircraft. We hold the part in inventory until it is needed, which could literally be years for certain parts. This means there could be hundreds of time periods (if each time period is a day) where we are holding the part. Estimating the value of the part now (which would determine whether we order the part to hold in inventory) using a single-pass algorithm could produce extremely slow convergence. A double-pass algorithm would work dramatically better. But if the part is used frequently, staying in inventory for only a few days, then the single-pass algorithm will work fine.

10.2.6 Value Iteration for Multidimensional Decision Vectors

In the previous section we saw that the use of the post-decision state variable meant that the process of choosing the best action required solving a deterministic optimization problem. This opens the door to considering problems where the decision is a vector x_t . For example, imagine that we have a problem of assigning an agent in a multiskill call center to customers requiring help with their computers. An agent of type i might have a particular set of language and technical skills. A customer has answered a series of automated questions, which we capture with a label j . Let

R_{ti} = number of agents available at time t with skill set i ,

D_{tj} = number of customers waiting at time t whose queries are characterized by j .

We let $R_t = (R_{ti})_i$ and $D_t = (D_{tj})_j$, and finally let our state variable be $S_t = (R_t, D_t)$. Let our decision vector be defined using

x_{tij} = number of agents of type i who are assigned to customers of type j at time t ,

$$x_t = (x_{tij})_{i,j}.$$

For realistic problems, it is easy to create vectors x_t with hundreds or thousands of dimensions. Finally, let

c_{ij} = estimated time required for an agent of type i to serve a customer of type j .

The state variables evolve according to

$$R_{t+1,i} = R_{ti} - \sum_j x_{tij} + \hat{R}_{t+1,i},$$

$$D_{t+1,j} = D_{tj} - \sum_i x_{tij} + \hat{D}_{t+1,j}.$$

Here $\hat{R}_{t+1,i}$ represents the number of agents that were busy but that became idle between t and $t+1$ because they completed their previous assignment. $\hat{D}_{t+1,j}$ represents arrival of new customers. We note that R_t and D_t are pre-decision state variables. Their post-decision counterparts are given by

$$R_{t+1,i}^x = R_{ti} - \sum_j x_{tij}, \quad (10.12)$$

$$D_{t+1,j}^x = D_{tj} - \sum_i x_{tij}. \quad (10.13)$$

We are going to have to create a value function approximation $\bar{V}(S_t^x)$. For the purpose of illustrating the basic idea, let us use a separable approximation, which we can write using

$$\bar{V}_t(R_t^x, D_t^x) = \sum_i \bar{V}_{ti}^R(R_{ti}^x) + \sum_j \bar{V}_{tj}^D(D_{tj}^x).$$

Since we are minimizing, we intend to create scalar, convex approximations for $\bar{V}_{ti}^R(R_{ti}^x)$ and $\bar{V}_{tj}^D(D_{tj}^x)$. Now, our VFA policy requires solving a linear (or nonlinear) math programming problem of the form

$$\min_{x_t} \sum_i \sum_j c_{ij} x_{tij} + \sum_i \bar{V}_{t-1,i}^R(R_{ti}^x) + \sum_j \bar{V}_{t-1,j}^D(D_{tj}^x), \quad (10.14)$$

where R_{ti}^x and D_{tj}^x are given by (10.12) and (10.13), respectively. Also our optimization problem has to be solved subject to the constraints

$$\sum_j x_{tij} \leq R_{ti}, \quad (10.15)$$

$$\sum_i x_{tij} \leq D_{tj}, \quad (10.16)$$

$$x_{tij} \geq 0. \quad (10.17)$$

The optimization problem described by equations (10.14) through (10.17) is a linear or nonlinear optimization problem. If the scalar, separable value function approximations are convex, this is generally fairly easy to solve, even when x_t has hundreds or thousands of dimensions.

Elsewhere we would let \hat{v}_t^n be the value of the optimal objective function, which we then use to update the value function approximation. For this problem class we are going to take advantage of the fact that the optimal solution will yield dual variables for the constraints (10.15) and (10.16). Call these dual variables \hat{v}_{ti}^R and \hat{v}_{tj}^D , respectively. Thus we can interpret \hat{v}_{ti}^R as an approximation of the marginal value of R_{ti} , while \hat{v}_{tj}^D is an approximation of the marginal value of D_{tj} . We can use these marginal values to update our value function approximations. However, we would use \hat{v}_t^R and \hat{v}_t^D to update the value function approximations $\bar{V}_{t-1}^R(R_{t-1}^x)$ and $\bar{V}_{t-1}^D(D_{t-1}^x)$, which means that we are using information from the problem we solve at time t to update value function approximations at time $t - 1$ around the previous, post-decision state variable.

This logic is described in much greater detail in Chapters 13 and 14. Our goal here is simply to illustrate the ability to use the post-decision value function to handle optimization problems with high-dimensional decision vectors.

10.3 STATISTICAL BIAS IN THE MAX OPERATOR

A subtle type of bias arises when we are optimizing because we are taking the maximum over a set of random variables. In algorithms such as Q -learning or approximate value iteration, we are computing \hat{q}_t^n by choosing the best of a set of decisions that depend on $\bar{Q}^{t-1}(S, a)$. The problem is that the estimates $\bar{Q}^{t-1}(S, a)$ are random variables. In the best of circumstances, assume that $\bar{Q}^{t-1}(S, a)$ is an unbiased estimate of the true value $V_t(S^a)$ of being in (post-decision) state S^a . Because it is still a statistical estimate with some degree of variation, some of the estimates will be too high while others will be too low. If a particular action takes us to a state where the estimate just happens to be too high (due to statistical variation), then we are more likely to choose this as the best action and use it to compute \hat{q}^n .

To illustrate, let us choose an action $a \in \mathcal{A}$, where $C(S, a)$ is the contribution earned by using decision a (given that we are in state S), which then takes us to state $S^a(S, a)$ where we receive an estimated value $\bar{V}(S^a(S, a))$. Normally we would update the value of being in state S by computing

$$\hat{v}^n = \max_{a \in \mathcal{A}} (C(S, a) + \bar{V}^{n-1}(S^a(S, a))).$$

We would then update the value of being in state S using our standard update formula

$$\bar{V}^n(S) = (1 - \alpha_{n-1})\bar{V}^{n-1}(S) + \alpha_{n-1}\hat{v}^n.$$

Since $\bar{V}^{n-1}(S^a(S, a))$ is a random variable, sometimes it will overestimate the true value of being in state $S^a(S, a)$ while other times it will underestimate the true value. Of course, we are more likely to choose an action that takes us to a state where we have overestimated the value.

We can quantify the error due to statistical bias as follows. Fix the iteration counter n (so that we can ignore it), and let

$$U_a = C(S, a) + \bar{V}(S^a(S, a))$$

be the estimated value of using action a . The statistical error, which we represent as β , is given by

$$\beta = \mathbb{E}\{\max_{a \in \mathcal{A}} U_a\} - \max_{a \in \mathcal{A}} \mathbb{E}U_a. \quad (10.18)$$

The first term on the right-hand side of (10.18) is the expected value of $\bar{V}(S)$, which is computed based on the best observed value. The second term is the correct answer (which we can only find if we know the true mean). We can get an estimate of the difference by using a strategy known as the “plug-in principle.” We assume that $\mathbb{E}U_a = \bar{V}(S^a(S, a))$, which means that we assume that the estimates $\bar{V}(S^a(S, a))$ are correct, and then try to estimate $\mathbb{E}\{\max_{a \in \mathcal{A}} U_a\}$. Thus computing the second term in (10.18) is easy.

The challenge is computing $\mathbb{E}\{\max_{a \in \mathcal{A}} U_a\}$. We assume that while we have been computing $\bar{V}(S^a(S, a))$, we have also been computing $\bar{\sigma}^2(a) = \text{Var}(U_a) = \text{Var}(\bar{V}(S^a(S, a)))$. Using the plug-in principle, we are going to assume that the estimates $\bar{\sigma}^2(a)$ represent the true variances of the value function approximations. Computing $\mathbb{E}\{\max_{a \in \mathcal{A}} U_a\}$ for more than a few decisions is computationally intractable, but we can use a technique called the Clark approximation to provide an estimate. This strategy finds the exact mean and variance of the maximum of two normally distributed random variables, and then assumes that this maximum is also normally distributed. Assume that the decisions can be ordered so that $\mathcal{A} = \{1, 2, \dots, |\mathcal{A}|\}$. Now let

$$\bar{U}_2 = \max\{U_1, U_2\}.$$

We can compute the mean and variance of \bar{U}_2 as follows. First compute

$$\text{Var}(\bar{U}_2) = \sigma_1^2 + \sigma_2^2 - 2\sigma_1\sigma_2\rho_{12},$$

where $\sigma_1^2 = \text{Var}(U_1)$, $\sigma_2^2 = \text{Var}(U_2)$, and ρ_{12} is the correlation coefficient between U_1 and U_2 (we allow the random variables to be correlated, but shortly we are going to approximate them as being independent). Next find

$$z = \frac{\mu_1 - \mu_2}{\sqrt{\text{Var}(\bar{U}_2)}},$$

where $\mu_1 = \mathbb{E}U_1$ and $\mu_2 = \mathbb{E}U_2$. Now let $\Phi(z)$ be the cumulative standard normal distribution (i.e., $\Phi(z) = \mathbb{P}[Z \leq z]$ where Z is normally distributed with mean 0 and variance 1), and let $\phi(z)$ be the standard normal density function. If we assume that U_1 and U_2 are normally distributed (a reasonable assumption when

they represent sample estimates of the value of being in a state), then it is a straightforward exercise to show that

$$\mathbb{E}\bar{U}_2 = \mu_1\Phi(z) + \mu_2\Phi(-z) + \sqrt{\text{Var}(\bar{U}_2)}\phi(z) \quad (10.19)$$

$$\begin{aligned} \text{Var}(\bar{U}_2) &= \left[(\mu_1^2 + \sigma_1^2)\Phi(z) + (\mu_1^2 + \sigma_2^2)\Phi(-z) + (\mu_1 + \mu_2)\sqrt{\text{Var}(\bar{U}_2)}\phi(z) \right] \\ &\quad - (\mathbb{E}\bar{U}_2)^2. \end{aligned} \quad (10.20)$$

Now suppose that we have a third random variable, U_3 , where we wish to find $\mathbb{E} \max\{U_1, U_2, U_3\}$. The Clark approximation solves this by using

$$\begin{aligned} \bar{U}_3 &= \mathbb{E} \max\{U_1, U_2, U_3\} \\ &\approx \mathbb{E} \max\{U_3, \bar{U}_2\}, \end{aligned}$$

where we find that \bar{U}_2 is normally distributed with the mean given by (10.19) and the variance given by (10.20). For our setting, it is unlikely that we would be able to estimate the correlation coefficient ρ_{12} (or ρ_{23}), so we are going to assume that the random estimates are independent. This idea can be repeated for large numbers of decisions by using

$$\begin{aligned} \bar{U}_a &= \mathbb{E} \max\{U_1, U_2, \dots, U_a\} \\ &\approx \mathbb{E} \max\{U_a, \bar{U}_{a-1}\}. \end{aligned}$$

We can apply this repeatedly until we find the mean of $\bar{U}_{|\mathcal{A}|}$, which is an approximation of $\mathbb{E}\{\max_{a \in \mathcal{A}} U_a\}$. We can then use this estimate to compute an estimate of the statistical bias β given by equation (10.18).

Figure 10.6 plots $\beta = \mathbb{E} \max_a U_a - \max_a \mathbb{E} U_a$ as it is being computed for 100 decisions, averaged over 30 sample realizations. The standard deviation of each U_a was fixed at $\sigma = 20$. The plot shows that the error increases steadily until the set \mathcal{A} reaches about 20 or 25 decisions, after which it grows much more slowly. Of course, in an approximate dynamic programming application, each U_a would have its own standard deviation which would tend to decrease as we sample a decision repeatedly (a behavior that the approximation above captures nicely).

This brief analysis suggests that the statistical bias in the max operator can be significant. However, it is highly data dependent. If there is a single dominant decision, then the error will be negligible. The problem only arises when there are many (as in 10 or more) decisions that are competitive, and where the standard deviation of the estimates is not small relative to the differences between the means. Unfortunately, this is likely to be the case in most large-scale applications (if a single decision is dominant, then it suggests that the solution is probably obvious).

The relative magnitudes of value iteration bias over statistical bias will depend on the nature of the problem. If we are using a pure forward pass ($\text{TD}(0)$), and if the value of being in a state at time t reflects rewards earned over many periods into the future, then the value iteration bias can be substantial (especially if the

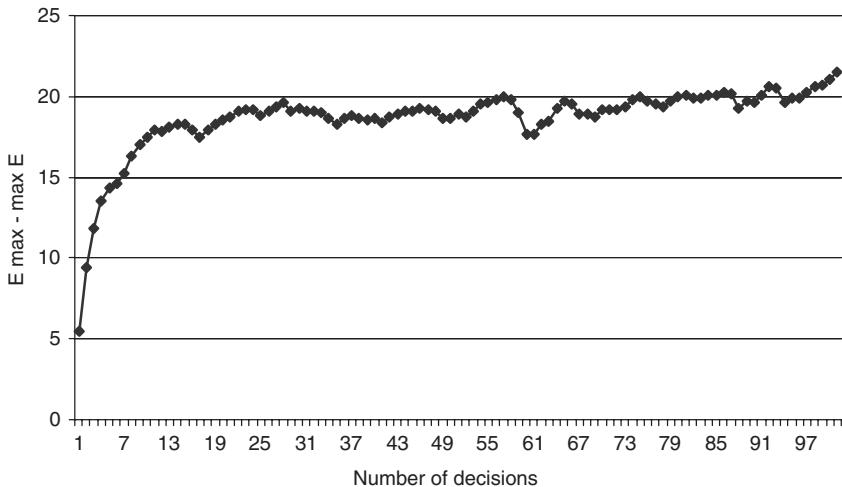


Figure 10.6 $\mathbb{E} \max_a U_a - \max_a \mathbb{E} U_a$ for 100 decisions, averaged over 30 sample realizations. The standard deviation of all sample realizations was 20.

stepsize is too small). Value iteration bias has long been recognized in the dynamic programming community. By contrast, statistical bias appears to have received almost no attention, and as a result we are not aware of any research addressing this problem. We suspect that statistical bias is likely to inflate value function approximations fairly uniformly, which means that the impact on the policy may be small. However, if the goal is to obtain the value function itself (e.g., to estimate the value of an asset or a contract), then the bias can distort the results.

10.4 APPROXIMATE VALUE ITERATION AND Q -LEARNING USING LINEAR MODELS

Approximate value iteration, Q -learning, and temporal-difference learning (with $\lambda = 0$) are clearly the simplest methods for updating an estimate of the value of being in a state. Linear models are the simplest methods for approximating a value function. Not surprisingly, then, there has been considerable interest in putting these two strategies together.

Figure 10.7 depicts a basic adaptation of approximate value iteration using a linear model to approximate the value function. The strategy is popular because it is so easy to implement. As of this writing, the design of provably convergent algorithms based on approximate value iteration using linear models is an active area of research. However, general-purpose algorithms such as the one sketched in Figure 10.7 are not convergent, and may behave badly. This is true even if the true value function can be perfectly represented using a set of basis functions.

The problem with approximate value iteration is that the update of the coefficient vector θ^n can be noisy, and this immediately impacts the policy, which further contributes to the noise. In short, the strategy is fundamentally unstable.

Step 0. Initialization.

Step 0a. Initialize \bar{V}^0 .

Step 0b. Initialize S^1 .

Step 0c. Set $n = 1$.

Step 1. Solve

$$\hat{v}^n = \max_{a \in \mathcal{A}^n} \left(C(S^n, a) + \gamma \sum_f \theta_f^{n-1} \phi_f(S^{M,a}(S^n, a)) \right), \quad (10.21)$$

and let a^n be the value of a that solves (10.21).

Step 2. Update the value function recursively using equations (9.24)–(9.28) from chapter 9 to obtain θ^n .

Step 3. Choose a sample $W^{n+1} = W(\omega^{n+1})$, and determine the next state using some policy such as

$$S^n = S^M(S^n, a^n, W^{n+1}).$$

Step 3. Increment n . If $n \leq N$, go to step 1.

Step 4. Return the value functions \bar{V}^N .

Figure 10.7 Approximate value iteration using a linear model.

It is possible to design algorithms using approximate value iteration that are provably convergent, but these require special structure. For example, Chapters 13 and 14 describe the use of approximate value iteration in the context of resource allocation problems, where we can exploit the property of concavity. This is an incredibly powerful property and allows us to obtain algorithms that are both practical and, under certain conditions, provably convergent. However, without these properties, there are few guarantees. It is for this reason that the research community has focused more on approximate policy iteration (see Section 10.5 below).

Despite the potentially poor performance of this algorithm, it remains a popular strategy because of its simplicity. In addition, while it may not work, it *might* work quite well! The important point here is that it is a strategy that may be worth trying, but caution needs to be used. Given these observations, this section is aimed at providing some guidance to improve the chances that the algorithm will work.

The most important step whenever a linear model is used, regardless of the setting, is to choose the basis functions carefully so that the linear model has a chance of representing the true value function accurately. The biggest strength of a linear model is also its biggest weakness. A large error can distort the update of θ^n , which then impacts the accuracy of the entire approximation. Since the value function approximation determines the policy (see step 1), a poor approximation leads to poor policies, which then distorts the observations \hat{v}^n . This can be a vicious circle from which the algorithm may never recover.

A second step is in the specific choice of recursive least squares updating. Figure 10.7 refers to the classic recursive least squares updating formulas in equations (9.24) through (9.28). However, buried in these formulas is the implicit

use of a stepsize rule of $1/n$. We show in Chapter 11 that a stepsize $1/n$ is particularly bad for approximate value iteration (as well as Q -learning and TD(0) learning). While this stepsize can work well (indeed it is optimal) for stationary data, it is very poorly suited for the backward learning that arises in approximate value iteration. Fortunately, the problem is easily fixed if we replace the updating equations for B^n and γ , which are given as

$$\begin{aligned} B^n &= B^{n-1} - \frac{1}{\gamma^n} (B^{n-1} \phi^n (\phi^n)^T B^{n-1}), \\ \gamma^n &= 1 + (\phi^n)^T B^{n-1} \phi^n, \end{aligned}$$

in equations (9.27) and (9.28) with

$$\begin{aligned} B^n &= \frac{1}{\lambda} \left(B^{n-1} - \frac{1}{\gamma^n} (B^{n-1} \phi^n (\phi^n)^T B^{n-1}) \right), \\ \gamma^n &= \lambda + (\phi^n)^T B^{n-1} \phi^n, \end{aligned}$$

in equations (9.30) and (9.31). Here λ discounts older errors. $\lambda = 1$ produces the original recursive formulas. When used with approximate value iteration, it is important to use $\lambda < 1$. In Section 9.3.2 we argued that if you choose a stepsize rule for α_n such as $\alpha_n = a/(a + n - 1)$, you should set λ_n at iteration n using

$$\lambda_n = \alpha_{n-1} \left(\frac{1 - \alpha_n}{\alpha_n} \right).$$

The last issue that needs special care is the rule for determining the next state to visit. As we discussed in Section 9.4, temporal-difference learning (TD(0)) is approximate value iteration for a fixed policy) is only guaranteed to converge while using on-policy learning, and can diverge when using off-policy learning. Off-policy learning is a cornerstone of convergence proofs when using lookup tables, but it appears to cause considerable problems when using a linear model (it gets worse if our model is nonlinear in the parameters). At the same time exploration should not be needed if our basis functions are carefully chosen (for a fixed policy). Interestingly it was recently shown that approximate value iteration may converge if we start with a policy that is close enough to the optimal policy. However, this condition is relaxed as the discount factor is decreased, suggesting that a good strategy is to start with a smaller discount factor, find a good policy for that discount factor, and then increase the discount factor toward the desired level.

10.5 APPROXIMATE POLICY ITERATION

One of the most important tools in the toolbox for approximate dynamic programming is approximate policy iteration. This algorithm is neither simpler nor more elegant than approximate value iteration, but it can offer convergence guarantees while using linear models to approximate the value function.

In this section we review several flavors of approximate policy iteration, including

1. Finite horizon problems using lookup tables.
2. Finite horizon problems using basis functions.
3. Infinite horizon problems using basis functions.

Finite horizon problems allow us to obtain Monte Carlo estimates of the value of a policy by simulating the policy until the end of the horizon. Note that a “policy” here always refers to decisions that are determined by value function approximations. We use the finite horizon setting to illustrate approximating value function approximations as well as lookup tables and basis functions, which allows us to highlight the strengths and weaknesses of the transition to basis functions.

We then present an algorithm based on least squares temporal differences (LSTD) and contrast the steps required for finite horizon and infinite horizon problems when using basis functions.

10.5.1 Finite Horizon Problems Using Lookup Tables

A fairly general-purpose version of an approximate policy iteration algorithm is given in Figure 10.8 for an infinite horizon problem. This algorithm helps to illustrate the choices that can be made when designing a policy iteration algorithm in an approximate setting. The algorithm features three nested loops. The innermost loop steps forward and backward in time from an initial state $S^{n,0}$. The purpose of this loop is to obtain an estimate of the value of a path. Normally we would choose T large enough so that γ^T is quite small (thereby approximating an infinite path). The next outer loop repeats this process M times to obtain a statistically reliable estimate of the value of a policy (determined by $\bar{V}^{\pi,n}$). The third loop, representing the outer loop, performs policy updates (in the form of updating the value function). In a more practical implementation we might choose states at random rather than looping over all states.

Readers should note that we have tried to index variables in a way that shows how they are changing (do they change with outer iteration n ? inner iteration m ? the forward look-ahead counter t ?). This does not mean that it is necessary to store, for example, each state or decision for every n , m , and t . In an actual implementation the software should be designed to store only what is necessary.

We can create different variations of approximate policy iteration by our choice of parameters. First, if we let $T \rightarrow \infty$, then we are evaluating a true infinite horizon policy. If we simultaneously let $M \rightarrow \infty$, then \hat{v}^n approaches the exact, infinite horizon value of the policy π determined by $\bar{V}^{\pi,n}$. Thus, for $M = T = \infty$, we have a Monte Carlo-based version of exact policy iteration.

We can choose a finite value of T that produces values $\hat{v}^{n,m}$ close to the infinite horizon results. We can also choose finite values of M , including $M = 1$. When we use finite values of M , this means that we are updating the policy before we have fully evaluated the policy. This variant is known in the literature as *optimistic*

Step 0. Initialization.

Step 0a. Initialize $\bar{V}^{\pi,0}$.

Step 0b. Set a lookahead parameter T and inner iteration counter M .

Step 0c. Set $n = 1$.

Step 1. Sample a state S_0^n and then do:

Step 2. Do for $m = 1, 2, \dots, M$:

Step 3. Choose a sample path ω^m (a sample realization over the lookahead horizon T).

Step 4. Do for $t = 0, 1, \dots, T$:

Step 4a. Compute

$$a_t^{n,m} = \arg \max_{a_t \in \mathcal{A}_t^{n,m}} (C(S_t^{n,m}, a_t) + \gamma \bar{V}^{\pi,n-1}(S_t^{n,m}, a_t)).$$

Step 4b. Compute

$$S_{t+1}^{n,m} = S^M(S_t^{n,m}, a_t^{n,m}, W_{t+1}(\omega^m)).$$

Step 5. Initialize $\hat{v}_{T+1}^{n,m} = 0$.

Step 6. Do for $t = T, T-1, \dots, 0$:

Step 6a. Accumulate $\hat{v}^{n,m}$:

$$\hat{v}_t^{n,m} = C(S_t^{n,m}, a_t^{n,m}) + \gamma \hat{v}_{t+1}^{n,m}.$$

Step 6b. Update the approximate value of the policy:

$$\bar{v}^{n,m} = \left(\frac{m-1}{m} \right) \bar{v}^{n,m-1} + \frac{1}{m} \hat{v}_0^{n,m}.$$

Step 8. Update the value function at S^n :

$$\bar{V}^{\pi,n}(S^n) = (1 - \alpha_{n-1}) \bar{V}^{\pi,n-1}(S^n) + \alpha_{n-1} \bar{v}_0^{n,M}.$$

Step 9. Set $n = n + 1$. If $n < N$, go to step 1.

Step 10. Return the value functions $(\bar{V}^{\pi,N})$.

Figure 10.8 Policy iteration algorithm for infinite horizon problems.

policy iteration because rather than wait until we have a true estimate of the value of the policy, we update the policy after each sample (presumably, although not necessarily, producing a better policy). We may also think of this as a form of partial policy evaluation, not unlike the hybrid value–policy iteration described in Section 3.6.

10.5.2 Finite Horizon Problems Using Basis Functions

The simplest demonstration of approximate policy iteration using basis functions is in the setting of a finite horizon problem. Figure 10.9 provides an adaption of

Step 0. Initialization.

Step 0a. Fix the basis functions $\phi_f(s)$.

Step 0b. Initialize $\theta_{tf}^{\pi,0}$ for all t . This determines the policy we simulate in the inner loop.

Step 0c. Set $n = 1$.

Step 1. Sample an initial starting state S_0^n :

Step 2. Initialize $\theta^{n,0}$ (if $n > 1$, use $\theta^{n,0} = \theta^{n-1}$), which is used to estimate the value of policy π produced by $\theta^{\pi,n}$. $\theta^{n,0}$ is used to approximate the value of following policy π determined by $\theta^{\pi,n}$.

Step 3. Do for $m = 1, 2, \dots, M$:

Step 4. Choose a sample path ω^m .

Step 5. Do for $t = 0, 1, \dots, T$:

Step 5a. Compute

$$a_t^{n,m} = \arg \max_{a_t \in \mathcal{A}_t^{n,m}} \left(C(S_t^{n,m}, a_t) + \gamma \sum_f \theta_{tf}^{\pi,n-1} \phi_f(S_t^{n,m}, a_t) \right).$$

Step 5b. Compute

$$S_{t+1}^{n,m} = S^M(S_t^{n,m}, a_t^{n,m}, W_{t+1}(\omega^m)).$$

Step 6. Initialize $\hat{v}_{T+1}^{n,m} = 0$.

Step 7. Do for $t = T, T-1, \dots, 0$:

$$\hat{v}_t^{n,m} = C(S_t^{n,m}, a_t^{n,m}) + \gamma \hat{v}_{t+1}^{n,m}.$$

Step 8. Update $\theta_t^{n,m-1}$ using recursive least squares to obtain $\theta_t^{n,m}$ (see Section 9.3).

Step 9. Set $n = n + 1$. If $n < N$, go to step 1.

Step 10. Return the regression coefficients θ_t^N for all t .

Figure 10.9 Policy iteration algorithm for finite horizon problems using basis functions.

the algorithm with lookup tables for when we are using basis functions. There is an outer loop over n where we fix the policy using

$$A_t^\pi(S_t) = \arg \max_a \left(C(S_t, a) + \gamma \sum_f \theta_{tf}^{\pi,n} \phi_f(S_t^{n,m}) \right). \quad (10.22)$$

We are assuming that the basis functions are not themselves time-dependent, although they depend on the post-decision state variable S_t^a , which, of course, is time dependent. The policy is determined by the parameters $\theta_{tf}^{\pi,n}$.

We update the policy $A_t^\pi(s)$ by performing repeated simulations of the policy in an inner loop that runs $m = 1, \dots, M$. Within this inner loop we use recursive least squares to update a parameter vector $\theta_{tf}^{n,m}$. This step replaces step 6b in Figure 10.8.

If we let $M \rightarrow \infty$, then the parameter vector $\theta_t^{n,M}$ approaches the best possible fit for the policy $A_t^\pi(s)$ determined by $\theta^{\pi,n-1}$. However, it is important to recognize that this is not equivalent to performing a perfect evaluation of a policy using a lookup table representation. The problem is that (for discrete states) lookup tables have the *potential* for perfectly approximating a policy, whereas this is not generally true when we use basis functions. If we have a poor choice of basis functions, we may be able find the best possible value of $\theta^{n,m}$ as m goes to infinity, but we may still have a terrible approximation of the policy produced by $\theta^{\pi,n-1}$.

10.5.3 LSTD for Infinite Horizon Problems Using Basis Functions

We have built the foundation for approximate policy iteration using lookup tables and basis functions for finite horizon problems. We now make the transition to infinite horizon problems using basis functions, where we introduce the dimension of projecting contributions over an infinite horizon. There are several ways of accomplishing this (see Section 9.1.2). We use least squares temporal differencing, since it represents the most natural extension of classical policy iteration for infinite horizon problems.

To begin, we let a sample realization of a one-period contribution, given state S^m , action a^m and random information W^{m+1} be given by

$$\hat{C}^m = C(S^m, a^m, W^{m+1}).$$

As in the past we let $\phi^m = \phi(S^m)$ be the column vector of basis functions evaluated at state S^m . We next fix a policy that chooses actions greedily based on a value function approximation given by $\bar{V}^n(s) = \sum_f \theta_f^n \phi_f(s)$ (see equation (10.22)). Imagine that we have simulated this policy over a set of iterations $i = (0, 1, \dots, m)$, giving us a sequence of contributions \hat{C}^i , $i = 1, \dots, m$. Drawing on the foundation provided in Section 9.5, we can use standard linear regression to estimate θ^m as

$$\theta^m = \left[\frac{1}{1+m} \sum_{i=0}^m \phi^i (\phi^i - \gamma \phi^{i+1})^T \right]^{-1} \left[\frac{1}{1+m} \sum_{i=1}^m \phi^i \hat{C}^i \phi^i \right]. \quad (10.23)$$

As a reminder, the term $\phi^i (\phi^i)^T - \gamma \phi^i (\phi^{i+1})^T$ can be viewed as a simulated, sample realization of $I - \gamma P^\pi$, projected onto the feature space. Just as we would use $(I - \gamma P^\pi)^{-1}$ in our basic policy iteration to project the infinite-horizon value of a policy π (for a review, see Section 3.5) we are using the term

$$\left[\frac{1}{1+m} \sum_{i=0}^m \phi^i (\phi^i - \gamma \phi^{i+1})^T \right]^{-1}$$

to produce an infinite horizon estimate of the feature-projected contribution

$$\left[\frac{1}{1+m} \sum_{i=1}^m \phi^i \hat{C}^i \phi^i \right].$$

Equation (10.23) requires solving a matrix inverse for every observation. It is much more efficient to use recursive least squares, which is done by using

$$\begin{aligned}\epsilon^m &= \hat{C}^m - (\phi^m - \gamma\phi^{m+1})^T\theta^{m-1}, \\ B^m &= B^{m-1} - \frac{B^m\phi^m(\phi^m - \gamma\phi^{m+1})^T B^{m-1}}{1 + (\phi^m - \gamma\phi^{m+1})^T B^{m-1}\phi^m}, \\ \theta^m &= \theta^{m-1} + \frac{\epsilon^m B^{m-1}\phi^m}{1 + (\phi^m - \gamma\phi^{m+1})^T B^{m-1}\phi^m}.\end{aligned}$$

Figure 10.10 provides a detailed summary of the complete algorithm. The algorithm has some nice properties if we are willing to assume that there is a vector θ^* such that the true value function $V(s) = \sum_{f \in \mathcal{F}} \theta_f^* \phi_f(s)$ (admittedly, a pretty strong assumption). First, if the inner iteration limit M increases as a function of n so that the quality of the approximation of the policy gets better and better, then the

Step 0. Initialization.

Step 0a. Initialize θ^0 .

Step 0b. Set the initial policy:

$$A^\pi(s|\theta^0) = \arg \max_{a \in \mathcal{A}} (C(s, a) + \gamma\phi(S^M(s, a))^T\theta^0).$$

Step 0c. Set $n = 1$.

Step 1. Do for $n = 1, \dots, N$.

Step 2. Initialize $S^{n,0}$.

Step 3. Do for $m = 0, 1, \dots, M$:

Step 4. Initialize $\theta^{n,m}$.

Step 5. Sample W^{m+1} .

Step 6. Do the following:

Step 6a. Computing the action $a^{n,m} = A^\pi(S^{n,m}|\theta^{n-1})$.

Step 6b. Compute the post-decision state $S^{n,m} = S^{M,a}(S^{n,m}, a^{n,m})$.

Step 6c. Compute the next pre-decision state $S^{n,m+1} = S^M(S^{n,m}, a^{n,m}, W^{m+1})$.

Step 6d. Compute the input variable $\phi(S^{n,m}) - \gamma\phi(S^{n,m+1})$ for equation (10.23).

Step 7: Do the following:

Step 7a. Compute the response variable $\hat{C}^m = C(S^{n,m}, a^{n,m}, W^{m+1})$.

Step 7b. Compute $\theta^{n,m}$ using equation (10.23).

Step 8. Update θ^n and the policy:

$$\theta^{n+1} = \theta^{n,M}$$

$$A^{\pi,n+1}(s|\theta^{n+1}) = \arg \max_{a \in \mathcal{A}} (C(s, a) + \gamma\phi(S^M(s, a))^T\theta^{n+1}).$$

Step 9. Return the $A^\pi(s|\theta^N)$ and parameter θ^N .

Figure 10.10 Approximate policy iteration for infinite horizon problems using least squares temporal differencing.

overall algorithm will converge to the true optimal policy. Of course, this means letting $M \rightarrow \infty$, but from a practical perspective, it means that the algorithm can find a policy arbitrarily close to the optimal policy.

Second, the algorithm can be used with vector-valued and continuous actions (we normally use the notation x_t in this case). There are several features of the algorithm that allow this. First, computing the policy $A^\pi(s|\theta^n)$ requires solving a deterministic optimization problem. If we are using discrete actions, it means simply enumerating the actions and choosing the best one. If we have continuous actions, we need to solve a nonlinear programming problem. The only practical issue is that we may not be able to guarantee that the objective function is concave (or convex if we are minimizing). Second, note that we are using trajectory following (also known as on-policy training) in step 6c, without an explicit exploration step. The algorithm does not require exploration, but it does require that we be able to solve the recursive least squares equations.

We can avoid exploration as long as there is enough variation in the states we visit that allows us to compute θ^m in equation (10.23). When we use lookup tables, we require exploration to guarantee that we eventually will visit every state infinitely often. When we use basis functions, we only need to visit states with sufficient diversity that we can estimate the parameter vector θ^m . In the language of statistics, the issue is one of *identification* (i.e., the ability to estimate θ) rather than exploration. This is a much easier requirement to satisfy, and one of the major advantages of parametric models.

10.6 THE ACTOR–CRITIC PARADIGM

It is very popular in some communities to view approximate dynamic programming in terms of an “actor” and a “critic.” In this setting a decision function that chooses a decision given the state is known as an actor. The process that determines the contribution (cost or reward) from a decision is known as the critic, from which we can compute a value function. The interaction of making decisions and updating the value function is referred to as an actor–critic framework. The slight change in vocabulary brings out the observation that the techniques of approximate dynamic programming closely mimic human behavior. This is especially true when we drop any notion of costs or contributions and simply work in terms of succeeding (or winning) and failing (or losing).

The policy iteration algorithm in Figure 10.11 provides one illustration of the actor-critic paradigm. The decision function is equation (10.24), where $V^{\pi,n-1}$ determines the policy (in this case). This is the actor. Equation (10.25), where we update our estimate of the value of the policy, is the critic. We fix the actor for a period of time and perform repeated iterations where we try to estimate value functions given a particular actor (policy). From time to time, we stop and use our value function to modify our behavior (something critics like to do). In this case we update the behavior by replacing V^π with our current \bar{V} .

In other settings the policy is a rule or function that does not directly use a value function (e.g., V^π or \bar{V}). For example, if we are driving through a transportation

Step 0. Initialization.

Step 0a. Initialize $V_t^{\pi,0}$, $t \in \mathcal{T}$.

Step 0b. Set $n = 1$.

Step 0c. Initialize S_0^1 .

Step 1. Do for $n = 1, 2, \dots, N$:

Step 2. Do for $m = 1, 2, \dots, M$:

Step 3. Choose a sample path ω^m .

Step 4. Initialize $\hat{v}^m = 0$

Step 5. Do for $t = 0, 1, \dots, T$:

Step 5a. Solve:

$$a_t^{n,m} = \arg \max_{a_t \in \mathcal{A}_t^n} (C_t(S_t^{n,m}, a_t) + \gamma V_t^{\pi,n-1}(S_t^{M,a}(S_t^{n,m}, a_t))). \quad (10.24)$$

Step 5b. Compute:

$$S_{t+1}^{n,m} = S^M(S_t^{a,n,m}, a^{n,m}, W_{t+1}(\omega^m)).$$

Step 6. Do for $t = T, \dots, 0$:

Step 6a. Accumulate the path cost (with $\hat{v}_{T+1}^m = 0$):

$$\hat{v}_t^m = C_t(S_t^{n,m}, a_t^m) + \gamma \hat{v}_{t+1}^m.$$

Step 6b. Update approximate value of the policy for $t > 0$:

$$\bar{V}_{t-1}^{n,m} \leftarrow U^V(\bar{V}_{t-1}^{n,m-1}, S_{t-1}^{a,n,m}, \hat{v}_t^m), \quad (10.25)$$

where we typically use $\alpha_{m-1} = 1/m$.

Step 7. Update the policy value function

$$V_t^{\pi,n}(S_t^a) = \bar{V}_t^{n,M}(S_t^a) \quad \forall t = 0, 1, \dots, T.$$

Step 8. Return the value functions $(V_t^{\pi,N})_{t=0}^T$.

Figure 10.11 Approximate policy iteration using value function-based policies.

network (or traversing a graph), the policy might be of the form “when at node i , go next to node j .” As we update the value function, we may decide the right policy at node i is to traverse to node k . Once we have updated our policy, the policy itself does not directly depend on a value function.

Another example might arise when determining how much of a resource we should have on hand. We might solve the problem by maximizing a function of the form $f(x) = \beta_0 - \beta_1(x - \beta_2)^2$. Of course, β_0 does not affect the optimal quantity. We might use the value function to update β_0 and β_1 . Once these are determined, we have a function that does not itself directly depend on a value function.

10.7 POLICY GRADIENT METHODS

Perhaps the cleanest illustration of the actor–critic framework arises when we parameterize both the value of being in a state as well as the policy. We use a standard strategy from the literature which uses Q -factors, and where the goal is to maximize the *average* contribution per time period (see Section 3.7 for a brief introduction using the classical derivation based on transition matrices). Our presentation here represents only a streamlined sketch of an idea that is simple in principle but which involves some fairly advanced principles. We assume that the Q -factors are parameterized using

$$\bar{Q}(s, a|\theta) = \sum_f \theta_f \phi_f(s, a).$$

The policy is represented using a function such as

$$A^\pi(s|\eta, \theta) = \frac{e^{\eta Q(s,a)}}{\sum_{a'} e^{\eta Q(s,a')}}.$$

This choice of policy has the important feature that the probability that an action is chosen is greater than zero. Also $A^\pi(s|\eta, \theta)$ is differentiable in the policy parameter vector η .

In the language of actor–critic algorithms, $\bar{Q}(s, a|\theta)$ is an approximation of the critic parameterized by θ , while $A^\pi(s|\eta, \theta)$ is an approximate policy parameterized by η . We can update θ and η using standard stochastic gradient methods. We begin by defining

$$\begin{aligned} \psi_\theta(s, a) &= \nabla_\theta \ln A^\pi(s|\eta, \theta) \\ &= \frac{\nabla_\theta A^\pi(s|\eta, \theta)}{A^\pi(s|\eta, \theta)}. \end{aligned}$$

Since we are maximizing the average reward per time period, we begin by estimating the average reward per time period using

$$\bar{c}^{n+1} = (1 - \alpha_n) \bar{c}^n + \alpha_n C(S^{n+1}, a^{n+1}).$$

We then compute the temporal difference in terms of the difference between the contribution of a state-action pair, and the average contribution, using

$$\delta^n = C(S^n, a^n) - \bar{c}^n + (\theta^n)^T \phi_{\theta^n}(S^{n+1}, a^{n+1}) - (\theta^n)^T \phi_{\theta^n}(S^n, a^n).$$

Say we are using a $TD(\lambda)$ updating procedure where we assume $0 < \lambda < 1$. We compute the eligibility trace using

$$Z^{n+1} = \lambda Z^n + \phi_{\theta^n}(S^{n+1}, a^{n+1}).$$

We can now present the updating equations for the actor (the policy) and the critic (Q -factors) in a simple and compact way. The actor update is given by

$$\eta^{n+1} = \eta^n - \beta^n \Gamma(\theta^n)(\theta^n)^T \phi_{\theta^n}(S^{n+1}, a^{n+1}) \psi_{\theta^n}(S^{n+1}, a^{n+1}), \quad (10.26)$$

where $\Gamma(\theta^n)$ is used to scale the stepsize β^n in a way that depends on the parameter estimates θ^n . The critic update is given by

$$\theta^{n+1} = \theta^n + \alpha_n \delta^n Z^n. \quad (10.27)$$

Equations (10.26) and (10.27) provide an elegant and compact illustration of an actor–critic updating equation, where both the value function and the policy are approximated using parametric models.

10.8 THE LINEAR PROGRAMMING METHOD USING BASIS FUNCTIONS

In Section 3.8 we showed that the determination of the value of being in each state can be found by solving the following linear program:

$$\min_v \sum_{s \in \mathcal{S}} \beta_s v(s) \quad (10.28)$$

subject to

$$v(s) \geq C(s, x) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, x) v(s') \quad \text{for all } s \text{ and } x. \quad (10.29)$$

The problem with this formulation arises because it requires that we enumerate the state space to create the value function vector $(v(s))_{s \in \mathcal{S}}$. Furthermore we have a constraint for each state-action pair, a set that will be huge even for relatively small problems.

We can partially solve this problem by replacing the discrete value function with a regression function such as

$$\bar{V}(s|\theta) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(s),$$

where $(\phi_f)_{f \in \mathcal{F}}$ is an appropriately designed set of basis functions. This produces a revised linear programming formulation

$$\min_{\theta} \sum_{s \in \mathcal{S}} \beta_s \sum_{f \in \mathcal{F}} \theta_f \phi_f(s)$$

subject to

$$v(s) \geq C(s, x) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, x) \sum_{f \in \mathcal{F}} \theta_f \phi_f(s') \quad \text{for all } s \text{ and } x.$$

This is still a linear program, but now the decision variables are $(\theta_f)_{f \in \mathcal{F}}$ instead of $(v(s))_{s \in \mathcal{S}}$. Notice that rather than use a stochastic iterative algorithm, we obtain θ directly by solving the linear program.

We still have a problem with a huge number of constraints. Since we no longer have to determine $|\mathcal{S}|$ decision variables (in (10.28)–(10.29) the parameter vector $(v(s))_{s \in \mathcal{S}}$ represents our decision variables), it is not surprising that we do not actually need all the constraints. One strategy that has been proposed is to simply choose a random sample of states and actions. Given a state space \mathcal{S} and set of actions (decisions) \mathcal{X} , we can randomly choose states and actions to create a smaller set of constraints.

Some care needs to be exercised when generating this sample. In particular, it is important to generate states roughly in proportion to the probability that they will actually be visited. Then, for each state that is generated, we need to randomly sample one or more actions. The best strategy for doing this is going to be problem-dependent.

This technique has been applied to the problem of managing a network of queues. Figure 10.12 shows a queueing network with three servers and eight queues. A server can serve only one queue at a time. For example, server A might be a machine that paints components one of three colors (e.g., red, green, and blue). It is best to paint a series of parts red before switching over to blue. There are customers arriving exogenously (denoted by the arrival rates λ_1 and λ_2). Other customers arrive from other queues (e.g., departures from queue 1 become arrivals to queue 2). The problem is to determine which queue a server should handle after each service completion.

If we assume that customers arrive according to a Poisson process and that all servers have negative exponential service times (which means that all processes are memoryless), then the state of the system is given by

$$S_t = R_t = (R_{ti})_{i=1}^8,$$

where R_{ti} is the number of customers in queue i . Let $\mathcal{K} = \{1, 2, 3\}$ be our set of servers, and let a_t be the attribute vector of a server given by $a_t = (k, q_t)$, where k is the identity of the server and q_t is the queue being served at time t . Each server can only serve a subset of queues (as shown in Figure 10.12). Let $\mathcal{D} = \{1, 2, \dots, 8\}$ represent a decision to serve a particular queue, and let \mathcal{D}_a be the decisions that can be used for a server with attribute a . Finally, let $x_{tad} = 1$ if we decide to assign a server with attribute a to serve queue $d \in \mathcal{D}_a$.

The state space is effectively infinite (i.e., too large to enumerate). But we can still sample states at random. Research has shown that it is important to sample

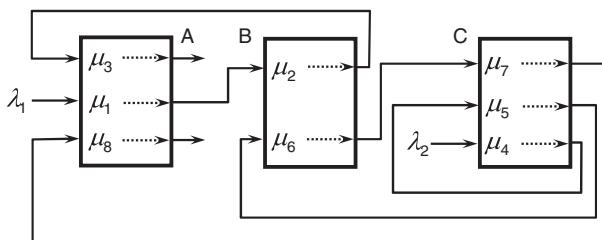


Figure 10.12 Queueing network with three servers serving a total of eight queues, two with exogenous arrivals (λ) and six with arrivals from other queues (from de Farias and Van Roy, 2003).

Table 10.3 Average cost estimated using simulation

Policy	Cost
ADP	33.37
Longest	45.04
FIFO	45.71

Source: From de Farias and Van Roy (2003).

states roughly in proportion to the probability they are visited. We do not know the probability a state will be visited, but it is known that the probability of having a queue with r customers (when there are Poisson arrivals and negative exponential servers) follows a geometric distribution. For this reason we have chosen to sample a state with $r = \sum_i R_{ti}$ customers with probability $(1 - \gamma)\gamma^r$, where γ is a discount factor (a value of 0.95 was used).

Further complicating this problem class is that we also have to sample actions. Let \mathcal{X} be the set of all feasible values of the decision vector x . The number of possible decisions for each server is equal to the number of queues it serves, so the total number of values for the vector x is $3 \times 2 \times 3 = 18$. In the experiments for this illustration, only 5000 states were sampled (in portion to $(1 - \gamma)\gamma^r$) but all the actions were sampled for each state, producing 90,000 constraints.

Once the value function is approximated, it is possible to simulate the policy produced by this value function approximation. The results were compared against two myopic policies: serving the longest queue, and first-in–first-out (i.e., serve the customer who had arrived first). The costs produced by each policy are given in Table 10.3, showing that the ADP-based strategy significantly outperforms these other policies.

Considerably more numerical work is needed to test this strategy on more realistic systems. For example, for systems that do not exhibit Poisson arrivals or negative exponential service times, it is still possible that sampling states based on geometric distributions may work quite well. More problematic is the rapid growth in the feasible region \mathcal{X} as the number of servers, and queues per server, increases.

An alternative to using constraint sampling is an advanced technique known as column generation. Instead of generating a full linear program that enumerates all decisions (i.e., $v(s)$ for each state), and all constraints (equation (10.29)), it is possible to generate sequences of larger and larger linear programs, adding rows (constraints) and columns (decisions) as needed. These techniques are beyond the scope of our presentation, but readers need to be aware of the range of techniques available for this problem class.

10.9 APPROXIMATE POLICY ITERATION USING KERNEL REGRESSION*

We build on the foundation provided in Section 9.8 that describes the use of kernel regression in the context of least squares temporal difference (LSTD) learning. As

we have done earlier, we let the one-period contribution be given by

$$\hat{C}^m = C(S^{n,m}, a^{n,m}, W^{m+1}).$$

Let $S^{a,i}, i = 1, \dots, m$ be the sample-path of post-decision states produced by following a policy. Let $k(S^{a,i}, S^{a,j})$ be the normalized kernel function given by

$$k(S^{a,i}, S^{a,j}) = \frac{K_h(S^{a,i}, S^{a,j})}{\sum_{i=0}^{m-1} K_h(S^{a,i}, S^{a,j})},$$

which means that $\sum_{i=0}^{m-1} k(S^{a,i}, S^{a,j}) = 1$. Then let $P^{\pi,n}$ be a $M \times M$ matrix where the (i, j) th entry is given by

$$P_{i,j}^{\pi,n} = k(S^{a,i-1}, S^{a,j}).$$

By construction, $P^{\pi,n}$ is a stochastic matrix (its rows sum to 1), which means that $I - \gamma P^{\pi,n}$ is invertible.

Define the kernel-based approximation of Bellman's operator for a fixed policy $\hat{M}^{\pi,m}$ from the sample path of post-decision states $S^{a,0}, \dots, S^{a,m+1}$ using

$$\hat{M}^{\pi,m} V(s) = \sum_{i=0}^{m-1} k(S^{a,i}, s)(\hat{C}^i + \gamma V(S^{M,a}(S^i, a^i, W^{i+1}))).$$

We would like to find the fixed point of the kernel-based Bellman equation defined by

$$\begin{aligned}\hat{V}^\pi &= \hat{M}^{\pi,m} \hat{V}^\pi \\ &= P^\pi [c^\pi + \gamma \hat{V}^\pi] \\ &= [I - \gamma P^\pi]^{-1} c^\pi.\end{aligned}$$

We can avoid the matrix inversion by using a value iteration approximation

$$\hat{V}^{\pi,k+1} = P^\pi (c^\pi + \gamma \hat{V}^{\pi,k}).$$

The vector \hat{V}^π has an element $\hat{V}^\pi(S^{a,i})$ for each of the (post-decision) states $S^{a,i}$ that we have visited. We then extrapolate from this vector of calculated values for the states we have visited, giving us the continuous function

$$\bar{V}^\pi(s) = \sum_{i=0}^{m-1} k(S^{a,i}, s) \left(\hat{C}^i + \gamma \hat{V}^\pi(S^{a,i+1}) \right).$$

This approximation forms the basis of our approximate policy iteration. The full algorithm is given in Figure 10.13.

Step 0. Initialization.

Step 0a. Initialize the policy $A^\pi(s)$.

Step 0b. Choose the kernel function $K_h(s, s')$.

Step 0b. Set $n = 1$.

Step 1. Do for $n = 1, \dots, N$:

Step 2. Choose an initial state S_0^n .

Step 3. Do for $m = 0, 1, \dots, M$:

Step 4. Let $a^{n,m} = A^{\pi,n}(S^{n,m})$.

Step 5. Sample W^{m+1} .

Step 6. Compute the post-decision state $S^{a,m} = S^{M,a}(S^{n,m}, a^{n,m})$ and the next state $S^{m+1} = S^M(S^{n,m}, a^{n,m}, W^{m+1})$.

Step 7. Let c^π be a vector of dimensionality M with element $\hat{C}^m = C(S^{n,m}, a^{n,m}, W^{m+1})$, $m = 1, \dots, M$.

Step 8. Let $P^{\pi,n}$ be a $M \times M$ matrix where the (i, j) th entry is given by $K_h(S^{a,i-1}, S^{a,j})$ for $i, j \in \{1, \dots, m\}$.

Step 9. Solve for $\hat{v}^n = (I - \gamma P^{\pi,n})^{-1}$, where \hat{v}^n is an m -dimensional vector with i th element $\hat{v}^n(S^{a,i})$ for $i = 1, \dots, m$. This can be approximated using value iteration.

Step 10. Let $\bar{V}^n(s) = \sum_{i=0}^{m-1} K_h(S^{a,i}, s)(\hat{C}^i + \gamma \hat{v}^n(S^{a,i}))$ be our kernel-based value function approximation.

Step 11. Update the policy

$$A^{\pi,n+1}(s) = \arg \max_a (C(s, a) + \gamma \bar{V}^n(S^{M,a}(s, a))).$$

Step 12. Return the $A^{\pi,N}(s)$ and parameter θ^N .

Figure 10.13 Approximate policy iteration using least squares temporal differencing and kernel regression.

10.10 FINITE HORIZON APPROXIMATIONS FOR STEADY-STATE APPLICATIONS

It is easy to assume that if we have a problem with stationary data (i.e., all random information is coming from a distribution that is not changing over time), then we can solve the problem as an infinite horizon problem, and use the resulting value function to produce a policy that tells us what to do in any state. If we can in fact find the optimal value function for every state, then this is true.

There are many applications of infinite horizon models to answer policy questions. Do we have enough doctors? What if we increase the buffer space for holding customers in a queue? What is the impact of lowering transaction costs on the amount of money a mutual fund holds in cash? What happens if a car rental company changes the rules allowing rental offices to give customers a better car if they run out of the type of car that a customer reserved? These are all dynamic programs controlled by a constraint (the size of a buffer or the number of doctors), a parameter (the transaction cost), or the rules governing the physics of the problem (the ability to substitute cars). We may be interested in understanding the behavior

of such a system as these variables are adjusted. For infinite horizon problems that are too complex to solve exactly, ADP offers a way to approximate these solutions.

Infinite horizon models also have applications in operational settings. Suppose that we have a problem governed by stationary processes. We could solve the steady-state version of the problem, and use the resulting value function to define a policy that would work from any starting state. This works if we have in fact found at least a close approximation of the optimal value function for any starting state. However, if you have made it this far in this book, then that means you are interested in working on problems where the optimal value function cannot be found for all states. Typically we are forced to approximate the value function, and it is always the case that we do the best job of fitting the value function around states that we visit most of the time.

When we are working in an operational setting, we start with some known initial state S_0 . From this state there are a range of “good” decisions, followed by random information, that will take us to a set of states S_1 that is typically heavily influenced by our starting state. Figure 10.14 illustrates the phenomenon. Assume that our true, steady-state value function approximation looks like the sine function. At time $t = 1$, the probability distribution of the state S_t that we can reach is shown as the shaded area. Assume that we have chosen to fit a quadratic function of the value function, using observations of S_t that we generate through Monte Carlo sampling. We might obtain the dotted curve labeled as $\bar{V}_1(S_1)$, which closely fits the true value function around the states S_1 that we have observed.

For times $t = 2$ and $t = 3$, the distribution of states S_2 and S_3 that we actually observe grows wider and wider. As a result the best fit of a quadratic function spreads as well. So, even though we have a steady-state problem, the best value function approximation depends on the initial state S_0 and how many time periods into the future that we are projecting. Such problems are best modeled as finite horizon problems, but only because we are forced to approximate the problem.

10.11 BIBLIOGRAPHIC NOTES

Section 10.2 Approximate value iteration using lookup tables encompasses the family of algorithms that depend on an approximation of the value of a future state to estimate the value of being in a state now, which includes Q -learning and temporal-difference learning. These methods represent the foundation of approximate dynamic programming and reinforcement learning, and we already saw an introduction to lookup table versions of these algorithms in Chapter 4.

Section 10.4 The problems with the use of linear models in the context of approximate value iteration (TD learning) are well known in the research literature. Good discussions of these issues are found in Bertsekas and Tsitsiklis (1996), Tsitsiklis et al. (1997), Baird (1995), and Precup et al. (2001), to name a few. Munos and Szepesvari (2008) prove convergence for an approximate value iteration algorithm using a linear model but impose technical conditions such as requiring that the algorithm start with a policy that is close to the optimal policy. This paper provides insights into the types of restrictions that may be required for approximate value iteration to work.

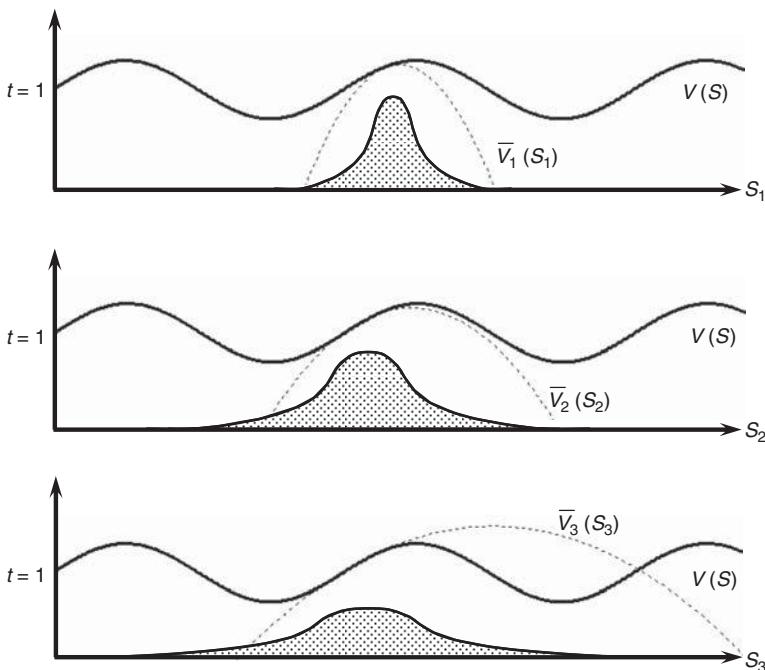


Figure 10.14 Exact value function (sine curve) and value function approximations for $t = 1, 2, 3$, which change with the probability distribution of the states that we can reach from S_0 .

Section 10.5 Bradtko and Barto (1996) introduced least squares temporal differencing as a way of approximating the one-period contribution using a linear model, and then projecting the infinite horizon performance. Lagoudakis and Parr (2003) describe the least squares policy iteration algorithm (LSPI), which uses a linear model to approximate the Q -factors that is then embedded in a model-free algorithm.

Section 10.6 There is a long history of referring to policies as “actors” and value functions as “critics” (e.g., see Barto et al., 1983; Williams and Baird, 1990; Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998). Borkar and Konda (1997) and Konda and Borkar (1999) analyze actor–critic algorithms as an updating process with two time scales, one for the inner iteration to evaluate a policy and one for the outer iteration where the policy is updated. Konda and Tsitsiklis (2003) discuss actor–critic algorithms using linear models to represent both the actor and the critic, and bootstrapping for the critic. Bhatnagar et al. (2009) suggest several new variations of actor–critic algorithms, and prove convergence when both the actor and the critic use bootstrapping.

Section 10.7 Policy gradient methods have received considerable attention in the reinforcement learning community. The material in this section is based on Konda and Tsitsiklis (2003), but we have provided only a streamlined presentation, and we urge readers to consult the original article before attempting to implement the equations given in this section. One of the earliest

policy-gradient algorithms is given in Williams (1992). Marbach and Tsitsiklis (2001) provide gradient-based algorithms for optimizing Markov reward processes, which is a mathematically equivalent problem. Sutton et al. (2000) provide a version of a policy-gradient algorithm, but in a form that is difficult to compute. Szepesvari (2010) provides a recent summary of policy gradient algorithms.

Section 10.8 Schweitzer and Seidmann (1985) describe the use of basis functions in the context of the linear programming method. The idea is further developed in de Farias and Van Roy (2003) who also develops performance guarantees. de Farias and Van Roy (2001) investigate the use of constraint sampling and prove results on the number of samples that are needed.

Section 10.9 This material is based on Ma and Powell (2010a).

Problems Exercise 10.1 is due to de Farias and Van Roy (2000).

PROBLEMS

- 10.1** Consider a Markov chain with two states, 1 and 2, and just one policy. There is a reward from being in state 1 of \$1, and if in state 1 there is a probability 0.2 of staying in state 1, and a probability 0.8 of transitioning to state 2. The reward for being in state 2 is \$2 with a probability 0.2 of transitioning to state 1, and a probability of 0.8 of staying in state 2. Let c^π be the column vector of rewards for being in each state, and let P be the one-step transition matrix. Let the basis functions be given by $\phi(1) = 1$ and $\phi(2) = 2$, and let $\Phi = [1, 2]^T$. Let Π be the projection operator (derived earlier). If we represent the value function using $\Phi\theta$, then Bellman's equation would be written

$$\begin{aligned}\Phi\theta &= \Pi\mathcal{M}^\pi\Phi\theta \\ &= \Pi(c^\pi + \gamma P\Phi\theta),\end{aligned}$$

where $\gamma = 5/4$.

- (a) First using value iteration and generate 5 iterations of

$$v^{n+1} = c^\pi + \gamma Pv^n,$$

starting with $v^0 = 0$. What can you say about the limiting performance of v^n if we were to repeat this indefinitely?

- (b) Now consider what happens when we use value iteration with basis functions, where

$$\theta = (\Phi^T\Phi)^{-1}\Phi^T(c^\pi + \gamma P\Phi\theta).$$

Use this function to show that value iteration for this problem would reduce to $\theta^{n+1} = 1 + \theta^n$. What can you say about the limiting behavior of approximate value iteration using basis functions?

Adaptive Estimation and Stepsizes

At the core of approximate dynamic programming is some form of iterative learning, whether we are doing policy search or approximating value functions. In Section 9.2 we introduced a stochastic optimization problem with the generic form

$$\min_v \mathbb{E}F(v, \hat{v}),$$

where

$$F(v, \hat{v}) = \frac{1}{2}(v - \hat{v})^2.$$

Here v is the value of being in some state that we are trying to estimate, and \hat{v} is a random variable that represents a noisy estimate of this state. An algorithm for estimating the best value of v (i.e., that minimizes the expected value of $F(v, \hat{v})$) is a stochastic gradient algorithm, which looks like

$$v^n = v^{n-1} - \alpha_{n-1} \nabla F(v^{n-1}, \hat{v}^n) \quad (11.1)$$

$$= v^{n-1} - \alpha_{n-1} (v^{n-1} - \hat{v}^n) \quad (11.2)$$

$$= (1 - \alpha_{n-1})v^{n-1} + \alpha_{n-1}\hat{v}^n. \quad (11.3)$$

Equation (11.2) shows the updating in the classical form that we have seen using temporal difference learning, and equation (11.3) shows the form most familiar when using recursive least squares. Both of these are easily seen as directly related to the classical stochastic gradient algorithm in equation (11.1).

This classical derivation based on stochastic optimization views the sequence of observations \hat{v}^n as if they are observations from some signal such as a stock price or wind speed. In approximate dynamic programming the observations \hat{v}^n are somewhat more complex, although their precise nature depends on the algorithm that we are using. For this reason we begin our presentation with a review of

recursive least squares, approximate value iteration, temporal difference learning (TD(0)), least squares policy evaluation (LSPE), and least squares temporal differences (LSTD). These algorithms are analyzed using a Markov decision process with a single state, which makes it possible to derive the updating equations in closed form. From these results it is possible to derive insights into the demands on a stepsize rule.

In other chapters we often use θ^n as the estimate of the true value θ after n iterations. In this chapter we have to deal with time-dependent series, and we use θ^n as the true value of our series at iteration n , and $\bar{\theta}^n$ is our estimate of θ^n based on the observations we have made so far.

11.1 LEARNING ALGORITHMS AND STEPSIZES

A useful exercise to understand the behavior of recursive least squares, LSTD and LSPE is to consider what happens when they are applied to a trivial dynamic program with a single state and a single action. This dynamic program is equivalent to computing the sum

$$F = \mathbb{E} \sum_{i=0}^{\infty} \gamma^i \hat{C}^i, \quad (11.4)$$

where \hat{C}^i is a random variable giving the i th contribution. If we let $\bar{c} = \mathbb{E} \hat{C}^i$, then clearly $F = \bar{c}/(1 - \gamma)$. But let us pretend that we do not know this, and we are using these various algorithms to compute the expectation.

We first used the single-state problem in Section 9.6 but did not focus on the implications for stepsizes. Here we use our ability to derive analytical solutions for the optimal value function for least squares temporal differences (LSTD), least squares policy evaluation (LSPE), and recursive least squares and temporal differences. These expressions allow us to understand the types of behaviors we would like to see in a stepsize formula.

In the remainder of this section, we start by assuming that the value function is approximated using a linear model

$$\bar{V}(s) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(s).$$

However, we are going to then transition to a problem with a single state, and a single basis function $\phi(s) = 1$. We assume that \hat{v} is a sampled estimate of the value of being in the single state.

11.1.1 Least Squares Temporal Differences

In Section 9.5 we showed that the LSTD method, when using a linear architecture applied to infinite horizon problems, required solving

$$\sum_{i=1}^n \phi_f(S^i)(\phi_f(S^i) - \gamma \phi_f(S^{i+1}))^T \theta = \sum_{i=1}^n \phi_f(S^i) \hat{C}^i$$

for each $f \in \mathcal{F}$. Let θ^n be the optimal solution. Again, since we have only one basis function $\phi(s) = 1$ for our single-state problem, this reduces to finding $v^n = \theta^n$:

$$v^n = \frac{1}{1 - \gamma} \left(\frac{1}{n} \sum_{i=1}^n \hat{C}^n \right). \quad (11.5)$$

Equation (11.5) shows that we are trying to estimate $\mathbb{E}\hat{C}$ using a simple average. If we let \bar{C}^n be the average over n observations, we can write this recursively using

$$\bar{C}^n = \left(1 - \frac{1}{n}\right) \bar{C}^{n-1} + \frac{1}{n} \hat{C}^n.$$

For the single-state (and single-action) problem, the sequence \hat{C}^n comes from a stationary sequence. In this case a simple average is the best possible estimator. In a dynamic programming setting with multiple states, and where we are trying to optimize over policies, v^n would depend on the state. Also, because the policy that determines the action we take when we are in a state that is changing over the iterations, the observations \hat{C}^n , even when we fix a state, would be nonstationary. In this setting simple averaging is no longer the best. Instead, it is better to use

$$\bar{C}^n = (1 - \alpha_{n-1}) \bar{C}^{n-1} + \alpha_{n-1} \hat{C}^n, \quad (11.6)$$

and also one of the stepsizes described in Section 11.2, 11.3, or 11.4. As a rule, these stepsize rules do not decline as quickly as $1/n$.

11.1.2 Least Squares Policy Evaluation

Least squares policy evaluation, which is developed using basis functions for infinite horizon applications, finds the regression vector θ by solving

$$\theta^n = \arg \max_{\theta} \sum_{i=1}^n \left(\sum_f \theta_f \phi_f(S^i) - (\hat{C}^i + \gamma \bar{V}^{n-1}(S^{i+1})) \right)^2.$$

When we have one state, the value of being in the single state is given by $v^n = \theta^n$, which we can write as

$$v^n = \max_{\theta} \sum_{i=1}^n \left(\theta - (\hat{C}^i + \gamma v^{n-1}) \right)^2.$$

This problem can be solved in closed form, giving us

$$v^n = \left(\frac{1}{n} \sum_{i=1}^n \hat{C}^i \right) + \gamma v^{n-1}.$$

Similar to LSTD, LSPE works to estimate $\mathbb{E}\hat{C}$. For a problem with a single state and action (and therefore only one policy), the best estimate of $\mathbb{E}\hat{C}$ is a simple average. However, as we already argued with LSTD, if we have multiple states and are searching for the best policy, the observation \hat{C} for a particular state will come from a nonstationary series. For such problems we should again adopt the updating formula in (11.6) and use one of the stepsize rules described Section 11.2, 11.3, or 11.4.

11.1.3 Recursive Least Squares

Using our linear model, we start with the following standard least squares model to fit our approximation:

$$\min_{\theta} \sum_{i=1}^n \left(\hat{v}^i - \left(\sum_{f \in \mathcal{F}} \theta_f \phi_f(S^i) \right) \right)^2.$$

As we have already seen in Chapter 9, we can fit the parameter vector θ using least squares, which can be computed recursively using

$$\theta^n = \theta^{n-1} - \frac{1}{1 + (x^n)^T B^{n-1} x^n} B^{n-1} x^n (\bar{V}^{n-1}(S^n) - \hat{v}^n),$$

where $x^n = (\phi_1(S^n), \dots, \phi_f(S^n), \dots, \phi_F(S^n))$, and the matrix B^n is computed using

$$B^n = B^{n-1} - \frac{1}{1 + (x^n)^T B^{n-1} x^n} (B^{n-1} x^n (x^n)^T B^{n-1}).$$

For the special case of a single state, we use the fact that we have only one basis function $\phi(s) = 1$ and one parameter $\theta^n = \bar{V}^n(s) = v^n$. In this case the matrix B^n is a scalar, and the updating equation for θ^n (now v^n), becomes

$$\begin{aligned} v^n &= v^{n-1} - \frac{B^{n-1}}{1 + B^{n-1}} (v^{n-1} - \hat{v}^n) \\ &= \left(1 - \frac{B^{n-1}}{1 + B^{n-1}} \right) v^{n-1} + \frac{B^{n-1}}{1 + B^{n-1}} \hat{v}^n. \end{aligned}$$

If $B^0 = 1$, $B^{n-1} = 1/n$, giving us

$$v^n = \left(1 - \frac{1}{n} \right) v^{n-1} + \frac{1}{n} \hat{v}^n. \quad (11.7)$$

Now imagine we are using approximate value iteration. In this case, $\hat{v}^n = \hat{C}^n + \gamma v^n$. Substituting this into equation (11.7) gives us

$$\begin{aligned} v^n &= \left(1 - \frac{1}{n}\right) v^{n-1} + \frac{1}{n}(\hat{C}^n + \gamma \hat{v}^n) \\ &= \left(1 - \frac{1}{n}(1 - \gamma)\right) v^{n-1} + \frac{1}{n} \hat{C}^n. \end{aligned} \quad (11.8)$$

Recursive least squares has the behavior of averaging the observations of \hat{v} . The problem is that $\hat{v}^n = \hat{C}^n + \gamma v^n$, since \hat{v}^n is also trying to be a discounted accumulation of the costs. Assume that the contribution was deterministic, where $\hat{C} = c$. If we were doing classical approximate value iteration, we would write

$$v^n = c + \gamma v^{n-1}. \quad (11.9)$$

Comparing (11.8) and (11.9), we see that the one-period contribution carries a coefficient of $1/n$ in (11.8) and a coefficient of 1 in (11.8). We can view equation (11.8) as a steepest ascent update with a stepsize of $1/n$. If we change the stepsize to 1, we obtain (11.9).

11.1.4 Bounding $1/n$ Convergence for Approximate Value Iteration

It is well known that a $1/n$ stepsize will produce a provably convergent algorithm when used with approximate value iteration. Experimentalists know that the rate of convergence can be quite slow, but people new to the field can sometimes be found using this stepsize rule. In this section we hope to present evidence that the $1/n$ stepsize should never be used with approximate value iteration or its variants.

Figure 11.1 is a plot of v^n computed using equation (11.8) as a function of $\log_{10}(n)$ for $\gamma = 0.7, 0.8, 0.9$, and 0.95 , where we have set $\hat{C} = 1$. For $\gamma = 0.90$, we need 10^{10} iterations to get $\bar{v}^n = 9$, which means we are still 10 percent from the optimal. For $\gamma = 0.95$, we are not even close to converging after 100 billion iterations.

It is possible to derive compact bounds, $v^L(n)$ and $v^U(n)$ for \bar{v}^n where

$$v^L(n) < v^n < v^U(n).$$

These are given by

$$v^L(n) = \frac{c}{1 - \gamma} \left(1 - \left(\frac{1}{1+n}\right)^{1-\gamma}\right), \quad (11.10)$$

$$v^U(n) = \frac{c}{1 - \gamma} \left(1 - \frac{1 - \gamma}{\gamma n} - \frac{1}{\gamma n^{1-\gamma}} (\gamma^2 + \gamma - 1)\right). \quad (11.11)$$

Using the formula for the lower bound (which is fairly tight when n is large enough that v^n is close to v^*), we can derive the number of iterations to achieve a

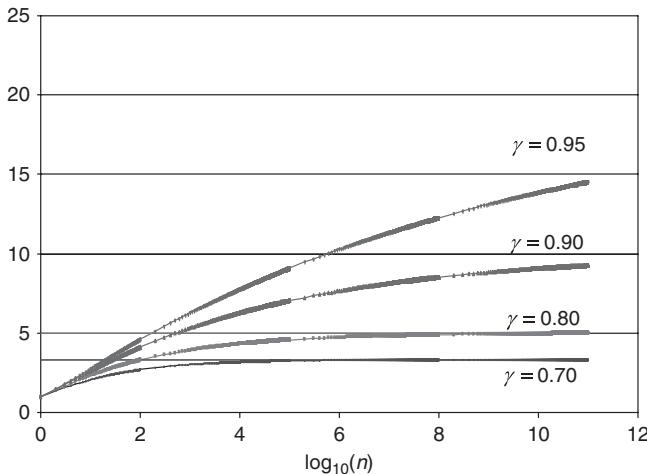


Figure 11.1 \bar{v}^n plotted against $\log_{10}(n)$ when we use a $1/n$ stepsize rule for updating.

particular degree of accuracy. Let $\hat{C} = 1$, which means that $v^* = 1/(1 - \gamma)$. For a value of $v < 1/(1 - \gamma)$, we would need at least $n(v)$ to achieve $\bar{v}^* = v$, where $n(v)$ is found (from (11.10)) to be

$$n(v) \geq [1 - (1 - \gamma)v]^{-1/(1-\gamma)}. \quad (11.12)$$

If $\gamma = 0.9$, we would need $n(v) = 10^{20}$ iterations to reach a value of $v = 9.9$, which gives us a one percent error. On a 3-GHz chip, assuming we can perform one iteration per clock cycle (i.e., 3×10^9 iterations per second), it would take 1000 years to achieve this result.

11.1.5 Discussion

We can now see the challenge of choosing stepsizes for approximate value iteration, temporal-difference learning, and Q -learning, compared to algorithms such as LSPE, LSTD, and approximate policy iteration (the finite horizon version of LSPE). If we observe \hat{C} with noise, and if the discount factor $\gamma = 0$ (which means that we are not trying to accumulate contributions over time), then a stepsize of $1/n$ is ideal. We are just averaging contributions to find the average value. As the noise in \hat{C} diminishes, and as γ increases, we would like a stepsize that approaches 1. In general, we have to strike a balance between accumulating contributions over time (which is more important as γ increases) and averaging the observations of contributions (for which a stepsize of $1/n$ is ideal).

By contrast, LSPE, LSTD, and approximate policy iteration are all trying to estimate the average contribution per period for each state. The values $\hat{C}(s, a)$ are nonstationary because the policy that chooses the action is changing, making the sequence $\hat{C}(s^n, a^n)$ nonstationary. But these algorithms are not trying to simultaneously accumulate contributions over time.

Sections 11.2, 11.3, and 11.4 describe stepsize formulas that are especially designed for estimating means in the presence of nonstationary data when estimating value functions based on some variation of policy evaluation. Section 11.2 describes simple deterministic stepsize rules, while Section 11.3 describes heuristic stochastic stepsize rules that adapt to the data. Section 11.4 presents results for optimal stepsize rules, including results for both stationary and nonstationary data series. Section 11.5 presents a new stepsize formula, called the “optimal stepsize for approximate value iteration” (OSAVI) that is specifically designed for approximate value iteration and its variants such as Q -learning and TD(0).

11.2 DETERMINISTIC STEPSIZE RECIPES

One of the challenges in Monte Carlo methods is finding the stepsize α_n . We refer to a method for choosing a stepsize as a *stepsize rule*, while other communities refer to them as *learning rate schedules*. A standard technique in deterministic problems (of the continuously differentiable variety) is to find the value of α_n so that θ^n gives the smallest possible objective function value (among all possible values of α). For a deterministic problem, this is generally not too hard. For a stochastic problem, it means calculating the objective function, which involves computing an expectation. For most applications expectations are computationally intractable, which makes it impossible to find an optimal stepsize.

Throughout our presentation, we assume that we are trying to estimate a parameter θ^n that is evolving over the iterations, just as would occur using exact value iteration (from Chapter 3) when we can compute expectations. In a dynamic programming setting, θ would be the value of being in some state s , but we are going to suppress the reference to state s . We let $\bar{\theta}^{n-1}$ be our estimate of θ^n computing using information from the first $n - 1$ iterations. We then let $\hat{\theta}^n$ be a random observation of θ^n in iteration n . (In Chapter 4, we used \hat{v}^n as our random observation of being in a state, as shown in Figure 4.2.)

Our updating equation looks like

$$\bar{\theta}^n = (1 - \alpha_{n-1})\bar{\theta}^{n-1} + \alpha_{n-1}\hat{\theta}^n. \quad (11.13)$$

Our iteration counter always starts at $n = 1$ (just as our first time interval starts with $t = 1$). The use of α_{n-1} in equation (11.13) means that we are computing α_{n-1} using information available at iteration $n - 1$ and before. Thus we have an explicit assumption that we are not using $\hat{\theta}^n$ to compute the stepsize in iteration n . This is irrelevant when we use a deterministic stepsize sequence, but it is critical in convergence proofs for stochastic stepsize formulas (introduced below). In most formulas, α_0 is a parameter that has to be specified, although we will generally assume that $\alpha_0 = 1$, which means that we do not have to specify $\bar{\theta}_0$. The only reason to use $\alpha_0 < 1$ is when we have some a priori estimate of $\bar{\theta}_0$ that is better than $\hat{\theta}^1$.

Stepsize rules can be important especially for algorithms such as approximate value iteration and Q -learning. As you experiment with ADP, it is possible to find

problems where provably convergent algorithms simply do not work, and the only reason is a poor choice of stepsizes. Inappropriate choices of stepsize rules have led many to conclude that “approximate dynamic programming does not work.”

There are two issues when designing a good stepsize rule. The first is the question of whether the stepsize will produce a provably convergent algorithm. While this is primarily of theoretical interest, these conditions do provide important guidelines to follow to produce good behavior. The second issue is whether the rule produces the fastest rate of convergence. Both are important issues in practice.

Below we start with a general discussion of stepsize rules. Following this, we provide a number of examples of deterministic stepsize rules. These are formulas that depend only on the iteration counter n (or more precisely, the number of times that we update a particular parameter). Section 11.3 then describes stochastic stepsize rules that adapt to the data.

The deterministic and stochastic rules presented in this section and Section 11.3 are, for the most part, heuristically designed to achieve good rates of convergence, but are not supported by any theory that they will produce the best rate of convergence. Later (Section 11.4) we provide a theory for choosing stepsizes that produce the fastest possible rate of convergence when estimating value functions based on policy evaluation. Finally, Section 11.5 presents a new optimal stepsize rule designed specifically for approximate value iteration.

11.2.1 Properties for Convergence

The theory for proving convergence of stochastic gradient algorithms was first developed in the early 1950s and has matured considerably since then (see Section 7.6). However, all the proofs require three basic conditions

$$\alpha_{n-1} \geq 0, \quad n = 1, 2, \dots, \quad (11.14)$$

$$\sum_{n=1}^{\infty} \alpha_{n-1} = \infty, \quad (11.15)$$

$$\sum_{n=1}^{\infty} (\alpha_{n-1})^2 < \infty. \quad (11.16)$$

Equation (11.14) obviously requires that the stepsizes be nonnegative. The most important requirement is (11.15), which states that the infinite sum of stepsizes must be infinite. If this condition did not hold, the algorithm might stall prematurely. Finally, condition (11.16) requires that the infinite sum of the squares of the stepsizes be finite. This condition in effect requires that the stepsize sequence converge “reasonably quickly.” A good intuitive justification for this condition is that it guarantees that the *variance* of our estimate of the optimal solution goes to zero in the limit. Sections 7.6.2 and 7.6.3 illustrate two proof techniques that both lead to these requirements on the stepsize. However, it is possible under certain conditions to replace equation (11.16) with the weaker requirement that $\lim_{n \rightarrow \infty} \alpha_n = 0$.

Conditions (11.15) and (11.16) effectively require that the stepsizes decline according to an arithmetic sequence such as

$$\alpha_{n-1} = \frac{1}{n}. \quad (11.17)$$

This rule has an interesting property. Exercise 11.3 asks you to show that a step-size of $1/n$ produces an estimate $\bar{\theta}^n$ that is simply an average of all previous observations, which is to say,

$$\bar{\theta}^n = \frac{1}{n} \sum_{m=1}^n \hat{\theta}^m. \quad (11.18)$$

Of course, we have a nice name for equation (11.18): it is called a sample average. And we are all aware that in general (some modest technical conditions are required) as $n \rightarrow \infty$, $\bar{\theta}^n$ will converge (in some sense) to the mean of our random variable $\hat{\theta}$.

The issue of the rate at which the stepsizes decrease is of considerable practical importance. Consider, for example, the stepsize sequence

$$\alpha_n = 0.5\alpha_{n-1},$$

which is a geometrically decreasing progression. This stepsize formula violates (11.5). More intuitively, the problem is that the stepsizes would decrease so quickly that it is likely that we would never reach the final solution.

Surprisingly, the “ $1/n$ ” stepsize formula, which works in theory, tends not to work in practice because it drops to zero too quickly when applied to approximate dynamic programming applications (as we discussed in Section 11.1). The reason is that we are usually updating the value function using biased estimates that are changing over time. For example, consider the updating expression we used for the post-decision state variable given in Section 4.6.5, which we repeat here for convenience

$$\begin{aligned} \hat{v}_t^n &= C_t(S_t^n, a_t^n) + \bar{V}_t^{n-1}(S_t^{a,n}), \\ \bar{V}_{t-1}^n(S_{t-1}^{a,n}) &= (1 - \alpha_{n-1})\bar{V}_{t-1}^{n-1}(S_{t-1}^{a,n}) + \alpha_{n-1}\hat{v}_t^n. \end{aligned}$$

\hat{v}_t^n is our sample observation of an estimate of the value of being in state S_t , which we then smooth into the current approximation $\bar{V}_t^{n-1}(S_t^{a,n})$. If \hat{v}_t^n were an unbiased estimate of the true value, then a stepsize of $1/n$ would be the best we could do (we show this later). However, \hat{v}_t^n depends on $\bar{V}_t^{n-1}(S_t^a)$, which is an imperfect estimate of the value function for time t . What typically happens is that the value functions undergo a transient learning phase. Since we have not found the correct estimate for $\bar{V}_t^{n-1}(S_t^{a,n})$, the estimates \hat{v}_t^n are biased, and the $1/n$ rule puts the highest weights on the early iterations when the estimates are the worst. The resulting behavior is illustrated in Figure 11.2.

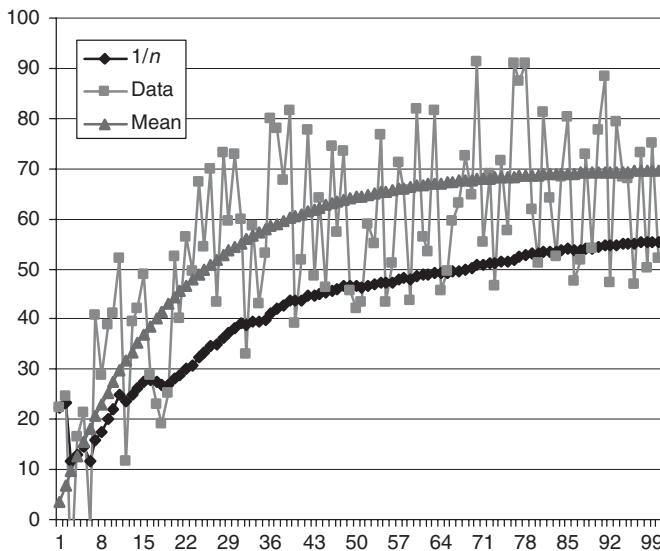


Figure 11.2 Illustration of poor convergence of the $1/n$ stepsize rule in the presence of transient data.

The remainder of this section presents a series of deterministic stepsize formulas designed to overcome this problem. These rules are the simplest to implement and are typically the best starting point when designing an ADP algorithm.

Constant Stepsizes

A constant stepsize rule is simply

$$\alpha_{n-1} = \begin{cases} 1 & \text{if } n = 1, \\ \bar{\alpha} & \text{otherwise,} \end{cases}$$

where $\bar{\alpha}$ is a stepsize that we have chosen. It is common to start with a stepsize of 1 so that we do not need an initial value $\bar{\theta}^0$ for our statistic.

Constant stepsizes are popular when we are estimating not one but many parameters (for large scale applications, these can easily number in the thousands or millions). In these cases no single rule is going to be right for all of the parameters, and there is enough noise that any reasonable stepsize rule will work well. Constant stepsizes are easy to code (no memory requirements) and, in particular, easy to tune (there is only one parameter). Perhaps the biggest point in their favor is that we simply may not know the rate of convergence, which means that we run the risk with a declining stepsize rule of allowing the stepsize to decline too quickly, producing a behavior we refer to as “apparent convergence.”

In dynamic programming we are typically trying to estimate the value of being in a state using observations that are not only random but also changing systematically as we try to find the best policy. As a rule, as the noise in the observations of

the values increases, the best stepsize decreases. But, if the values are increasing rapidly, we want a larger stepsize. Choosing the best stepsize requires striking a balance between stabilizing the noise and responding to the changing mean. Figure 11.3 illustrates observations that are coming from a process with relatively low noise but where the mean is changing quickly (Figure 11.3a), and observations that are very noisy but where the mean is not changing at all (Figure 11.3b). For

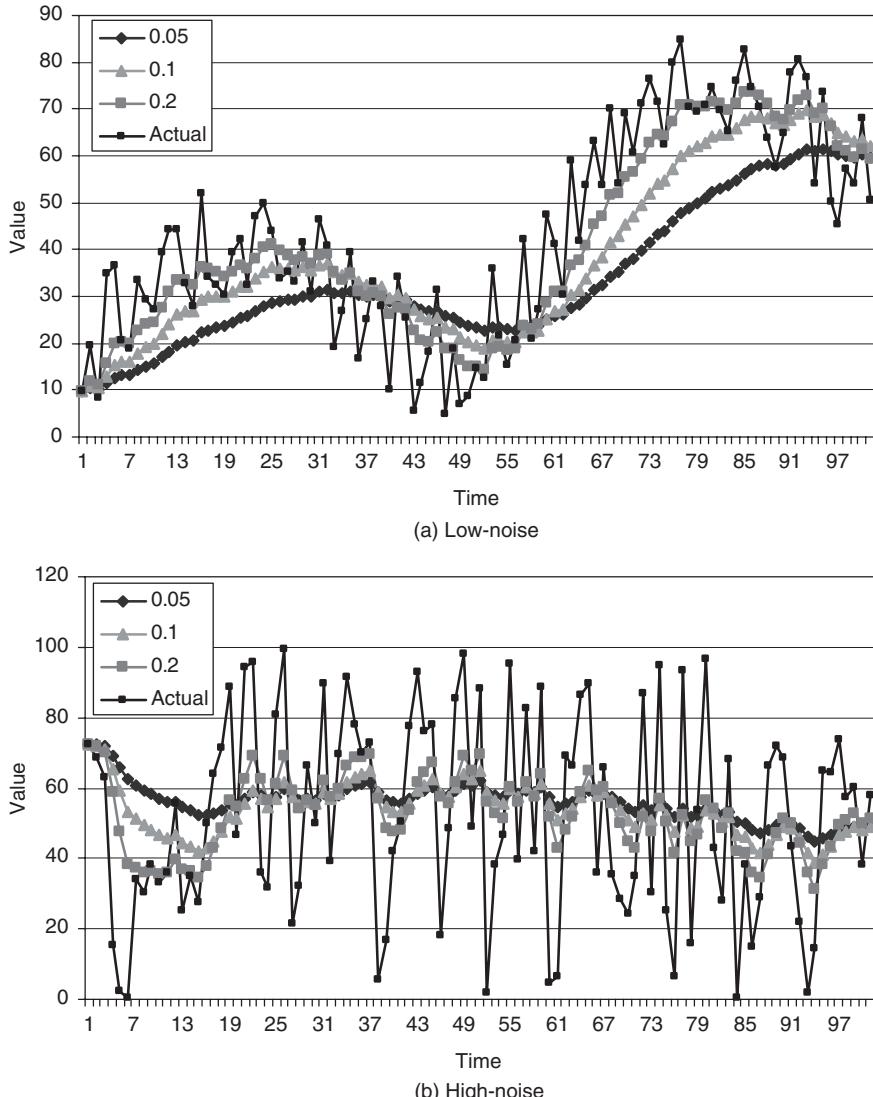


Figure 11.3 Illustration of the effects of smoothing using constant stepsizes. (a) Low-noise dataset with an underlying nonstationary structure; (b) high-noise dataset from a stationary process.

the first, the ideal stepsize is relatively large, while for the second, the best stepsize is quite small.

Generalized Harmonic Stepsizes

A generalization of the $1/n$ rule is the generalized harmonic sequence given by

$$\alpha_{n-1} = \frac{a}{a + n - 1}. \quad (11.19)$$

This rule satisfies the conditions for convergence but produces larger stepsizes for $a > 1$ than the $1/n$ rule. Increasing a slows the rate at which the stepsize drops to zero, as illustrated in Figure 11.4. Choosing the best value of a requires understanding the rate of convergence of the application. Some problems converge to good solutions in a few hundred iterations, while others require hundreds of thousands or millions of iterations. The best value of a might be as small as 5 or 10, or as large as 10,000 (or greater). When you get a sense of how many iterations are required, choose a so that the stepsize is less than .05 as the algorithm approaches convergence.

Polynomial Learning Rates

An extension of the basic harmonic sequence is the stepsize

$$\alpha_{n-1} = \frac{1}{(n)^\beta}, \quad (11.20)$$

where $\beta \in (\frac{1}{2}, 1]$. Smaller values of β slow the rate at which the stepsizes decline, which improves the responsiveness in the presence of initial transient conditions. The best value of β depends on the degree to which the initial data is transient, and as such is a parameter that needs to be tuned.

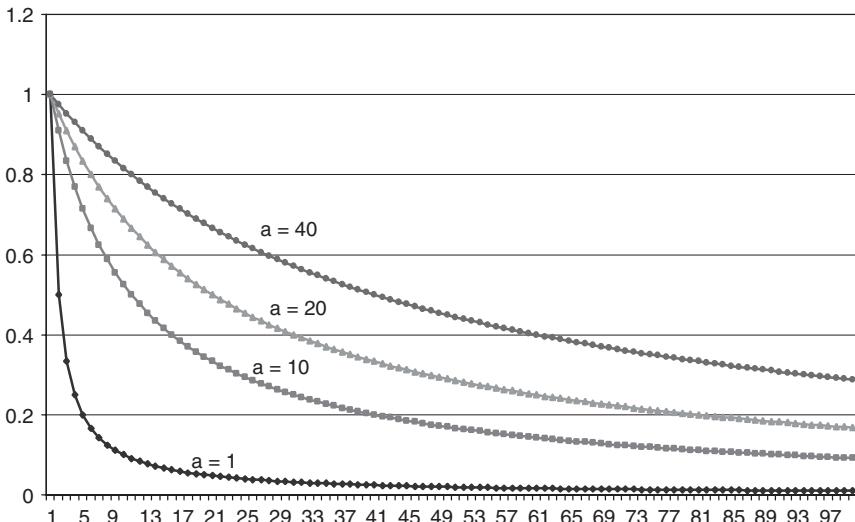


Figure 11.4 Stepsizes for $a/(a + n)$ while varying a .

McClain's Formula

McClain's formula (McClain, 1974) is an elegant way of obtaining $1/n$ behavior initially but approaching a specified constant in the limit. The formula is given by

$$\alpha_n = \frac{\alpha_{n-1}}{1 + \alpha_{n-1} - \bar{\alpha}}, \quad (11.21)$$

where $\bar{\alpha}$ is a specified parameter. Note that steps generated by this model satisfy the following properties:

$$\alpha_n > \alpha_{n+1} > \bar{\alpha} \quad \text{if } \alpha > \bar{\alpha},$$

$$\alpha_n < \alpha_{n+1} < \bar{\alpha} \quad \text{if } \alpha < \bar{\alpha}.$$

McClain's rule, illustrated in Figure 11.5, combines the features of the “ $1/n$ ” rule that are ideal for stationary data, and constant stepsizes for nonstationary data. If we set $\bar{\alpha} = 0$, then it is easy to verify that McClain's rule produces $\alpha_{n-1} = 1/n$. In the limit, $\alpha_n \rightarrow \bar{\alpha}$. The value of the rule is that the $1/n$ averaging generally works quite well in the very first iterations (this is a major weakness of constant stepsize rules) but avoids going to zero. The rule can be effective when you are not sure how many iterations are required to start converging, and it can also work well in nonstationary environments.

Search–Then–Converge Learning Rule

The search–then–converge (STC) stepsize rule (Darken and Moody, 1992) is a variation on the harmonic stepsize rule that produces delayed learning. It was

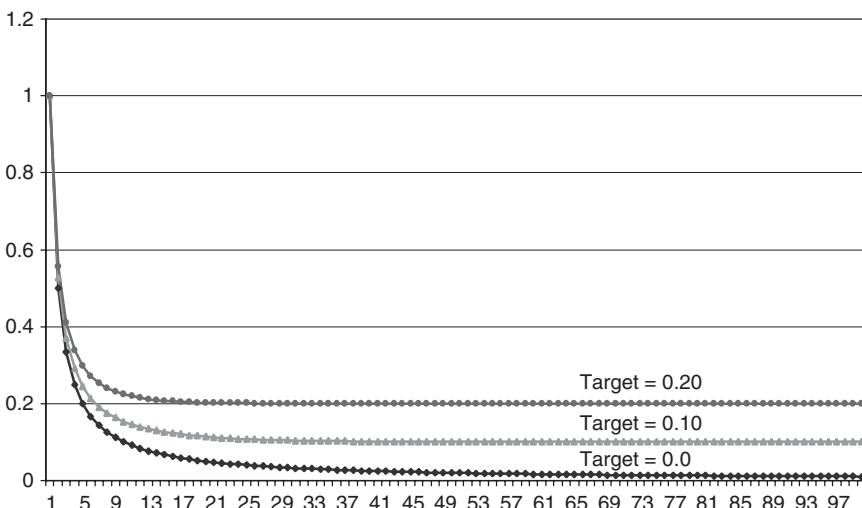


Figure 11.5 McClain stepsize rule with varying targets.

originally proposed as

$$\alpha_n = \alpha_0 \frac{\left(1 + \frac{\beta}{\alpha_0} \frac{n}{\tau}\right)}{\left(1 + \frac{\beta}{\alpha_0} \frac{n}{\tau} + \frac{n^2}{\tau}\right)}, \quad (11.22)$$

where α_0 , β and τ are parameters to be determined. A more compact and slightly more general version of this formula is

$$\alpha_{n-1} = \alpha_0 \frac{\left(\frac{b}{n} + a\right)}{\left(\frac{b}{n} + a + n^\beta\right)}. \quad (11.23)$$

If $\beta = 1$, then this formula is similar to the STC rule. In addition, if $b = 0$, then it is the same as the $a/(a+n)$ rule. The addition of the term b/n to the numerator and the denominator can be viewed as a kind of $a/(a+n)$ rule, where a is very large but declines with n . The effect of the b/n term then is to keep the stepsize larger for a longer period of time, as illustrated in Figure 11.6. This can help algorithms that have to go through an extended learning phase when the values being estimated are relatively unstable. The relative magnitude of b depends on the number of iterations that are expected to be run, which can range from several dozen to several million.

This class of stepsize rules is termed search–then–converge because they provide for a period of high stepsizes (while searching is taking place) after which

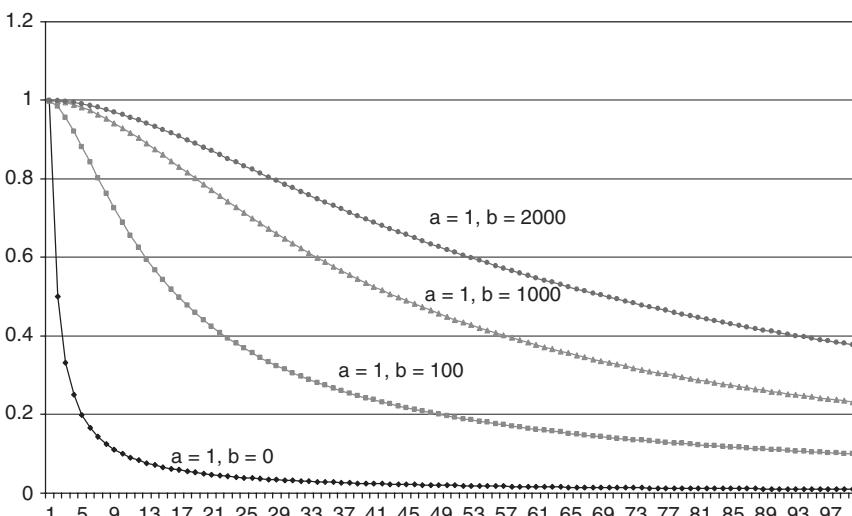


Figure 11.6 Stepsizes for $(b/n + a)/(b/n + a + n)$ while varying b .

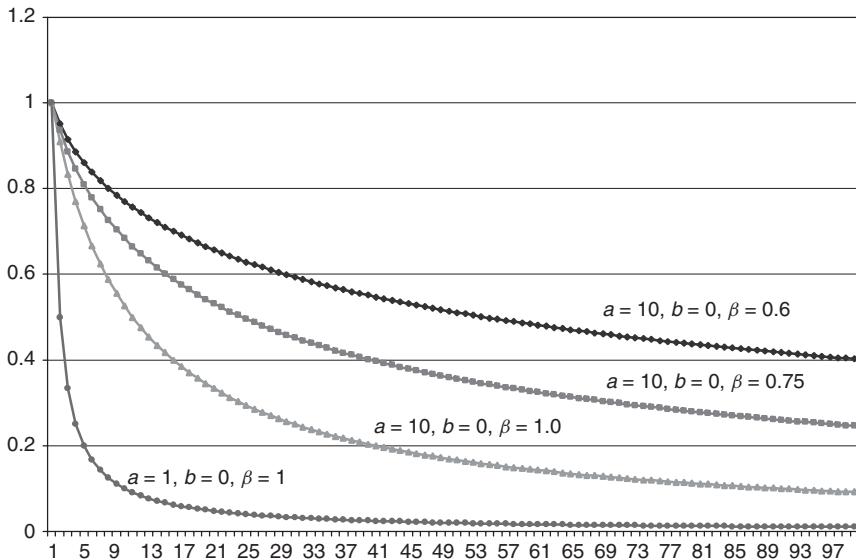


Figure 11.7 Stepsizes for $(b/n + a)/(b/n + a + n^\beta)$ while varying β .

the stepsize declines (to achieve convergence). The degree of delayed learning is controlled by the parameter b , which can be viewed as playing the same role as the parameter a but which declines as the algorithm progresses.

The exponent β in the denominator has the effect of increasing the stepsize in later iterations (see Figure 11.7). With this parameter it is possible to accelerate the reduction of the stepsize in the early iterations (by using a smaller a) but then slow the descent in later iterations (to sustain the learning process). This may be useful for problems where there is an extended transient phase requiring a larger stepsize for a larger number of iterations.

11.3 STOCHASTIC STEPSIZES

There is considerable appeal to the idea that the stepsize should depend on the actual trajectory of the algorithm. For example, if we are consistently observing that our estimate $\hat{\theta}^{n-1}$ is smaller (or larger) than the observations $\hat{\theta}^n$, then the stepsize suggests that we are trending upward (or downward). When this happens, we typically would like to use a larger stepsize to increase the speed at which we reach a good estimate. When the stepsizes depend on the observations $\hat{\theta}^n$, then we say that we are using a *stochastic stepsize*.

In this section we first review the case for stochastic stepsizes, then present the revised theoretical conditions for convergence, and finally outline a series of recipes that have been suggested in the literature (including some that have not).

11.3.1 The Case for Stochastic Stepsizes

Suppose that our estimates are consistently under or consistently over the actual observations. This can easily happen during early iterations because of either a poor initial starting point or the use of biased estimates (which is common in dynamic programming) during the early iterations. For large problems it is possible that we have to estimate thousands of parameters. It seems unlikely that all the parameters will approach their true value at the same rate. Figure 11.8 shows the change in estimates of the value of being in different states, illustrating the wide variation in learning rates that can occur within the same dynamic program.

Stochastic stepsizes try to adjust to the data in a way that keeps the stepsize larger while the parameter being estimated is still changing quickly. Balancing noise against the change in the underlying signal, particularly when both of these are unknown, is a difficult challenge.

11.3.2 Convergence Conditions

When the stepsize depends on the history of the process, the stepsize itself becomes a random variable. This change requires some subtle modifications to our requirements for convergence (equations (11.15) and (11.16). For technical reasons our convergence criteria change to

$$\alpha_n \geq 0, \quad \text{almost surely,} \quad (11.24)$$

$$\sum_{n=0}^{\infty} \alpha_n = \infty \quad \text{almost surely,} \quad (11.25)$$

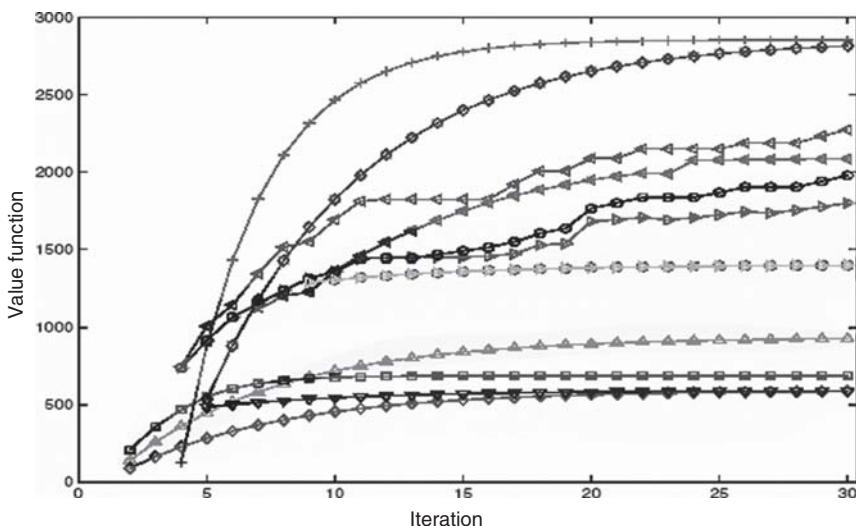


Figure 11.8 Different parameters can undergo significantly different initial rates.

$$\mathbb{E} \left\{ \sum_{n=0}^{\infty} (\alpha_n)^2 \right\} < \infty. \quad (11.26)$$

The condition “almost surely” (universally abbreviated “a.s.”) means that equation (11.25) holds for every sample path ω , and not just on average. More precisely, we mean every sample path ω that might actually happen (we exclude sample paths where the probability that the sample path would happen is zero).

For the reasons behind these conditions, go to our “Why does it work” section (Section 7.6). It is important to emphasize, however, that these conditions are completely unverifiable and are purely for theoretical reasons. The real issue with stochastic stepsizes is whether they contribute to the rate of convergence.

11.3.3 Recipes for Stochastic Stepsizes

The desire to find stepsize rules that adapt to the data has become a small cottage industry that has produced a variety of formulas with varying degrees of sophistication and convergence guarantees. This section provides a brief sample of these efforts.

To present our stochastic stepsize formulas, we need to define a few quantities. Recall that our basic updating expression is given by

$$\bar{\theta}^n = (1 - \alpha_{n-1})\bar{\theta}^{n-1} + \alpha_{n-1}\hat{\theta}^n.$$

$\bar{\theta}^{n-1}$ is our estimate of the next observation, given by $\hat{\theta}^n$. The difference between the estimate and the actual can be treated as the error, given by

$$\varepsilon^n = \bar{\theta}^{n-1} - \hat{\theta}^n.$$

We may wish to smooth the error in the estimate, which we designate by the function

$$S(\varepsilon^n) = (1 - \beta)S(\varepsilon^{n-1}) + \beta\varepsilon^n.$$

Some formulas depend on tracking changes in the sign of the error. This can be done using the indicator function

$$1_{\{\cdot\}} = \begin{cases} 1 & \text{if the logical condition } X \text{ is true,} \\ 0 & \text{otherwise.} \end{cases}$$

Thus $1_{\varepsilon^n \varepsilon^{n-1} < 0}$ indicates if the sign of the error has changed in the last iteration.

Below we summarize three classic rules. Kesten’s rule is the oldest and is perhaps the simplest illustration of a stochastic stepsize rule. Trigg’s formula is a simple rule widely used in the demand-forecasting community. Finally, the stochastic gradient adaptive stepsize rule enjoys a theoretical convergence proof, but is controlled by several tunable parameters that complicate its use in practice.

Kesten's Rule

Kesten's rule (Kesten, 1958) was one of the earliest stepsize rules that took advantage of a simple principle: If we are far from the optimal, the errors tend to all have the same sign; as we get close, the errors tend to alternate. Exploiting this simple observation, Kesten proposed the following simple rule:

$$\alpha_{n-1} = \frac{a}{a + K^n - 1}, \quad (11.27)$$

where a is a parameter to be calibrated. K^n counts the number of times that the sign of the error has changed, where we would use

$$K^n = \begin{cases} n & \text{if } n = 1, 2, \\ K^{n-1} + 1_{\{\varepsilon^n \varepsilon^{n-1} < 0\}} & \text{if } n > 2. \end{cases} \quad (11.28)$$

Kesten's rule is particularly well suited to initialization problems. It slows the reduction in the stepsize as long as the error exhibits the same sign (an indication that the algorithm is still climbing into the correct region). However, the stepsize declines monotonically. This is typically fine for most dynamic programming applications, but problems can be encountered in situations with delayed learning.

Trigg's Formula

Trigg's formula (Trigg and Leach, 1967) is given by

$$\alpha_n = \frac{|S(\varepsilon^n)|}{S(|\varepsilon^n|)}. \quad (11.29)$$

The formula takes advantage of the simple property that smoothing on the absolute value of the errors is greater than or equal to the absolute value of the smoothed errors. If there is a series of errors with the same sign, that can be taken as an indication that there is a significant difference between the true mean and our estimate of the mean, which means we would like larger stepsizes.

Stochastic Gradient Adaptive Stepsize Rule

This class of rules uses stochastic gradient logic to update the stepsize. We first compute

$$\psi^n = (1 - \alpha_{n-1})\psi^{n-1} + \varepsilon^n. \quad (11.30)$$

The stepsize is then given by

$$\alpha_n = [\alpha_{n-1} + v\psi^{n-1}\varepsilon^n]_{\alpha_-}^{\alpha_+}, \quad (11.31)$$

where α_+ and α_- are, respectively, upper and lower limits on the stepsize. $[\cdot]_{\alpha_-}^{\alpha_+}$ represents a projection back into the interval $[\alpha_-, \alpha_+]$, and v is a scaling factor. $\psi^{n-1}\varepsilon^n$ is a stochastic gradient that indicates how we should change the stepsize to improve the error. Since the stochastic gradient has units that are the square of

the units of the error, while the stepsize is unitless, ν has to perform an important scaling function. The equation $\alpha_{n-1} + \nu\psi^{n-1}\varepsilon^n$ can easily produce stepsizes that are larger than 1 or smaller than 0, so it is customary to specify an allowable interval (which is generally smaller than $(0,1)$). This rule has provable convergence, but in practice, ν , α_+ , and α_- all have to be tuned.

11.3.4 Experimental Notes

Throughout our presentation, we represent the stepsize at iteration n using α_{n-1} . For discrete, lookup table representations of value functions (as we are doing here), the stepsize should reflect how many times we have visited a specific state. If $n(S)$ is the number of times we have visited state S , then the stepsize for updating $\bar{V}(S)$ should be $\alpha_{n(S)}$. For notational simplicity we suppress this capability, but it can have a significant impact on the empirical rate of convergence.

A word of caution is offered when testing out stepsize rules. It is quite easy to test out these ideas in a controlled way in a simple spreadsheet on randomly generated data, but there is a big gap between showing a stepsize that works well in a spreadsheet and one that works well in specific applications. Stochastic stepsize rules work best in the presence of transient data where the degree of noise is not too large compared to the change in the signal (the mean). As the variance of the data increases, stochastic stepsize rules begin to suffer and simpler (deterministic) rules tend to work better.

11.4 OPTIMAL STEPSIZES FOR NONSTATIONARY TIME SERIES

Given the variety of stepsize formulas we can choose from, it seems natural to ask whether there is an optimal stepsize rule. Before we can answer such a question, we have to define exactly what we mean by it. Suppose that we are trying to estimate a parameter (e.g., a value of being in a state or the slope of a value function) that may be changing over time; we denote that parameter by θ^n . At iteration n , our estimate of θ^n , $\bar{\theta}^n$, is a random variable that depends on our stepsize rule. To express this dependence, let α represent a stepsize rule, and let $\bar{\theta}^n(\alpha)$ be the estimate of the parameter μ after iteration n using stepsize rule α . We would like to choose a stepsize rule to minimize

$$\min_{\alpha} \mathbb{E}(\bar{\theta}^n(\alpha) - \theta^n)^2. \quad (11.32)$$

Here the expectation is over the entire history of the algorithm and requires, in principle, knowing the true value of the parameter being estimated. If we could compute the expectation, we would obtain a deterministic stepsize sequence. In practice, we have to estimate certain parameters from data, and this gives us a stochastic stepsize rule.

There are other objective functions we could use. For example, instead of minimizing the distance to an unknown parameter sequence θ^n , we could solve the minimization problem

$$\min_{\alpha} \mathbb{E}_{\alpha} \left\{ (\bar{\theta}^n(\alpha) - \hat{\theta}^{n+1})^2 \right\}, \quad (11.33)$$

where we are trying to minimize the deviation between our prediction, obtained at iteration n , and the actual observation at $n + 1$. Here we are again proposing an unconditional expectation, which means that $\bar{\theta}^n(\alpha)$ is a random variable within the expectation. Alternatively, we could condition on our history up to iteration n :

$$\min_{\alpha} \mathbb{E}^n \left\{ (\bar{\theta}^n(\alpha) - \hat{\theta}^{n+1})^2 \right\}, \quad (11.34)$$

where \mathbb{E}^n means that we are taking the expectation given what we know at iteration n (i.e., $\bar{\theta}^n(\alpha)$ is a constant). (For readers familiar with the material in Section 5.9, we would write the expectation as $\mathbb{E} \left\{ (\bar{\theta}^n(\alpha) - \hat{\theta}^{n+1})^2 | \mathfrak{F}^n \right\}$, where \mathfrak{F}^n is the sigma-algebra generated by the history of the process up through iteration n .) In this formulation $\bar{\theta}^n(\alpha)$ is now deterministic at iteration n (because we are conditioning on the history up through iteration n), whereas in (11.33), $\bar{\theta}^n(\alpha)$ is random (since we are not conditioning on the history). The difference between these two objective functions is subtle but significant.

We begin our discussion of optimal stepsizes in Section 11.4.1 by addressing the case of estimating a constant parameter that we observe with noise. Section 11.4.2 considers the case where we are estimating a parameter that is changing over time, but where the changes have mean zero. Finally, Section 11.4.3 addresses the case where the mean may be drifting up or down with nonzero mean, a situation that we typically face when approximating a value function.

11.4.1 Optimal Stepsizes for Stationary Data

Suppose that we observe $\hat{\theta}^n$ at iteration n and that the observations $\hat{\theta}^n$ can be described by

$$\hat{\theta}^n = \theta + \varepsilon^n,$$

where θ is an unknown constant and ε^n is a stationary sequence of independent and identically distributed random deviations with mean 0 and variance σ^2 . We can approach the problem of estimating θ from two perspectives: choosing the best stepsize and choosing the best linear combination of the estimates. That is, we may choose to write our estimate $\bar{\theta}^n$ after n observations in the form

$$\bar{\theta}^n = \sum_{m=1}^n a_m^n \hat{\theta}_m.$$

For our discussion we will fix n and work to determine the coefficients a_m (recognizing that they can depend on the iteration). We would like our statistic to have

two properties: it should be unbiased, and it should have minimum variance (i.e., it should solve (11.32)). To be unbiased, it should satisfy

$$\begin{aligned}\mathbb{E} \left[\sum_{m=1}^n a_m \hat{\theta}_m \right] &= \sum_{m=1}^n a_m \mathbb{E} \hat{\theta}_m \\ &= \sum_{m=1}^n a_m \theta \\ &= \theta,\end{aligned}$$

which implies that we must satisfy

$$\sum_{m=1}^n a_m = 1.$$

The variance of our estimator is given by

$$\text{Var}(\bar{\theta}^n) = \text{Var} \left[\sum_{m=1}^n a_m \hat{\theta}_m \right].$$

We use our assumption that the random deviations are independent, which allows us to write

$$\begin{aligned}\text{Var}(\bar{\theta}^n) &= \sum_{m=1}^n \text{Var}[a_m \hat{\theta}_m] \\ &= \sum_{m=1}^n a_m^2 \text{Var}[\hat{\theta}_m] \\ &= \sigma^2 \sum_{m=1}^n a_m^2.\end{aligned}\tag{11.35}$$

Now we face the problem of finding a_1, \dots, a_n to minimize (11.35) subject to the requirement that $\sum_m a_m = 1$. This problem is easily solved using the Lagrange multiplier method. We start with the nonlinear programming problem

$$\min_{\{a_m\}} \sum_{m=1}^n a_m^2$$

subject to

$$\sum_{m=1}^n a_m = 1,\tag{11.36}$$

$$a_m \geq 0.\tag{11.37}$$

We relax constraint (11.36) and add it to the objective function

$$\min_{\{a_m\}} L(a, \lambda) = \sum_{m=1}^n a_m^2 - \lambda \left(\sum_{m=1}^n a_m - 1 \right)$$

subject to (11.37). We are now going to try to solve $L(a, \lambda)$ (known as the “Lagrangian”) and hope that the coefficients a are all nonnegative. If this is true, then we can take derivatives and set them equal to zero

$$\frac{\partial L(a, \lambda)}{\partial a_m} = 2a_m - \lambda. \quad (11.38)$$

The optimal solution (a^*, λ^*) would then satisfy

$$\frac{\partial L(a, \lambda)}{\partial a_m} = 0.$$

This means that at optimality

$$a_m = \frac{\lambda}{2},$$

which tells us that the coefficients a_m are all equal. Combining this result with the requirement that they sum to one gives the expected result:

$$a_m = \frac{1}{n}.$$

In other words, our best estimate is a sample average. From this (somewhat obvious) result we can obtain the optimal stepsize, since we already know that $\alpha_{n-1} = 1/n$ is the same as using a sample average.

This result tells us that if the underlying data are stationary, and we have no prior information about the sample mean, then the best stepsize rule is the basic $1/n$ rule. Using any other rule requires that there be some violation in our basic assumptions. In practice, the most common violation is that the observations are not stationary because they are derived from a process where we are searching for the best solution.

11.4.2 Optimal Stepsizes for Nonstationary Data—I

Suppose now that our parameter evolves over time (iterations) according to the process

$$\theta^n = \theta^{n-1} + \xi^n, \quad (11.39)$$

where $\mathbb{E}\xi^n = 0$ is a zero mean drift term with variance $(\sigma_\xi^2)^2$. As before, we measure θ^n with an error according to

$$\hat{\theta}^n = \theta^n + \varepsilon^n.$$

We want to choose a stepsize so that we minimize the mean squared error. This problem can be solved using the Kalman filter. The Kalman filter is a powerful recursive regression technique, but we adapt it here for the problem of estimating a single parameter. Typical applications of the Kalman filter assume that the variance of ξ^n , given by $(\sigma^\xi)^2$, and the variance of the measurement error, ε^n , given by σ^2 , are known. In this case the Kalman filter would compute a stepsize (generally referred to as the gain) using

$$\alpha_n = \frac{(\sigma^\xi)^2}{v^n + \sigma^2}, \quad (11.40)$$

where v^n is computed recursively using

$$v^n = (1 - \alpha_{n-1})v^{n-1} + (\sigma^\xi)^2. \quad (11.41)$$

Remember that $\alpha_0 = 1$, so we do not need a value of v^0 . For our application we do not know the variances, so these have to be estimated from data. We first estimate the bias using

$$\bar{\beta}^n = (1 - \eta_{n-1})\bar{\beta}^{n-1} + \eta_{n-1}(\bar{\theta}^{n-1} - \hat{\theta}^n), \quad (11.42)$$

where η_{n-1} is a simple stepsize rule such as the harmonic stepsize rule or McClain's formula. We then estimate the total error sum of squares using

$$\bar{v}^n = (1 - \eta_{n-1})\bar{v}^{n-1} + \eta_{n-1}(\bar{\theta}^{n-1} - \hat{\theta}^n)^2. \quad (11.43)$$

Finally, we estimate the variance of the error using

$$(\bar{\sigma}^n)^2 = \frac{\bar{v}^n - (\bar{\beta}^n)^2}{1 + \bar{\lambda}^{n-1}}, \quad (11.44)$$

where $\bar{\lambda}^{n-1}$ is computed using

$$\lambda^n = \begin{cases} (\alpha_{n-1})^2, & n = 1, \\ (1 - \alpha_{n-1})^2 \lambda^{n-1} + (\alpha_{n-1})^2, & n > 1. \end{cases}$$

We use $(\bar{\sigma}^n)^2$ as our estimate of σ^2 . We then propose to use $(\bar{\beta}^n)^2$ as our estimate of $(\sigma^\xi)^2$. This is purely an approximation, but experimental work suggests that it performs quite well, and it is relatively easy to implement.

11.4.3 Optimal Stepsizes for Nonstationary Data—II

In dynamic programming we are trying to estimate the value of being in a state (call it v) by \bar{v} which is estimated from a sequence of random observations \hat{v} . The problem we encounter is that \hat{v} might depend on a value function approximation that is steadily increasing, which means that the observations \hat{v} are nonstationary. Furthermore, unlike the assumption made by the Kalman filter that the mean of \hat{v} is varying in a zero-mean way, our observations of \hat{v} might be steadily increasing. This would be the same as assuming that $\mathbb{E}\xi = \mu > 0$ in the section above. In this section we derive the Kalman filter learning rate for biased estimates.

Our challenge is to devise a stepsize that strikes a balance between minimizing error (which prefers a smaller stepsize) and responding to the nonstationary data (which works better with a large stepsize). We return to our basic model

$$\hat{\theta}^n = \theta^n + \varepsilon^n,$$

where θ^n varies over time, but it might be steadily increasing or decreasing. This would be similar to the model in the previous section (equation 11.39) but where ξ^n has a nonzero mean. As before, we assume that $\{\varepsilon^n\}_{n=1,2,\dots}$ are independent and identically distributed with mean value of zero and variance, σ^2 . We perform the usual stochastic gradient update to obtain our estimates of the mean

$$\bar{\theta}^n(\alpha_{n-1}) = (1 - \alpha_{n-1})\bar{\theta}^{n-1} + \alpha_{n-1}\hat{\theta}^n. \quad (11.45)$$

We wish to find α_{n-1} that solves

$$\min_{\alpha_{n-1}} F(\alpha_{n-1}) = \mathbb{E} \left[\left(\bar{\theta}^n(\alpha_{n-1}) - \theta^n \right)^2 \right]. \quad (11.46)$$

It is important to realize that we are trying to choose α_{n-1} to minimize the *unconditional* expectation of the error between $\bar{\theta}^n$ and the true value θ^n . For this reason our stepsize rule will be deterministic, since we are not allowing it to depend on the information obtained up through iteration n .

We assume that the observation at iteration n is unbiased, which is to say,

$$\mathbb{E}[\hat{\theta}^n] = \theta^n. \quad (11.47)$$

But the smoothed estimate is biased because we are using simple smoothing on nonstationary data. We denote this bias as

$$\begin{aligned} \beta^{n-1} &= \mathbb{E} [\bar{\theta}^{n-1} - \theta^n] \\ &= \mathbb{E} [\bar{\theta}^{n-1}] - \theta^n. \end{aligned} \quad (11.48)$$

We note that β^{n-1} is the bias computed after iteration $n-1$ (i.e., after we have computed $\bar{\theta}^{n-1}$). β^{n-1} is the bias when we use $\bar{\theta}^{n-1}$ as an estimate of θ^n .

The variance of the observation $\hat{\theta}^n$ is computed as follows:

$$\begin{aligned}\text{Var}[\hat{\theta}^n] &= \mathbb{E}\left[\left(\hat{\theta}^n - \theta^n\right)^2\right] \\ &= \mathbb{E}\left[(\varepsilon^n)^2\right] \\ &= \sigma^2.\end{aligned}\tag{11.49}$$

We now have what we need to derive an optimal stepsize for nonstationary data with a mean that is steadily increasing (or decreasing). We refer to this as the *bias-adjusted Kalman filter* stepsize rule (or BAKF), in recognition of its close relationship to the Kalman filter learning rate. We state the formula in the following theorem:

Theorem 11.4.1 *The optimal stepsizes $(\alpha_m)_{m=0}^n$ that minimize the objective function in equation (11.46) can be computed using the expression*

$$\alpha_{n-1} = 1 - \frac{\sigma^2}{(1 + \lambda^{n-1})\sigma^2 + (\beta^n)^2},\tag{11.50}$$

where λ is computed recursively using

$$\lambda^n = \begin{cases} (\alpha_{n-1})^2, & n = 1, \\ (1 - \alpha_{n-1})^2 \lambda^{n-1} + (\alpha_{n-1})^2, & n > 1. \end{cases}\tag{11.51}$$

Proof We present the proof of this result because it brings out some properties of the solution that we exploit later when we handle the case where the variance and bias are unknown. Let $F(\alpha_{n-1})$ denote the objective function from the problem stated in (11.46):

$$F(\alpha_{n-1}) = \mathbb{E}\left[\left(\bar{\theta}^n(\alpha_{n-1}) - \theta^n\right)^2\right]\tag{11.52}$$

$$= \mathbb{E}\left[\left((1 - \alpha_{n-1})\bar{\theta}^{n-1} + \alpha_{n-1}\hat{\theta}^n - \theta^n\right)^2\right]\tag{11.53}$$

$$= \mathbb{E}\left[\left((1 - \alpha_{n-1})\left(\bar{\theta}^{n-1} - \theta^n\right) + \alpha_{n-1}\left(\hat{\theta}^n - \theta^n\right)\right)^2\right]\tag{11.54}$$

$$\begin{aligned}&= (1 - \alpha_{n-1})^2 \mathbb{E}\left[\left(\bar{\theta}^{n-1} - \theta^n\right)^2\right] + (\alpha_{n-1})^2 \mathbb{E}\left[\left(\hat{\theta}^n - \theta^n\right)^2\right] \\ &\quad + 2\alpha_{n-1}(1 - \alpha_{n-1}) \underbrace{\mathbb{E}\left[\left(\bar{\theta}^{n-1} - \theta^n\right)\left(\hat{\theta}^n - \theta^n\right)\right]}_I.\end{aligned}\tag{11.55}$$

Equation (11.52) is true by definition, while (11.53) is true by definition of the updating equation for $\bar{\theta}^n$. We obtain (11.54) by adding and subtracting $\alpha_{n-1}\theta^n$. To

obtain (11.55), we expand the quadratic term and then use the fact that the stepsize rule, α_{n-1} , is deterministic, which allows us to pull it outside the expectations. Then the expected value of the cross-product term, I , vanishes under the assumption of independence of the observations and the objective function reduces to the following form:

$$F(\alpha_{n-1}) = (1 - \alpha_{n-1})^2 \mathbb{E} \left[(\bar{\theta}^{n-1} - \theta^n)^2 \right] + (\alpha_{n-1})^2 \mathbb{E} \left[(\hat{\theta}^n - \theta^n)^2 \right]. \quad (11.56)$$

In order to find the optimal stepsize, α_{n-1}^* , that minimizes this function, we obtain the first-order optimality condition by setting $\partial F(\alpha_{n-1})/\partial \alpha_{n-1} = 0$, which gives us

$$-2(1 - \alpha_{n-1}^*) \mathbb{E} \left[(\bar{\theta}^{n-1} - \theta^n)^2 \right] + 2\alpha_{n-1}^* \mathbb{E} \left[(\hat{\theta}^n - \theta^n)^2 \right] = 0. \quad (11.57)$$

Solving this for α_{n-1}^* gives us the result

$$\alpha_{n-1}^* = \frac{\mathbb{E} \left[(\bar{\theta}^{n-1} - \theta^n)^2 \right]}{\mathbb{E} \left[(\bar{\theta}^{n-1} - \theta^n)^2 \right] + \mathbb{E} \left[(\hat{\theta}^n - \theta^n)^2 \right]}. \quad (11.58)$$

Recall that we can write $(\bar{\theta}^{n-1} - \theta^n)^2$ as the sum of the variance plus the bias squared using

$$\mathbb{E} \left[(\bar{\theta}^{n-1} - \theta^n)^2 \right] = \lambda^{n-1} \sigma^2 + (\beta^n)^2. \quad (11.59)$$

Using (11.59) and $\mathbb{E} \left[(\hat{\theta}^n - \theta^n)^2 \right] = \sigma^2$ in (11.58) gives us

$$\begin{aligned} \alpha_{n-1} &= \frac{\lambda^{n-1} \sigma^2 + (\beta^n)^2}{\lambda^{n-1} \sigma^2 + (\beta^n)^2 + \sigma^2} \\ &= 1 - \frac{\sigma^2}{(1 + \lambda^{n-1}) \sigma^2 + (\beta^n)^2}, \end{aligned}$$

which is our desired result (equation (11.50)). \square

A brief remark is in order. We have taken considerable care to make sure that α_{n-1} , used to update $\bar{\theta}^{n-1}$ to obtain $\bar{\theta}^n$, is a function of information available up through iteration $n-1$. Yet, the expression for α_{n-1} in equation (11.50) includes β^n . At this point in our derivation, however, β^n is deterministic, which means that it is known (in theory at least) at iteration $n-1$. Below we have to address the problem that we do not actually know β^n and have to estimate it from data.

Before we turn to the problem of estimating the bias and variance, there are two interesting corollaries that we can easily establish: the optimal stepsize if the data comes from a stationary series, and the optimal stepsize if there is no noise. Earlier we proved this case by solving a linear regression problem, and noted that the best estimate was a sample average, which implied a particular stepsize. Here we directly find the optimal stepsize as a corollary of our earlier result.

Corollary 11.4.1 *For a sequence with a static mean, the optimal stepsizes are given by*

$$\alpha_{n-1} = \frac{1}{n} \quad \forall n = 1, 2, \dots \quad (11.60)$$

Proof In this case, the mean $\theta^n = \mu$ is a constant. Therefore the estimates of the mean are unbiased, which means that $\beta^n = 0$, for all $t = 0, 1, \dots$. This allows us to write the optimal stepsize as

$$\alpha_{n-1} = \frac{\lambda^{n-1}}{1 + \lambda^{n-1}}. \quad (11.61)$$

Substituting (11.61) into (11.51) gives us

$$\alpha_n = \frac{\alpha_{n-1}}{1 + \alpha_{n-1}}. \quad (11.62)$$

If $\alpha_0 = 1$, it is easy to verify (11.60). \square

For the case where there is no noise ($\sigma^2 = 0$), we have the following:

Corollary 11.4.2 *For a sequence with zero noise, the optimal stepsizes are given by*

$$\alpha_{n-1} = 1 \quad \forall n = 1, 2, \dots \quad (11.63)$$

The corollary is proved by simply setting $\sigma^2 = 0$ in equation (11.50). As a final result we obtain

Corollary 11.4.3 *In general,*

$$\alpha_{n-1} \geq \frac{1}{n} \quad \forall n = 1, 2, \dots$$

Proof We leave this more interesting proof as an exercise to the reader (see exercise (11.11)). \square

Corollary 11.4.3 is significant since it establishes one of the conditions needed for convergence of a stochastic approximation method, namely that $\sum_{n=1}^{\infty} \alpha_n = \infty$.

An open theoretical question, as of this writing, is whether the BAKF stepsize rule also satisfies the requirement that $\sum_{n=1}^{\infty} (\alpha_n)^2 < \infty$.

The problem with using the stepsize formula in equation (11.50) is that it assumes that the variance σ^2 and the bias $(\beta^n)^2$ are known. This can be problematic in real instances, especially the assumption of knowing the bias, since knowing the bias is the same as knowing the real function. If we have this information, we do not need this algorithm.

As an alternative, we can try to estimate these quantities from data. Let

- $(\bar{\sigma}^2)^n$ = estimate of the variance of the error after iteration n ,
- $\bar{\beta}^n$ = estimate of the bias after iteration n ,
- \bar{v}^n = estimate of the variance of the bias after iteration n .

To make these estimates, we need to smooth new observations with our current best estimate, something that requires the use of a stepsize formula. We could attempt to find an optimal stepsize for this purpose, but it is likely that a reasonably chosen deterministic formula will work fine. One possibility is McClain's formula (equation 11.21):

$$\eta_n = \frac{\eta_{n-1}}{1 + \eta_{n-1} - \bar{\eta}}.$$

A limit point such as $\bar{\eta} \in (0.05, 0.10)$ appears to work well across a broad range of functional behaviors. The property of this stepsize that $\eta_n \rightarrow \bar{\eta}$ can be a strength, but it does mean that the algorithm will not tend to converge in the limit, which requires a stepsize that goes to zero. If this is needed, we suggest a harmonic stepsize rule:

$$\eta_{n-1} = \frac{a}{a + n - 1},$$

where a in the range between 5 and 10 seems to work quite well for many dynamic programming applications.

Care needs to be used in the early iterations. For example, if we let $\alpha_0 = 1$, then we do not need an initial estimate for $\bar{\theta}^0$ (a trick we have used throughout). However, since the formulas depend on an estimate of the variance, we still have problems in the second iteration. For this reason we recommend forcing η_1 to equal 1 (in addition to using $\eta_0 = 1$). We also recommend using $\alpha_n = 1/(n + 1)$ for the first few iterations, since the estimates of $(\bar{\sigma}^2)^n$, $\bar{\beta}^n$, and \bar{v}^n are likely to be very unreliable in the very beginning.

Figure 11.9 summarizes the entire algorithm. Note that the estimates have been constructed so that α_n is a function of information available up through iteration n . Figure 11.10 illustrates the behavior of the bias-adjusted Kalman filter stepsize rule for two signals: very low noise (Figure 11.10a) and with higher noise (Figure 11.10b). For both cases the signal starts small and rises toward an upper limit of 1.0 (on average). In both figures we also show the stepsize $1/n$. For the

Step 0. Initialization.

Step 0a. Set the baseline to its initial value, $\bar{\theta}_0$.

Step 0b. Initialize the parameters $-\bar{\beta}_0$, \bar{v}_0 and $\bar{\lambda}_0$.

Step 0c. Set initial stepsizes $\alpha_0 = \eta_0 = 1$, and specify the stepsize rule for η .

Step 0d. Set the iteration counter, $n = 1$.

Step 1. Obtain the new observation, $\hat{\theta}^n$.

Step 2. Smooth the baseline estimate.

$$\bar{\theta}^n = (1 - \alpha_{n-1})\bar{\theta}^{n-1} + \alpha_{n-1}\hat{\theta}^n.$$

Step 3. Update the following parameters:

$$\begin{aligned}\varepsilon^n &= \bar{\theta}^{n-1} - \hat{\theta}^n, \\ \bar{\beta}^n &= (1 - \eta_{n-1})\bar{\beta}^{n-1} + \eta_{n-1}\varepsilon^n, \\ \bar{v}^n &= (1 - \eta_{n-1})\bar{v}^{n-1} + \eta_{n-1}(\varepsilon^n)^2, \\ (\bar{\sigma}^2)^n &= \frac{\bar{v}^n - (\bar{\beta}^n)^2}{1 + \lambda^{n-1}}.\end{aligned}$$

Step 4. Evaluate the stepsizes for the next iteration.

$$\begin{aligned}\alpha_n &= \begin{cases} \frac{1}{n+1}, & n = 1, 2, \\ 1 - \frac{(\bar{\sigma}^2)^n}{\bar{v}^n}, & n > 2, \end{cases} \\ \eta_n &= \frac{a}{a+n-1} \quad \text{(note that this gives us } \eta_1 = 1\text{).}\end{aligned}$$

Step 5. Compute the coefficient for the variance of the smoothed estimate of the baseline.

$$\bar{\lambda}^n = (1 - \alpha_{n-1})^2\bar{\lambda}^{n-1} + (\alpha_{n-1})^2.$$

Step 6. If $n < N$, then $n = n + 1$ and go to step 1; else stop.

Figure 11.9 Bias-adjusted Kalman filter stepsize rule.

low-noise case, the stepsize stays quite large. For the high noise case, the stepsize roughly tracks $1/n$ (notice that it never goes below $1/n$).

11.5 OPTIMAL STEPSIZES FOR APPROXIMATE VALUE ITERATION

All the stepsize rules that we have presented so far are designed to estimate the mean of a nonstationary series. In this section we develop a stepsize rule that is specifically designed for approximate value iteration. We use as our foundation a dynamic program with a single state and single action. We use the same theoretical foundation that we used in Section 11.4. However, given the complexity of the derivation, we simply provide the expression.

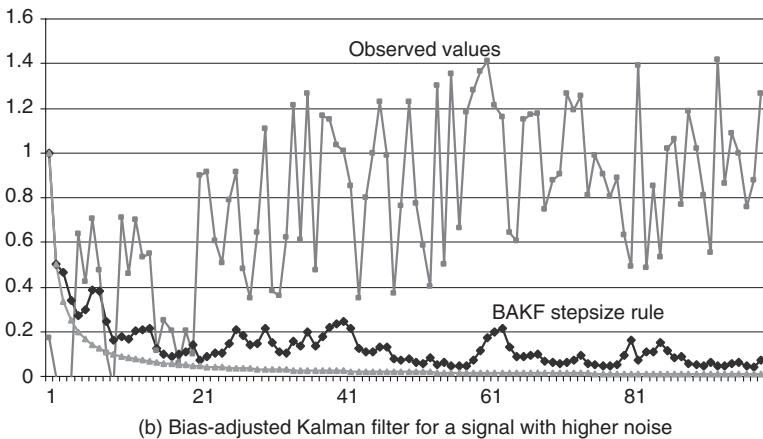
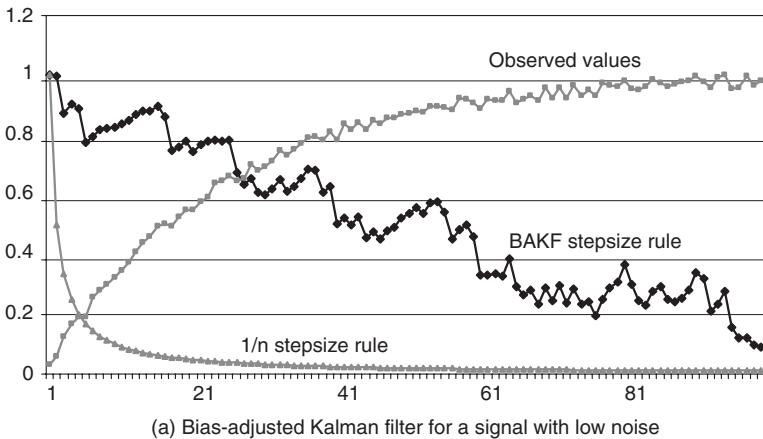


Figure 11.10 BAKF stepsize rule for low-noise (a) and high-noise (b). Each panel shows the signal, the BAKF stepsizes, and the stepsizes produced by the $1/n$ stepsize rule.

We start with the basic relationship for our single-state problem

$$v^n(\alpha_{n-1}) = (1 - (1 - \gamma)\alpha_{n-1})v^{n-1} + \alpha_{n-1}\hat{C}^n. \quad (11.64)$$

Let $c = \hat{C}$ be the expected one-period contribution for our problem, and let $\text{Var}(\hat{C}) = \sigma^2$. For the moment, we assume c and σ^2 are known. We next define the iterative formulas for two series, λ^n and δ^n , as follows:

$$\begin{aligned} \lambda^n &= \begin{cases} \alpha_0^2, & n = 1, \\ \alpha_{n-1}^2 + (1 - (1 - \gamma)\alpha_{n-1})^2\lambda^{n-1}, & n > 1. \end{cases} \\ \delta^n &= \begin{cases} \alpha_0, & n = 1 \\ \alpha_{n-1} + (1 - (1 - \gamma)\alpha_{n-1})\delta^{n-1}, & n > 1. \end{cases} \end{aligned}$$

It is possible to then show that

$$\mathbb{E}(v^n) = \delta^n c,$$

$$\text{Var}(v^n) = \lambda^n \sigma^2.$$

Let $v^n(\alpha_{n-1})$ be defined as in equation (11.64). Our goal is to solve the optimization problem

$$\min_{\alpha_{n-1}} \mathbb{E} \left[(v^n(\alpha_{n-1}) - \mathbb{E}\hat{v}^n)^2 \right]. \quad (11.65)$$

The optimal solution can be shown to be given by

$$\alpha_{n-1} = \frac{(1-\gamma)\lambda^{n-1}\sigma^2 + (1-(1-\gamma)\delta^{n-1})^2 c^2}{(1-\gamma)^2 \lambda^{n-1} \sigma^2 + (1-(1-\gamma)\delta^{n-1})^2 c^2 + \sigma^2}. \quad (11.66)$$

We refer to equation (11.64) as the *optimal stepsize for approximate value iteration* (OSAVI). Of course, it is only optimal for our single-state problem, and it assumes that we know the expected contribution per time period c , and the variance in the contribution \hat{C} , σ^2 .

OSAVI has some desirable properties. If $\sigma^2 = 0$, then $\alpha_{n-1} = 1$. Also, if $\gamma = 0$, then $\alpha_{n-1} = 1/n$. It is also possible to show that $\alpha_{n-1} \geq (1-\gamma)/n$ for any sample path.

All that remains is adapting the formula to more general dynamic programs with multiple states and where we are searching for optimal policies. We suggest the following adaptation. We propose to estimate a single constant \bar{c} representing the average contribution per period, averaged over all states. If \hat{C}^n is the contribution earned in period n , let

$$\begin{aligned} \bar{c}^n &= (1 - v_{n-1}) \bar{c}^{n-1} + v_{n-1} \hat{C}^n, \\ (\bar{\sigma}^n)^2 &= (1 - v_{n-1})(\bar{\sigma}^{n-1})^2 + v_{n-1}(\bar{c}^n - \hat{C}^n)^2. \end{aligned}$$

Here v_{n-1} is a separate stepsize rule. Our experimental work suggests that a constant stepsize works well, and that the results are quite robust with respect to the value of v_{n-1} . We suggest a value of $v_{n-1} = 0.2$. Now let \bar{c}^n be our estimate of c , and let $(\bar{\sigma}^n)^2$ be our estimate of σ^2 .

We could also consider estimating $\bar{c}^n(s)$ and $(\bar{\sigma}^n)^2(s)$ for each state so that we can estimate a state-dependent stepsize $\alpha_{n-1}(s)$. There is not enough experimental work to support the value of this strategy, and lacking this, we favor simplicity over complexity.

11.6 CONVERGENCE

A practical issue that arises with all stochastic approximation algorithms is that we simply do not have reliable, implementable stopping rules. Proofs of convergence

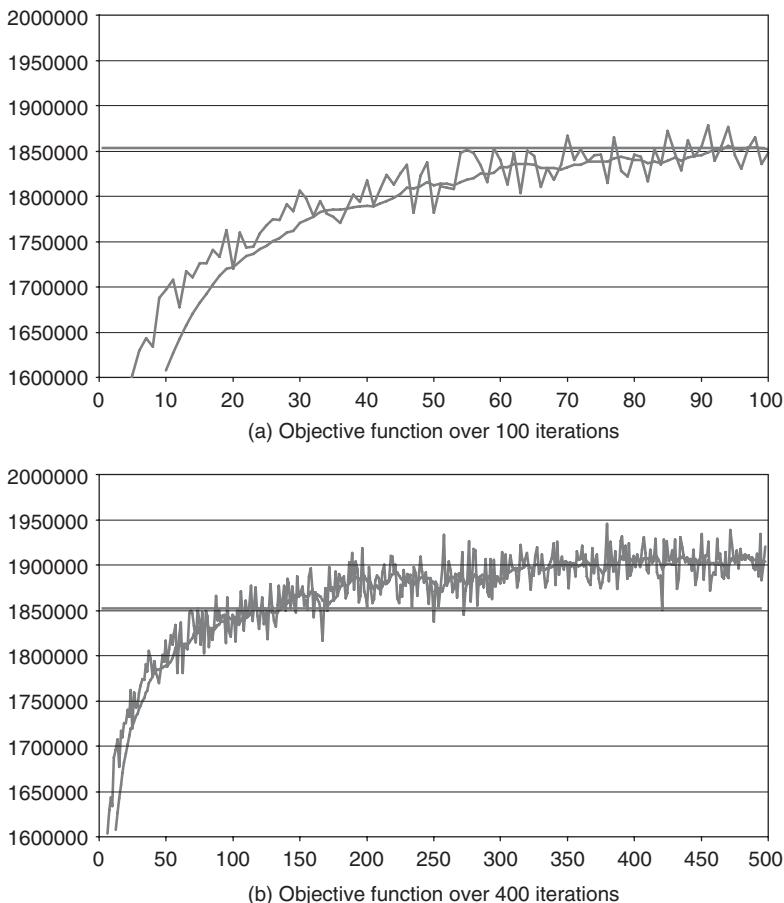


Figure 11.11 Objective function plotted over 100 iterations (a), showing “apparent convergence.” The same algorithm, continued over 400 iterations (b), showing significant improvement.

in the limit are an important theoretical property, but they provide no guidelines or guarantees in practice. A good illustration of the issue is given in Figure 11.11. Figure 11.11a shows the objective function for a dynamic program over 100 iterations (in this application a single iteration required approximately 20 minutes of CPU time). The figure shows the objective function for an ADP algorithm that was run for 100 iterations, at which point it appeared to be flattening out (evidence of convergence). Figure 11.11b is the objective function for the same algorithm run for 400 iterations. A solid line that shows the best objective function after 100 iterations is shown at the same level on the graph where the algorithm was run for 400 iterations. As we see, the algorithm was nowhere near convergence after 100 iterations.

We refer to this behavior as “apparent convergence,” and it is particularly problematic on large-scale problems where run times are long. Typically the number of

iterations needed before the algorithm “converges” requires a level of subjective judgment. When the run times are long, wishful thinking can interfere with this process.

Complicating the analysis of convergence in approximate dynamic programming is the behavior in some problems to go through periods of stability which are simply a precursor to breaking through to new plateaus. During periods of exploration an ADP algorithm might discover a strategy that opens up new opportunities, moving the performance of the algorithm to an entirely new level.

Special care has to be made in the choice of stepsize rule. In any algorithm using a declining stepsize, it is possible to show a stabilizing objective function simply because the stepsize is decreasing. This problem is exacerbated when using algorithms based on value iteration, where updates to the value of being in a state depend on estimates of the values of future states, which can be biased. We recommend that initial testing of an ADP algorithm start with inflated stepsizes. After getting a sense for the number of iterations needed for the algorithm to stabilize, decrease the stepsize (keeping in mind that the number of iterations required to convergence may increase) to find the right trade-off between noise and rate of convergence.

11.7 GUIDELINES FOR CHOOSING STEPSIZE FORMULAS

Given the plethora of strategies for computing stepsizes, it is perhaps not surprising that there is a need for general guidance when choosing a stepsize formula. Strategies for stepsizes are problem-dependent, and as a result any advice reflects the experience of the individual giving the advice.

We first suggest that the reader review the material in Section 11.1. Are you using some variation of approximate value iteration or Q -learning, or are you using a variation of approximate policy iteration for a finite or infinite horizon problem? The implications on stepsizes are significant, and appear to have been largely overlooked in the research community. Stepsizes for variations of policy iteration tend to be closer to $1/n$, while stepsizes for value iteration and Q -learning need to be larger, especially for discount factors closer to 1.

With this in mind, we offer the following general strategies for choosing stepsizes:

Step 1. Start with a constant stepsize α and test out different values. Problems with a relatively high amount of noise will require smaller stepsizes. Periodically stop updating the value function approximation and test your policy. Plot the results to see roughly how many iterations are needed before your results stop improving.

Step 2. Next try the harmonic stepsize $a/(a + n - 1)$. $a = 1$ produces the $1/n$ stepsize rule that is provably convergent but is likely to decline too quickly. $1/n$ absolutely should not be used for approximate value iteration or Q -learning. To choose a , look at how many iterations seemed to be needed

when using a constant stepsize. If 100 iterations appears to be enough for a stepsize of 0.1, then try $a \approx 10$, as it produces a stepsize of roughly 0.1 after 100 iterations. If you need 10,000 iterations, choose $a \approx 1000$. But you will need to tune a . An alternative rule is the polynomial stepsize rule $\alpha = 1/n^\beta$ with $\beta \in (0.5, 1]$ (we suggest 0.7 as a good starting point).

Step 3. Now try either the BAKF stepsize rule (Section 11.4.3) for policy iteration, LSPE and LSTD, or the OSAVI rule (Section 11.5) for approximate value iteration, TD learning, and Q -learning. These are both fairly robust, stochastic stepsize rules that adapt to the data. Both have a single tunable parameter (a stepsize), but we have found that the stepsize smoothing for OSAVI is more robust.

There is always the temptation to do something simple. A constant stepsize, or a harmonic rule, is extremely simple to implement. Keep in mind that either rule has a tunable parameter, and that the constant stepsize rule will not converge to anything (although the final solution may be acceptable). A major issue is that the best tuning of a stepsize not only depends on a problem but also on the parameters of a problem such as the discount factor. BAKF and OSAVI are more difficult to implement but are more robust to the setting of the single tunable parameter. Tunable parameters can be a major headache in the design of algorithms, and it is good strategy to absolutely minimize the number of tunable parameters your algorithm needs. Stepsize rules should be something you code once and forget about.

11.8 BIBLIOGRAPHIC NOTES

Sections 11.1–11.3 A number of different communities have studied the problem of “stepsizes,” including the business forecasting community (Brown, 1959; Holt et al., 1960; Brown, 1963; Giffin, 1971; Trigg, 1964; Gardner, 1983), artificial intelligence (Jaakkola et al., 1994; Darken and Moody, 1991; Darken et al., 1992; Sutton and Singh, 1994), stochastic programming (Kesten, 1958; Mirozahmedov and Uryasev, 1983; Pflug 1988; Ruszcynski and Syski, 1986), and signal processing (Goodwin and Sin, 1984; Benveniste et al., 1990; Stengel, 1994; Douglas and Mathews, 1995). The neural network community refers to “learning rate schedules”; see Haykin (1999). Even-dar and Mansour (2003) provide a thorough analysis of convergence rates for certain types of stepsize formulas, including $1/n$ and the polynomial learning rate $1/n^\beta$, for Q -learning problems. These sections are based on the presentation in George and Powell (2006).

Section 11.4 This section is based on the review in George and Powell (2006), along with the development of the optimal stepsize rule. Our proof that the optimal stepsize is $1/n$ for stationary data is based on Kmenta (1997).

Section 11.5 The optimal stepsize for approximate value iteration was derived in Ryzhov et al. (2009).

PROBLEMS

- 11.1** We are going to solve a classic stochastic optimization problem known as the newsvendor problem. Assume that we have to order x assets after which we try to satisfy a random demand D for these assets, where D is randomly distributed between 100 and 200. If $x > D$, we have ordered too much and we pay $5(x - D)$. If $x < D$, we have an underage, and we have to pay $20(D - x)$.

- (a) Write down the objective function in the form $\min_x \mathbb{E}f(x, D)$.
- (b) Derive the stochastic gradient for this function.
- (c) Find the optimal solution analytically [Hint: Take the expectation of the stochastic gradient, set it equal to zero and solve for the quantity $\mathbb{P}(D \leq x^*)$. From this, find x^* .]
- (d) Since the gradient is in units of dollars while x is in units of the quantity of the asset being ordered, we encounter a scaling problem. Choose as a stepsize $\alpha_{n-1} = \alpha_0/n$ where α_0 is a parameter that has to be chosen. Use $x^0 = 100$ as an initial solution. Plot x^n for 1000 iterations for $\alpha_0 = 1, 5, 10, 20$. Which value of α_0 seems to produce the best behavior?
- (e) Repeat the algorithm (1000 iterations) 10 times. Let $\omega = (1, \dots, 10)$ represent the 10 sample paths for the algorithm, and let $x^n(\omega)$ be the solution at iteration n for sample path ω . Let $\text{Var}(x^n)$ be the variance of the random variable x^n where

$$\bar{V}(x^n) = \frac{1}{10} \sum_{\omega=1}^{10} (x^n(\omega) - x^*)^2$$

Plot the standard deviation as a function of n for $1 \leq n \leq 1000$.

- 11.2** A customer is required by her phone company to pay for a minimum number of minutes per month for her cell phone. She pays 12 cents per minute of guaranteed minutes, and 30 cents per minute that she goes over her minimum. Let x be the number of minutes she commits to each month, and let M be the random variable representing the number of minutes she uses each month, where M is normally distributed with mean 300 minutes and a standard deviation of 60 minutes.

- (a) Write down the objective function in the form $\min_x \mathbb{E}f(x, M)$.
- (b) Derive the stochastic gradient for this function.
- (c) Let $x^0 = 0$ and choose as a stepsize $\alpha_{n-1} = 10/n$. Use 100 iterations to determine the optimum number of minutes the customer should commit to each month.

- 11.3** Show that if we use a stepsize rule $\alpha_{n-1} = 1/n$, then $\bar{\theta}^n$ is a simple average of $\hat{\theta}^1, \hat{\theta}^2, \dots, \hat{\theta}^n$ (thus proving equation 11.18).

- 11.4** We are going to again try to use approximate dynamic programming to estimate a discounted sum of random variables (we first saw this in Chapter 4):

$$F^T = \mathbb{E} \sum_{t=0}^T \gamma^t R_t,$$

where R_t is a random variable that is uniformly distributed between 0 and 100 (you can use this information to randomly generate outcomes, but otherwise, you cannot use this information). This time we are going to use a discount factor of $\gamma = 0.95$. We assume that R_t is independent of prior history. We can think of this as a single-state Markov decision process with no decisions.

- (a) Using the fact that $\mathbb{E}R_t = 50$, give the exact value for F^{100} .
 - (b) Propose an approximate dynamic programming algorithm to estimate F^T . Give the value function updating equation, using a stepsize $\alpha_t = 1/t$.
 - (c) Perform 100 iterations of the approximate dynamic programming algorithm to produce an estimate of F^{100} . How does this compare to the true value?
 - (d) Compare the performance of the following stepsize rules: Kesten's rule, the stochastic gradient adaptive stepsize rule (use $v = 0.001$), $1/n^\beta$ with $\beta = 0.85$, the Kalman filter rule, and the optimal stepsize rule. For each one, find both the estimate of the sum and the variance of the estimate.
- 11.5** Consider a random variable given by $R = 10U$ (which would be uniformly distributed between 0 and 10). We wish to use a stochastic gradient algorithm to estimate the mean of R using the iteration $\bar{\theta}^n = \bar{\theta}^{n-1} - \alpha_{n-1}(R^n - \bar{\theta}^{n-1})$, where R^n is a Monte Carlo sample of R in the n th iteration. For each of the stepsize rules below, use equation (7.12) to measure the performance of the stepsize rule to determine which works best, and compute an estimate of the bias and variance at each iteration. If the stepsize rule requires choosing a parameter, justify the choice you make (you may have to perform some test runs).
- (a) $\alpha_{n-1} = 1/n$.
 - (b) Fixed stepsizes of $\alpha_n = 0.05, 0.10$, and 0.20 .
 - (c) The stochastic gradient adaptive stepsize rule (equations (11.30–11.31)).
 - (d) The Kalman filter (equations (11.40–11.44)).
 - (e) The optimal stepsize rule (algorithm 11.9).
- 11.6** Repeat exercise 11.5 using

$$R^n = 10(1 - e^{-0.1n}) + 6(U - 0.5).$$

11.7 Repeat exercise 11.5 using

$$R^n = \left(\frac{10}{1 + e^{-0.1(50-n)}} \right) + 6(U - 0.5).$$

11.8 Let U be a uniform $[0, 1]$ random variable, and let

$$\mu^n = 1 - \exp(-\theta_1 n).$$

Now let $\hat{R}^n = \mu^n + \theta_2(U^n - .5)$. We wish to try to estimate μ^n using

$$\bar{R}^n = (1 - \alpha_{n-1})\bar{R}^{n-1} + \alpha_{n-1}\hat{R}^n.$$

In the exercises below, estimate the mean (using \bar{R}^n) and compute the standard deviation of \bar{R}^n for $n = 1, 2, \dots, 100$ for each of the following stepsize rules:

- $\alpha_{n-1} = 0.10$.
- $\alpha_{n-1} = a/(a + n - 1)$ for $a = 1, 10$.
- Kesten's rule.
- The bias-adjusted Kalman filter stepsize rule.

For each of the parameter settings below, compare the rules based on the average error (1) over all 100 iterations and (2) in terms of the standard deviation of \bar{R}^{100} .

- (a) $\theta_1 = 0, \theta_2 = 10$.
- (b) $\theta_1 = 0.05, \theta_2 = 0$.
- (c) $\theta_1 = 0.05, \theta_2 = 0.2$.
- (d) $\theta_1 = 0.05, \theta_2 = 0.5$.
- (e) Now pick the single stepsize that works the best on all four of the exercises above.

11.9 An oil company covers the annual demand for oil using a combination of futures and oil purchased on the spot market. Orders are placed at the end of year $t - 1$ for futures that can be exercised to cover demands in year t . If too little oil is purchased this way, the company can cover the remaining demand using the spot market. If too much oil is purchased with futures, then the excess is sold at 70 percent of the spot market price (it is not held to the following year—oil is too valuable and too expensive to store). To write down the problem, model the exogenous information using

\hat{D}_t = demand for oil during year t ,

\hat{p}_t^s = spot price paid for oil purchased in year t ,

$\hat{p}_{t,t+1}^f$ = futures price paid in year t for oil to be used in year $t + 1$.

The demand (in millions of barrels) is normally distributed with mean 600 and standard deviation of 50. The decision variables are given by

$\overline{\theta}_{t,t+1}^f$ = number of futures to be purchased at the end of year t to be used in year $t + 1$,

$\overline{\theta}_t^s$ = spot purchases made in year t .

- (a) Set up the objective function to minimize the expected total amount paid for oil to cover demand in a year $t + 1$ as a function of $\overline{\theta}_t^f$. List the variables in your expression that are not known when you have to make a decision at time t .
- (b) Give an expression for the stochastic gradient of your objective function. That is, what is the derivative of your function for a particular sample realization of demands and prices (in year $t + 1$)?
- (c) Generate 100 years of random spot and futures prices as follows:

$$\hat{p}_t^f = 0.80 + 0.10U_t^f,$$

$$\hat{p}_{t,t+1}^s = \hat{p}_t^f + 0.20 + 0.10U_t^s,$$

where U_t^f and U_t^s are random variables uniformly distributed between 0 and 1. Run 100 iterations of a stochastic gradient algorithm to determine the number of futures to be purchased at the end of each year. Use $\overline{\theta}_0^f = 30$ as your initial order quantity, and use as your stepsize $\alpha_t = 20/t$. Compare your solution after 100 years to your solution after 10 years. Do you think you have a good solution after 10 years of iterating?

- 11.10** The proof in Section 7.6.3 was performed assuming that μ is a scalar. Repeat the proof assuming that μ is a vector. You will need to make adjustments such as replacing assumption 2 with $\|g^n\| < B$. You will also need to use the triangle inequality, which states that $\|a + b\| \leq \|a\| + \|b\|$.
- 11.11** Prove Corollary 11.4.3.

Exploration Versus Exploitation

A fundamental challenge with approximate dynamic programming is that our ability to estimate a value function may require that we visit states just to estimate the value of being in (or near) the state. Should we make a decision because we think it is the best decision (based on our current estimate of the values of states the decision may take us to), or do we make a decision just to try something new? This is a decision we face in day-to-day life, so it is not surprising that we face this problem in our algorithms.

This choice is known in the approximate dynamic programming literature as the *exploration versus exploitation* problem. Do we make a decision to explore a state? Or do we “exploit” our current estimates of downstream values to make what we think is the best possible decision? It can cost time and money to visit a state, so we have to consider the future value of an action in terms of improving future decisions.

When we are using approximate dynamic programming, we always have to think about how well we know the value of being in a state, and whether a decision will help us learn this value better. The science of learning in approximate dynamic programming is quite young, with many unanswered questions. This chapter provides a peek into the issues and some of the simpler results that can be put into practice.

12.1 A LEARNING EXERCISE: THE NOMADIC TRUCKER

A nice illustration of the exploration versus exploitation problem is provided by our nomadic trucker example. Assume that the state of our nomadic trucker is his location R_t . From a location, our trucker is able to choose from a set of demands \hat{D}_t . Thus our state variable is $S = \{R_t, \hat{D}_t\}$. Our actions \mathcal{A}_t consist of the decision to assign to one of the loads in \hat{D}_t , or to do nothing and wait for another time period. Let $\hat{C}(S_t, a_t)$ be the contribution earned from being in location R_t and taking

Approximate Dynamic Programming: Solving the Curses of Dimensionality, Second Edition.

Warren B. Powell.

© 2011 John Wiley & Sons, Inc. Published 2011 by John Wiley & Sons, Inc.

action a_t . Any demands not covered in \hat{D}_t at time t are lost. After implementing action a_t , the driver will either stay in his current location (if he does nothing) or move to a location that corresponds to the destination of the load the driver moved in the set \hat{D}_t . Let R_t^a be this downstream location (which corresponds to the post-decision state variable for R_t). The post-decision state variable $S_t^a = (R_t^a)$. We assume that the action a deterministically determines the downstream destination, so $R_{t+1} = R_t^a$.

The driver makes his decision by solving

$$\hat{v}_t^n = \max_a (C(S_t, a) + \gamma \bar{V}^{n-1}(R_t^a)),$$

where $R_t^a = S^{M,a}(S_t, a)$ is the downstream location. Let a_t be the optimal action. We update the value of our previous, post-decision state using

$$\bar{V}^n(R_{t-1}^a) = (1 - \alpha_{n-1})\bar{V}^{n-1}(R_{t-1}^a) + \alpha_{n-1}\hat{v}_t^n.$$

We start by initializing the value of being in each location to zero, and use a pure exploitation strategy. If we simulate 500 iterations of this process, we produce the pattern shown in Figure 12.1a. Here the circles at each location are proportional to the value $\bar{V}^{500}(R)$ of being in that location. The small circles indicate places where the trucker never visited. Out of 30 cities our trucker has ended up visiting seven.

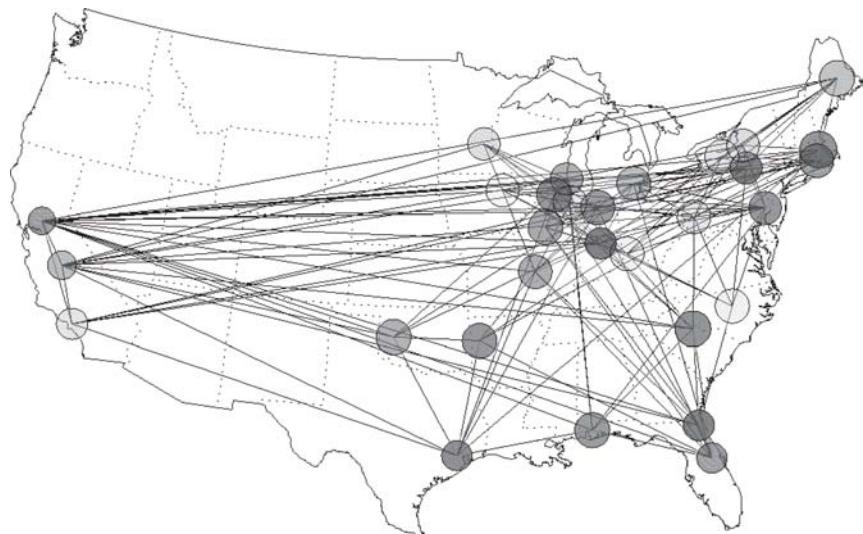
An alternative strategy is to initialize $\bar{V}^0(R)$ to a large number. For our illustration, where rewards tends to average several hundred dollars per iteration (we are using a discount factor of 0.80), we might initialize the value function to \$2000, which is higher than we would expect the optimal solution to be. The same strategy applied to visiting a state generally produces a reduction in the estimate of the value of being in the state. Not surprisingly, the logic tends to favor visiting locations that we have never visited before (or have visited the least). The resulting behavior is shown in Figure 12.1b. Here the pattern of lines shows that after 500 iterations, we have visited every city.

How do these strategies compare? We also ran an experiment where we estimated the value functions by using a pure exploration strategy, where we ran five iterations of sampling every single location. Then, for all three methods of estimating the value function, we simulated the policy produced by these value functions for 200 iterations. The results are shown in Table 12.1. Notice that for this example pure exploitation with a high initial estimate for the value function works better than when we use a low initial estimate, but estimating the value functions using a pure exploration strategy works best of all.

The difficulty with using a high initial estimate for the value function is that it leads us to visit every state (at least every state that we can reach). If the problem has a large state space, this will produce very slow convergence, since we have to visit many states just to determine that the states should not have been visited. In effect, exploitation with a high initial value will behave (at least initially) like pure exploration. For applications such as the nomadic trucker (where the state space is



(a) Low initial estimate of the value function



(b) High initial estimate of the value function

Figure 12.1 Effect of value function initialization on search process. (a) Low initial estimate, which produces limited exploration; (b) high initial estimate, which forces exploration of the entire state space.

Table 12.1 Comparison of pure exploitation with low and high initial estimates to one which uses exploration

Logic	Profits
Low initial estimate	120
High initial estimate	208
Explore all	248

relatively small), this works quite well. But the logic will not generalize to more complex problems.

12.2 AN INTRODUCTION TO LEARNING

Before we tackle the full complexity of learning within approximate dynamic programming, it helps to start with some fundamental concepts in learning, a topic we first saw in Chapter 7. Let x be a design, a path, a policy, or an action that we take in a dynamic program. For now, let μ_x be the true value of x , and let θ_x^n be our best estimate of μ_x after n measurements.

12.2.1 Belief Structures

The literature on learning can be separated between *frequentist* and *Bayesian* belief models. We touched on this earlier in Chapter 7, but we provide a brief review and a discussion for the context of approximate dynamic programming.

Frequentist Belief Model

In a frequentist model our belief about μ_x is formed purely from our observations. For example, let θ_x^n be our estimate of μ_x after n measurements, and let $\hat{\sigma}_x^{2,n}$ be our estimate of the variance of a measurement W_x of x . Also let N_x^n be the number of times we have measured x . We would update θ_x^n and $\hat{\sigma}_x^{2,n}$ recursively using

$$\theta_x^n = \begin{cases} \left(1 - \frac{1}{N_x^n}\right)\theta_x^{n-1} + \frac{1}{N_x^n}W_x^n & \text{if } x^n = x, \\ \theta_x^{n-1} & \text{otherwise;} \end{cases}$$

$$\hat{\sigma}_x^{2,n} = \begin{cases} \frac{1}{N_x^n}(W_x^n - \theta_x^{n-1})^2, & \text{if } x^n = x \text{ and } N_x^n = 2 \\ \frac{N_x^n - 2}{N_x^n - 1}\hat{\sigma}_x^{2,n-1}x + \frac{1}{N_x^n}(W_x^n - \theta_x^{n-1})^2, & \text{if } x^n = x \text{ and } N_x^n > 2 \\ \hat{\sigma}_x^{2,n-1} & x^n \neq x. \end{cases}$$

The statistics θ_x^n and $\hat{\sigma}_x^{2,n}$ are random variables only because the observations W_x^n are random. For example, the variance of θ_x^n is computed using

$$\overline{\sigma}_x^{2,n} = \frac{1}{N_x^n}\hat{\sigma}_x^{2,n}.$$

Bayesian Belief Model

In a Bayesian model we view μ_x as a random variable, where we assume that we already know an initial distribution of belief about μ_x . For example, we often assume that our initial distribution of belief is normal with mean θ_x^0 and precision β_x^0 (recall that the precision is the inverse of the variance). Then we make each measurement of W_x (with precision β_x^W) using

$$\begin{aligned}\mu_x^{n+1} &= \begin{cases} \frac{\beta_x^n \mu_x^n + \beta_x^W W_x^{n+1}}{\beta_x^n + \beta_x^W} & \text{if } x^n = x, \\ \mu_x^n & \text{otherwise;} \end{cases} \\ \beta_x^{n+1} &= \begin{cases} \beta_x^n + \beta_x^W & \text{if } x^n = x, \\ \beta_x^n & \text{otherwise.} \end{cases}\end{aligned}$$

After n measurements, our uncertainty about μ_x reflects a mixture of our confidence in our initial belief (captured by β_x^0) and the precision in the measurements W_x^n .

Belief Architectures

As we saw in Chapters 7 and 8, we can represent our belief about the value of a state, or the value of a state-action pair, using three fundamental architectures: lookup table, parametric (especially linear models with basis functions), and nonparametric. Each of these carries specific implications in terms of our need to explore and learn. Lookup tables have a parameter (a value) for each state, and as a result algorithms for learning often carry requirements that we visit each state infinitely often if we want to guarantee convergence. Parametric models require that we visit states with sufficient diversity so that we can statistically identify the regression parameters. Nonparametric models are closer in behavior to lookup tables, but with continuous states the requirement to visit (discrete) states infinitely often is replaced with conditions to visit neighborhoods of regions with sufficient density.

Discussion

Bayesian belief models are fairly common in the optimal learning literature, but less so in approximate dynamic programming. One major reason is that we may not have a natural prior distribution of belief about the value of being in a state. While the ability to express a prior distribution of belief can be a major benefit, it can also represent a major data hurdle if this is neither readily apparent nor easy to express. Bayesians often address this issue using a method known as *empirical Bayes*, where an initial sample of observations is used to build the prior distribution.

12.2.2 Online versus Offline Learning

We begin by introducing two fundamental problem classes in learning, which we refer to as offline learning and online learning.

Offline Learning

Imagine that we have a budget of N measurements, after which we have to use what we have learned to pick the best design. A design could be the layout of a manufacturing facility, the structure of a molecular compound for a new drug, or a policy for optimizing a dynamic program. If x is a design and $\theta_x^{\pi, N}$ is our estimate of how well x performs based on N measurements that were allocated using policy π (in this setting, π is the rule for making measurements), then a sample estimate of how well policy π has performed is given by

$$F^\pi = \max_x \mu_{x^N}, \quad (12.1)$$

where $x^N = \arg \max_x \theta_x^N$ is the choice that appears to be best based on our estimate θ_x^N of each alternative after N measurements. We write the problem of finding the best measurement policy using

$$\max_\pi \mathbb{E}^\pi F^\pi.$$

Here we write the expectation \mathbb{E}^π as a function of π with the understanding that the measurement policy determines what we measure next.

Offline learning when the measurements x are discrete and finite is known as the *ranking and selection problem*. If x is continuous, the problem is generally referred to as *stochastic search*.

Online Learning

Imagine again that we have again a budget of N measurements, but now we have to learn as we go. Perhaps we have to find the best path to get from our apartment to our new job, or we have to find the best price for selling a product, or we have to determine the best dosage for a new patient. In all these cases we have to learn as we go. As a result we care about the rewards we earn (or costs we incur) as we proceed. Let $\theta_x^{\pi, n}$ be our estimate of the value of alternative x after n measurements while using policy π to determine what to measure next. Our objective function is given by

$$\max_\pi \mathbb{E}^\pi \sum_{n=1}^N \gamma^n \mu_{x^n}, \quad (12.2)$$

where x^n is the measurement decision we make when using policy π . For example, we could choose to measure $x^n = \arg \max_x \theta_x^n$, which is to say, we choose to measure the alternative that appears to be best given what we know after n measurements. We should also note that the expectation in equation (12.2) is over two sets of random variables: a) the true values μ_x giving the value of each alternative and b) the observations that are made.

Online learning problems are conventionally referred to as bandit problems. If the measurements x are discrete and finite, the problem is known as the *multi-armed bandit problem*, described in greater detail below.

Online versus Offline Dynamic Programming

Dynamic programs also arise in online and offline settings. An online setting arises when we are trying to optimize a problem as it happens. Perhaps we want to optimize the control of a thermostat or the storage of energy in a battery while a system is in operation. We may be interested in testing a strategy that does not look good to us now but might be better than we think. So it means that we have to run the risk of incurring lower contributions (or higher costs) while learning about the benefit of a new policy.

At the same time there are many settings where we have the ability to learn a value function approximation using simulations. During these simulations we do not really care if we make poor decisions while learning a better value function. Once the simulations are finished, we use the value function approximations in a production setting.

12.2.3 Physical States versus Belief States

Imagine that we are taking measurements of a set of discrete alternatives $x \in \mathcal{X}$. In Chapter 7 we introduced the idea of a *belief state* or state of knowledge K^n that captures what we know (or believe) about the performance of each choice x . If we are using a frequentist perspective, our knowledge state can be represented using

$$K^n = (\theta_x^n, \hat{\sigma}_x^{2,n}, N_x^n)_{x \in \mathcal{X}}.$$

If we are using a Bayesian belief model, then we would represent our state of knowledge using

$$K^n = (\theta_x^n, \hat{\sigma}_x^{2,n})_{x \in \mathcal{X}}.$$

If we are using a parametric model with a linear architecture (with a frequentist perspective), our state of knowledge would be given by

$$K^n = (\theta^n, B^n).$$

If we are using a nonparametric model, the knowledge state is literally the entire history of observations,

$$K^n = (\hat{v}^i, x^i)_{i=1}^n.$$

We could just as easily let S^n be our state of knowledge. However, now consider a problem where we have a person, vehicle, sensor, or storage device that exhibits what we would call a *physical state*. In Section 12.1 the physical state of our nomadic trucker might be its location, although it could also include other attributes of the truck (e.g., fuel level and maintenance status). Perhaps we are moving around a graph, and as we traverse arcs in the graph we collect information about the cost of the arc. We may have a belief about the expected cost, but as we gain experience,

we are willing to modify our beliefs. However, if we are at node i , we can only observe links that are emanating from node i . If we are at node i and choose link (i, j) , we observe a realization of \hat{C}_{ij} , but in the process we also transition to node j .

Since the state of our system captures everything, let R^n be our physical state after n transition. In our graph problem, R^n would capture the node at which we are located. Let \bar{c}_{ij}^n be our estimate of the expected cost on link (i, j) , and let $\sigma_{ij}^{2,n}$ be our estimate of the variance. We could say that $K^n = (\bar{c}^n, \sigma^{2,n})_{ij}$ is our state of knowledge about the network, while R^n is our physical state (the node where we are currently positioned). The state of our system would be written

$$S^n = (R^n, K^n).$$

Now we have a problem with a physical state and a knowledge state. Reflecting the confusion in the research community over the definition of a state variable, some authors (including Bellman) refer to S^n as the *hyperstate*, but given our definitions (see Section 5.4), this is a classic state variable since it captures everything we need to know to model the future evolution of our system.

12.3 HEURISTIC LEARNING POLICIES

Much of the challenge of estimating a value function is identical to that facing any statistician trying to fit a model to data. The biggest difference is that in dynamic programming we may choose what data to collect by controlling what states to visit. Further complicating the problem is that it takes time (and may cost money) to visit these states to collect the information. Do we take the time to visit the state and better learn the value of being in the state? Or do we live with what we know?

Below we review several simple strategies (which we first saw in Chapter 7), any of which can be effective for specific problem classes.

12.3.1 Pure Exploitation

A pure exploitation strategy assumes that we have to make decisions by solving

$$x_t^n = \arg \max_{x_t \in \mathcal{X}_t} (C(S_t^n, x_t) + \bar{V}_t^{n-1}(S^{M,x}(S_t^n, x_t))).$$

This decision then determines the next state that we visit. Some authors refer to this as a *greedy* strategy, since we are doing the best that we think we can given what we know.

A pure exploitation strategy may be needed for practical reasons. For example, consider a large resource allocation problem where we have a resource vector R_t that we act on with a decision vector x_t . For some applications the dimensionality

of R_t may be in the thousands, while x_t may be in the tens of thousands. For problems of this size, exploration may be of little or no value (what is the point in randomly sampling even a million states out of a population of 10^{100} ?). Exploitation strategies focus our attention on states that offer some likelihood of being states we are interested in.

The problem with pure exploitation is that it is easy to become stuck in a local solution because we have poor estimates of the value of being in some states. While it is not hard to construct small problems where this problem is serious, the errors can be substantial on virtually any problem that lacks specific structure that can be exploited to ensure convergence. As a rule, optimal solutions are not available for large problems, so we have to be satisfied with doing the best we can. But just because your algorithm appears to have converged, do not fool yourself into believing that you have reached an optimal, or even near-optimal, solution.

12.3.2 Pure Exploration

Let us use an exogenous process (e.g., random selection) to choose either a state to visit or a state-action pair (which leads to a state). Once in a state, we sample information and obtain an estimate of the value of being in the state, which is then used to update our estimate.

In a pure exploration strategy we can guarantee that we visit every state, or at least have a chance of visiting every state. This property is often used to produce convergence proofs. We need to remember that some problems have 10^{100} states or more, so even if we run a million iterations, we may sample only a fraction of the complete state space. For problems with large state spaces, random exploration (i.e., choosing states or actions purely at random) is unlikely to work well. This is a reminder that provably convergent algorithms can work terribly in practice.

The amount of exploration we undertake depends in large part on the cost of collecting the information (how much time does it take to run each iteration) and the value of the information we collect. It is important to explore the most important states (of course, we may not know which states are important).

12.3.3 Persistent Excitation

In engineering applications we often encounter problems where states and actions are continuous, and possibly vector-valued. For example, S_t might be the coordinates (three dimensions) and velocity (three dimensions) of a robot or aircraft. An action x_t might be a three dimensional force applied to the object, where we might be willing to assume that the effect of the action is deterministic. We might write our transition equation using

$$S_{t+1} = S_t + Ax_t,$$

where A is a suitably dimensioned matrix that translates our three-dimensional force to our six-dimensional state. We can force our system to visit other states by adding a noise term

$$S_{t+1} = S_t + Ax_t + \varepsilon_{t+1},$$

where ε_{t+1} is a suitably dimensioned column vector of random variables that perturbs the state that we thought we would visit. This strategy is known in the control theory community as *persistent excitation*.

12.3.4 Epsilon-Greedy Exploration

A common strategy is to mix exploration and exploitation. We might specify an exploration rate ϵ where ϵ is the fraction of iterations where decisions should be chosen at random (exploration). The intuitive appeal of this approach is that we maintain a certain degree of forced exploration, while the exploitation steps focus attention on the states that appear to be the most valuable.

In practice, using a mix of exploration steps only adds value for problems with relatively small state or action spaces. The only exception arises when the problem lends itself to an approximation that is characterized by a relatively small number of parameters. Otherwise, performing, say, 1000 exploration steps for a problem with 10^{100} states may prove to have little or no practical value.

A useful variation is to let ϵ decrease with the number of iterations. For example, let

$$\epsilon^n(s) = \frac{c}{N^n(s)},$$

where $0 < c < 1$ and where $N^n(s)$ is the number of times we have visited state s by iteration n . When we explore, we will choose an action a with probability $1/|\mathcal{A}|$. We choose to explore with probability $\epsilon^n(s)$. This means that the probability we choose decision a when we are in state s , given by $P^n(s, a)$, is at least $\epsilon^n(s)/|\mathcal{A}|$. This guarantees that we will visit every state infinitely often since

$$\sum_{n=1}^{\infty} P^n(s, a) = \frac{\sum_{n=1}^{\infty} \epsilon^n(s)}{|\mathcal{A}|} = \infty.$$

12.3.5 Interval Estimation

Interval estimation sets the value of a decision to the 90th or 95th percentile of the estimate of the value of a decision. Thus our decision problem would be

$$\max_a (\theta_a^n + z_\alpha \bar{\sigma}_a^n).$$

Here $\bar{\sigma}_a^n$ is our estimate of the standard deviation of θ_a^n . As the number of times we observe action a goes to infinity, $\bar{\sigma}_a^n$ goes to zero. The parameter z_α is best viewed as a tunable parameter, although it is common to choose values around 2 or 3. This has the effect of valuing each action at its 90th or 95th percentile.

12.3.6 Upper Confidence Bound Sampling Algorithm

The upper confidence bound (UCB) uses the same idea as interval estimation, but with a somewhat different scaling factor for the standard deviation. If n is our iteration counter (which means the total number of samples across all decisions), and N_a^n is the number of times we have sampled action a after n iterations, then the UCB algorithm makes a decision by solving

$$\max_a \left(\theta_a^n + C^{\max} \sqrt{\frac{2 \ln n}{N_a^n}} \right),$$

where C^{\max} is the maximum possible contribution, and N_a^n is the number of times we have sampled action a . Instead of C^{\max} , we might use an estimate of the 95th percentile for \hat{C} (not the 95th percentile of θ^n).

12.3.7 Boltzmann Exploration

The problem with exploration steps is that you are choosing an action $a \in \mathcal{A}$ at random. Sometimes this means that you are choosing really poor decisions where you are learning nothing of value. An alternative is *Boltzmann exploration* where from state s , an action a is chosen with a probability proportional to the estimated value of an action. For example, let $Q(s, a) = C(s, a) + \bar{V}^n(s, a)$ be the value of choosing decision a when we are in state s . Using Boltzmann exploration, if we are in state s at iteration n we would choose action a with probability

$$P^n(s, a) = \frac{e^{Q(s,a)/T}}{\sum_{a' \in \mathcal{A}} e^{Q(s,a')/T}}.$$

T is known as the temperature, since in physics (where this idea has its origins), electrons at high temperatures are more likely to bounce from one state to another (we generally replace $1/T$ with a scaling factor β). As the parameter T increases, the probability of choosing different actions becomes more uniform. As $T \rightarrow 0$, the probability of choosing the best decision approaches 1.0. It makes sense to start with T relatively large and steadily decrease it as the algorithm progresses.

It is common to write $P^n(s, a)$ in the form

$$P^n(s, a) = \frac{e^{\beta^n(s) Q(s,a)}}{\sum_{a' \in \mathcal{A}} e^{\beta^n(s) Q(s,a')}}. \quad (12.3)$$

where $\beta^n(s)$ is a state-specific scaling coefficient. Let $N^n(s)$ be the number of visits to state s after n iterations. We can ensure convergence if we choose $\beta^n(s)$

so that $P^n(s, a) \geq c/n$ for some $c \leq 1$ (which we choose below). This means that we want

$$\begin{aligned} \frac{e^{\beta^n(s)Q(S,a)}}{\sum_{a' \in \mathcal{A}} e^{\beta^n(s)Q(S,a')}} &\geq \frac{c}{N^n(s)}, \\ N^n(s)e^{\beta^n(s)Q(S,a)} &\geq c \sum_{a' \in \mathcal{A}} e^{\beta^n(s)Q(S,a')}. \end{aligned}$$

Let $|\mathcal{A}|$ be the number of actions, and let

$$a^{\max}(S) = \arg \max_{a \in \mathcal{A}} Q(S, a)$$

be the decision that returns the highest value of $Q(S, a)$. Then we can require that $\beta^n(s)$ satisfy

$$\begin{aligned} N^n(s)e^{\beta^n(s)Q(S,a)} &\geq c|\mathcal{A}|e^{\beta^n(s)Q(S,a^{\max}(S))}, \\ \frac{N^n(S)}{c|\mathcal{A}|} &\geq e^{\beta^n(s)(Q(S,a^{\max}(S))-Q(S,a))}, \\ \ln N^n(s) - \ln c|\mathcal{A}| &\geq \beta^n(s)(Q(S,a^{\max}(S))-Q(S,a)). \end{aligned}$$

Now choose $c = 1/|\mathcal{A}|$ so that $\ln c|\mathcal{A}| = 0$. Solving for $\beta^n(s)$ gives us

$$\beta^n(s) = \frac{N^n(s)}{\delta Q(s)},$$

where $\delta Q = \max_{a \in \mathcal{A}} |Q(S, a^{\max}(S)) - Q(S, a)|$. Since δQ is bounded, this choice guarantees that $\beta^n(s) \rightarrow \infty$, which means that in the limit we will be only visiting the best decision (a pure greedy policy).

Boltzmann exploration provides for a more rational choice of decision that focuses our attention on better decisions, but provides for some degree of exploration. The probability of exploring decisions that look really bad is small, limiting the amount of time that we spend exploring poor decisions (but this assumes we have reasonable estimates of the value of these state/action pairs). Those that appear to be of lower value are selected with a lower probability. We focus our energy on the decisions that appear to be the most beneficial, but provide for intelligent exploration. However, this idea only works for problems where the number of decisions \mathcal{A} is relatively small, and it requires that we have some sort of estimate of the values $Q(S, a)$.

12.3.8 Remarks

The trade-off between exploration and exploitation is illustrated in Figure 12.2 where we are estimating the value of being in each state for a small problem with a few dozen states. For this problem we are able to compute the exact value function, which allows us to compute the value of a policy using the approximate value

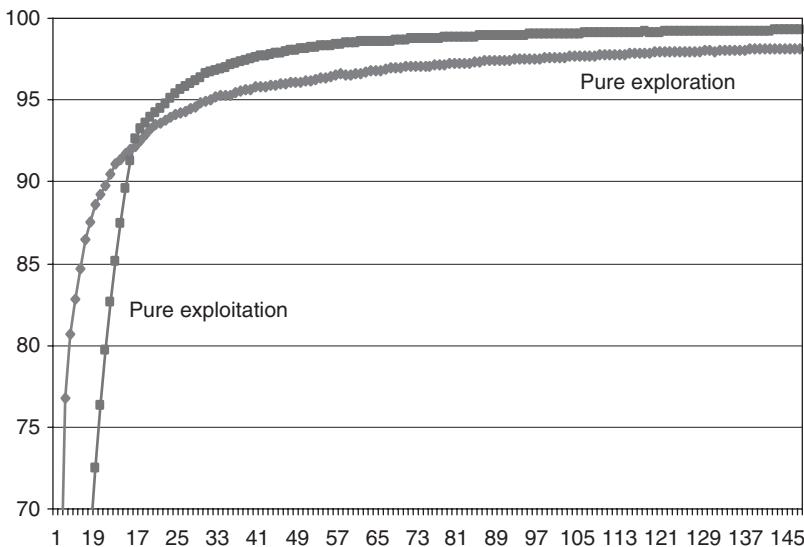


Figure 12.2 Pure exploration outperforms pure exploitation initially, but slows as the iterations progress.

function as a percentage of the optimal. This graph nicely shows that pure exploration has a much faster initial rate of convergence, whereas the pure exploitation policy works better as the function becomes more accurate.

This behavior, however, is very problem-dependent. The value of any exploration strategy drops as the number of parameters increases. If a mixed strategy is used, the best fraction of exploration iterations depends on the characteristics of each problem and may be difficult to ascertain without access to an optimal solution. Tests on smaller, computationally tractable problems (where exploration is more useful) will not tell us the right balance for larger problems.

Exploration strategies are sometimes chosen because they guarantee that, in the limit, actions will be chosen infinitely often. This property is then used to prove that estimates will reach their true value. Unfortunately, “provably convergent” algorithms can produce extremely poor solutions. A proof of convergence may bring some level of comfort, but in practice, it may provide almost no guarantee at all of a high-quality solution.

A serious problem arises when the action space is large (or infinite). In particular, consider problems where the decision is a vector x_t . Even if x_t is discrete, the number of potential decisions may be extremely large. For example, it is not that hard to create action spaces with 10^{100} actions (the curse of dimensionality creates problems like these fairly easily). By contrast, running a dynamic programming algorithm for more than 100,000 iterations can get very expensive, even for small applications. As of this writing, the literature on effective exploration strategies is largely limited to small action spaces.

12.4 GITTINS INDICES FOR ONLINE LEARNING

For the most part the best balance of exploration and exploitation is ad hoc, problem-dependent and highly experimental. There is, however, one body of theory that offers some very important insights into how to best make the trade-off between exploring and exploiting. This theory is often referred to as *multi-armed bandits*, which is the name given to the underlying mathematical model, or *Gittins Indices*, that refers to an elegant method for solving the problem.

There are two perspectives of the information acquisition problem. First, this is a problem in its own right that can be modeled and solved as a dynamic program. There are many situations where the problem is to collect information, and we have to balance the cost of collecting information against the benefits of using it. The second perspective is that this is a problem that arises in virtually every approximate dynamic programming problem. As we illustrated with our nomadic trucking application (which is, after all, a generic discrete dynamic program), we have to visit states to estimate the value of being in a state. We depend on our value function approximation $\bar{V}_t(S_t^a)$ to approximate the value of a decision that takes us to state S_t^a . Our estimate may be low, but the only way we are going to find out is to actually visit the state.

In this section we begin with a pure application of information acquisition and then make the transition to its application in a broad range of dynamic programming applications.

12.4.1 Foundations

Consider the problem faced by a gambler playing a set of slot machines (often referred to as “one-armed bandits”) in a casino. Now pretend that the probability of winning is different for each slot machine, but we do not know what these probabilities are. We can, however, obtain information about the probabilities by playing a machine and watching the outcomes. Because our observations are random, the best we can do is to obtain statistical estimates of the probabilities, but as we play a machine more, the quality of our estimates improves.

Since we are looking at a set of slot machines, the problem is referred to as the multi-armed bandit problem. This is a pure exercise in information acquisition, since after every round, our player is faced with the same set of choices. Contrast this situation with most dynamic programs that involve allocating an asset where making a decision changes the attribute (state) of the asset. In the multi-armed bandit problem, after every round the player faces the same decisions with the same rewards. All that has changed is what she *knows* about the system.

This problem, which is extremely important in approximate dynamic programming, provides a nice illustration of what might be called the *knowledge state* (some refer to it as the *information state*). The difference between the state of the resource (in this case, the player) and the state of what we know has confused authors since Bellman first raised the issue. The vast majority of papers in dynamic programming implicitly assume that the state variable is the state of the resource or

the physical state of the system. This is precisely the reason that our presentation in Chapter 5 adopted the term ‘‘resource state’’ to be clear about what we were referring to. In other areas of engineering we might use the term ‘‘physical state.’’

In our multi-armed bandit problem let \mathcal{A} be the set of slot machines, and let \hat{C}_a be the random variable that gives the amount that we win if we play bandit a . Most of our presentation assumes that \hat{C}_a is normally distributed. Let μ_a be the true mean of \hat{C}_a (which is unknown), and let σ_a^2 be the variance (which we may assume is known or unknown). Let $(\theta_a^n, \hat{\sigma}_a^{2,n})$ be our estimate of the mean and variance of \hat{C}_a after n iterations. Under our assumption of normality the mean and variance completely determine the distribution.

We next need to specify our transition equations. When we were managing physical assets, we used equations such as $R_{t+1} = [R_t + a_t - D_{t+1}]^+$ to capture the quantity of assets available. In our bandit problem we have to model how our distribution of belief evolves over time.

The theory surrounding bandit problems, and much of the literature on information collection, has evolved in a Bayesian framework, where we view μ_a , the true value of action a , to be a random variable. We assume that we begin with a prior distribution of belief where μ_a is normally distributed with mean θ_a^0 and precision β_a^0 (recall that this is the inverse of the variance).

After n measurements we assume that our belief has evolved to where μ_a is normally distributed with mean θ_a^n and precision β_a^n . We write our *belief state* as $S^n = (\theta_a^n, \beta_a^n)_{a \in \mathcal{A}}$, with the assumption of normality implicit. We assume that we choose to measure $a^n = A^\pi(S^n)$, after which we observe \hat{C}^{n+1} , where we assume that the precision of our observation of \hat{C} has known precision β^C . We would then use this information to update our belief using our Bayesian updating formulas (first presented in Section 7.3.3), given by

$$\mu_a^{n+1} = \begin{cases} \frac{\beta_a^n \mu_a^n + \beta_a^C \hat{C}_a^{n+1}}{\beta_a^n + \beta_a^C} & \text{if } a^n = a, \\ \mu_a^n & \text{otherwise;} \end{cases} \quad (12.4)$$

$$\beta_a^{n+1} = \begin{cases} \beta_a^n + \beta_a^C & \text{if } a^n = a, \\ \beta_a^n & \text{otherwise.} \end{cases} \quad (12.5)$$

Our goal is to find a policy that determines which action to take to collect information. Let $a^n = A^\pi(S^n)$ be the action we take after making n observations using policy π . Note that using our notational system, we have a^0 as the first action (before we have any observations), and so a^n means that we observe $\hat{C}_{a^n}^{n+1}$. Let μ_a be the (unknown) expected true value of choosing action a . θ^n is our best estimate of this expectation after n measurements, and \hat{C}^{n+1} is a random variable whose mean is θ^n (which we know) that is also equal to μ_a (which we do not know). We can state the problem of finding the best policy in terms of solving

$$\max_{\pi} \mathbb{E} \sum_{n=0}^{\infty} \gamma^n \mu_{a^n},$$

where $a^n = A^\pi(S^n)$. We can equivalently write this as

$$\max_{\pi} \mathbb{E} \sum_{n=0}^{\infty} \gamma^n \theta_{a^n}^n,$$

keeping in mind that a^n must depend on the information in S^n .

One way to solve the problem is to use Bellman's equation

$$V^n(S^n) = \max_{a \in \mathcal{A}} (C(S^n, a) + \gamma \mathbb{E}\{V^{n+1}(S^{n+1})|S^n\}).$$

It is important to keep in mind that S^n is our *belief state* (or state of knowledge), and that $V^n(S^n)$ is the expected value of our earnings given our current state of belief. The problem is that if we have $|\mathcal{A}|$ actions, then our state variable has a state variable with $2|\mathcal{A}|$ continuous dimensions. Of course, an interesting line of research would be to use approximate dynamic programming, which allows us to solve the exploration-exploitation problem of approximate dynamic programming.

In a landmark paper (Gittins and Jones, 1974) it was shown that an optimal policy can be designed using an index policy. That is, it is possible to compute a number v_a^n for each bandit a , using information from just this bandit. It is then optimal to choose which bandit to play next by simply finding the largest v_a^n for all $a \in \mathcal{A}$. This is known as an index policy, and the values v_a^n are widely known as *Gittins indices*. We develop these next.

12.4.2 Basic Theory of Gittins Indices

Say we face the choice of playing a single slot machine, or stopping and converting to a process that pays a fixed reward ρ in each time period until infinity. If we choose to stop sampling and accept the fixed reward, the total future reward is $\rho/(1 - \gamma)$. Alternatively, if we play the slot machine, we not only win a random amount \hat{C} , we also learn something about the parameter μ that characterizes the distribution of \hat{C} (for our presentation, $\mathbb{E}\hat{C} = \mu$, but μ could be a vector of parameters that characterizes the distribution of \hat{C}). θ^n represents our state variable (i.e., our current estimate of the mean). Let $C(\theta^n) = \mathbb{E}\hat{C} = \theta^n$ be our expected reward given our estimate θ^n . We use the format $C(\theta^n)$ for consistency with our earlier models, where we wrote the contribution as a function of the state (we do not have an action for this simple problem). The optimality equations can now be written

$$V(\theta^n, \rho) = \max [\rho + \gamma V(\theta^n, \rho), C(\theta^n) + \gamma \mathbb{E}\{V(\theta^{n+1}, \rho)|\theta^n\}], \quad (12.6)$$

where we have written the value function to express the dependence on ρ .

Since we have an infinite horizon problem, the value function must satisfy the optimality equations

$$V(\theta, \rho) = \max [\rho + \gamma V(\theta, \rho), C(\theta) + \gamma \mathbb{E}\{V(\theta', \rho)|\theta\}],$$

where θ' is defined by equation (12.4). It can be shown that if we choose to stop sampling in iteration n and accept the fixed payment ρ , then that is the optimal

strategy for all future rounds. This means that starting at iteration n , our optimal future payoff (once we have decided to accept the fixed payment) is

$$\begin{aligned} V(\theta, \rho) &= \rho + \gamma\rho + \gamma^2\rho + \dots \\ &= \frac{\rho}{1 - \gamma}, \end{aligned}$$

which means that we can write our optimality recursion in the form

$$V(\theta^n, \rho) = \max \left[\frac{\rho}{1 - \gamma}, C(\theta^n) + \gamma \mathbb{E} \{ V(\theta^{n+1}, \rho) | \theta^n \} \right]. \quad (12.7)$$

Now for the magic of Gittins indices. Let v be the value of ρ that makes the two terms in the brackets in (12.7) equal. That is,

$$\frac{v}{1 - \gamma} = C(\theta) + \gamma \mathbb{E} \{ V(\theta', v) | \theta \}. \quad (12.8)$$

Let $v^{Gitt}(\theta, \sigma, \sigma_W, \gamma)$ be the solution of (12.7). The optimal solution depends on the current estimate of the mean θ , its variance σ^2 , the variance of our measurements σ_W^2 , and the discount factor γ . (For notational simplicity we are assuming that the measurement noise σ_W^2 is independent of the action a , but this assumption is easily relaxed.) Next assume that we have a family of slot machines \mathcal{A} , and let $v_a^{Gitt,n}(\theta_a^n, \bar{\sigma}_a^n, \sigma_W, \gamma)$ be the value of v that we compute for each slot machine $a \in \mathcal{A}$. An optimal policy for selecting slot machines is to choose the slot machine with the highest value for $v_a^{Gitt,n}(\theta_a^n, \bar{\sigma}_a^n, \sigma_W, \gamma)$. Such policies are known as *index policies*, and they refer to the property that the parameter $v_a^{Gitt,n}(\theta_a^n, \bar{\sigma}_a^n, \sigma_W, \gamma)$ for alternative a depends only on the characteristics of alternative a . For this problem, the parameters $v_a^{Gitt,n}(\theta_a^n, \bar{\sigma}_a^n, \sigma_W, \gamma)$ are called Gittins indices.

To put this solution in more familiar notation, imagine that we are facing the problem of choosing a decision $a \in \mathcal{A}$. Assume that at each iteration, we face the same decisions, and we are learning the value of making a decision just as we were learning the value of a reward from using a slot machine. Let $v_a^{Gitt,n}(\theta_a^n, \bar{\sigma}_a^n, \sigma_W, \gamma)$ be the “value” of making decision a , given our current belief (captured by $(\theta_a^n, \bar{\sigma}_a^n)$) about the potential reward we would receive from this decision. When we ignore the value of acquiring information (as we have done in our presentation of approximate dynamic programming algorithms up to now), we would make a decision by solving

$$\max_a \theta_a^n.$$

This solution might easily lead us to avoid a decision that might be quite good, but that we currently think is poor (and we are unwilling to learn anything more about the decision). Gittins theory tells us to solve

$$\max_a v_a^{Gitt,n}(\theta_a^n, \bar{\sigma}_a^n, \sigma_W, \gamma).$$

The computation of Gittins indices highlights a subtle issue when computing expectations for information-collection problems. The proper computation of the expectation needed to solve the optimality equations requires, in theory, knowledge of exactly the distribution that we are trying to compute. To illustrate, the expected winnings are given by $C(\theta^n) = \mathbb{E}\hat{C} = \mu$, but μ is unknown. Instead, we adopt a Bayesian approach that our expectation is computed with respect to the distribution *we believe* to be true. Thus at iteration n we believe that our winnings are normally distributed with mean θ^n , so we would use $C(\theta^n) = \theta^n$. The term $\mathbb{E}\{V(\theta^{n+1}, \rho) | \theta^n\}$ captures what we believe the effect of observing \hat{C}^{n+1} will have on our estimate θ^{n+1} , but this belief is based on what we think the distribution of \hat{C}^{n+1} is, rather than the true distribution.

The beauty of Gittins indices (or any index policy) is that it reduces N -dimensional problems into a series of one-dimensional problems. The problem is that solving equation (12.7) (or equivalently, (12.8)) offers its own challenges. Finding $v_a^{Gitt,n}(\theta, \sigma, \sigma_W, \gamma)$ requires solving the optimality equation in (12.7) for different values of ρ until (12.8) is satisfied. Although algorithmic procedures have been designed for this, they are not simple.

12.4.3 Gittins Indices for Normally Distributed Rewards

The calculation of Gittins indices is simplified for special classes of distributions. In this section we consider the case where the observations of rewards \hat{C} are normally distributed. This is the case we are most interested in, since in the next section we are going to apply this theory to the problem where the unknown reward is in fact the value of being in a future state $\bar{V}(s')$. Since $\bar{V}(s')$ is computed using averages of random observations, the distribution of $\bar{V}(s')$ will be closely approximated by a normal distribution.

Students learn in their first statistics course that normally distributed random variables satisfy a nice property. If Z is normally distributed with mean 0 and variance 1 and if

$$X = \mu + \sigma Z,$$

then X is normally distributed with mean μ and variance σ^2 . This property simplifies what are otherwise difficult calculations about probabilities of events. For example, computing $\mathbb{P}[X \geq x]$ is difficult because the normal density function cannot be integrated analytically. We instead have to resort to numerical procedures. But because of the above-mentioned translationary and scaling properties of normally distributed random variables, we can perform the difficult computations for the random variable Z (the “standard normal deviate”), and use this to answer questions about any random variable X . For example, we can write

$$\begin{aligned} \mathbb{P}[X \geq x] &= \mathbb{P}\left[\frac{X - \mu}{\sigma} \geq \frac{x - \mu}{\sigma}\right] \\ &= \mathbb{P}\left[Z \geq \frac{x - \mu}{\sigma}\right]. \end{aligned}$$

Thus the ability to answer probability questions about Z allows us to answer the same questions about any normally distributed random variable.

The same property applies to Gittins indices. Although the proof requires some development, it is possible to show that

$$v^{Gitt,n}(\theta^n, \bar{\sigma}^n, \sigma_W, \gamma) = \theta + \Gamma\left(\frac{\bar{\sigma}^n}{\sigma_W}, \gamma\right)\sigma_W,$$

where

$$\Gamma\left(\frac{\bar{\sigma}^n}{\sigma_W}, \gamma\right) = v^{Gitt,n}(0, \sigma, 1, \gamma)$$

is a “standard normal Gittins index” for problems with mean 0 and variance 1. Note that $\bar{\sigma}^n/\sigma_W$ increases with n , and that $\Gamma(\bar{\sigma}^n/\sigma_W, \gamma)$ decreases toward zero as $\bar{\sigma}^n/\sigma_W$ increases. As $n \rightarrow \infty$, $v^{Gitt,n}(\theta^n, \bar{\sigma}^n, \sigma_W, \gamma) \rightarrow \theta^n$.

Unfortunately, as of this writing, there do not exist easy-to-use software utilities for computing standard Gittins indices. The situation is similar to doing statistics before computers when students had to look up the cumulative distribution for the standard normal deviate in the back of a statistics book. Table 12.2 is exactly such a table for Gittins indices. The table gives indices for both the variance-known and variance-unknown cases, but only for the case where $\sigma^n/\sigma_W = 1/n$. In the variance-known case we assume that σ^2 is given, which allows us to calculate the variance of the estimate for a particular slot machine just by dividing by the number of observations.

Lacking standard software libraries for computing Gittins indices, researchers have developed simple approximations. As of this writing, the most recent of these works as follows. First, it is possible to show that

$$\Gamma(s, \gamma) = \sqrt{-\log \gamma} \cdot b\left(-\frac{s^2}{\log \gamma}\right). \quad (12.9)$$

A good approximation of $b(s)$, which we denote by $\tilde{b}(s)$, is given by

$$\tilde{b}(s) = \begin{cases} s/\sqrt{2}, & s \leq \frac{1}{7}, \\ e^{-0.02645(\log s)^2 + 0.89106 \log s - 0.4873}, & \frac{1}{7} < s \leq 100, \\ \sqrt{s} (2 \log s - \log \log s - \log 16\pi)^{1/2}, & s > 100. \end{cases}$$

Thus the approximate version of (12.9) is

$$v^{Gitt,n}(\theta, \sigma, \sigma_W, \gamma) \approx \theta^n + \sigma_W \sqrt{-\log \gamma} \cdot \tilde{b}\left(-\frac{\bar{\sigma}^{2,n}}{\sigma_W^2 \log \gamma}\right). \quad (12.10)$$

Table 12.2 Gittins indices $\Gamma(\sigma^n/\sigma_W, \gamma)$ for the case of observations that are normally distributed with mean 0, variance 1, and where $\sigma^n/\sigma_W = 1/n$

	Discount Factor			
	Known Variance	Unknown Variance		
Observations	0.95	0.99	0.95	0.99
1	0.9956	1.5758	—	—
2	0.6343	1.0415	10.1410	39.3343
3	0.4781	0.8061	1.1656	3.1020
4	0.3878	0.6677	0.6193	1.3428
5	0.3281	0.5747	0.4478	0.9052
6	0.2853	0.5072	0.3590	0.7054
7	0.2528	0.4554	0.3035	0.5901
8	0.2274	0.4144	0.2645	0.5123
9	0.2069	0.3808	0.2353	0.4556
10	0.1899	0.3528	0.2123	0.4119
20	0.1058	0.2094	0.1109	0.2230
30	0.0739	0.1520	0.0761	0.1579
40	0.0570	0.1202	0.0582	0.1235
50	0.0464	0.0998	0.0472	0.1019
60	0.0392	0.0855	0.0397	0.0870
70	0.0339	0.0749	0.0343	0.0760
80	0.0299	0.0667	0.0302	0.0675
90	0.0267	0.0602	0.0269	0.0608
100	0.0242	0.0549	0.0244	0.0554

Source: From Gittins (1989).

12.4.4 Upper Confidence Bounding

A strategy that has received considerable theoretical interest is known as the *upper confidence bound*, which produces an easily computable policy. The idea is to create an upper bound on the potential reward from a particular action. If θ_a^n is our estimate of the value of the reward from action a after n observations, the UCB policy is given by

$$A^{UCB} = \theta_a^n + \sqrt{\frac{2}{N_a^n} g\left(\frac{N_a^n}{n}\right)}, \quad (12.11)$$

where N_a^n is the number of times we have measured action a after n measurements, and $g(w)$ is given by

$$g(w) = \log \frac{1}{2} - \frac{1}{2} \log \log \frac{1}{t} - \frac{1}{2} \log 16\pi.$$

The UCB policy has been proved to be the best possible policy in terms of minimizing expected regret, which is the degree to which the policy underperforms the optimal. Curiously, despite this theory, the policy seems to underperform other

heuristics in practice. A variant of the UCB policy, dubbed the UCB1 policy, is given by

$$A^{UCP} = \theta_a^n + 4\sigma_w \sqrt{\frac{\log n}{N_a^n}}, \quad (12.12)$$

where σ_w is the standard deviation of the observation error.

UCB policies work on a very similar principle to Gittins indices, which is that each action is valued based on the current estimate of the value of the action θ_a^n , plus a term that reflects in some way the level of uncertainty in the estimate of the contribution. Despite the theoretical support of UCB strategies, it is common practice to test them with a tunable parameter in front of the second term.

12.5 THE KNOWLEDGE GRADIENT POLICY

We first saw the knowledge gradient in Chapter 7 (Section 7.4) in the context of policy search, which is an offline application. We present the knowledge gradient in more detail in this section, and show how it can also be adapted to online applications. The knowledge gradient is based on a Bayesian belief model, but it can be used for problems without a formal prior by performing an initial sample to build an initial belief (a method known as empirical Bayes).

As with our presentation of Gittins indices, we are going to start with a pure learning model, which means that our state variable consists purely of our belief about the performance of a series of options. For example, we need to find the configuration of features for a laptop that will produce the highest sales. At each iteration we face the same set of choices and the goal is to learn the most about the value of each choice. We defer until Section 12.6 a discussion of the much more difficult problem of learning in the presence of a physical state.

Gittins indices were derived in the context of online learning problems, which is to say we receive rewards from choices as we learn information about the performance of each choice. In Chapter 7 we introduced the concept of the knowledge gradient in the context of offline learning, and we start by briefly reviewing this material here. But the knowledge gradient can also be adapted to online learning problems, and we show how to do this here.

12.5.1 The Knowledge Gradient for Offline Learning

We first introduced the knowledge gradient in Section 7.4 in the context of searching over a discrete set of parameters to optimize a policy. Here we re-introduce the knowledge gradient, but now in the context of taking an action that yields information about the value of the action. We begin by focusing on problems without a physical state (sometimes described as a problem with a single physical state) in an offline setting.

The knowledge gradient seeks to learn about the value of different actions by maximizing the value of information from a single observation. Let S^n be our state

of knowledge about the value of each action a . The knowledge gradient uses a Bayesian model, so

$$S^n = (\theta_a^n, \hat{\sigma}_a^{2,n})_{a \in \mathcal{A}}.$$

The value of being in (knowledge) state S^n is given by

$$V^n(S^n) = \mu_{a^n},$$

where a^n is the choice that appears to be best given what we know after n measurements, calculated using

$$a^n = \max_{a' \in \mathcal{A}} \theta_{a'}^n.$$

If we choose action $a^n = a$, we observe \hat{C}_a^{n+1} , which we then use to update our estimate of our belief about μ_a using our Bayesian updating equations:

$$\theta_a^{n+1} = \begin{cases} \frac{\beta_a^n \theta_a^n + \beta_a^W W_a^{n+1}}{\beta_a^n + \beta_a^W} & \text{if } a^n = x, \\ \theta_a^n & \text{otherwise;} \end{cases} \quad (12.13)$$

$$\beta_a^{n+1} = \begin{cases} \beta_a^n + \beta_a^W & \text{if } a^n = a, \\ \beta_a^n & \text{otherwise.} \end{cases} \quad (12.14)$$

Here β_a^n is the precision (one over the variance) of our belief about μ_a , and β_a^W is the precision of the observation of a . Using the updating equations (12.13) and (12.14), the value of state $S^{n+1}(a)$ when we trying action $a^n = a$ is given by

$$V^{n+1}(S^{n+1}(a)) = \max_{a' \in \mathcal{A}} \theta_{a'}^n.$$

The knowledge gradient is then given by

$$v_a^{KG,n} = \mathbb{E}V^{n+1}(S^M(S^n, a, W^{n+1})) - V^n(S^n).$$

Computing a knowledge gradient policy is extremely easy. We assume that all rewards are normally distributed, and that we start with an initial estimate of the mean and variance of the value of decision d , given by

θ_a^0 = initial estimate of the expected reward from making decision a ,

$\bar{\sigma}_a^0$ = initial estimate of the standard deviation of θ_a^0 .

Suppose that each time we make a decision we receive a reward given by

$$\hat{C}_a^{n+1} = \mu_a + \varepsilon^{n+1},$$

where μ_a is the true expected reward from action a (which is unknown) and ε is the measurement error with standard deviation σ_W (which we assume is known).

Assume that $(\theta_a^n, \bar{\sigma}_a^{2,n})$ is the mean and variance of our belief about μ_a after n observations. If we take action a and observe a reward \hat{C}_a^{n+1} , we can use Bayesian updating (which produces estimates that are most likely given what we knew before and what we just learned) to obtain new estimates of the mean and variance for action a .

Suppose that we try an action a where $\beta_a^n = 1/\bar{\sigma}_a^{2,n} = 1/(20^2) = 0.0025$, and $\beta_W = 1/\sigma_W^2 = 1/(40^2) = 0.000625$. Assume that $\theta_a^n = 200$ and that we observe $\hat{C}_a^{n+1} = 250$. The updated mean and precision are given by

$$\begin{aligned}\theta_a^{n+1} &= \frac{\beta_a^n \theta_a^n + \beta_W \hat{C}_a^{n+1}}{\beta_a^n + \beta_W} \\ &= \frac{(0.0025)(200) + (0.000625)(250)}{0.0025 + 0.000625} \\ &= 210. \\ \beta^{n+1} &= \beta_a^n + \beta_W \\ &= 0.0025 + 0.000625 \\ &= 0.003125.\end{aligned}$$

We next find the variance of the *change* in our estimate of μ_a assuming we choose to sample action a in iteration n . For this we define

$$\tilde{\sigma}_a^{2,n} = \text{Var}[\theta_a^{n+1} - \theta_a^n | S^n]. \quad (12.15)$$

With a little work, we can write $\tilde{\sigma}_a^{2,n}$ in different ways, including

$$\tilde{\sigma}_a^{2,n} = \bar{\sigma}_a^{2,n} - \bar{\sigma}_a^{2,n+1}, \quad (12.16)$$

$$= \frac{(\bar{\sigma}_a^{2,n})}{1 + \sigma_W^2 / \bar{\sigma}_a^{2,n}}. \quad (12.17)$$

Equation (12.16) expresses the (perhaps unexpected) result that $\tilde{\sigma}_a^{2,n}$ measures the change in the estimate of the standard deviation of the reward from decision a from iteration $n - 1$ to n . Equation (12.17) closely parallels equation (7.26). Using our numerical example, equations (12.16) and (12.17) both produce the result

$$\begin{aligned}\tilde{\sigma}_a^{2,n} &= 400 - 320 = 80 \\ &= \frac{40^2}{1 + (10^2/40^2)} = 80.\end{aligned}$$

We next compute

$$\xi_a^n = - \left| \frac{\theta_a^n - \max_{a' \neq a} \theta_{a'}^n}{\tilde{\sigma}_a^n} \right|.$$

ξ_a^n is called the *normalized influence* of decision a . It measures the number of standard deviations from the current estimate of the value of decision a , given by θ_a^n , and the best alternative other than decision a . We then find

$$f(\xi) = \xi \Phi(\xi) + \phi(\xi),$$

where $\Phi(\xi)$ and $\phi(\xi)$ are, respectively, the cumulative standard normal distribution and the standard normal density. Thus, if Z is normally distributed with mean 0, variance 1, then $\Phi(\xi) = \mathbb{P}[Z \leq \xi]$ while

$$\phi(\xi) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{\xi^2}{2}\right).$$

The knowledge gradient algorithm chooses the decision a with the largest value of $v_a^{KG,n}$ given by

$$v_a^{KG,n} = \tilde{\sigma}_a^n f(\xi_a^n).$$

The knowledge gradient algorithm is quite simple to implement. Table 12.3 illustrates a set of calculations for a problem with five options. θ represents the current estimate of the value of each action, while $\bar{\sigma}$ is the current standard deviation of θ . Options 1, 2, and 3 have the same value for $\bar{\sigma}$, but with increasing values of θ . The table illustrates that when the variance is the same, the knowledge gradient prefers the decisions that appear to be the best. Decisions 3 and 4 have the same value of θ , but decreasing values of $\bar{\sigma}$, illustrating that the knowledge gradient prefers decisions with the highest variance. Finally, decision 5 appears to be the best of all the decisions, but has the lowest variance (meaning that we have the highest confidence in this decision). The knowledge gradient is the smallest for this decision out of all of them.

Figure 12.3 compares the performance of the knowledge gradient algorithm against interval estimation (using $\alpha/2 = 0.025$), Boltzmann exploration, Gittins exploration, uniform exploration and pure exploitation. The results show steady improvement as we collect more observations. The knowledge gradient and interval estimation work the best for this problem. However, it is possible to fool interval estimation by providing an option where the interval is very tight (for an option that is good but not the best). Both of these methods outperform the Gittins exploration.

Table 12.3 Calculations behind the knowledge gradient algorithm

Decision	θ	$\bar{\sigma}$	$\tilde{\sigma}$	ξ	$f(z)$	KG index
1	1.0	2.5	1.569	-1.275	0.048	0.075
2	1.5	2.5	1.569	-0.956	0.090	0.142
3	2.0	2.5	1.569	-0.637	0.159	0.249
4	2.0	2.0	1.400	-0.714	0.139	0.195
5	3.0	1.0	0.981	-1.020	0.080	0.079

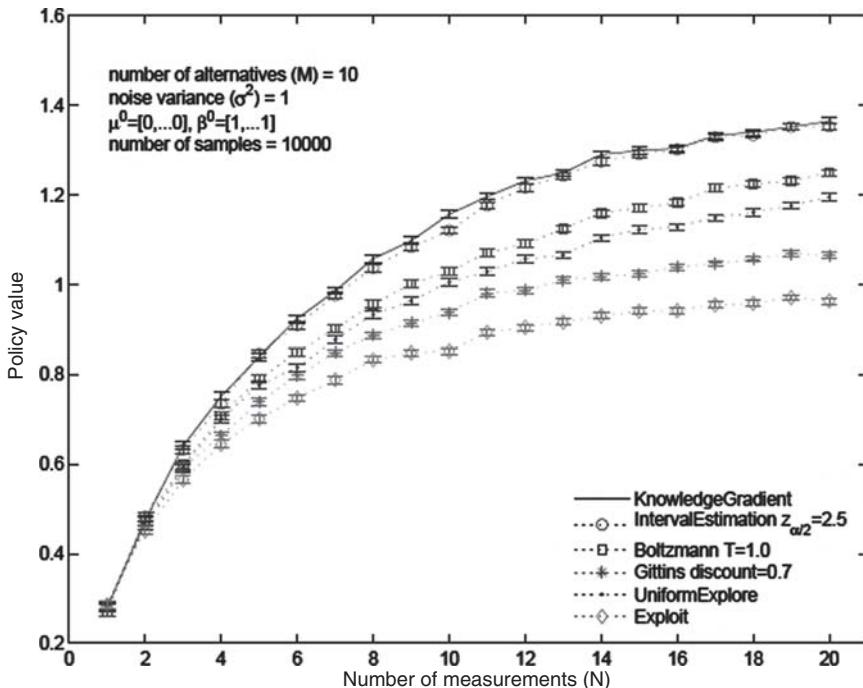


Figure 12.3 Comparison of knowledge gradient against other policies as a function of the number of observations.

12.5.2 The Knowledge Gradient for Correlated Beliefs

The knowledge gradient is particularly simple when we are evaluating discrete actions, and where there is no structure among the actions. There are many problems where this would not be true. For example, an action may be a discretization of a continuous variable such as a price, location, or drug dosage. Imagine that we currently believe that $\theta_a = \theta_{0.012}^n = 22.7$ and $\theta_a^n = 22.5$. Now assume we sample $\hat{C}_{0.012}^{n+1} = 28.1$. Imagine that from this observation, we update our belief about $a = 0.012$ to $\theta_{0.012}^{n+1} = 24.3$. It is likely that the value for $a = 0.012$ is highly correlated with the value for $a = 0.014$, and hence we would expect that our observation of $\hat{C}_{0.012}^{n+1}$ would also lead us to increase our estimate of $\theta_{0.014}^n$.

We showed how the knowledge gradient can be adapted to problems with correlated beliefs in Section 7.4.3. If there are $|\mathcal{A}|$ actions, this logic has complexity $|\mathcal{A}|^2$, which is not an issue for problems with small action spaces but can become a serious problem when the number of actions approaches 1000.

The knowledge gradient can be applied to a variety of problems where beliefs are represented as lookup tables, parametric models (see Section 7.4.4), and non-parametric models. We withhold further discussion until Section 12.6 when we consider problems with a physical state.

12.5.3 The Knowledge Gradient for Online Learning

We have derived the knowledge gradient for offline learning problems. Our choice of action does not depend directly on how good the action is: we only care about how much information we gain from trying an action.

There are many online applications of dynamic programming where there is an operational system that we would like to optimize. In these settings we have to strike a balance between the value of an action and the information we gain that may improve our choice of actions in the future. This is precisely the trade-off that is made by Gittins indices for the multiarmed bandit problem.

It turns out that the knowledge gradient is easily adapted for online problems. As before, let $v_a^{KG,n}$ be the offline knowledge gradient, giving the value of observing action $a^n = a$, measured in terms of the improvement in a single decision. Now imagine that we have a budget of N decisions. After having made n decisions (which means n observations of the value of different actions), if we observe $a^n = a$, which allows us to observe \hat{C}_a^{n+1} , then we received an expected reward of $\mathbb{E}^n \hat{C}_a^{n+1} = \theta_a^n$ and obtain information that improves the contribution from a single decision by $v_a^{KG,n}$. However, we have $N - n$ more decisions to make. Suppose that we learn from the observation of \hat{C}_a^{n+1} by choosing $a^n = a$, but we do not allow ourselves to learn anything from future decisions. This means that the remaining $N - n$ decisions have access to the same information.

From this analysis the knowledge gradient for online applications consists of the expected value of the single-period contribution of the measurement plus the improvement in all the remaining decisions in our horizon. This implies that

$$v_a^{OLKG,n} = \theta_a^n + (N - n)v_a^{KG,n}. \quad (12.18)$$

If we have an infinite horizon problem with discount factor γ , then

$$v_a^{OLKG,n} = \theta_a^n + \frac{\gamma}{1 - \gamma} v_a^{KG,n}. \quad (12.19)$$

We note that the logic for incorporating correlated beliefs for the offline knowledge gradient can now be directly applied to online problems.

12.6 LEARNING WITH A PHYSICAL STATE

Up to now we have focused on the problem of determining how to collect information to make a choice that returns an uncertain reward. This is an important problem in its own right, and it also lays the foundation for solving the exploration–exploitation problem in approximate dynamic programming when we have a physical state. However, the presence of a physical state dramatically complicates the formal study of information collection.

In Section 12.1 we used our nomadic trucker example to provide an illustration of what happens if we ignore learning and use a pure exploitation strategy. The results are depicted in Figure 12.1, which demonstrates how a pure exploitation

strategy can become stuck in a small number of states. To overcome this behavior, we have to design a strategy that forces the model to visit a diversity of states.

We can put this problem in the setting of our multi-armed bandit problem if we think of the contribution of an action \hat{C}_a as consisting of

$$\hat{C}_a^n = C(s, a) + \gamma \bar{V}^n(S^{M,a}(s, a)).$$

Now let θ^n be our estimate of the mean of \hat{C}_a .

12.6.1 Heuristic Exploration Policies

We have already reviewed a series of heuristic exploration strategies in Section 12.3 that can be applied in the multi-armed bandit setting:

- ϵ -greedy exploration
- Persistent excitation
- Boltzmann exploration
- Interval estimation

For discrete actions, ϵ -greedy is probably the most popular, but Boltzmann exploration and interval estimation are also widely used. Note that all these strategies have tunable parameters.

There are some issues we need to address when choosing an action:

1. *Bias due to learning.* If we use approximate value iteration, Q -learning, or least squares policy evaluation, the value function grows with the iterations because we are adding contributions over time. This means that visiting a state tends to increase the value of being in the state.
2. *The problem of vectors.* Imagine that our decision is a vector x , which makes the action space (feasible region) \mathcal{X} very large or infinite in size. Choosing an action at random teaches us virtually nothing.
3. *The uncertainty in our estimate of a value.* Classic exploration strategies do not consider the variance in the estimate of the value of the state $S^{M,a}(S, a)$ that we next visit as a result of our action. Interval estimation is the only exception.

ϵ -greedy avoids the bias issue because it ignores the estimated value of an action, θ_a^n , when making an exploration step. Boltzmann exploration and interval estimation are sensitive to the estimate θ_a^n , reducing the likelihood of testing an action if the estimate is too low, ignoring the fact that trying an action may also raise its value. None of the three major heuristics— ϵ -greedy, Boltzmann, or interval estimation—can be applied to problems with vector-valued action spaces. ϵ -greedy and Boltzmann, which are probably the most popular heuristics used in practice for discrete actions (along with persistent excitation for continuous problems), ignore the uncertainty in the estimate of an action, an issue that has proved to be of central

importance in the design of more formal strategies such as Gittins indices and the knowledge gradient, as well as the highly effective interval estimation.

It is our position that it is going to be difficult to design a principled exploration strategy for algorithms based on approximate value iteration, which includes Q -learning and least squares policy evaluation. As of this writing, we are not aware of any rigorous theory for learning problems such as the multi-armed bandit that can handle not only the uncertainty in our belief about the value of each choice but also the property that trying an action increases the expected value of the action.

12.6.2 Gittins Heuristics

There has been an attempt to apply Gittins indices to problems with a physical state, resulting in a method called the *local bandit approximation* (LBA), which is an adaptation of Gittins indices to problems with a physical state. The LBA works roughly as follows: Imagine that you are in a state s and you are considering action a that might take you to a state s' , which allows you to update your belief about the value of being in state s' . Assuming an ergodic process (under whatever policy we are following), we will eventually return to state s , which will allow us to use our newly acquired information again. Let $R(s, a)$ be the expected reward that we earn during this time, and let $\tau(s, a)$ be the expected time it takes to return to state s . The Gittins index for action a is then

$$v_a^{LBA} = \frac{R(s, a)}{\tau(s, a)}.$$

This can be thought of as a bandit problem that is local to state s , hence the name. Not surprisingly, this is fairly hard to compute.

As a simpler alternative, we might apply the concept of Gittins indices in a purely heuristic fashion, a method we call *Gittins exploration*. For example, we might choose an action a^n using

$$a^n = \arg \max_a \left[C(s, a) + \gamma \bar{V}^n(S^M(s, a)) + \Gamma\left(\frac{\bar{\sigma}^n}{\sigma_W}, \gamma\right) \sigma_W \right], \quad (12.20)$$

where σ_W is the standard deviation of an observation, and $\Gamma(\bar{\sigma}^n/\sigma_W, \gamma)$ is the standard normal Gittins index that tells us how many standard deviations away from the mean we should use (which is a function of the number of observations n). This is a purely heuristic use of Gittins indices, and as a result it would make sense to replace $\Gamma(\bar{\sigma}^n/\sigma_W, \gamma)$ with a simple formula that declines with n such as

$$\Gamma\left(\frac{\bar{\sigma}^n}{\sigma_W}, \gamma\right) = \frac{a}{b + n},$$

where a and b would have to be tuned for a particular application.

An advantage of equation (12.20) is that it can be used as if we are pursuing an exploitation strategy with a modified contribution function that captures the value of collecting information for future decisions. Unlike combined

exploration-exploitation policies, we never choose an action purely at random (regardless of how bad we think it might be).

A significant limitation of all these heuristics is the presence of tunable parameters. In the next section we show how the knowledge gradient can be adapted without using any tunable parameters.

12.6.3 The Knowledge Gradient Using Lookup Tables

We can apply the concept of the knowledge gradient to infinite horizon, discrete dynamic programs using the following line of thinking. If we choose action a^n while in state S^n , this will take us to the post-decision state $S^{n,a}$ after which we transition to the next pre-decision state S^{n+1} . The sequence of pre-decision states, actions and post-decision states is depicted in Figure 12.4. We illustrate the calculations in this section for a lookup table representation, since these are easiest to understand. As we point out in the section that follows, adapting this strategy to other belief models such as linear architectures using basis functions is fairly minor.

Define the Q -factors using the post-decision state as

$$Q^n(S^n, a) = C(S^n, a) + \gamma V^n(S^{M,a}(S^n, a)).$$

We need to interpret $V^n(s)$ (where s is a post-decision state) as our belief about V^n after n iterations. When we apply the knowledge gradient, we are asking about the value of learning from a single measurement. Thus, if we choose action a^n , we are going to update our belief about $V^n(s)$ to $V^{n+1}(s)$, after which we are going to stop and apply our policy from that point on, using $V^{n+1}(s)$ to guide the choice of decisions. This means that $V^{n'}(s) = V^{n+1}(s)$ for all $n' \geq n + 1$.

If we assume we are not going to do any more learning, we would make our decision by solving

$$a^n = \arg \max_a (C(S^n, a) + \gamma V^n(S^{M,a}(S^n, a))). \quad (12.21)$$

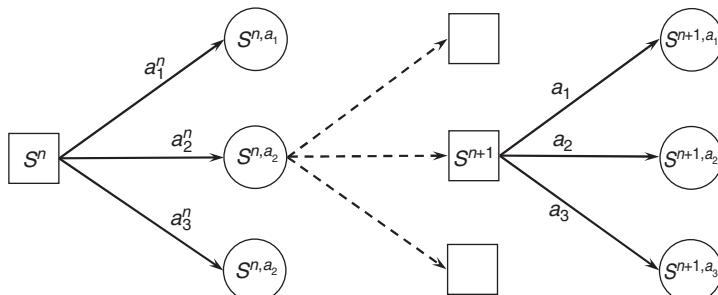


Figure 12.4 Decision tree showing pre-decision states S_t , actions a_t , post-decision states S_t^a , and subsequent states and actions.

However, if we choose action a^n , we will transition to state $S^{n,a} = S^{M,a}(S^n, a^n)$, after which we observe W^{n+1} which takes us to state $S^{n+1} = S^M(S^n, a^n, W^{n+1})$. Once at state S^{n+1} , we are going to compute

$$\hat{v}^{n+1} = \max_{a'} (C(S^{n+1}, a') + \gamma V^n(S^{M,a}(S^{n+1}, a'))).$$

Our intent is to use \hat{v}^{n+1} to update our belief $V^n(s)$ using the Bayesian updating equations we first introduced in Section 9.9.2. Using our (presumably known) covariance matrix Σ^n , we can update our belief about the value $V^n(s)$ (where s is the post-decision state) of each state as

$$\bar{V}^{n+1}(s) = V^n(s) + \frac{\hat{v}^{n+1} - V^n(s)}{\lambda(S^{a,n}) + \Sigma^n(S^{a,n}, S^{a,n})} \Sigma^n(s, S^{a,n}), \quad (12.22)$$

where $\lambda(S^{a,n})$ is our (again, presumably known) variance of \hat{v}^{n+1} . Note that while we have only visited one (post-decision) state $S^{a,n}$, we update all the states s (or at least all the states where $\Sigma^n(s, S^{a,n}) > 0$). We also update our covariance matrix using

$$\Sigma^{n+1}(s, s') = \Sigma^n(s, s') - \frac{\Sigma^n(s, S^{a,n}) \Sigma^n(S^{a,n}, s')}{\lambda(S^{a,n}) + \Sigma^n(S^{a,n}, S^{a,n})}. \quad (12.23)$$

Our idea is to replace equation (12.21) with a rule that recognizes that we will learn from our action, but only once. That is, by transitioning to $S^{a,n}$ and then observing \hat{v}^{n+1} , we are going to update our beliefs, and we would like to recognize this before we choose our action. However, we are going to assume that we *only* learn from this single decision. This means that we want to find our action by solving

$$a^n = \arg \max_a (C(S^n, a) + \gamma \mathbb{E}_a^n V^{n+1}(S^{M,a}(S^n, a))). \quad (12.24)$$

where \mathbb{E}_a^n represents our expectation given the information we have up through iteration n , and given that we choose action a . $V^{n+1}(S^{M,a}(S^n, a))$ represents our updated value function *after* we observe \hat{v}^{n+1} . We recognize that since W^{n+1} is a random variable, so is S^{n+1} . For this reason we can write

$$\begin{aligned} Q^n(S^n, a) &= C(S^n, a) + \gamma \mathbb{E}_a^n V^{n+1}(S^{M,a}(S^n, a)) \\ &= C(S^n, a) + \gamma \sum_{s'=S^{n+1}} P(s'|S^{a,n}) \mathbb{E}_a^n \max_{a'} Q^{n+1}(S^{n+1}, a'). \end{aligned}$$

When we observe \hat{v}^{n+1} , we are going to update $V^n(s)$ to obtain $V^{n+1}(s)$, which we then use to compute

$$Q^{n+1}(S^{n+1}, a') = C(S^{n+1}, a') + \gamma V^{n+1}(S^{M,a}(S^{n+1}, a')).$$

In other words, by choosing action a when we are in state S^n means that we are going to improve our solution of $\max_{a'} Q^{n+1}(S^{n+1}, a')$. We would like to capture

the value of observing \hat{v}^{n+1} in terms of improving our choice of a^{n+1} in the future, when we make our decision about a^n . We now describe the steps for doing this.

Let Z be a normally distributed random variable with mean 0, variance 1. When we are in state S^n , V^{n+1} is a random variable that can be written as (see Section 7.4.3 for more details)

$$V^{n+1} \propto V^n + \frac{\Sigma^n(s, S^{a,n})}{\sqrt{\lambda(S^{a,n}) + \Sigma^n(S^{a,n}, S^{a,n})}} Z.$$

This means that we can write $Q^{n+1}(S^{n+1}, a')$ as

$$Q^{n+1}(S^{n+1}, a') = Q^n(S^{n+1}, a') + \beta^n(a')Z,$$

where

$$\beta^n(a') = \gamma \frac{\Sigma^n(S^{M,a}(S^{n+1}, a'), S^{a,n})}{\sqrt{\lambda(S^{a,n}) + \Sigma^n(S^{a,n}, S^{a,n})}}.$$

Next we need to compute

$$\mathbb{E}^n \max_{a'} Q^{n+1}(S^{n+1}, a') = \mathbb{E}^n \max_{a'} (Q^n(S^{n+1}, a') + \beta^n(a')Z).$$

We have already seen how to do this in Section 7.4.3, and we refer the reader back to this section for details. The method requires sorting the actions a into a list $A = \{1, 2, \dots, a, a+1, \dots\}$ where $\beta^n(a'+1) > \beta^n(a')$. To compute the list A , we also find numbers $z_{a'}$ using

$$z_{a'} = \frac{Q^n(S^{n+1}, a') - Q^n(S^{n+1}, a'+1)}{\beta^n(a'+1) - \beta^n(a')}.$$

We require that $z_{a'+1} > z_{a'}$, and eliminate any actions a' where this is not satisfied. Given this construction, we can compute the knowledge gradient using

$$\begin{aligned} v^{KG,n}(S^{a,n}, S^{n+1}) &= \text{marginal value of information gained by an action that takes us to state } S^{a,n} \text{ after which there is a transition to } S^{n+1}, \\ &= \mathbb{E}^n \max_{a'} Q^{n+1}(S^{n+1}, a') - \max_{a'} Q^n(S^{n+1}, a') \\ &= \mathbb{E}^n \max_{a'} (Q^n(S^{n+1}, a') + \beta^n(a')Z) - \max_{a'} Q^n(S^{n+1}, a') \\ &= \sum_{a' \in A} (\beta^n(a'+1) - \beta^n(a')) f(-|z_{a'}|), \end{aligned}$$

where $f(z) = z\Phi(z) + \phi(z)$, and where $\Phi(z)$ is the cumulative distribution function for the standard normal distribution, and $\phi(z)$ is the standard normal density (this is standard notation for these distributions, and we regret the overlap with our also standard notation for basis functions). We emphasize that $Q^{n+1}(S^{n+1}, a')$ is our estimate of the Q factor after the benefits of learning from the first $n+1$

measurements, evaluated at the state-action pair (S^{n+1}, a') , while $Q^n(S^{n+1}, a')$ is our estimate of the Q factor after learning from the first n measurements, but also evaluated at the state action pair (S^{n+1}, a') . This also allows us to write

$$\mathbb{E}^n \max_{a'} Q^{n+1}(S^{n+1}, a') = \max_{a'} Q^n(S^{n+1}, a') + v^{KG,n}(S^{a,n}, S^{n+1}).$$

Of course, S^{n+1} is a random variable, and as a result we have to compute the expectation

$$\begin{aligned} \sum_{S^{n+1}} P(S^{n+1}|S^{a,n}) \mathbb{E}_a^n \max_{a'} Q^{n+1}(S^{n+1}, a') &= \sum_{S^{n+1}} P(S^{n+1}|S^{a,n}) \max_{a'} Q^n(S^{n+1}, a') \\ &\quad + \sum_{S^{n+1}} P(S^{n+1}|S^{a,n}) v^{KG,n}(S^{a,n}, S^{n+1}). \end{aligned}$$

We note that we can write

$$\sum_{S^{n+1}} P(S^{n+1}|S^{a,n}) \max_{a'} Q^n(S^{n+1}, a') = V^n(S^{a,n}),$$

since $V^n(S^{a,n})$ is the expected value of being in post-decision state $S^{a,n}$. This allows us to write equation (12.24) as

$$\begin{aligned} A^{KG-online,n} &= \arg \max_a (C(S^n, a) + \gamma V^n(S^{a,n}) \\ &\quad + \gamma \sum_{S^{n+1}} P(S^{n+1}|S^{a,n}) v^{KG,n}(S^{a,n}, S^{n+1})), \end{aligned} \quad (12.25)$$

which we can write equivalently as

$$A^{KG-online,n} = \arg \max_a \left(Q^n(S^n, a) + \gamma \sum_{S^{n+1}} P(S^{n+1}|S^{a,n}) v^{KG,n}(S^{a,n}, S^{n+1}) \right). \quad (12.26)$$

Equation (12.26) is an equation that is very similar to an interval estimation, Gittins indices, and the knowledge gradient for online learning that we first saw in Section 12.5.3, with one small difference. In Section 12.5.2 our formula for the online knowledge gradient was given by

$$v_a^{OLKG,n} = \theta_a^n + (N - n)v_a^{KG,n}$$

for finite horizon problems, or

$$v_a^{OLKG,n} = \theta_a^n + \frac{\gamma}{1 - \gamma} v_a^{KG,n}$$

for infinite horizon problems. The reason we do not factor the knowledge gradient term in (12.26) by $N - n$ or $\gamma/(1 - \gamma)$ is that we are computing the value of

information on the value function, which itself is the finite or infinite horizon value of being in a state.

The policy in (12.26) (or (12.25)) would be best suited for online dynamic programs, which is to say, problems where we are actually incurring contributions as we are making decisions. Imagine now that we are simply training value functions in a simulated environment, and our plan is to then use the value functions we estimated to make decisions at a later time. This is offline training, and the policy we would use is given by

$$A^{KG-offline,n} = \arg \max_a \left(\sum_{S^{n+1}} P(S^{n+1}|S^{a,n}) v^{KG,n}(S^{a,n}, S^{n+1}) \right). \quad (12.27)$$

Both our online policy $A^{KG-online,n}$ and offline policy $A^{KG-offline,n}$ require computing an expectation using the probability $P(S^{n+1}|S^{a,n})$. In practice, this is likely to be difficult to execute, and as a result we suggest using a simulated approximation

$$\sum_{S^{n+1}} P(S^{n+1}|S^{a,n}) v^{KG,n}(S^{a,n}, S^{n+1}) \approx \frac{1}{K} \sum_{k=1}^K v^{KG,n}(S^{a,n}, S_k^{n+1})$$

where $S_k^{n+1} = S^{M,W}(S^{a,n}, W_k^{n+1})$ and where W_k^{n+1} the k th simulated observation of W^{n+1} and $S^{M,W}(S^{a,n}, W_k^{n+1})$ is the function we use to capture the evolution from post-decision state to pre-decision state.

The adaptation of the knowledge gradient to dynamic programs with a physical state is quite new (as of this writing). Our decision to include such new research reflects what is a paucity of practical algorithms to do efficient learning in a principled way. The method was first suggested in Ryzhov et al. (2010) who report on experiments for two applications: a newsvendor problem and an energy storage problem. Both are fairly small dynamic programs (which make lookup tables possible) and serve only as an initial test. The online and offline knowledge gradient policies were tested using both online and offline settings. That is, in the online setting, we accumulated the contributions over 150 iterations, using a discount factor of $\gamma = 0.99$. In the offline setting, we trained the value function for 150 iterations, and then evaluated the policy produced by the resulting value function approximation using additional simulations without any subsequent updating.

These policies were compared against the following alternatives

- Bayesian Q -learning. This strategy uses equation (12.21) to make decisions, and uses Bayesian updating after each step.
- Bayesian learning with ϵ -greedy exploration.
- ADP using ϵ -greedy exploration.

For the newsvendor problem, it was also possible to calculate the optimal policy for the offline version of the problem as a benchmark.

Table 12.4 Comparison of different learning policies for online and offline versions of a newsvendor problem

Policy	Offline Objective		Online Objective	
	Mean	Avg. Std. Dev.	Mean	Avg. Std. Dev.
Optimal	10795	3.31	—	—
Offline KG	10639	3.45	-1645	59.90
Online KG	10188	3.16	3961	31.44
Bayesian Q -learning	10633	3.10	3820	46.15
Bayesian ϵ -greedy	10587	3.29	3328	35.28
ADP ϵ -greedy	9626	3.22	2449	23.98

Note: Also shown are the standard deviations in the estimates of the performance of each policy.

These policies were tested on 1000 different problem instances, where each was run for 150 iterations. The results are shown in Table 12.4. As we would expect, the offline KG policy worked best on the offline problems, while the online KG policy worked best on the online problems. The KG policies outperformed the other policies, although the Bayesian Q -learning policy for this example was close.

The policies were also compared on an energy storage problem with continuous states, which eliminated the option of finding an optimal policy. The results are shown in Table 12.5, which again shows that the offline KG works best on the offline problem, while the online KG works best on the online problem. In this case the KG policies appeared to significantly outperform the other policies.

The major bottleneck computationally with this method is not the calculation of the knowledge gradient but rather the updating of the beliefs using equations (12.22) and (12.23). We circumvent this problem in the next section when we make the transition to using a linear belief model.

12.6.4 The Knowledge Gradient with Parametric Beliefs*

It is possible to adapt the knowledge gradient to problems with other types of belief structures. We focus our attention here on the most popular belief structure that uses a set of pre-specified basis functions to create a linear model.

Table 12.5 Comparison of different learning policies for online and offline versions of an energy storage problem

Policy	Offline Objective		Online Objective	
	Mean	Avg. Std. Dev.	Mean	Avg. Std. Dev.
Offline KG	208.39	0.33	-260.59	16.86
Online KG	68.10	0.24	155.30	6.01
Bayesian Q -learning	133.65	0.31	76.03	1.87
Bayesian ϵ -greedy	85.65	0.31	67.10	3.00
ADP ϵ -greedy	154.47	0.35	7.09	2.57

The steps required for using a linear model are basically the same as we used in Section 12.6.3 or a lookup table, with the following changes:

- *Updating the belief $V^n(s)$.* We are using a linear model with basis functions, so we use the steps for updating the regression vector θ^n described in Section 9.9, where it is adapted for a Bayesian setting.
- *Computing the knowledge gradient $v^{KG,n}$.* We use the adaptation of the knowledge gradient calculation presented in Section 7.4.4 when using a linear model.

The use of a linear model is computationally much faster than a lookup table, even when dealing with small problems. As long as the number of features is not too large, updating both θ^n and the covariance matrix Σ^θ is quite fast.

The online and offline variations of the knowledge gradient were again applied to the energy storage problem, but this time using a simple linear model with five basis functions. These policies were compared against ADP based on approximate value iteration, and a pure exploitation policy. The results are shown in Table 12.6. This time the knowledge gradient policy seems to be having a significant impact.

12.6.5 Discussion

Considerable caution should be used before drawing conclusions based on the very limited experimental work presented in Tables 12.4, 12.5, and 12.6. The results are encouraging, and this appears to be the first learning policy for dynamic programs with a physical state that scales to problems of realistic size (using a linear belief model). Reflecting the derivation of the policy based on first principles, the knowledge gradient does not have any tunable parameters.

The biggest challenge with Bayesian learning, especially when using a linear belief model, is creating the prior. If you pick a poor prior, and especially if you do not accurately capture your confidence in the prior, you will get a poor policy regardless of your exploration policy.

This said, we feel there is considerable potential in the use of the knowledge gradient in a Bayesian setting. In addition to not needing any tunable parameters, it can be used in both online and offline settings. For example, we may use $A^{KG\text{-online},n}$ not only to determine what action to take that determines the next

Table 12.6 Comparison of different policies using a linear belief model on the energy storage problem

Policy	Offline Objective			Online Objective		
	Mean	Avg.	Std. Dev.	Mean	Avg.	Std. Dev.
Offline KG	702.62		2.58	-162.72		32.54
Online KG	320.49		1.96	6.01		39.01
ADP ϵ -greedy	-350.21		6.65	0.71		25.10
Pure exploitation	-618.78		0.85	-125.53		35.69

downstream state (the behavior or sampling policy), we can also use it to update our estimate of the value of being in a state by computing

$$\begin{aligned}\hat{v}^n = \max_a & (C(S^n, a) + \gamma V^n(S^{a,n})) \\ & + \gamma \sum_{S^{n+1}} P(S^{n+1}|S^{a,n}) v^{KG,n}(S^{a,n}, S^{n+1})\end{aligned}\right).$$

Although the value of information term would appear to distort the estimate of the value of being in a state, this term tends to zero as the algorithm progresses. However, this strategy means that we are using an online policy, thereby avoiding the difficulties inherent in offline policies that arise when we are trying to evaluate a learning policy while following a different sampling policy.

12.7 BIBLIOGRAPHIC NOTES

Section 12.3 A nice introduction to various learning strategies is contained in Kaelbling (1993) and Sutton and Barto (1998). Thrun (1992) contains a good discussion of exploration in the learning process. The discussion of Boltzmann exploration and epsilon-greedy exploration is based on Singh et al. (2000). Interval estimation is due to Kaelbling (1993). The upper confidence bound is due to Lai and Robbins (1985). We use the version of the UCB rule given in Lai (1987). The UCB1 policy is given in Auer et al. (2002). Analysis of UCB policies are given in Lai and Robbins (1985) and Auer et al. (2002), as well as Chang et al. (2007).

Section 12.4 What came to be known as “Gittins indices” was introduced in Gittins and Jones (1974) to solve bandit problems (see DeGroot 1970, for a discussion of bandit problems before the development of Gittins indices). This was more thoroughly developed in Gittins (1979, 1981, 1989). Whittle (1982) and Ross (1983) provide very clear tutorials on Gittins indices, helping launch an extensive literature on the topic (e.g., see Lai and Robbins, 1985; Berry and Fristedt, 1985; Weber, 1992). The work on approximating Gittins indices is due to Brezzi and Lai (2002), Yao (2006), and Chick and Gans (2009).

Section 12.5 The knowledge gradient policy for normally distributed rewards and independent beliefs was introduced by Gupta and Miescke (1996), and subsequently analyzed in greater depth by Powell et al. (2008). The knowledge gradient for correlated beliefs was introduced by Frazier et al. (2009). The adaptation of the knowledge gradient for online problems is due to Ryzhov and Powell (2009).

Section 12.6 The problem of introducing active learning in dynamic programs (implicitly, where the state represents a physical state) has been an area of

interest for some time, but characterized primarily by heuristic algorithms. Mike Duff and Andy Barto (Duff and Barto, 1997, 2003; Duff, 2002) developed an adaptation of Gittins indices for dynamic programs using the concept of the local bandit approximation. Poupart et al. (2006) give an analytic solution of the online learning problem and proposes an algorithm called BEETLE, which is demonstrated on some very small problems. The adaptation of the knowledge gradient to problems with a physical state was first developed in Ryzhov and Powell (2010a). The adaptation of the knowledge gradient for parametric beliefs was first given in Negoescu et al. (2010) but adapted to dynamic programming in Ryzhov and Powell (2010a).

PROBLEMS

- 12.1** Joe Torre, former manager of the great Yankees, had to struggle with the constant game of guessing who his best hitters are. The problem is that he could only observe a hitter if he put him in the order. Say he had four batters that he was looking at. Table 12.7 shows their actual batting averages (batter 1 will produce hits 30 percent of the time, batter 2 will get hits 32 percent of the time, etc.). Unfortunately, Joe did not know these numbers. As far as he was concerned, these were all 0.300 hitters.

For each man at bat, Joe had to pick one of these hitters to hit. Table 12.7 shows what would have happened if each batter were given a chance to hit (1 = hit, 0 = out). Again, Joe did not get to see all these numbers. He only got to observe the outcome of the hitter who gets to hit.

Assume that Joe always let the batter hit with the best batting average. Assume that he used an initial batting average of 0.300 for each hitter

Table 12.7 Data for problem 12.1

Day	Actual Batting Average			
	0.300	0.320	0.280	0.260
	A	B	C	D
1	0	1	1	1
2	1	0	0	0
3	0	0	0	0
4	1	1	1	1
5	1	1	0	0
6	0	0	0	0
7	0	0	1	0
8	1	0	0	0
9	0	1	0	0
10	0	1	0	1

(in case of a tie, used batter 1 over batter 2 over batter 3 over batter 4). Whenever a batter got to hit, calculate a new batting average by putting an 80 percent weight on your previous estimate of his average plus a 20 percent weight on how he did for his at bat. By this logic, you would choose batter 1 first. Since he did not get a hit, his updated average would be $0.80(0.200) + 0.20(0) = 0.240$. For the next at bat, you would choose batter 2 because your estimate of his average is still 0.300, while your estimate for batter 1 is now 0.240.

After 10 at bats, who would you conclude is your best batter? Comment on the limitations of this way of choosing the best batter. Do you have a better idea? (It would be nice if your idea is practical.)

- 12.2** There are four paths you can take to get to your new job. On the map, they all seem reasonable, and as far as you can tell, they all take 20 minutes, but the actual times vary quite a bit. The value of taking a path is your current estimate of the travel time on that path. In Table 12.8 we show the travel time on each path if you had traveled that path. Start with an initial estimate of each value function of 20 minutes with your tie-breaking rule to use the lowest numbered path. At each iteration take the path with the best estimated value, and update your estimate of the value of the path based on your experience. After 10 iterations, compare your estimates of each path to the estimate you obtain by averaging the “observations” for each path over all 10 days. Use a constant stepsize of 0.20. How well did you do?

Table 12.8 Data for exercise 12.2

Day	Paths			
	1	2	3	4
1	37	29	17	23
2	32	32	23	17
3	35	26	28	17
4	30	35	19	32
5	28	25	21	26
6	24	19	25	31
7	26	37	33	30
8	28	22	28	27
9	24	28	31	30
10	33	29	17	29

- 12.3** We are going to try again to solve our asset selling problem. We assume that we are holding a real asset and we are responding to a series of offers. Let \hat{p}_t be the t th offer, which is uniformly distributed between 500 and 600 (all prices are in thousands of dollars). We also assume that each offer is independent of all prior offers. You are willing to consider up to 10 offers,

and your goal is to get the highest possible price. If you have not accepted the first nine offers, you must accept the 10th offer.

- (a) Write out the decision function you would use in an approximate dynamic programming algorithm in terms of a Monte Carlo sample of the latest price and a current estimate of the value function approximation.
 - (b) Write out the updating equations (for the value function) you would use after solving the decision problem for the t th offer.
 - (c) Implement an approximate dynamic programming algorithm using *synchronous* state sampling. Using 1000 iterations, write out your estimates of the value of being in each state immediately after each offer. For this exercise you will need to discretize prices for the purpose of approximating the value function. Discretize the value function in units of 5 dollars.
 - (d) From your value functions, infer a decision rule of the form “sell if the price is greater than \bar{p}_t .”
- 12.4** Suppose that you are considering five options. The actual value θ_d , the initial estimate $\bar{\theta}_d^0$, and the initial standard deviation $\bar{\sigma}_d^0$ of each $\bar{\theta}_d^0$ are given in Table 12.9. Perform 20 iterations of each of the following algorithms:
- (a) Gittins exploration using $\Gamma(n) = 2$.
 - (b) Interval exploration using $\alpha/2 = 0.025$.
 - (c) The upper confidence bound algorithm using $W^{\max} = 6$.
 - (d) The knowledge gradient algorithm.
 - (e) Pure exploitation.
 - (f) Pure exploration.
- Each time you sample a decision, randomly generate an observation $W_d = \theta_d + \sigma^\varepsilon Z$, where $\sigma^\varepsilon = 1$ and Z is normally distributed with mean 0 and variance 1. [Hint: You can generate random observations of Z in Excel by using =NORMSINV(RAND())].

Table 12.9 Data for exercise 12.4

Decision	μ	$\bar{\theta}^0$	$\bar{\sigma}^0$
1	1.4	1.0	2.5
2	1.2	1.2	2.5
3	1.0	1.4	2.5
4	1.5	1.0	1.5
5	1.5	1.0	1.0

- 12.5** Repeat exercise 12.4 using the data in Table 12.10, with $\sigma^\varepsilon = 10$.

Table 12.10 Data for exercise 12.5

Decision	μ	$\bar{\theta}^0$	$\bar{\sigma}^0$
1	100	100	20
2	80	100	20
3	120	100	20
4	110	100	10
5	60	100	30

12.6 Repeat exercise 12.4 using the data in Table 12.11, with $\sigma^\varepsilon = 20$.

Table 12.11 Data for exercise 12.6

Decision	μ	$\bar{\theta}^0$	$\bar{\sigma}^0$
1	120	100	30
2	110	105	30
3	100	110	30
4	90	115	30
5	80	120	30

12.7 Nomadic trucker project revisited. We are going to again solve the nomadic trucker problem first posed in exercise 4.18, but this time we are going to experiment with our more advanced information acquisition policies. Assume that we are solving the finite horizon problem with a discount factor of $\gamma = 0.1$. Compare the following policies for determining the next state to visit:

- (a) Pure exploitation. This is the policy that you would have implemented in exercise 4.18.
 - (b) Pure exploration. After solving a decision problem (and updating the value of being in a city), choose the next city at random.
 - (c) Boltzmann exploration. Use equation (12.3) to determine the next city to visit.
 - (d) Gittins exploration. Use equation (12.20) to decide which city to visit next, and use equation (12.10) to compute $\Gamma(n)$. You will have to experiment to find the best value of ρ^G .
 - (e) [(e)] Interval estimation. Use $\alpha/2 = 0.025$.
 - (f) [(f)] The knowledge gradient policy.
- 12.8** Repeat exercise 12.7 using a discount factor $\gamma = 0.8$. How does this affect the behavior of the search process? How does it change the performance of the different methods for collecting information?

Value Function Approximations for Resource Allocation Problems

In Chapter 8 we introduced general-purpose approximation tools for calculating value functions without needing to assume any special structural properties. In this chapter we focus on approximating value functions that arise in resource allocation problems. For example, if R is the amount of resource available (water, oil, money, vaccines) and $V(R)$ is the value of having R units of our resource, we often find that $V(R)$ might be linear (or approximately linear), nonlinear (concave), piecewise linear, or in some cases, simply something continuous. Value functions with this structure yield to special approximation strategies.

We consider a series of strategies for approximating the value function using increasing sophistication:

Linear approximations. These are typically the simplest nontrivial approximations that work well when the functions are approximately linear over the range of interest. It is important to realize that we mean “linear in the state” as opposed to the more classical “linear in the parameters” model that we considered earlier.

Separable, piecewise linear, concave (convex if minimizing). These functions are especially useful when we are interested in integer solutions. Separable functions are relatively easy to estimate and offer special structural properties when solving the optimality equations.

Auxiliary functions. This is a special class of algorithms that fixes an initial approximation and uses stochastic gradients to adjust the function.

General nonlinear regression equations. Here we bring the full range of tools available from the field of regression.

Cutting planes. This is a technique for approximating multidimensional, piecewise linear functions that has proved to be particularly powerful for multistage linear programs such those that arise in dynamic resource allocation problems.

An important dimension of this chapter will be our use of derivatives to estimate value functions, rather than just the value of being in a state. When we want to determine how much oil should be sent to a storage facility, what matters most is the marginal value of additional oil. For some problem classes this is a particularly powerful device that dramatically improves convergence.

13.1 VALUE FUNCTIONS VERSUS GRADIENTS

It is common in dynamic programming to talk about the problem of estimating the value of being in a state. There are many applications where it is more useful to work with the derivative or gradient of the value function. In one community, where “heuristic dynamic programming” represents approximate dynamic programming based on estimating the value of being in a state, “dual heuristic programming” refers to approximating the gradient.

We are going to use the context of resource allocation problems to illustrate the power of using the gradient. In principal, the challenge of estimating the slope of a function is the same as that of estimating the function itself (the slope is simply a different function). However, there can be important practical advantages to estimating slopes. First, if the function is approximately linear, it may be possible to replace estimates of the value of being in each state (or set of states) with a single parameter that is the estimate of the slope of the function. Estimating constant terms is typically unnecessary.

A second and equally important difference is that if we estimate the value of being in a state, we get one estimate of the value of being in a state when we visit that state. When we estimate a gradient, we get an estimate of a derivative for each type of resource. For example, if $R_t = (R_{tr})_{r \in \mathcal{R}}$ is our resource vector and $V_t(R_t)$ is our value function, then the gradient of the value function with respect to R_t would look like

$$\nabla_{R_t} V_t(R_t) = \begin{pmatrix} \hat{v}_{tr_1} \\ \hat{v}_{tr_2} \\ \vdots \\ \hat{v}_{tr_{|\mathcal{A}|}} \end{pmatrix},$$

where

$$\hat{v}_{tr_i} = \frac{\partial V_t(R_t)}{\partial R_{tr_i}}.$$

There may be additional work required to obtain each element of the gradient, but the incremental work can be far less than the work required to get the value

function itself. This is particularly true when the optimization problem naturally returns these gradients (e.g., dual variables from a linear program), and even when we have to resort to numerical derivatives. After we have all the calculations to solve a problem once, solving small perturbations can be relatively inexpensive.

There is one important problem class where finding the value of being in a state is equivalent to finding the derivative. That is the case of managing a single resource (see Section 5.3.1). In this case the state of our system (the resource) is the attribute vector r , and we are interested in estimating the value $V(r)$ of our resource being in state r . Alternatively, we can represent the state of our system using the vector R_t , where $R_{tr} = 1$ indicates that our resource has attribute r (we assume that $\sum_{r \in \mathcal{R}} R_{tr} = 1$). In this case the value function can be written

$$V_t(R_t) = \sum_{r \in \mathcal{R}} v_{tr} R_{tr}.$$

Here the coefficient v_{tr} is the derivative of $V_t(R_t)$ with respect to R_{tr} .

In a typical implementation of an approximate dynamic programming algorithm, we would only estimate the value of a resource when it is in a particular state (given by the vector r). This is equivalent to finding the derivative \hat{v}_r only for the value of r where $R_{tr} = 1$. By contrast, computing the gradient $\nabla_{R_t} V_t(R_t)$ implicitly assumes that we are computing \hat{v}_r for each $r \in \mathcal{R}$. There are some algorithmic strategies (we will describe an example of this in Section 13.7) where this assumption is implicit in the algorithm. Computing \hat{v}_r for all $r \in \mathcal{R}$ is reasonable if the attribute state space is not too large (e.g., if r is a physical location among a set of several hundred locations). If r is a vector, then enumerating the attribute space can be prohibitive (it is, in effect, the “curse of dimensionality” revisited).

Given these issues, it is critical to first determine whether it is necessary to estimate the slope of the value function, or the value function itself. The result can have a significant impact on the algorithmic strategy.

13.2 LINEAR APPROXIMATIONS

There are a number of problems where we are allocating resources of different types. As in the past we let r be the attributes of a resource and R_{tr} be the quantity of resources with attribute r in our system at time t with $R_t = (R_{tr})_{r \in \mathcal{R}}$. R_t may describe our investments in different resource classes (growth stocks, value stocks, index funds, international mutual funds, domestic stock funds, bond funds). Or R_t might be the amount of oil we have in different reserves or the number of people in a management consulting firm with particular skill sets. We want to make decisions to acquire or sell resources of each type, and we want to capture the impact of decisions now on the future through a value function $V_t(R_t)$.

Rather than attempt to estimate $V_t(R_t)$ for each value of R_t , it may make more sense to estimate a linear approximation of the value function with respect to the resource vector. Linear approximations can work well when the single-period contribution function is continuous and increases or decreases monotonically over

the range we are interested in (the function may or may not be differentiable). They can also work well in settings where the value function increases or decreases monotonically, even if the value function is neither convex nor concave, nor even continuous.

To illustrate, consider the problem of purchasing a commodity. Let

$$\hat{D}_t = \text{random demand during time interval } t,$$

$$R_t = \text{resources on hand at time } t \text{ just before we make an ordering decision},$$

$$x_t = \text{quantity ordered at time } t \text{ to be used during time interval } t+1,$$

$$R_t^x = \text{resources available just after we make a decision},$$

$$\hat{p}_t = \text{market price for selling commodities during time interval } t,$$

$$c_t = \text{purchase cost for commodities purchased at time } t.$$

At time t , we know the price \hat{p}_t and demand \hat{D}_t for time interval t , but we have to choose how much to order for the next time interval. The transition equations are given by

$$R_t^x = R_t + x_t,$$

$$R_{t+1} = [R_t^x - \hat{D}_{t+1}]^+.$$

The value of being in state R_t is given by

$$V_t(R_t) = \max_{x_t} \mathbb{E}(\hat{p}_{t+1} \min\{R_t + x_t, \hat{D}_{t+1}\} - c_t x_t + V_t^x(R_t + x_t)), \quad (13.1)$$

where $V_t^x(R_t^x)$ is the post-decision value function, while $V_t(R_t)$ is the traditional value function around the pre-decision state. Now assume that we introduce a linear value function approximation

$$\bar{V}_t^x(R_t^x) \approx \bar{V}_t R_t^x.$$

The resulting approximation can be written

$$\begin{aligned} \tilde{V}_t(R_t) &= \max_{x_t} \mathbb{E}(\hat{p}_{t+1} \min\{R_t + x_t, \hat{D}_{t+1}\} - c_t x_t + \bar{v}_t R_t^x) \\ &= \max_{x_t} \mathbb{E}(\hat{p}_{t+1} \min\{R_t + x_t, \hat{D}_{t+1}\} - c_t x_t + \bar{v}_t(R_t + x_t)). \end{aligned} \quad (13.2)$$

We assume that we can compute, or at least approximate, the expectation in equation (13.2). If this is the case, we may approximate the gradient at iteration n using a numerical derivative, as in

$$\hat{v}_t = \tilde{V}_t(R_t + 1) - \tilde{V}_t(R_t).$$

We now use \hat{v}_t to update the value function \bar{V}_{t-1} using

$$\bar{v}_t \leftarrow (1 - \alpha)\bar{v}_{t-1} + \alpha \hat{v}_t.$$

Normally we would use \hat{v}_t to update $\bar{V}_{t-1}(R_{t-1}^x)$ around the previous post-decision state variable R_{t-1}^x . Linear approximations, however, are a special case, since the slope is the same for all R_{t-1}^x , which means that it is also the same for $R_{t-1} = R_{t-1}^x - x_{t-1}$.

Linear approximations are useful in two settings. First, the value function may be approximately linear over the range that we are interested in. Imagine, for example, that you are trying to decide how many shares of stock you want to sell, where the range is between 0 and 1000. As an individual investor, it is unlikely that selling all 1000 shares will change the market price. However, if you are a large mutual fund and you are trying to decide how many of your 50 million shares you want to sell, it is likely that such a high volume would move the market price. When this happens, we need a nonlinear function.

A second use of linear approximations arises when managing resources such as people and complex equipment such as locomotives or aircraft. Let r be the attributes of a resource and R_{tr} be the number of resources with attribute r at time t . Then it is likely that R_{tr} will be 0 or 1, implying that a linear function is all we need. For these problems a linear value function is particularly convenient because it means that we need one parameter, \bar{v}_{tr} , for each attribute r .

13.3 PIECEWISE-LINEAR APPROXIMATIONS

There are many problems where we have to estimate the value of having a quantity R of some resource (where R is a scalar). We might want to know the value of having R dollars in a budget, R pieces of equipment, or R units of some inventory. R may be discrete or continuous, but we are going to focus on problems where R is either discrete or is easily discretized.

Suppose now that we want to estimate a function $V(R)$ that gives the value of having R resources. There are applications where $V(R)$ increases or decreases monotonically in R . There are other applications where $V(R)$ is piecewise linear, concave (or convex) in R , which means that the slopes of the function are monotonically decreasing (if the function is concave) or increasing (if it is convex). When the function (or the slopes of the function) is steadily increasing or decreasing, we say that the function is *monotone*. If the function is increasing in the state variable, we say that it is “monotonically increasing,” or that it is *isotone* (although the latter term is not widely used). To say that a function is “monotone” can mean that it is monotonically increasing or decreasing.

Say we have a function that is monotonically decreasing, which means that while we do not know the value function exactly, we know that $V(R+1) \leq V(R)$ (for scalar R). If our function is piecewise linear concave, then we will assume that $V(R)$ refers to the slope at R (more precisely, to the right of R). Assume that our current approximation $\bar{V}^{n-1}(R)$ satisfies this property, and that at iteration n , we have a sample observation of $V(R)$ for $R = R^n$. If our function is piecewise linear concave, then \hat{v}^n would be a sample realization of a derivative of the function. If

we use our standard updating algorithm, we would write

$$\bar{V}^n(R^n) = (1 - \alpha_{n-1})\bar{V}^{n-1}(R^n) + \alpha_{n-1}\hat{v}^n.$$

After the update, it is quite possible that our updated approximation no longer satisfies our monotonicity property. In this section we review three strategies for maintaining monotonicity:

The leveling algorithm. A simple method that imposes monotonicity by simply forcing elements of the series that violate monotonicity to a larger or smaller value so that monotonicity is restored.

The SPAR algorithm. SPAR takes the points that violate monotonicity and simply averages them.

The CAVE algorithm. If there is a monotonicity violation after an update, CAVE simply expands the range of the function over which the update is applied.

The leveling algorithm and the SPAR algorithm enjoy convergence proofs, but the CAVE algorithm is the one that works the best in practice. We report on all three to provide a sense of algorithmic choices, and an enterprising reader may design a modification of one of the first two algorithms to incorporate the features of CAVE that make it work so well.

The Leveling Algorithm

The leveling algorithm uses a simple updating logic that can be written as follows:

$$\bar{V}^n(y) = \begin{cases} (1 - \alpha_{n-1})\bar{V}^{n-1}(R^n) + \alpha_{n-1}\hat{v}^n & \text{if } y = R^n, \\ \bar{V}^n(y) \vee \left\{ (1 - \alpha_{n-1})\bar{V}^{n-1}(R^n) + \alpha_{n-1}\hat{v}^n \right\} & \text{if } y > R^n, \\ \bar{V}^n(y) \wedge \left\{ (1 - \alpha_{n-1})\bar{V}^{n-1}(R^n) + \alpha_{n-1}\hat{v}^n \right\} & \text{if } y < R^n, \end{cases} \quad (13.3)$$

where $x \wedge y = \max\{x, y\}$, and $x \vee y = \min\{x, y\}$. Equation (13.3) starts by updating the slope $\bar{V}'^n(y)$ for $y = R^n$. We then want to make sure that the slopes are declining. So, if we find a slope to the right that is larger, then we simply bring it down to our estimated slope for $y = R^n$. Similarly, if there is a slope to the left that is smaller, then we simply raise it to the slope for $y = R^n$. The steps are illustrated in Figure 13.1.

The leveling algorithm is easy to visualize, but it is unlikely to be the best way to maintain monotonicity. For example, we may update a value at $y = R^n$ for which there are very few observations. But because it produces an unusually high or low estimate, we find ourselves simply forcing other slopes higher or lower just to maintain monotonicity.

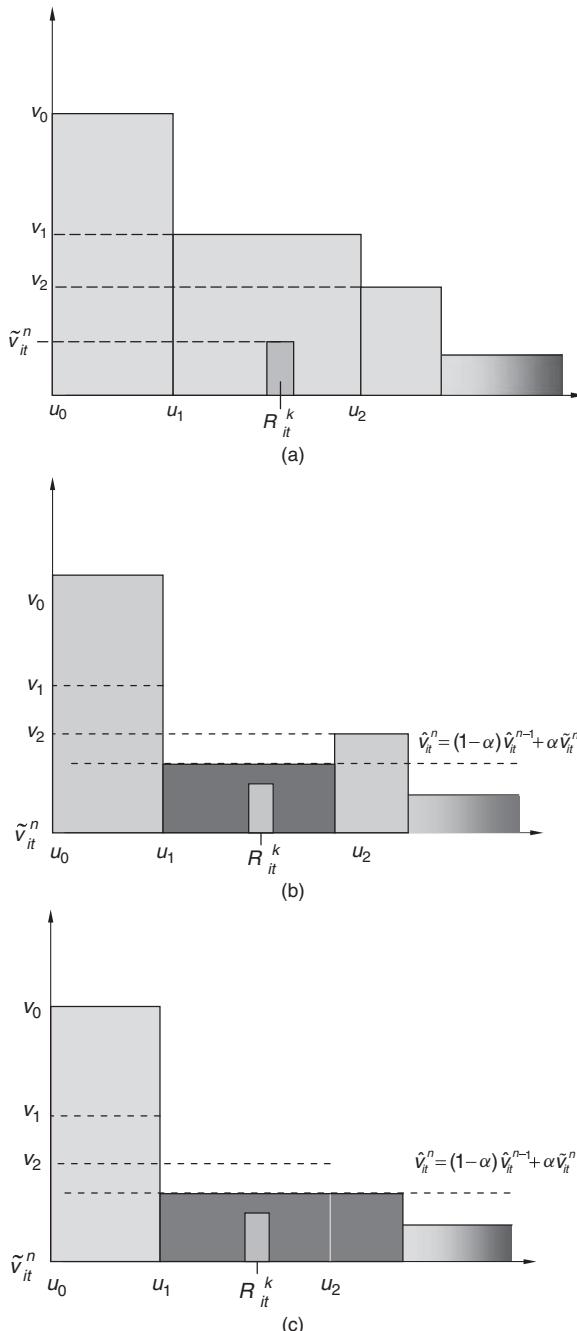


Figure 13.1 Steps of the leveling algorithm. (a) The initial monotone function, with the observed R and observed value of the function \hat{v} ; (b) the function after updating the single segment, producing a nonmonotone function; (c) the function after monotonicity restored by leveling the function.

The SPAR Algorithm

A more elegant strategy is the SPAR (separable projective approximation routine), which works as follows. Say that we start with our original set of values $(\bar{V}^{n-1}(y))_{y \geq 0}$, and that we sample $y = R^n$ and obtain an estimate of the slope \hat{v}^n . After the update we obtain the set of values (which we store temporarily in the function $\bar{z}^n(y)$):

$$\bar{z}^n(y) = \begin{cases} (1 - \alpha_{n-1})\bar{V}^{n-1}(y) + \alpha_{n-1}\hat{v}^n, & y = R^n, \\ \bar{V}^{n-1}(y) & \text{otherwise.} \end{cases} \quad (13.4)$$

If $\bar{z}^n(y) \geq \bar{z}^n(y+1)$ for all y , then we are in good shape. If not, then either $\bar{z}^n(R^n) < \bar{z}^n(R^n+1)$ or $\bar{z}^n(R^n-1) < \bar{z}^n(R^n)$. We can fix the problem by solving the projection problem

$$\min_v \|v - \bar{z}^n\|^2 \quad (13.5)$$

subject to

$$v(z+1) - v(z) \leq 0. \quad (13.6)$$

Solving this projection is especially easy. Imagine that after our update, we have a violation to the left. The projection is achieved by averaging the updated cell with all the cells to the left that create a monotonicity violation. This means that we want to find the largest $i \leq R^n$ such that

$$\bar{z}^n(i-1) \geq \frac{1}{R^n - i + 1} \sum_{y=i}^{R^n} \bar{z}^n(y).$$

In other words, we can start by averaging the values for R^n and $R^n - 1$ and checking to see if we now have a concave function. If not, we keep lowering the left end of the range until we either restore monotonicity or reach $y = 0$. If our monotonicity violation is to the right, then we repeat the process to the right.

The steps of the algorithm are given in Figure 13.2, with an illustration given in Figure 13.3. We start with a monotone set of values (Figure 13.3a), then update one of the values to produce a monotonicity violation (Figure 13.3b), and finally average the violating values together to restore monotonicity (Figure 13.3c).

There are a number of variations of these algorithms that help with convergence. For example, in the SPAR algorithm we can solve a weighted projection that gives more weight to slopes that have received more observations. To do this, we weight each value of $\bar{y}(r)$ by the number of observations that segment r has received when computing $\bar{y}^n(i-1)$.

The CAVE Algorithm

A particularly useful variation is to perform an initial update (when we compute \bar{y}) over a wider interval than just $y = R^n$. Say we are given a parameter δ^0 that

Step 0. Initialize \bar{V}^0 and set $n = 1$.

Step 1. Sample R^n .

Step 2. Observe a sample of the value function \hat{v}^n .

Step 3. Calculate the vector z^n as follows

$$z^n(y) = \begin{cases} (1 - \alpha_{n-1})V_{R^n}^{n-1} + \alpha_{n-1}\hat{v}^n & \text{if } y = R^n, \\ v^{n-1}(y) & \text{otherwise.} \end{cases}$$

Step 4. Project the updated estimate onto the space of monotone functions,

$$v^n = \Pi(z^n),$$

by solving (13.5)–(13.6). Increase n by one and go to step 1.

Figure 13.2 Learning form of the separable projective approximation routine (SPAR).

has been chosen so that it is approximately 20 to 50 percent of the maximum value that R^n might take. Now compute $\bar{z}(y)$ using

$$\bar{z}^n(y) = \begin{cases} (1 - \alpha_{n-1})\bar{V}^{n-1}(y) + \alpha_{n-1}\hat{v}^n, & R^n - \delta^n \leq y \leq R^n + \delta^n, \\ \bar{V}^{n-1}(y) & \text{otherwise.} \end{cases}$$

Here we are using \hat{v}^n to update a wider range of the interval. We then apply the same logic for maintaining monotonicity (concavity if these are slopes). We start with the interval $R^n \pm \delta^0$, but we have to periodically reduce δ^0 . We might, for example, track the objective function (call it F^n), and update the range using

$$\delta^n = \begin{cases} \delta^{n-1} & \text{if } F^n \geq F^{n-1} - \epsilon, \\ \max\{1, 0.5\delta^{n-1}\} & \text{otherwise.} \end{cases}$$

While the rules for reducing δ^n are generally ad hoc, we have found that this is critical for fast convergence. The key is that we have to pick δ^0 so that it plays a critical scaling role, since it has to be set so that it is roughly on the order of the maximum value that R^n can take. If SPAR or the leveling algorithm are going to be successful, then these will have to be adapted to solve these scaling problems.

13.4 SOLVING A RESOURCE ALLOCATION PROBLEM USING PIECEWISE-LINEAR FUNCTIONS

Scalar piecewise-linear functions have proved to be an exceptionally powerful way of solving a large number of stochastic resource allocation problems. We can describe the algorithm with a minimum of technical details using what is known as a “plant–warehouse–customer” model, which is a form of multidimensional newsvendor problem. Imagine that we have the problem depicted in Figure 13.4a.

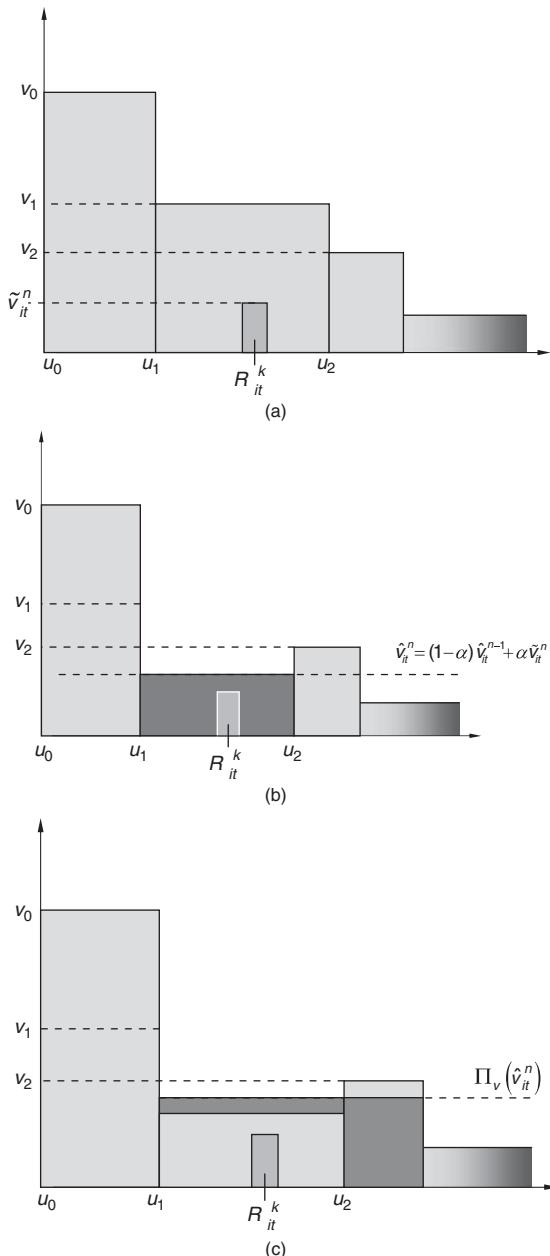


Figure 13.3 Steps of the SPAR algorithm. (a) The initial monotone function, with the observed R and observed value of the function \hat{v} ; (b) the function after updating the single segment, producing a nonmonotone function; (c) the function after the projection operation.

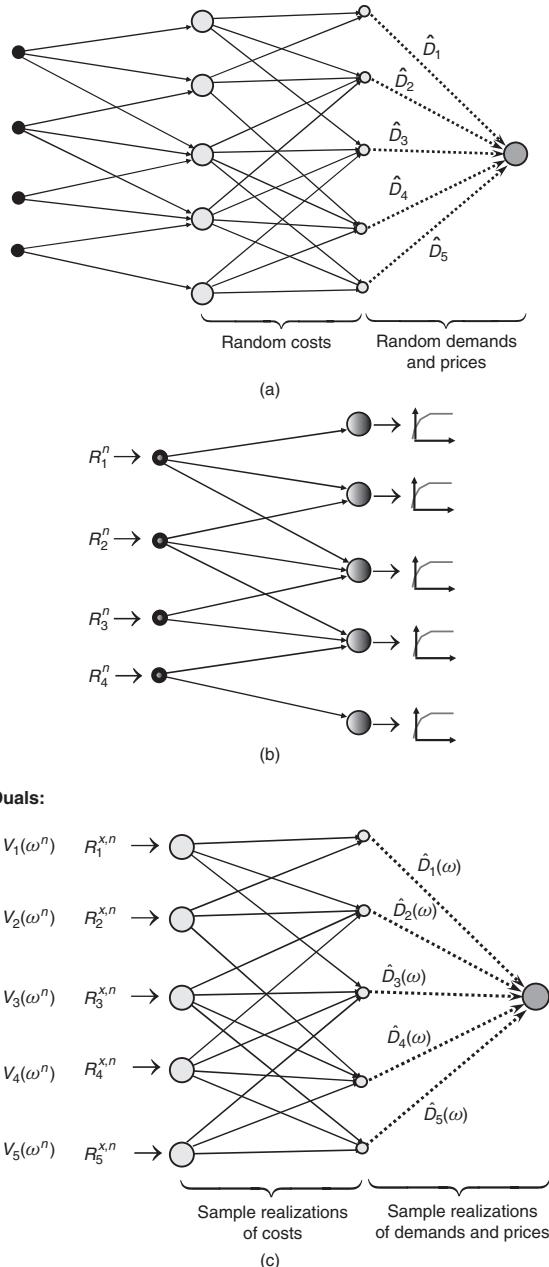


Figure 13.4 Steps in estimating separable, piecewise-linear approximations for two-stage stochastic programs. (a) The two-stage problem with stochastic second-stage data. (b) Solving the first stage using a separable, piecewise-linear approximation of the second stage. (c) Solving a Monte Carlo realization of the second stage and obtaining dual variables.

We start by shipping “product” out of the four “plant” nodes on the left, and we have to decide how much to send to each of the five “warehouse” nodes in the middle. After making this decision, we observe the demands at the five “customer” nodes on the right.

We can solve this problem using separable, piecewise-linear value function approximations. Say we have an initial estimate of a piecewise-linear value function for resources at the warehouses (setting these equal to zero is fine). This gives us the network shown in Figure 13.4b, which is a small linear program (even when we have hundreds of plant and warehouse nodes). Solving this problem gives us a solution of how much to send to each node.

We then use the solution to the first stage (which gives us the resources available at each warehouse node), take a Monte Carlo sample of each of the demands, and solve a second linear program that sends product from each warehouse to each customer. What we want from this stage is the dual variable for each warehouse node, which gives us an estimate of the marginal value of resources at each node. Note that some care needs to be used here because these dual variables are not actually estimates of the value of one more resources, rather they are subgradients, which means that they may be the value of the last resource or the next resource, or something in between.

Finally, we use these dual variables to update the piecewise linear value functions using the methods described above. This process is repeated until the solution no longer seems to be improving.

Although we have described this algorithm in the context of a two-stage problem, the same basic strategy can be applied for problems with many time periods. Using approximate value iteration (TD(0)), we would step forward in time, and after solving each linear program, we would stop and use the duals to update the value functions from the previous time period (more specifically, around the previous post-decision state). For a finite horizon problem we would proceed until the last time period, then repeat the entire process until the solution seems to be converging.

With more work, we can implement a backward pass (TD(1)) by avoiding any value function updates until we reach the final time period, but we would have to retain information about the effect of incrementing the resources at each node by one unit (this is best done with a numerical derivative). We would then need to step back in time, computing the marginal value of one more resource at time t using information about the value of one more resource at time $t + 1$. These marginal values would be used to update the value function approximations.

This algorithmic strategy has some nice features:

- This is a very general model with applications that span equipment, people, product, money, energy, and vaccines. It is ideally suited for “single-layer” resource allocation problems (one type of resource, rather than pairs such as pilots and aircraft, locomotives and trains, or doctors and patients), although many two-layer problems can be reasonably approximated as single-layer problems.

- The methodology scales to very large problems, with hundreds or thousands of nodes, and tens of thousands of dimensions in the decision vector.
- We do not need to solve the exploration–exploitation problem. A pure exploitation strategy works fine. The reason has to do with the concavity of the value function approximations, which has the effect of pushing suboptimal value functions toward the correct solution.
- Piecewise linear value function approximations are quite robust, and avoid making any simplifying assumptions about the shapes of the value functions.

Chapter 14 provides more details on using these ideas for resource allocation problems, and closes with some descriptions of large, industrial applications. This brief sketch of an algorithm provides a hint of the power of separable piecewise-linear approximations in the contextual domain of resource allocation problems.

13.5 THE SHAPE ALGORITHM

A particularly simple algorithm for approximating continuous value functions starts with an initial approximation and then “tilts” this function to improve the approximation. The concept is most effective if it is possible to build an initial approximation, perhaps using some simplifications, that produces a “pretty good” solution.

13.5.1 The Basic Idea

The idea works as follows. Suppose that we are trying to solve the stochastic optimization problem we first saw in Section 7.2, which is given by

$$\min_{x \in \mathcal{X}} \mathbb{E} F(x, W), \quad (13.7)$$

where x is our decision variable and W is a random variable. We already know that we can solve this problem using a stochastic gradient algorithm that takes the form

$$x^n = x^{n-1} - \alpha_{n-1} \nabla_x F(x^{n-1}, W(\omega^n)). \quad (13.8)$$

The challenge we encounter with this algorithm is that the units of the decision variable and the gradient may be different, which means that our stepsize has to be properly scaled. Also, while these algorithms may work in the limit, convergence can be slow and choppy.

Now assume that we have a rough approximation of $\mathbb{E} F(x, W)$ as a function of x , which we call $\bar{F}^0(x)$. For example, if x is a scalar and we think that the optimal solution is approximately equal to a , we might let $\bar{F}^0(x) = (x - a)^2$. The SHAPE algorithm iteratively adjusts this initial approximation by first finding a stochastic gradient $\nabla_x F(x^{n-1}, W(\omega^n))$, and then computing an updated approximation using

$$\bar{F}^n(x) = \bar{F}^{n-1}(x) + \underbrace{\alpha_{n-1} (\nabla_x F(x^{n-1}, W(\omega^n)) - \nabla_x \bar{F}^{n-1}(x^{n-1}))}_I x.$$

Our updated approximation is the original approximation plus a linear correction term. Note that x is a variable while the term I is a constant. Since the correction term has the same units as the approximation, we can assume that $0 < \alpha_n \leq 1$. The linear correction term is simply the difference between the slope of the original function at a sample realization $W(\omega^n)$ and the exact slope of the approximation at x^{n-1} . Typically we would pick an approximation that is easy to differentiate.

The SHAPE algorithm is depicted graphically in Figure 13.5. Figure 13.5a shows the initial true function and the approximation $\bar{F}^0(x)$. Figure 13.5b shows the gradient of each function at the point x^{n-1} , with the difference between the two slopes. If there were no noise, and if our approximation were exact, these two gradients would match (at least at this point). Finally, Figure 13.5c shows the updated approximation after the correction has been added in. At this point the two slopes will match only if the stepsize $\alpha_{n-1} = 1$.

We can illustrate the SHAPE algorithm using a simple numerical example. Suppose that our problem is to solve

$$\max_{x \geq 0} \mathbb{E}\bar{F}(x, W) = \mathbb{E} \left\{ \frac{1}{2} \ln(x + W) - 2(x + W) \right\},$$

where W represents random measurement error, which is normally distributed with mean 0 and variance 4. Assume that we start with a convex approximation such as

$$\bar{F}^0(x) = 6\sqrt{x} - 2x.$$

We begin by obtaining the initial solution x^0

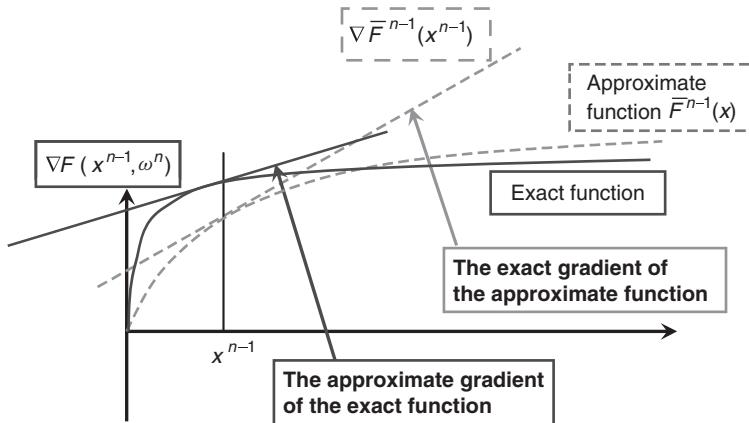
$$x^0 = \arg \max_{x \geq 0} (6\sqrt{x} - 2x).$$

Note that our solution to the approximate problem may be unbounded, requiring us to impose artificial limits. Since our approximation is concave, we can set the derivative equal to zero to find

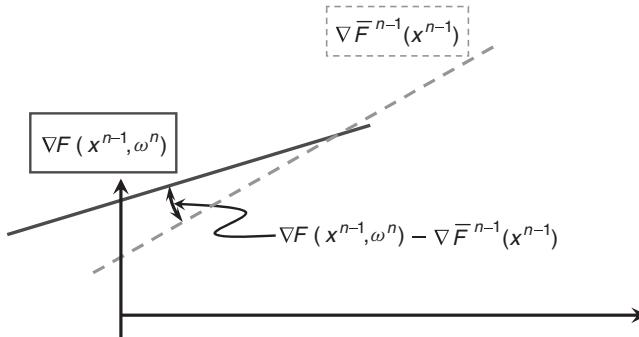
$$\begin{aligned} \nabla \bar{F}^0(x) &= \frac{3}{\sqrt{x}} - 2 \\ &= 0, \end{aligned}$$

which gives us $x^0 = 2.25$. Since $x^0 \geq 0$, it is optimal. To find the stochastic gradient, we have to sample the random variable W . Assume that $W(\omega^1) = 1.75$. Our stochastic gradient is then

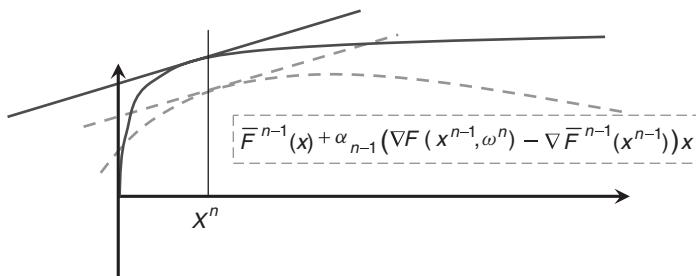
$$\begin{aligned} \nabla \bar{F}(x, W(\omega^1)) &= \frac{1}{2(x^0 + W(\omega^1))} - 2 \\ &= \frac{1}{2(2.25 + 1.75)} \\ &= 0.1250. \end{aligned}$$



(a) True function and initial approximation.



(b) Difference between stochastic gradient of true function and actual gradient of approximation.



(c) Updated approximation.

Figure 13.5 Illustration of the steps of the SHAPE algorithm.

Thus, while we have found the optimal solution to the approximate problem (which produces a zero slope), our estimate of the slope of the true function is positive, so we update $\bar{F}^1(s)$ with the adjustment

$$\begin{aligned}\bar{F}^1(x) &= 6\sqrt{x} - 2x - \alpha_0(0.1250 - 0)x \\ &= 6\sqrt{x} - 3.125x,\end{aligned}$$

where we have used $\alpha_0 = 1$.

13.5.2 A Dynamic Programming Illustration

Consider what happens when we have to solve a dynamic program (approximately). For this illustration, assume that R_t and x_t are scalars, although everything we are going to do works fine with vectors. Still using the value function around the post-decision state, we would start with the previous post-decision state R_{t-1}^x . We then use our sample path ω^n to find the pre-decision state $R_t^n = R^{M,W}(R_{t-1}^x, W_t(\omega^n))$ and solve

$$\tilde{V}_t^n(R_t^n) = \max_{x_t \in \mathcal{X}_t^n} (C(R_t^n, x_t) + \bar{V}_t^{n-1}(R^{M,x}(R_t^n, x_t))).$$

We need the derivative of $\tilde{V}_t^n(R_t^n)$. There are different ways of getting this depending on the structure of the optimization problem, but for the moment we will simply let

$$\hat{v}_t^n = \tilde{V}_t^n(R_t^n + 1) - \tilde{V}_t^n(R_t^n).$$

It is important to remember that embedded in the calculation of $\tilde{V}_t^n(R_t^n + 1)$ is the need to reoptimize x_t . Recall that we use \hat{v}_t^n , which is computed at $R_t = R_t^n$, to update $\bar{V}_{t-1}(R_{t-1}^x)$ as a function of the post-decision state R_{t-1}^x . Using the SHAPE algorithm, we update our approximation by way of

$$\bar{V}_{t-1}^n(R_{t-1}^x) = \bar{V}_{t-1}^{n-1}(R_{t-1}^x) + \alpha_{n-1}(\hat{v}_t^n - \nabla_R \bar{V}_{t-1}^{n-1}(R_{t-1}^x))R_{t-1}^x.$$

The steps of the SHAPE algorithm are given in Figure 13.6.

It is useful to consider how decisions are made using the SHAPE algorithm versus a stochastic gradient procedure. Using SHAPE, we would choose x_t by way of

$$x_t^n = \arg \max_{x_t \in \mathcal{X}_t^n} (C(R_t^n, x_t) + \bar{V}_t^{n-1}(R^{M,x}(R_t^n, x_t))). \quad (13.9)$$

Contrast determining the decision x_t^n from (13.9) to equation (13.8), which depends heavily on the choice of stepsize. This property is true of most algorithms that involve finding nonlinear approximations of the value function.

Step 0. Initialization.

Step 0a. Initialize $\bar{V}_t^0, \tilde{\omega}_t \in \mathcal{T}$.

Step 0b. Set $n = 1$.

Step 0c. Initialize R_0^1 .

Step 1. Choose a sample path ω^n .

Step 2. Do for $t = 0, 1, 2, \dots, T$:

Step 2a. Solve

$$\tilde{V}_t^n(R_t^n) = \max_{x_t \in \mathcal{X}_t^n} (C(R_t^n, x_t) + \bar{V}_t^{n-1}(R^{M,x}(R_t^n, x_t))),$$

and let x_t^n be the value of x_t that solves the maximization problem.

Step 2b. Compute the derivative

$$\hat{v}_t^n = \tilde{V}_t^n(R_t^n + 1) - \tilde{V}_t^n(R_t^n).$$

Step 2c. If $t > 0$, update the value function

$$\bar{V}_{t-1}^n(R_{t-1}^x) = \bar{V}_{t-1}^{n-1}(R_{t-1}^x) + \alpha_{n-1}(\hat{v}_t^n - \nabla_R \bar{V}_{t-1}^{n-1}(R_{t-1}^x)) R_{t-1}^x.$$

Step 2d. Update the states

$$R_t^{x,n} = R^{M,x}(R_t^n, x_t^n),$$

$$R_{t+1}^n = R^{M,W}(R_t^{x,n}, W_{t+1}(\omega^n)).$$

Step 3. Increment n . If $n \leq N$, go to step 1.

Step 4. Return the value functions $(\bar{V}_t^N)_{t=1}^T$.

Figure 13.6 SHAPE algorithm for approximate dynamic programming.

13.6 REGRESSION METHODS

As in Chapter 8 we can create regression models where the basis functions are manipulations of the number of resources of each type. For example, we might use

$$\bar{V}(R) = \theta_0 + \sum_{a \in \mathcal{A}} \theta_{1a} R_a + \sum_{a \in \mathcal{A}} \theta_{2a} R_a^2, \quad (13.10)$$

where $\theta = (\theta_0, (\theta_{1r})_{r \in \mathcal{R}}, (\theta_{2r})_{r \in \mathcal{R}})$ is a vector of parameters that are to be determined. The choice of explanatory terms in our approximation will generally reflect an understanding of the properties of our problem. For example, equation (13.10) assumes that we can use a mixture of linear and separable quadratic terms. A more general representation is to assume that we have developed a family \mathcal{F} of basis functions $(\phi_f(R))_{f \in \mathcal{F}}$. Examples of a basis function are

$$\phi_f(R) = R_{rf}^2,$$

$$\begin{aligned}\phi_f(R) &= \left(\sum_{r \in \mathcal{R}_f} R_r \right)^2 \quad \text{for some subset } \mathcal{R}_f, \\ \phi_f(R) &= (R_{r_1} - R_{r_2})^2, \\ \phi_f(R) &= |R_{r_1} - R_{r_2}|.\end{aligned}$$

A common strategy is to capture the number of resources at some level of aggregation. For example, if we are purchasing emergency equipment, we may care about how many pieces we have in each region of the country, and we may also care about how many pieces of a type of equipment we have (regardless of location). These issues can be captured using a family of aggregation functions G_f , $f \in \mathcal{F}$, where $G_f(r)$ aggregates an attribute vector r into a space $\mathcal{R}^{(f)}$ where for every basis function f there is an element $r_f \in \mathcal{R}^{(f)}$. Our basis function might then be expressed using

$$\phi_f(R) = \sum_{r \in \mathcal{R}} 1_{\{G_f(r)=r_f\}} R_r.$$

As we originally introduced in Section 8.2.2, the explanatory variables used in the examples above, which are generally referred to as independent variables in the regression literature, are typically referred to as basis functions by the approximate dynamic programming community. A basis function can be linear, nonlinear separable, nonlinear nonseparable, and even nondifferentiable, although the nondifferentiable case will introduce additional technical issues. The challenge, of course, is that it is the responsibility of the modeler to devise these functions for each application. We have written our basis functions purely in terms of the resource vector, but it is possible for them to be written in terms of other parameters in a more complex state vector, such as asset prices.

Given a set of basis functions, we can write our value function approximation as

$$\bar{V}(R|\theta) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(R). \quad (13.11)$$

It is important to keep in mind that $\bar{V}(R|\theta)$ (or more generally, $\bar{V}(S|\theta)$), is any functional form that approximates the value function as a function of the state vector parameterized by θ . Equation (13.11) is a classic linear-in-the-parameters function. We are not constrained to this form, but it is the simplest and offers some algorithmic shortcuts.

The issues that we encounter in formulating and estimating $\bar{V}(R|\theta)$ are the same as those that any student of statistical regression would face when modeling a complex problem. The major difference is that our data arrive over time (iterations), and we have to update our formulas recursively. Also it is typically the case that our observations are nonstationary. This is particularly true when an update of a value function depends on an approximation of the value function in the future

(as occurs with value iteration or any of the TD(λ) classes of algorithms). When we are estimating parameters from nonstationary data, we do not want to equally weight all observations.

The problem of finding θ can be posed in terms of solving the following stochastic optimization problem:

$$\min_{\theta} \mathbb{E}_{\omega} \frac{1}{2} (\bar{V}(R|\theta) - \hat{V})^2.$$

We can solve this using a stochastic gradient algorithm, which produces updates of the form

$$\begin{aligned} \bar{\theta}^n &= \bar{\theta}^{n-1} - \alpha_{n-1} (\bar{V}(R^n|\bar{\theta}^{n-1}) - \hat{V}(\omega^n)) \nabla_{\theta} \bar{V}(R^n|\theta^n) \\ &= \bar{\theta}^{n-1} - \alpha_{n-1} (\bar{V}(R^n|\bar{\theta}^{n-1}) - \hat{V}(\omega^n)) \begin{pmatrix} \phi_1(R^n) \\ \phi_2(R^n) \\ \vdots \\ \phi_F(R^n) \end{pmatrix}. \end{aligned}$$

If our value function is linear in R_t , we would write

$$\bar{V}(R|\theta) = \sum_{r \in \mathcal{R}} \theta_r R_r.$$

In this case our number of parameters has shrunk from the number of possible realizations of the entire vector R_t to the size of the attribute space (which, for some problems, can still be large, but nowhere near as large as the original state space). For this problem $\phi(R^n) = R^n$.

It is not necessarily the case that we will always want to use a linear-in-the-parameters model. We may consider a model where the value increases with the number of resources, but at a declining rate that we do not know. Such a model could be captured with the representation

$$\bar{V}(R|\theta) = \sum_{r \in \mathcal{R}} \theta_{1r} R_r^{\theta_{2r}},$$

where we expect $\theta_2 < 1$ to produce a concave function. Now our updating formula will look like

$$\begin{aligned} \theta_1^n &= \theta_1^{n-1} - \alpha_{n-1} (\bar{V}(R^n|\bar{\theta}^{n-1}) - \hat{V}(\omega^n))(R^n)^{\theta_2}, \\ \theta_2^n &= \theta_2^{n-1} - \alpha_{n-1} (\bar{V}(R^n|\bar{\theta}^{n-1}) - \hat{V}(\omega^n))(R^n)^{\theta_2} \ln R^n, \end{aligned}$$

where we assume the exponentiation operator in $(R^n)^{\theta_2}$ is performed component-wise.

We can put this updating strategy in terms of temporal differencing. As before, the temporal difference is given by

$$\delta_{\tau} = C_{\tau}(R_{\tau}, x_{\tau+1}) + \bar{V}_{\tau+1}^{n-1}(R_{\tau+1}) - \bar{V}_{\tau}^{n-1}(R_{\tau}).$$

The original parameter updating formula (equation (9.7)) when we had one parameter per state now becomes

$$\bar{\theta}^n = \bar{\theta}_t^{n-1} + \alpha_{n-1} \sum_{\tau=t}^T \lambda^{\tau-t} \delta_\tau \nabla_{\theta} \bar{V}(R^n | \bar{\theta}^n).$$

It is important to understand that in contrast with most of our other applications of stochastic gradients, updating the parameter vector using gradients of the objective function requires mixing the units of θ with the units of the value function. In these applications the stepsize α_{n-1} has to also perform a scaling role.

13.7 CUTTING PLANES*

Cutting planes represent a powerful strategy for representing concave (or convex if we are minimizing), piecewise-linear functions for multidimensional problems. This method evolved originally not as a method for approximating dynamic programs, but instead as a technique for solving linear programs in the presence of uncertainty. In the 1950s the research community recognized that many optimization problems involve different forms of uncertainty, with the most common being the challenge of allocating resources now to serve demands in the future that have not yet been realized. For this reason a subcommunity within math programming, known as the stochastic programming community, has developed a rich theory and some powerful algorithmic strategies for handling uncertainty within linear programs and, more recently, integer programs.

Historically, dynamic programming has been viewed as a technique for small, discrete optimization problems, whereas stochastic programming has been the field that handles uncertainty within math programs (which are typically characterized by high-dimensional decision vectors and large numbers of constraints). The connections between stochastic programming and dynamic programming, historically viewed as diametrically competing frameworks, have been largely overlooked. This section is designed to bridge the gap between stochastic programming and approximate dynamic programming. Our presentation is facilitated by notational decisions (in particular, the use of x as our decision vector) that we made at the beginning of the book.

The material in this section is somewhat more advanced and requires a fairly strong working knowledge of linear programming and duality theory. Our presentation is designed primarily to establish the relationship between the fields of approximate dynamic programming and stochastic programming.

We begin our presentation by introducing a classical model in stochastic programming known as the two-stage resource allocation problem.

13.7.1 A Two-Stage Resource Allocation Problem

We use the same notation we have been using for resource allocation problems, where

R_{tr} = number of resources with attribute $r \in \mathcal{R}$ in the system at time t .

$$R_t = (R_{tr})_{r \in \mathcal{R}}.$$

D_{tb} = number of demands of type $b \in \mathcal{B}$ in the system at time t .

$$D_t = (D_{tb})_{b \in \mathcal{B}}.$$

R_0 and D_0 capture the initial state of our system. Decisions are represented using

\mathcal{D}^D = decision to satisfy a demand with attribute b (each decision $d \in \mathcal{D}^D$ corresponds to a demand attribute $b_d \in \mathcal{B}$).

\mathcal{D}^M = decision to modify a resource (each decision $d \in \mathcal{D}^M$ has the effect of modifying the attributes of the resource where \mathcal{D}^M includes the decision to “do nothing”).

$$\mathcal{D} = \mathcal{D}^D \cup \mathcal{D}^M.$$

x_{trd} = number of resources that initially have attribute r that we act on with decision d .

$$x_t = (x_{trd})_{r \in \mathcal{R}, d \in \mathcal{D}}.$$

x_0 is the vector of decisions we have to make now given $S_0 = (R_0, D_0)$. The post-decision state is given by

$R_{0r'}^x$ = number of resources with attribute r' available in the second stage after we have made our original decision x_0

$$= \sum_{r \in \mathcal{R}} \sum_{d \in \mathcal{D}} \delta_{r'}(r, d) x_{0rd},$$

$$R_0^x = \Delta x_0,$$

where Δ is a matrix with $\delta_{r'}(r, d)$ in row r' , column (r, d) . Before we solve the second-stage problem (at time $t = 1$), we observe random changes to the resource vector as well as random new demands. For our illustration, we assume that unsatisfied demands from the first stage are lost. This means that

$$\begin{aligned} R_1 &= R_0^x + \hat{R}_1, \\ D_1 &= \hat{D}_1. \end{aligned}$$

The resource vector R_1 can be used to satisfy demands that first become known during time interval 1. In the second stage we need to choose x_1 to minimize the cost function $C_1(x_1)$. At this point we assume that our problem is finished. Thus the problem over both stages would be written

$$\max_{x_0, x_1} (C_0(x_0) + \mathbb{E} C_1(x_1)). \quad (13.12)$$

This problem has to be solved subject to the first-stage constraints:

$$\sum_{d \in \mathcal{D}} x_{0rd} = R_{0r}, \quad (13.13)$$

$$\sum_{r \in \mathcal{R}} x_{0rd} \leq \hat{D}_{0b_d}, \quad d \in \mathcal{D}^D, \quad (13.14)$$

$$x_{0rd} \geq 0, \quad (13.15)$$

$$\sum_{r \in \mathcal{R}} \sum_{d \in \mathcal{D}} \delta_{r'}(r, d) x_{0rd} - R_{0r'}^x = 0. \quad (13.16)$$

The second-stage decisions have to be made subject to constraints for each outcome $\omega \in \Omega$, given by

$$\sum_{d \in \mathcal{D}} x_{1rd}(\omega) = R_{1r}(\omega) = R_{0r}^x + \hat{R}_{1r}(\omega), \quad (13.17)$$

$$\sum_{r \in \mathcal{R}} x_{1rd}(\omega) \leq \hat{D}_{1b_d}(\omega), \quad d \in \mathcal{D}^D, \quad (13.18)$$

$$x_{1rd}(\omega) \geq 0. \quad (13.19)$$

Constraint (13.13) limits our decisions by the resources that are initially available. We may use these resources to satisfy initial demands, contained in \hat{D}_0 , where flows are limited by constraint (13.14). Constraint (13.15) enforces nonnegativity, while constraint (13.16) defines the post-decision resource vector R_0^x .

The constraints for the second stage are similar to those for the first stage, except that there is a set of constraints for every outcome ω . Note that equation (13.12) is written as if we are using x_1 as the state variable for the second stage. Again, this is the standard notational style of stochastic programming.

Equations (13.12) through (13.19) describe a fairly general model. It is useful to see the model formulated at a detailed level, but for what we are about to do, it is convenient to express it in matrix form.

First-stage constraints:

$$A_0 x_0 = R_0, \quad (13.20)$$

$$B_0 x_0 \leq \hat{D}_0, \quad (13.21)$$

$$x_0 \geq 0, \quad (13.22)$$

$$\Delta x_0 - R_0^x = 0. \quad (13.23)$$

Second-stage constraints:

$$A_1 x_1(\omega) = R_1(\omega) \quad \forall \omega \in \Omega, \quad (13.24)$$

$$B_1 x_1(\omega) \leq \hat{D}_1(\omega) \quad \forall \omega \in \Omega, \quad (13.25)$$

$$x_1(\omega) \geq 0 \quad \forall \omega \in \Omega. \quad (13.26)$$

We note that in the classical language of math programming, R_0^x is a decision variable defined by equation (13.23). In dynamic programming, we view $R_0^x = R^{M,x}(R_0, x_0)$ as a function that depends on R_0 and x_0 .

The second-stage contribution function depends on the first-stage allocation, $R_0^x(x_0)$, so we can write it as

$$V_0(R_0^x) = \mathbb{E}C_1(x_1),$$

which allows us to rewrite (13.12) as

$$\max_{x_0} (C_0(x_0) + V_0(R_0^x)). \quad (13.27)$$

This shows that our two-stage problem consists of a one-period contribution function (using the information we know now) and a value function that captures the expected contribution, which depends on the decisions that we made in the first stage.

We pause to note a major stylistic departure between stochastic and dynamic programming. In writing equation (13.27), it is clear that we have written our expected value function in terms of the resource state variable R_0^x . Of course, R_0^x is a function of x_0 , which means that we could write (13.27) as

$$\max_{x_0} (C_0(x_0) + V_0(x_0)). \quad (13.28)$$

This is the style favored by the stochastic programming community, where $V_0(x_0)$ is referred to as the *recourse function* which is typically denoted Q . Mathematically $V_0(x_0)$ and $V_0(R_0^x)$ are equivalent, but computationally they can be quite different. If $|\mathcal{R}|$ is the size of our resource attribute space and $|\mathcal{D}|$ is the number of types of decisions, the dimensionality of x_0 is typically on the order of $|\mathcal{R}| \times |\mathcal{D}|$, whereas the dimensionality of R_0^x will be on the order of $|\mathcal{R}|$. As a rule, it is computationally much easier to approximate $V_0(R_0^x)$ than $V_0(x_0)$ simply because functions of lower dimensional variables are easier to approximate than functions of higher dimensional ones. In some of our discussions in this chapter, we adopt the convention of writing V_0 as a function of x_0 in order to clarify the relationship with stochastic programming.

As we have seen in this volume, approximating the value function involves exploiting structure in the state variable. The stochastic programming community does not use the term “state variable,” but some authors will use the term *tenders*, which is to say that the first stage “tenders” R_0^x to the second stage. However, whereas the ADP community puts tremendous energy into exploiting the structure of the state variable when developing an approximation, state variables play either a minor or nonexistent role in stochastic programming.

Figure 13.4 illustrates solving the first-stage problem using a particular approximation for the second stage. The stochastic programming community has developed algorithmic strategies that solve these problems optimally (although the term “optimal” carries different meanings). The next two sections describe algorithmic strategies that have been developed within the stochastic programming community.

13.7.2 Benders' Decomposition

Benders' decomposition is most easily described in the context of our two-stage resource allocation problem. Initially we have to solve

$$\max_{x_0} (c_0 x_0 + V_0(R_0^x)) \quad (13.29)$$

subject to constraints (13.20) through (13.23). We note in passing that the stochastic programming community would normally write (13.29) in the form

$$\max_{x_0} (c_0 x_0 + V_0(x_0)),$$

which is mathematically correct (after all, R_0^x is a function of x_0), but we prefer the form in (13.29) to capture the second-stage dependence on R_0^x .

As before, the value function is given by

$$V_0(R_0^x) = \sum_{\omega \in \Omega} p(\omega) V_0(R_0^x, \omega),$$

where $V_0(R_0^x, \omega)$ is given by

$$V_0(R_0^x, \omega) = \max_{x_1(\omega)} c_1 x_1(\omega) \quad (13.30)$$

subject to constraints (13.24) through (13.26). For ease of reference we write the second-stage constraints as

$$\begin{array}{lll} A_1 x_1(\omega) & = & \Delta x_0 + \hat{R}_1(\omega), & \text{Dual} \\ B_1 x_1(\omega) & \leq & \hat{D}_1(\omega), & \hat{v}_1^R(\omega), \\ x_1(\omega) & \geq & 0, & \hat{v}_1^D(\omega), \end{array}$$

where $\hat{v}^R(\omega)$ and $\hat{v}^D(\omega)$ are the dual variables for the resource constraints and demand constraints under outcome (scenario) ω . For our discussion we assume that we have a discrete set of sample outcomes Ω , where the probability of outcome ω is given by $p(\omega)$. The linear programming dual of the second-stage problem takes the form

$$z^*(x_0, \omega) = \min_{\hat{v}_1^R(\omega), \hat{v}_1^D(\omega)} (\Delta x_0 + \hat{R}_1(\omega))^T \hat{v}_1^R(\omega) + (\hat{D}_1(\omega))^T \hat{v}_1^D(\omega) \quad (13.31)$$

subject to

$$A_1^T \hat{v}_1^R(\omega) + B_1^T \hat{v}_1^D(\omega) \geq c_1, \quad (13.32)$$

$$\hat{v}_1^D(\omega) \geq 0. \quad (13.33)$$

The dual is also a linear program, and the optimal $\hat{v}_1^*(\omega) = (\hat{v}_1^{R*}(\omega), \hat{v}_1^{D*}(\omega))$ must occur at one of a set of vertices, which we denote by \mathcal{V}_1 . Note that the set

of feasible dual solutions \mathcal{V}_1 is independent of ω , a property that arises because we have assumed that our only source of randomness is in the right-hand side constraints (we would lose this if our costs were random).

Since $z^*(x_0, \omega)$ is the optimal solution of the dual, we have

$$z^*(x_0, \omega) = (\Delta x_0 + \hat{R}_1(\omega))^T \hat{v}_1^{R*}(\omega) + (\hat{D}_1(\omega))^T \hat{v}_1^{D*}(\omega).$$

Next define

$$z(x_0, \hat{v}_1, \omega) = (\Delta x_0 + \hat{R}_1(\omega))^T \hat{v}_1^R + (\hat{D}_1(\omega))^T \hat{v}_1^D.$$

Note that while $z(x_0, \hat{v}_1, \omega)$ depends on ω , it is written for any set of duals $\hat{v}_1 \in \mathcal{V}_1$ (we do not need to index them by ω). We observe that

$$\begin{aligned} z^*(x_0, \omega) &= z(x_0, \hat{v}_1^*(\omega), \omega) \\ &\leq z(x_0, \hat{v}_1, \omega) \quad \forall \hat{v}_1 \in \mathcal{V}_1, \end{aligned} \quad (13.34)$$

since $z^*(x_0, \omega)$ is the best we can do. Furthermore equation (13.34) is true for all $\hat{v}_1 \in \mathcal{V}_1$, and all outcomes ω . We know from the theory of linear programming that our primal must always be less than or equal to our dual, which means that

$$\begin{aligned} V_0(x_0, \omega) &\leq z(x_0, \hat{v}_1, \omega) \quad \forall \hat{v}_1 \in \mathcal{V}_1, \omega \in \Omega, \\ &= z^*(x_0, \omega), \quad \forall \omega \in \Omega. \end{aligned}$$

This means that for all $\hat{v}_1 \in \mathcal{V}_1$,

$$V_0(x_0) \leq \sum_{\omega \in \Omega} p(\omega) \left((\Delta x_0 + \hat{R}_1(\omega))^T \hat{v}_1^R + (\hat{D}_1(\omega))^T \hat{v}_1^D \right), \quad (13.35)$$

where the inequality (13.35) is tight for $\hat{v}_1 = \hat{v}_1^*(\omega)$. Now let

$$\alpha_1(\hat{v}_1) = \sum_{\omega \in \Omega} p(\omega) (\hat{R}_1(\omega))^T \hat{v}_1^R + (\hat{D}_1(\omega))^T \hat{v}_1^D, \quad (13.36)$$

$$\beta_1(\hat{v}_1) = \sum_{\omega \in \Omega} p(\omega) \Delta^T \hat{v}_1^R. \quad (13.37)$$

For historical reasons we use α as a coefficient rather than a stepsize in our discussion of Benders' decomposition. This allows us to write (13.35) in the more compact form

$$V_0(x_0) \leq \alpha_1(\hat{v}_1) + (\beta_1(\hat{v}_1))^T x_0. \quad (13.38)$$

The right-hand side of (13.38) is called a *cut*, since it is a plane (actually, an n -dimensional hyperplane) that represents an upper bound on the value function. Using these cuts, we can replace (13.29) with

$$\max_{x_0} (c_0 x_0 + z) \quad (13.39)$$

subject to (13.20) through (13.23) plus

$$z \leq \alpha_1(\hat{v}_1) + (\beta_1(\hat{v}_1))^T x_0 \quad \forall \hat{v}_1 \in \mathcal{V}_1. \quad (13.40)$$

Unfortunately, equation (13.40) can be computationally problematic. The set \mathcal{V}_1 may be extremely large, so enforcing this constraint for each vertex $\hat{v}_1 \in \mathcal{V}_1$ is prohibitive. Furthermore, even if Ω is finite, it may be quite large, making summations over the elements in Ω expensive for some applications.

A simple strategy overcomes the problem of enumerating the set of dual vertices. Assume that after iteration $n - 1$ we have a set of cuts $(\alpha_1^m, \beta_1^m)_{m=1}^n$ that we have generated from previous iterations. Using these cuts, we solve equation (13.39):

$$z \leq \alpha_1^m + (\beta_1^m)^T x_0, \quad m = 1, 2, \dots, n - 1, \quad (13.41)$$

to obtain a first-stage solution x_0^n . Given x_0^n , we then solve the dual (13.31) to obtain $\hat{v}_1^n(\omega)$ (the optimal duals) for each $\omega \in \Omega$. Using this information, we obtain α_1^n and β_1^n . Thus, instead of having to solve (13.39) subject to the entire set of constraints (13.40), we use only the cuts in equation (13.41).

This algorithm is known as the “L-shaped” algorithm (more precisely, “L-shaped decomposition”). For a finite Ω it has been proved to converge to the optimal solution. The problem is the requirement that we calculate $\hat{v}_1^*(\omega)$ for all $\omega \in \Omega$, which means that we must solve a linear program for each outcome at each iteration. For most problems this is computationally pretty demanding.

13.7.3 Variations

Two variations of this algorithm that avoid the computational burden of computing $\hat{v}_1^*(\omega)$ for each ω have been proposed. These algorithms vary only in how the cuts are computed and updated. The first is known as *stochastic decomposition*. At iteration n , after solving the first-stage problem, we would solve (for sample realization ω^n)

$$\left(\hat{v}_1^{R,n}, \hat{v}_1^{D,n} \right) = \arg \min_{\hat{v}_1^R(\omega^n), \hat{v}_1^D(\omega^n)} (\Delta x_0^n + \hat{R}_1(\omega^n))^T \hat{v}_1^R(\omega^n) + \hat{D}_1(\omega^n)^T \hat{v}_1^D(\omega^n) \quad (13.42)$$

for a single outcome ω^n . We then update all previous cuts by first computing for $m = 1, 2, \dots, n$

$$(v_m^{R,n}, v_m^{D,n}) = \arg \min_{\hat{v}_1^R, \hat{v}_1^D} \left((\Delta x_0^n + \hat{R}_1(\omega^m))^T \hat{v}_1^R + \hat{D}_1(\omega^m)^T \hat{v}_1^D \mid (\hat{v}_1^R, \hat{v}_1^D) \in \mathcal{V}^n \right).$$

We then compute the next cut using

$$\alpha_n^n = \frac{1}{n} \sum_{m=1}^n (\hat{R}_1(\omega^m))^T v_m^{R,n} + \hat{D}_1(\omega^m)^T v_m^{D,n}, \quad (13.43)$$

$$\beta_n^n = \frac{1}{n} \sum_{m=1}^n \Delta^T v_m^{R,n}. \quad (13.44)$$

Then all the previous cuts are updated using

$$\alpha_m^n = \frac{n-1}{n} \alpha_m^{n-1}, \quad \beta_m^n = \frac{n-1}{n} \beta_m^{n-1}, \quad m = 1, \dots, n-1. \quad (13.45)$$

The complete algorithm is given in Figure 13.7. The beauty of stochastic decomposition is that we never have to loop over all outcomes, and we solve a single linear program for the second stage at each iteration. There is a requirement that we loop over all the cuts that we have previously generated (in equation (13.45)), but the calculation here is fairly trivial (contrast the need to solve a complete linear program for every outcome in the L-shaped algorithm).

A second variation, called the CUPPS algorithm (for “cutting plane and partial sampling” algorithm), finds new dual extreme points using

$$\alpha_n^n = \frac{1}{|\Omega|} \sum_{\omega \in \Omega} (\hat{R}_1(\omega)^T v^{R,n}(\omega) + \hat{D}_1(\omega^m)^T v^{D,n}(\omega)), \quad (13.46)$$

Step 0. Set $\mathcal{V}^0 = \phi, n = 1$

Step 1. Solve the following master problem:

$$x_0^n = \arg \max_{x_0 \in \mathcal{X}_0} (c_0 x_0 + z),$$

where $\mathcal{X}_0 = \{A_0 x_0 = R_0, B_0 x_0 \leq \hat{D}_0, z \leq \alpha_m^n + \beta_m^n x_0, m = 1, \dots, n-1, x_0 \geq 0\}$.

Step 2. Sample $\omega^n \in \Omega$ and solve the second-stage problem,

$$\max\{c_1 x_1 : A_1 x_1 = \Delta x_0^n + \hat{R}_1(\omega^n), B_1 x_1 \leq \hat{D}_1(\omega^n), x_1 \geq 0\},$$

to obtain the optimal dual solution from equation (13.42) and store the dual vertex

$$\mathcal{V}^n \leftarrow \mathcal{V}^{n-1} \cup (\hat{v}_1^{R,n}, \hat{v}_1^{D,n}).$$

Step 3. Update the cuts:

Step 3a. Compute for $m = 1, \dots, n$:

$$(v_m^{R,n}, v_m^{D,n}) = \arg \min_{\hat{v}_1^R, \hat{v}_1^D} \left((\Delta x_0^n + \hat{R}_1(\omega^m))^T \hat{v}_1^R + \hat{D}_1^T(\omega^m) \hat{v}_1^D \mid (\hat{v}_1^R, \hat{v}_1^D) \in \mathcal{V}^n \right).$$

Step 3b. Construct the coefficients α_n^n and β_n^n of the n th cut to be added to the master problem using (13.43) and (13.44).

Step 3c. Update the previously generated cuts by

$$\alpha_m^n = \frac{n-1}{n} \alpha_m^{n-1}, \quad \beta_m^n = \frac{n-1}{n} \beta_m^{n-1}, \quad m = 1, \dots, n-1.$$

Step 4. Check for convergence; if not, set $n = n + 1$ and return to step 1.

Figure 13.7 Sketch of the stochastic decomposition algorithm.

$$\beta_n^n = \frac{1}{|\Omega|} \sum_{\omega \in \Omega} \Delta^T v^{R,n}(\omega), \quad (13.47)$$

where for each $\omega \in \Omega$, we compute the duals

$$(v^{R,n}, v^{D,n})(\omega) = \arg \min_{\hat{v}_1^R, \hat{v}_1^D} \left((\Delta x_0^n + \hat{R}_1(\omega^m))^T \hat{v}_1^R + \hat{D}_1^T(\omega^m) \hat{v}_1^D \mid (\hat{v}_1^R, \hat{v}_1^D) \in \mathcal{V}^n \right) \quad (13.48)$$

Equations (13.46) through (13.48) represent the primary computational burden of CUPPS (outside of solving the linear program for each time period in each iteration). These have to be computed for each sample realization in Ω . Stochastic decomposition, by contrast, requires that we update all previously generated cuts, a step that CUPPS does not share. The steps of this algorithm are given in Figure 13.8.

L-shaped decomposition, stochastic decomposition, and the CUPPS algorithm offer three contrasting strategies for generating cuts, which are illustrated in Figure 13.9. L-shaped decomposition is computationally the most demanding, but it produces tight cuts and will, in general, produce the fastest convergence (measured in terms of the number of iterations). The cuts produced by stochastic decomposition are neither tight nor valid, but steadily converge to the correct function. The cuts generated by CUPPS are valid but not tight.

Stochastic decomposition and CUPPS require roughly the same amount of work as any of the other approximation strategies we have presented for approximate dynamic programming, but these strategies offer a proof of convergence. Notice

Step 0. Set $\mathcal{V}^0 = \phi$, $n = 1$.

Step 1. Solve the following master problem:

$$x_0^n = \arg \max_{x_0 \in \mathcal{X}_0} (c_0 x_0 + z),$$

where $\mathcal{X}_0 = \{A_0 x_0 = R_0, B_0 x_0 \leq \hat{D}_0, z \leq \alpha^m + \beta^m x_0, m = 1, \dots, n-1, x_0 \geq 0\}$.

Step 2. Sample $\omega^n \in \Omega$ and solve the second-stage problem,

$$\max\{c_1 x_1 : A_1 x_1 = \Delta x_0^n + \hat{R}_1(\omega^n), B_1 x_1 \leq \hat{D}_1(\omega^n), x_1 \geq 0\},$$

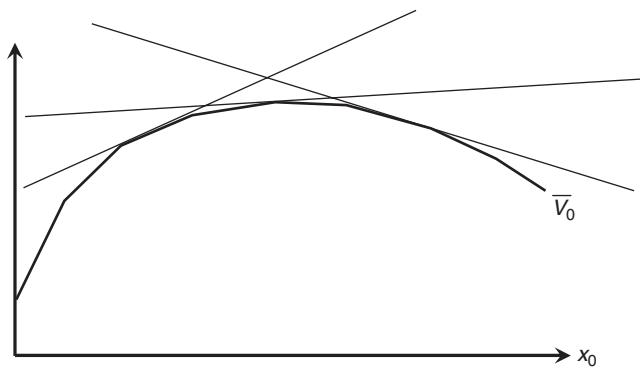
to obtain the optimal dual solution from equation (13.42) and store the dual vertex

$$\mathcal{V}^n \leftarrow \mathcal{V}^{n-1} \cup (\hat{v}_1^{R,n}, \hat{v}_1^{D,n}).$$

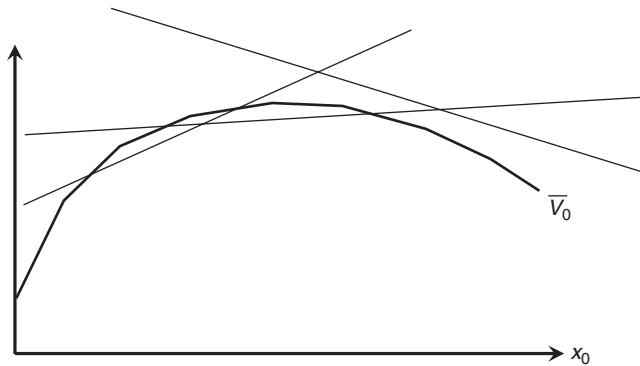
Step 3. Construct the coefficients of the n th cut to be added to the master problem using (13.46) and (13.47).

Step 4. Check for convergence; if not, set $n = n+1$ and return to step 1.

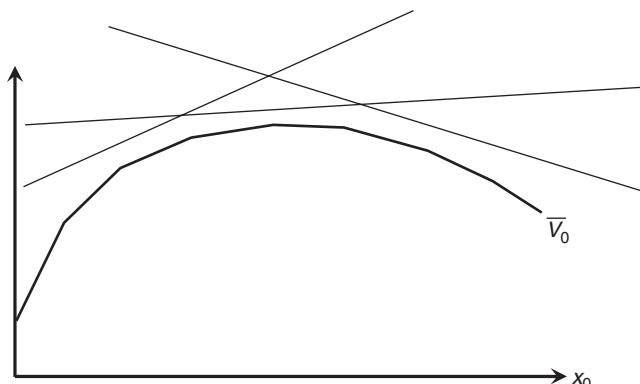
Figure 13.8 Sketch of the CUPPS algorithm.



(a) Cuts generated by L-shaped decomposition.



(b) Cuts generated by the stochastic decomposition algorithm.



(c) Cuts generated by the CUPPS algorithm.

Figure 13.9 Cuts generated by (a) L-shaped decomposition, (b) stochastic decomposition, and (c) CUPPS.

also that it does not require exploration, but it does require that we generate a dual variable for every second-stage constraint (which requires that our attribute space \mathcal{A} be small enough to enumerate). The real question is rate of convergence. The next section provides some experimental work that hints at answers to this question, although the real answer depends on the characteristics of individual problems.

13.7.4 Experimental Comparisons

For resource allocation problems it is important to evaluate approximations two ways. The first is to consider “two-stage” problems (a terminology from stochastic programming) where we make a decision, see a random outcome, make one more decision, and then stop. For resource allocation problems the first decision consists of meeting known demands and then acting on resources so that they are in position to satisfy demands that will become known in the second stage (at time $t = 1$). Two-stage problems are important because we solve multiperiod (multistage) problems as sequences of two-stage problems.

Powell et al. (2004) reports on an extensive set of experimental comparisons of separable piecewise-linear value function approximations (denoted the “SPAR” algorithm since it uses the projection operator for maintaining concavity described in Section 13.3) and variations of Benders’ decomposition (which can be viewed as nonseparable value function approximations). These variations include L-shaped (which requires solving a linear program for every sample outcome), CUPPS (which requires looping over all sample outcomes and performing a trivial calculation), and stochastic decomposition (which only requires looping over the sample outcomes that we have actually sampled, performing modest calculations for each outcome). Stochastic decomposition can be viewed as a form of approximate dynamic programming (without a state variable). All of our experiments were done (by necessity) with a finite set of outcomes. L-shaped is computationally expensive, but for smaller problems it could be generally counted on to produce the optimal solution, which was then used to evaluate the quality of the solutions produced by the other algorithms.

The experiments were conducted for a distribution problem where resources were distributed to a set of locations, after which they were used to satisfy a demand. The number of locations determines the dimensionality of R_0^x and R_1 . The problems tested used 10, 25, 50, and 100 locations. L-shaped decomposition found the optimal solution for all the problems except the one with 100 locations.

The results are shown in Table 13.1, which shows the percentage distance from the optimal solution for all three algorithms as a function of the number of iterations. The results show that L-shaped decomposition generally has the fastest rate of convergence (in terms of iterations), but the computational demand of solving linear programs for every outcome usually makes this computationally intractable for most applications. All the methods based on Benders’ decomposition show progressively slower convergence as the number of locations increase, suggesting that this method will struggle with higher dimensional state variables. As the problems became larger, the separable approximation showed much faster convergence with near optimal performance in the limit.

Table 13.1 Numerical results for two-stage distribution problems while varying the number of locations

Locations	Method	Number of Independent Demand Observations						
		25	50	100	250	500	1000	2500
10	SPAR	18.65	12.21	7.07	2.25	0.48	0.28	0.04
	L-Shaped	3.30	0.19	0.00				
	CUPPS	5.84	4.19	0.50	0.29	0.00		
	SD	45.45	9.18	11.35	2.35	2.30	1.12	0.30
25	SPAR	11.73	4.04	2.92	0.89	0.34	0.13	0.19
	L-Shaped	19.88	15.98	2.14	0.11	0.00		
	CUPPS	8.27	13.40	4.33	4.02	1.47	0.16	0.00
	SD	40.55	29.79	22.22	12.80	4.24	4.80	1.06
50	SPAR	9.99	2.60	1.18	0.48	0.26	0.30	0.12
	L-Shaped	42.56	20.30	6.07	1.49	0.52	0.04	0.00
	CUPPS	34.93	9.91	19.30	11.71	5.09	1.38	0.32
	SD	43.18	29.81	17.94	8.09	5.91	6.25	2.73
100 ^a	SPAR	8.74	4.61	1.20	0.45	0.16	0.05	0.01
	L-Shaped	74.52	29.79	26.21	7.30	2.32	0.85	0.11
	CUPPS	54.59	35.54	23.99	17.58	14.68	14.13	5.36
	SD	62.63	34.82	40.73	12.14	15.22	17.43	17.49

^aOptimal solution not found, figures represent the deviation from the best objective value known.

Note: The table gives the percent over optimal (where available) for SPAR (separable, piecewise linear value functions), along with three flavors of Benders' decomposition: L-shaped, CUPPS, and stochastic decomposition (SD).

Source: From Powell et al. (2004).

It is impossible to make general conclusions about the performance of one method over another, since this is an experimental question that depends on the characteristics of the datasets used for testing. However, it appears from these experiments that all flavors of Benders' decomposition become much slower as the dimensionality of the state variable grows. Stochastic decomposition and approximate dynamic programming (SPAR) require approximately the same computational effort per iteration. Stochastic decomposition (SD) is dramatically faster than the L-shaped decomposition in terms of the work required per iteration, but the price is that it will have a slower rate of convergence (in terms of iterations). However, even L-shaped decomposition exhibits a very slow rate of convergence, especially as the problem size grows (where problem size is measured in terms of the dimensionality of the state variable).

Benders' decomposition (in one of its various forms) offers significant value since it provides a provably convergent algorithm. SPAR (which uses piecewise-linear separable value function approximations) has been found to work on large-scale resource allocation problems, but the error introduced by the use of a separable approximation will be problem-dependent.

13.8 WHY DOES IT WORK?**

13.8.1 Proof of the SHAPE Algorithm

The convergence proof of the SHAPE algorithm is a nice illustration of a martingale proof. We start with the martingale convergence theorem (Doob, 1953; Neveu, 1975; Taylor, 1990), which has been the basis of convergence proofs for stochastic subgradient methods (as illustrated in Gladyshev, 1965). We then list some properties of the value function approximation $\hat{V}(\cdot)$ and produce a bound on the difference between x^n and x^{n+1} . Finally, we prove the theorem. This section is based on Powell and Cheung (2000).

Let ω^n be the information that we sample in iteration n , and let $\omega = (\omega^1, \omega^2, \dots)$ be an infinite sequence of observations of sample information where $\omega \in \Omega$. Let $h^n = \omega^1, \omega^2, \dots, \omega^n$ be the history up to (and including) iteration n . Let \mathfrak{F} be the σ -algebra on Ω , and let $\mathfrak{F}^n \subseteq \mathfrak{F}^{n+1}$ be the sequence of increasing subsigma-algebras on Ω representing the information we know up through iteration n .

We assume the following:

- (A.1) \mathcal{X} is convex and compact.
- (A.2) $\mathbb{E}V(x, W)$ is convex, finite and continuous on \mathcal{X} .
- (A.3) g^n is bounded such that $\|g^n\| \leq c_1$.
- (A.4) $\hat{V}^n(x)$ is strongly convex, meaning that

$$\hat{V}^n(y) - \hat{V}^n(x) \geq \hat{v}^n(x)(y - x) + b\|x - y\|^2, \quad (13.49)$$

where b is a positive number that is a constant throughout the optimization process. The term $b\|x - y\|^2$ is used to ensure that the slope $\hat{v}^n(x)$ is a monotone function of x . When we interchange y and x in (13.49) and add the resulting inequality to (13.49), we obtain

$$(\hat{v}^n(y) - \hat{v}^n(x))(y - x) \geq 2b\|x - y\|^2. \quad (13.50)$$

- (A.5) The stepsizes α_n are \mathfrak{F}^n measurable and satisfy

$$0 < \alpha_n < 1, \quad \sum_{n=0}^{\infty} \mathbb{E}\{(\alpha_n)^2\} < \infty$$

and

$$\sum_{n=0}^{\infty} \alpha_n = \infty \quad \text{a.s.}$$

- (A.6) $\hat{V}^0(x)$ is bounded and continuous, and $\hat{v}^0(x)$ (the derivative of $\hat{V}^0(x)$) is bounded for $x \in \mathcal{X}$.

Note that we require that the expected sum of squares be bounded, whereas we must impose the almost sure condition that the sum of stepsizes is infinite. We now state our primary theorem.

Theorem 13.8.1 *If (A.1) through (A.6) are satisfied, then the sequence x^n , produced by algorithm SHAPE, converges almost surely to an optimal solution $x^* \in \mathcal{X}^*$ of problem (13.7).*

The proof illustrates several proof techniques that are commonly used for these problems. Students interested in doing more fundamental research in this area may be able to use some of the devices in their own work.

To prove the theorem, we need to use the Martingale convergence theorem and two lemmas.

Martingale Convergence Theorem

A sequence of random variables $\{W^n\}$ that are \mathfrak{F}^n -measurable is said to be a *supermartingale* if the sequence of conditional expectations $\mathbb{E}\{W^{n+1}|\mathfrak{F}^n\}$ exists and satisfies

$$\mathbb{E}\{W^{n+1}|\mathfrak{F}^n\} \leq W^n.$$

Theorem 13.8.2 (Neveu (1975), p. 26) *Let W^n be a positive supermartingale. Then W^n converges to a finite random variable a.s.*

From the definition, W^n is essentially the stochastic analog of a decreasing sequence.

Property of Approximations

In addition to equations (13.49) and (13.50) of assumption (A.4), the optimal solution for problem (13.7) at iteration n can be characterized by the variational inequality

$$\hat{v}^n(x^n)(x - x^n) \geq 0 \quad \forall x \in \mathcal{X}. \quad (13.51)$$

Furthermore, at iteration $k + 1$,

$$(\hat{v}^n(x^{n+1}) + \alpha_n(g^n - \hat{v}^n(x^n))) (x - x^{n+1}) \geq 0 \quad \forall x \in \mathcal{X}. \quad (13.52)$$

The first lemma below provides a bound on the difference between two consecutive solutions. The second lemma establishes that $V^n(x)$ is bounded.

Lemma 13.8.1 *The solutions x^n produced by algorithm SHAPE satisfy*

$$\|x^n - x^{n+1}\| \leq \frac{\alpha_n}{2b} \|g^n\|,$$

where b satisfies equation (13.49).

Proof Substituting x by x^n in (13.52), we have

$$\alpha_n (g^n - \hat{v}^n(x^n)) (x^n - x^{n+1}) \geq \hat{v}^n(x^{n+1})(x^{n+1} - x^n).$$

Rearranging the terms, we obtain

$$\begin{aligned} \alpha_n g^n(x^n - x^{n+1}) &\geq \hat{v}^n(x^{n+1})(x^{n+1} - x^n) - \alpha_n \hat{v}^n(x^{n+1} - x^n) \\ &= (\hat{v}^n(x^{n+1}) - \hat{v}^n(x^n))(x^{n+1} - x^n) \\ &\quad + (1 - \alpha_n) \hat{v}^n(x^n)(x^{n+1} - x^n). \end{aligned}$$

Combining (13.50), (13.51), and $0 < \alpha_n < 1$ gives us

$$\begin{aligned} \alpha_n g^n(x^n - x^{n+1}) &\geq 2b \|x^n - x^{n+1}\|^2 + (1 - \alpha_n) \hat{v}^n(x^n)(x^{n+1} - x^n) \\ &\geq 2b \|x^n - x^{n+1}\|^2. \end{aligned}$$

Applying Schwarz's inequality, we have that

$$\alpha_n \|g^n\| \cdot \|x^n - x^{n+1}\| \geq \alpha_n g^n(x^n - x^{n+1}) \geq 2b \|x^n - x^{n+1}\|^2.$$

Dividing both sides by $\|x^n - x^{n+1}\|$, it follows that $\|x^n - x^{n+1}\| \leq \alpha_n / 2b \|g^n\|$. \square

Lemma 13.8.2 *The approximation function $\hat{V}^n(x)$ in iteration n can be written as*

$$\hat{V}^n(x) = \hat{V}^0(x) + r^n x,$$

where r^n is a finite vector.

Proof The algorithm proceeds by adding linear terms to the original approximation. Thus, at iteration n , the approximation is the original approximations plus the linear term

$$\hat{V}^n(x) = \hat{V}^0(x) + r^n x, \tag{13.53}$$

where r^n is the sum of the linear correction terms. We just have to show that r^n is finite.

When taking the first derivative of $\hat{V}^n(x)$ in equation (13.53), we have

$$\hat{v}^n(x) = \hat{v}^0(x) + r^n.$$

With that, we can write $\hat{V}^{n+1}(x)$ in terms of $\hat{V}^0(x)$

$$\begin{aligned} \hat{V}^{n+1}(x) &= \hat{V}^n(x) + \alpha_n (g^n - \hat{v}^n(x)) x \\ &= \hat{V}^0(x) + r^n x + \alpha_n (g^n - \hat{v}^n(x)) x \\ &= \hat{V}^0(x) + r^n x + \alpha_n (g^n - \hat{v}^0(x_k) - r^n) x. \end{aligned}$$

Therefore r^{n+1} and r^n are related as follows:

$$r^{n+1} = \alpha_n(g^n - \hat{v}^0(x_k)) + (1 - \alpha_n)r^n. \quad (13.54)$$

So, the total change in our initial approximation is a weighted sum of $g^n - \hat{v}^0(x^n)$ and the current cumulative change. Since both g^n and $\hat{v}^0(x^n)$ are finite, there exists a finite, positive vector such that

$$\hat{d} \geq \max_k |g^n - \hat{v}^0(x^n)|. \quad (13.55)$$

We can now use induction to show that $r^n \leq \hat{d}$ for all n . For $n = 1$, we have $r^1 = a_0(g^0 - \hat{v}_0^0) \leq a_0\hat{d}$. Since $a_0 < 1$ and is positive, we have $r^1 \leq \hat{d}$. Assuming that $r^n \leq \hat{d}$, we want to show $r^{n+1} \leq \hat{d}$. By using this assumption and the definition of \hat{d} , equation (13.54) implies that

$$r^{n+1} \leq \alpha_n|g^n - \hat{v}^0(x_k)| + (1 - \alpha_n)r^n \leq \alpha_n\hat{d} + (1 - \alpha_n)\hat{d} = \hat{d}.$$

We now return to our main result. \square

Proof of Theorem 13.8.1

Our algorithm proceeds in three steps. First, we establish a supermartingale that provides a basic convergence result. Then, we show that there is a convergent subsequence. Finally, we show that the entire sequence is convergent. For simplicity, we write $\hat{v}^n = \hat{v}^n(x^n)$.

Step 1. Establish a supermartingale for theorem 13.8.2.

Let $T^n = \hat{V}^n(x^*) - \hat{V}^n(x^n)$, and consider the difference of T^{n+1} and T^n :

$$\begin{aligned} T^{n+1} - T^n &= \hat{V}^{n+1}(x^*) - \hat{V}^{n+1}(x^{n+1}) - \hat{V}^n(x^*) + \hat{V}^n(x^n) \\ &= \hat{V}^n(x^*) + \alpha_n(g^n - \hat{v}^n)x^* - \hat{V}^n(x^{n+1}) - \alpha_n(g^n - \hat{v}^n)x^{n+1} \\ &\quad - \hat{V}^n(x^*) + \hat{V}^n(x^n). \end{aligned}$$

If we write $x^* - x^{n+1}$ as $x^* - x^n + x^n - x^{n+1}$, we get

$$\begin{aligned} T^{n+1} - T^n &= \underbrace{\hat{V}^n(x^n) - \hat{V}^n(x^{n+1})}_{(I)} - \underbrace{\alpha_n\hat{v}^n(x^n - x^{n+1})}_{(II)} - \underbrace{\alpha_n\hat{v}^n(x^* - x^n)}_{(III)} \\ &\quad + \underbrace{\alpha_n g^n(x^* - x^n)}_{(IV)} + \underbrace{\alpha_n g^n(x^n - x^{n+1})}. \end{aligned}$$

Consider each part individually. First, by the convexity of $\hat{V}^n(x)$, it follows that

$$\begin{aligned}\hat{V}^n(x^n) - \hat{V}^n(x^{n+1}) &\leq \hat{v}^{n\sim}(x^n - x^{n+1}) \\ &= (1 - \alpha_n)\hat{v}^{n\sim}(x^n - x^{n+1}) + \alpha_n\hat{v}^{n\sim}(x^n - x^{n+1}).\end{aligned}$$

From equation (13.51) and $0 < \alpha_n < 1$, we know that (I) ≤ 0 . Again, from equation (13.51) and $0 < \alpha_n < 1$, we show that (II) ≥ 0 .

For (III), by the definition that $g^n \in \partial V(x^n, \omega^{n+1})$,

$$g^n(x^* - x^n) \leq V(x^*, \omega^{n+1}) - V(x^n, \omega^{n+1}),$$

where $V(x, \omega^{n+1})$ is the recourse function given outcome ω^{n+1} .

For (IV), lemma 13.8.1 implies that

$$\alpha_n g^n(x^n - x^{n+1}) \leq \alpha_n \|g^n\| \cdot \|x^n - x^{n+1}\| \leq \frac{(\alpha_n)^2 \|g^n\|^2}{2b} \leq \frac{(\alpha_n)^2 c_1^2}{2b}.$$

Therefore, the difference $T^{n+1} - T^n$ becomes

$$T^{n+1} - T^n \leq -\alpha_n (V(x^n, \omega^{n+1}) - V(x^*, \omega^{n+1})) + \frac{(\alpha_n)^2 c_1^2}{2b}. \quad (13.56)$$

Taking the conditional expectation with respect to \mathfrak{F}^n on both sides, it follows that

$$\mathbb{E}\{T^{n+1}|\mathfrak{F}^n\} \leq T^n - \alpha_n (\bar{V}(x^n) - \bar{V}(x^*)) + \frac{(\alpha_n)^2 c_1^2}{2b}, \quad (13.57)$$

where T^n , α_n and x_k on the right-hand side are deterministic given the conditioning on \mathfrak{F}^n . We replace $V(x, \omega^{n+1})$ (for $x = x^n$ and $x = x^*$) with its expectation $\bar{V}(x)$ since conditioning on \mathfrak{F}^n tells us nothing about ω^{n+1} . Since $\bar{V}(x^n) - \bar{V}(x^*) \geq 0$, the sequence

$$W^n = T^n + \frac{c_1^2}{2b} \sum_{i=k}^{\infty} a_i^2$$

is a positive supermartingale. Theorem 13.8.2 implies the almost sure convergence of W^n . Thus

$$T^n \rightarrow T^* \quad \text{a.s.}$$

Step 2. Show that there exists a subsequence n_j of n such that $x^{n_j} \rightarrow x^ \in \mathcal{X}^*$ a.s.*

Summing equation (13.56) over n up to N and canceling the alternating terms of T^n gives

$$T^{N+1} - T^0 = - \sum_{n=0}^N \alpha_n (V(x^n, \omega^{n+1}) - V(x^*, \omega^{n+1})) + \sum_{n=0}^N \frac{(\alpha_n)^2 c_1^2}{2b}.$$

Take expectations of both sides. For the first term on the right-hand side, we take the conditional expectation first conditioned on \mathfrak{F}^n and then over all \mathfrak{F}^n , giving us

$$\begin{aligned}\mathbb{E}\{T^{N+1} - T^0\} &= - \sum_{n=0}^N \mathbb{E}\{\mathbb{E}\{\alpha_n(V(x^n, \omega^{n+1}) - V(x^*, \omega^{n+1})) | \mathfrak{F}^n\}\} \\ &\quad + \mathbb{E}\left\{\sum_{n=0}^N \frac{(\alpha_n)^2 c_1^2}{2b}\right\} \\ &= - \sum_{n=0}^N \mathbb{E}\{\alpha_n(\bar{V}(x^n) - \bar{V}(x^*))\} + \frac{c_1^2}{2b} \sum_{n=0}^N \mathbb{E}\{(\alpha_n)^2\}.\end{aligned}$$

Taking the limit as $N \rightarrow \infty$ and using the finiteness of T^n and $\sum_{n=0}^{\infty} \mathbb{E}\{\alpha_n^2\}$, we have

$$\sum_{n=0}^{\infty} \mathbb{E}\{\alpha_n(\bar{V}(x^n) - \bar{V}(x^*))\} < \infty.$$

Since $\bar{V}(x^n) - \bar{V}(x^*) \geq 0$ and $\sum_{n=0}^{\infty} \alpha_n = \infty$ (a.s.), there exists a subsequence n_j of n such that

$$\bar{V}(x^{n_j}) \rightarrow \bar{V}(x^*) \quad \text{a.s.}$$

By continuity of \bar{V} , this sequence converges. That is,

$$x^{n_j} \rightarrow x^* \in \mathcal{X}^* \quad \text{a.s.}$$

Step 3. Show that $x^n \rightarrow x^ \in \mathcal{X}^*$ a.s.*

Consider the convergent subsequence x^{n_j} in step 2. By using the expression of \hat{V}^n in lemma 13.8.2, we can write T^{n_j} as

$$\begin{aligned}T^{n_j} &= \hat{V}^{n_j}(x^{n_j}) - \hat{V}^{n_j}(x^*) \\ &= \hat{V}^0(x^{n_j}) - \hat{V}^0(x^*) + r^{n_j}(x^{n_j} - x^*) \\ &\leq \hat{V}^0(x^{n_j}) - \hat{V}^0(x^*) + r^{n_j}|x^{n_j} - x^*| \\ &\leq \hat{V}^0(x^{n_j}) - \hat{V}^0(x^*) + \hat{d}|x^{n_j} - x^*|,\end{aligned}$$

where \hat{d} is the positive, finite vector defined in (13.55). When $x^{n_j} \rightarrow x^*$, both the terms $\hat{d}|x^{n_j} - x^*|$ and $\hat{V}^0(x^{n_j}) - \hat{V}^0(x^*)$ (by continuity of \hat{V}^0) go to 0. Since T^{n_j} is positive, we obtain that $T^{n_j} \rightarrow 0$ a.s. Combining this result and the result in step 1 (that T^n converges to a unique nonnegative T^* a.s.), we have $T^n \rightarrow T^* = 0$ a.s. Finally, we know from the strong convexity of $\hat{V}^n(\cdot)$ that

$$T^n = \hat{V}^n(x^n) - \hat{V}^n(x^*) \geq b\|x^n - x^*\|^2 \geq 0.$$

Therefore $x^n \rightarrow x^*$ a.s.

13.8.2 The Projection Operation

Let v_s^{n-1} be the value (or the marginal value) of being in state s at iteration $n-1$ and assume that we have a function where we know that we should have $v_{s+1}^{n-1} \geq v_s^{n-1}$ (we refer to this function as *monotone*). For example, if this monotone is the marginal value, we would expect this if we are describing a concave function. Now assume that we have a sample realization \hat{v}_s^n that is the value (or marginal value) of being in state s . We would then smooth this new observation with the previous estimates using

$$z_s^n = \begin{cases} (1 - \alpha_{n-1})v_s^{n-1} + \alpha_{n-1}\hat{v}_s^n & \text{if } s = s^n, \\ v_s^n & \text{otherwise.} \end{cases} \quad (13.58)$$

Since \hat{v}_s^n is random, we cannot expect z_s^n to also be monotone. In this section we want to restore monotonicity by defining an operator $v^n = \Pi_V(z)$ where $v_{s+1}^n \geq v_s^n$. There are several ways to do this. In this section we define the operator $v = \Pi_V(z)$, which takes a vector z (not necessarily monotone) and produces a monotone vector v . If we wish to find v that is as close as possible to z , we would solve

$$\begin{aligned} \min \frac{1}{2} \|v - z\|^2 \\ \text{subject to } v_{s+1} - v_s \leq 0, \quad s = 0, \dots, M. \end{aligned} \quad (13.59)$$

Assume that v_0 is bounded above by B , and v_{M+1} , for $s < M$, is bounded from below by $-B$. Let $\lambda_s \geq 0$, $s = 0, 1, \dots, M$ be the Lagrange multipliers associated with equation (13.59). It is easy to see that the optimality equations are

$$v_s = z_s + \lambda_s - \lambda_{s-1}, \quad s = 1, 2, \dots, M, \quad (13.60)$$

$$\lambda_s(v_{s+1} - v_s) = 0, \quad s = 0, 1, \dots, M. \quad (13.61)$$

Let i_1, \dots, i_2 be a sequence of states where

$$v_{i_1-1} > v_{i_1} = v_{i_1+1} = \dots = c = \dots = v_{i_2-1} = v_{i_2} > v_{i_2+1}.$$

We can then add equations (13.60) from i_1 to i_2 to yield

$$c = \frac{1}{i_2 - i_1 + 1} \sum_{s=i_1}^{i_2} z_s.$$

If $i_1 = 1$, then c is the smaller of the above and B . Similarly, if $i_2 = M$, then c is the larger of the above and $-B$.

We also note that $v^{n-1} \in \mathcal{V}$ and z^n computed by differs from v^{n-1} in just one coordinate. If $z^n \notin \mathcal{V}$ then either $z_{s^n-1}^n < z_{s^n}^n$, or $z_{s^n+1}^n > z_{s^n}^n$.

If $z_{s^n-1}^n < z_{s^n}^n$, then we need to find the largest $1 < i \leq s^n$, where

$$z_{i-1}^n \geq \frac{1}{s^n - i + 1} \sum_{s=i}^{s^n} z_s^n.$$

If i cannot be found, then we use $i = 1$. We proceed to compute

$$c = \frac{1}{s^n - i + 1} \sum_{s=i}^{s^n} z_s^n$$

and let

$$v_j^{n+1} = \min(B, c), \quad j = i, \dots, s^n.$$

We have $\lambda_0 = \max(0, c - B)$, and

$$\lambda_s = \begin{cases} 0, & s = 1, \dots, i - 1, \\ \lambda_{s-1} + z_s - v_s, & s = i, \dots, s^n - 1, \\ 0, & s = s^n, \dots, M. \end{cases}$$

It is easy to show that the solution found and the Lagrange multipliers satisfy equations (13.60) and (13.61).

If $z_{s^n}^n < z_{s^n+1}^n$, then the entire procedure is basically the same with appropriate inequalities reversed.

13.9 BIBLIOGRAPHIC NOTES

Section 13.1 The decision of whether to estimate the value function or its derivative is often overlooked in the dynamic programming literature, especially within the operations research community. In the controls community, use of gradients is sometimes referred to as *dual heuristic dynamic programming* (see Werbos, 1992a; Venayagamoorthy et al. (2002)).

Section 13.3 The theory behind the projective SPAR algorithm is given in Powell et al. (2004). A proof of convergence of the leveling algorithm is given in Topaloglu and Powell (2003).

Section 13.5 The SHAPE algorithm was introduced by Powell and Cheung (2000).

Section 13.7 The first paper to formulate a math program with uncertainty appears to be Dantzig and Ferguson (1956). For a broad introduction to the field of stochastic optimization, see Ermoliev (1988) and Pflug (1996). For complete treatments of the field of stochastic programming, see Infanger (1994), Kall and Wallace (1994), Birge and Louveaux (1997), and Kall and Mayer (2005). For an easy tutorial on the subject, see Sen and Higle (1999). A very thorough introduction to stochastic programming is given in Ruszcynski and Shapiro (2003). Mayer (1998) provides a detailed presentation of computational work for stochastic programming. There has been special interest in the types of network problems we have considered (see Wallace, 1986a, b, 1987, Rockafellar and Wets (1991) presents specialized algorithms for stochastic programs formulated using scenarios. This modeling framework

has been of particular interest in the area of financial portfolios (Mulvey and Ruszcynski, 1995). Benders' decomposition for two-stage stochastic programs was first proposed by Van Slyke and Wets (1969) as the “L-shaped” method. Higle and Sen (1991) introduce stochastic decomposition, which is a Monte Carlo based algorithm that is most similar in spirit to approximate dynamic programming. Powell and Chen (1999) present a variation of Benders that falls between stochastic decomposition and the L-shaped method. The relationship between Benders' decomposition and dynamic programming is often overlooked. A notable exception is Pereira and Pinto (1991), who use Benders to solve a resource allocation problem arising in the management of reservoirs. This paper presents Benders as a method for avoiding the curse of dimensionality of dynamic programming. For an excellent review of Benders' decomposition for multistage problems, see Ruszcynski (2003). Benders has been extended to multistage problems in Birge (1985), Ruszcynski (1993), and Powell and Chen (1999), which can be viewed as a form of approximate dynamic programming using cuts for value function approximations.

Section 13.8.1 The proof of convergence of the SHAPE algorithm is based on Powell and Cheung (2000).

Section 13.8.2 The proof of the projection operation is based on Powell et al. (2004).

PROBLEMS

13.1 Consider a news vendor problem where we solve

$$\max_x \mathbb{E}F(x, \hat{D}),$$

where

$$F(x, \hat{D}) = p \min(x, \hat{D}) - cx.$$

We have to choose a quantity x before observing a random demand \hat{D} . For our problem, assume that $c = 1$, $p = 2$, and that \hat{D} follows a discrete uniform distribution between 1 and 10 ($\hat{D} = d$, $d = 1, 2, \dots, 10$ with probability 0.10). Approximate $\mathbb{E}F(x, \hat{D})$ as a piecewise linear function using the methods described in Section 13.3, using a stepsize $\alpha_{n-1} = 1/n$. Note that you are using derivatives of $F(x, \hat{D})$ to estimate the slopes of the function. At each iteration, randomly choose x between 1 and 10. Use sample realizations of the gradient to estimate your function. Compute the exact function and compare your approximation to the exact function.

- 13.2** Repeat exercise 13.6, but this time approximate $\mathbb{E}F(x, \hat{D})$ using a linear approximation:

$$\bar{F}(x) = \theta x.$$

Compare the solution you obtain with a linear approximation to what you obtained using a piecewise-linear approximation. Now repeat the exercise using demands that are uniformly distributed between 500 and 1000. Compare the behavior of a linear approximation for the two different problems.

- 13.3** Repeat exercise 13.3, but this time approximate $\mathbb{E}F(x, \hat{D})$ using the SHAPE algorithm. Start with an initial approximation given by

$$\bar{F}^0(x) = \theta_0(x - \theta_1)^2.$$

Use the recursive regression methods of Sections 13.6 and 9.3 to fit the parameters. Justify your choice of stepsize rule. Compute the exact function and compare your approximation to the exact function.

- 13.4** Repeat exercise 13.1, but this time approximate $\mathbb{E}F(x, \hat{D})$ using the regression function given by

$$\bar{F}(x) = \theta_0 + \theta_1 x + \theta_2 x^2.$$

Use the recursive regression methods of Sections 13.6 and 9.3 to fit the parameters. Justify your choice of stepsize rule. Compute the exact function and compare your approximation to the exact function. Estimate your value function approximation using two methods:

- (a) Use observations of $F(x, \hat{D})$ to update your regression function.
- (b) Use observations of the derivative of $F(x, \hat{D})$, so that $\bar{F}(x)$ becomes an approximation of the derivative of $\mathbb{E}F(x, \hat{D})$.

- 13.5** Approximate the function $\mathbb{E}F(x, \hat{D})$ in exercise 13.1, but now assume that the random variable $\hat{D} = 1$ (that is, it is deterministic). Using the following approximation strategies:

- (a) Use a piecewise-linear value function approximation. Try using both left and right derivatives to update your function.
- (b) Use the regression $\bar{F}(x) = \theta_0 + \theta_1 x + \theta_2 x^2$.

- 13.6** We are going to solve the basic asset acquisition problem (Section 2.2.5) where we purchase assets (at a price p^P) at time t to be used in time interval $t+1$. We sell assets at a price p^S to satisfy the demand \hat{D}_t that arises during time interval t . The problem is to be solved over a finite time horizon T .

Assume that the initial inventory is 0 and that demands follow a discrete uniform distribution over the range $[0, D^{\max}]$. The problem parameters are given by

$$\begin{aligned}\gamma &= 0.8, \\ D^{\max} &= 10, \\ T &= 20, \\ p^p &= 5, \\ p^s &= 8.\end{aligned}$$

Solve this problem by estimating a piecewise linear value function approximation (Section 13.3). Choose $\alpha_{n+1} = a/(a+n)$ as your stepsize rule, and experiment with different values of a (e.g., 1, 5, 10, and 20). Use a single-pass algorithm, and report your profits (summed over all time periods) after each iteration. Compare your performance for different stepsize rules. Run 1000 iterations and try to determine how many iterations are needed to produce a good solution (the answer may be substantially less than 1000).

- 13.7** Repeat exercise 13.6, but this time use the SHAPE algorithm to approximate the value function. Use as your initial value function approximation the function

$$\overline{V}_t^0(R_t) = \theta_0(R_t - \theta_2)^2.$$

For each of the exercises below, you may have to tweak your stepsize rule. Try to find a rule that works well for you (we suggest stick with a basic $a/(a+n)$ strategy). Determine an appropriate number of training iterations, and then evaluate your performance by averaging results over 100 iterations (testing iterations) where the value function is not changed.

- (a) Solve the problem using $\theta_0 = 1, \theta_1 = 5$.
- (b) Solve the problem using $\theta_0 = 1, \theta_1 = 50$.
- (c) Solve the problem using $\theta_0 = 0.1, \theta_1 = 5$.
- (d) Solve the problem using $\theta_0 = 10, \theta_1 = 5$.
- (e) Summarize the behavior of the algorithm with these different parameters.

- 13.8** Repeat exercise 13.6, but this time assume that your value function approximation is given by

$$\overline{V}_t^0(R_t) = \theta_0 + \theta_1 R_t + \theta_2 R_t^2.$$

Use the recursive regression techniques of sections 13.6 and 9.3 to determine the values for the parameter vector θ .

- 13.9** Repeat exercise 13.6, but this time assume you are solving an infinite horizon problem (which means you only have one value function approximation).
- 13.10** Repeat exercise 13.8, but this time assume an infinite horizon.
- 13.11** Repeat exercise 13.6, but now assume the following problem parameters:

$$\gamma = 0.99,$$

$$T = 200,$$

$$p^p = 5,$$

$$p^s = 20.$$

For the demand distribution, assume that $\hat{D}_t = 0$ with probability 0.95, and that $\hat{D}_t = 1$ with probability 0.05. This is an example of a problem with low demands, where we have to hold inventory for a fairly long time.

Dynamic Resource Allocation Problems

There is a vast array of problems that fall under the umbrella of “resource allocation.” We might be managing a team of medical specialists who have to respond to emergencies, or technicians who have to provide local support for the installation of sophisticated medical equipment. Alternatively, a transportation company might be managing a fleet of vehicles, or an investment manager might be trying to determine how to allocate funds among different asset classes. We can even think of playing a game of backgammon as a problem of managing a single resource (the board), although in this chapter we are only interested in problems that involve multiple resources where quantity is an important element of the decision variable.

Aside from the practical importance of these problems, this problem class provides a special challenge. In addition to the usual challenge of making decisions over time under uncertainty (the theme of this entire book) we now have to deal with the fact that our decision x_t is suddenly of very high dimensionality, requiring that we use the tools of math programming (linear, nonlinear, and integer programming). For practical applications it is not hard to find problems in this class where x_t has thousands, or even tens of thousands, of dimensions. This problem class clearly suffers from all three “curses of dimensionality.”

We illustrate the ideas by starting with a scalar problem, after which we move to a sequence of multidimensional (and in several cases, very high-dimensional) resource allocation problems. Each problem offers different features, but they can all be solved using the ideas we have been developing throughout this volume.

14.1 AN ASSET ACQUISITION PROBLEM

Perhaps one of the simplest resource allocation problems is the basic asset acquisition problem where we acquire assets (money, oil, water, aircraft) to satisfy a

Approximate Dynamic Programming: Solving the Curses of Dimensionality, Second Edition.

Warren B. Powell.

© 2011 John Wiley & Sons, Inc. Published 2011 by John Wiley & Sons, Inc.

future, random demand. In our simple illustration we purchase assets at time t at a price that can be used to satisfy demands in the next time period (which means we do not know what they are). If we buy too much, the leftover inventory is held to future time periods. If we cannot satisfy all the demand, we lose revenue, but we do not allow ourselves to hold these demands to be satisfied at a later time period.

We use the simple model presented in Section 2.2.5 so that we can focus our attention on using derivatives to estimate a piecewise linear function in the context of a dynamic program.

14.1.1 The Model

Our model is formulated as follows:

R_t = Assets on hand at time t before we make a new ordering decision, and before we have satisfied any demands arising in time interval t .

x_t = Amount of product purchased at time t to be used during time interval $t + 1$.

\hat{D}_t = Random demands that arise between $t - 1$ and t .

\hat{R}_t = Random exogenous changes to our asset levels (donations of blood, theft of product, leakage).

New assets are purchased at a fixed price p^p and are sold at a fixed price $p^s > p^p$. The contribution earned during time interval t is given by

$$C_t(R_t, x_t) = p^s \min\{R_t, \hat{D}_t\} - p^p x_t.$$

The transition function for R_t is given by

$$\begin{aligned} R_t^x &= R_t - \min\{R_t, \hat{D}_t\} + x_t, \\ R_{t+1} &= R_t^x + \hat{R}_{t+1}, \end{aligned}$$

where we assume that any unsatisfied demands are lost to the system.

This problem can be solved using Bellman's equation. For this problem, R_t is our state variable. Let $V_t(R_t^x)$ be the value of being in post-decision state R_t^x . Following our usual approximation strategy, the decision problem at time t , given that we are in state R_t^n , is given by

$$x_t^n = \arg \max_{x_t \in \mathcal{X}_t^n} (C_t(R_t^n, x_t) + \gamma \bar{V}_t^{n-1}(R_t^x)). \quad (14.1)$$

Recall that solving (14.1) is a form of decision function that we represent by $X_t^\pi(R_t)$, where the policy is to solve (14.1) using the value function approximation $\bar{V}_t^{n-1}(R_t^x)$. Assume, for example, that our approximation is of the form

$$\bar{V}_t^{n-1}(R_t^x) = \theta_0 + \theta_1 R_t^x + \theta_2 (R_t^x)^2,$$

where $R_t^x = R_t - \min\{R_t, \hat{D}_t\} + x_t$. In this case, (14.1) looks like

$$x_t^n = \arg \max_{x_t \in \mathcal{X}_t^n} (p^s \min\{R_t, \hat{D}_t\} - p^p x_t + \gamma(\theta_0 + \theta_1 R_t^x + \theta_2 (R_t^x)^2)). \quad (14.2)$$

We find the optimal value of x_t^n by taking the derivative of the objective function and setting it equal to zero, giving us

$$\begin{aligned} 0 &= \frac{dC_t(R_t^n, x_t)}{dx_t} + \gamma \frac{d\bar{V}_t^{n-1}(R_t^x)}{dx_t} \\ &= -p^p + \gamma \frac{d\bar{V}_t^{n-1}(R_t^x)}{dR_t^x} \frac{dR_t^x}{dx_t} \\ &= -p^p + \gamma(\theta_1 + 2\theta_2 R_t^x) \\ &= -p^p + \gamma(\theta_1 + 2\theta_2(R_t - \min\{R_t, \hat{D}_t\} + x_t)), \end{aligned}$$

where we used $dR_t^x/dx_t = 1$. Solving for x_t gives

$$x_t^n = \frac{1}{2\theta_2} \left(\frac{p^p}{\gamma} - \theta_1 \right) - (R_t^n - \min\{R_t^n, \hat{D}_t^n\}).$$

Of course, we assume that $x_t^n \geq 0$.

14.1.2 An ADP Algorithm

The simplest way to compute a gradient is using a pure forward-pass implementation. At time t , we have to solve

$$\tilde{V}_t^n(R_t^n) = \max_{x_t} \left(p^s \min\{R_t, \hat{D}_t\} - p^p x_t + \gamma \bar{V}_t^{n-1}(R_t^x) \right).$$

Here $\tilde{V}_t^n(R_t^n)$ is just a placeholder. We are going to compute a derivative using a finite difference. We begin by recalling that the pre-decision resource state is given by

$$\begin{aligned} R_t^n &= R^{M,W}(R_{t-1}^{x,n}, W_t(\omega^n)) \\ &= R_{t-1}^{x,n} + \hat{R}_t^n. \end{aligned}$$

Now let

$$R_t^{n+} = R^{M,W}(R_{t-1}^{x,n} + 1, W_t(\omega^n))$$

be the pre-decision resource vector when $R_{t-1}^{x,n}$ is incremented by one. We next compute a derivative using the finite difference

$$\hat{v}_t^n = \tilde{V}_t^n(R_t^{n+}) - \tilde{V}_t^n(R_t^n).$$

Note that when we solve the perturbed value $\tilde{V}_t^n(R_t^{n+})$ we have to re-optimize x_t . For example, it is entirely possible that if we increase $R_{t-1}^{x,n}$ by one, then x_t^n will

decrease by one. As is always the case with a single-pass algorithm, \hat{v}_t^n depends on $\bar{V}_{t-1}^n(R_t^x)$, and as a result will typically be biased.

Once we have a sample estimate of the gradient \hat{v}_t^n , we next have to update the value function. We can represent this updating process generically using

$$\bar{V}_{t-1}^n = U^V(\bar{V}_{t-1}^{n-1}, R_{t-1}^{x,n}, \hat{v}_t^n).$$

The actual updating process depends on whether we are using a piecewise-linear approximation, the SHAPE algorithm, or a general regression equation. Remember that we are using \hat{v}_t^n to update the post-decision value function at time $t-1$.

The complete algorithm is outlined in Figure 14.1, which is an adaptation of our original single-pass algorithm. A simple but critical conceptual difference is that we are now explicitly assuming that we are using a continuous functional approximation.

14.1.3 How Well Does It Work?

This approximation strategy is provably convergent. It also works in practice (!!). Unfortunately, while a proof of convergence can be reassuring, it is not hard to find provably convergent algorithms for problems that would never actually work (typically because the rate of convergence is far too slow).

The asset acquisition problem is simple enough that we can find the optimal solution using the exact methods provided in Chapter 3. For our approximation we used a piecewise-linear value function approximation. We used a pure exploitation strategy since the concave structure of the value function will eventually push us toward the correct solution.

Figure 14.2 shows the solution quality for a finite horizon problem (undiscounted). We are within 5 percent of optimal after about 50 iterations, and within 1 percent after 100 iterations. For many problems in practice (where we do not even know the optimal solution) this is extremely good.

14.1.4 Variations

This problem class is special because we are controlling the quantity of a single type of asset to serve random demands. Using approximate dynamic programming to fit piecewise-linear value function approximations produces an algorithm that is provably convergent (at least for the finite horizon case) without the need for exploration. We can use our current value function approximation to determine our decision at time t . Even if the value function is off, it will eventually converge to the correct value function (for the states that an optimal policy visits infinitely often).

This problem class seems quite simple, but it actually includes a number of variations that arise in industry.

- *Cash balance for mutual funds.* Imagine that we want to determine how much cash to have on hand in a mutual fund in order to handle requests for redemptions. If we have too much cash, we can invest it in equities do obtain a

Step 0. Initialize:

Step 0a. Initialize $\bar{V}_t^0(S_t^x)$ for all time periods t .

Step 0b. Initialize S_0^1 .

Step 0c. Let $n = 1$.

Step 1. Choose ω^n .

Step 2. Do for $t = 0, 1, 2, \dots, T$:

Step 2a. Let the state variable be

$$S_t^n = (R_t^n, \hat{D}_t(\omega^n)),$$

and let

$$S_t^{n+} = (R_t^n + 1, \hat{D}_t(\omega^n))$$

Step 2b. Solve

$$\tilde{V}_t^n(S_t^n) = \max_{x_t} (p^s \min\{R_t, \hat{D}_t(\omega^n)\} - p^p x_t + \gamma \bar{V}_t^{n-1}(R_t^x)),$$

where $R_t^x = R_t - \min\{R_t, \hat{D}_t\} + x_t$. Let x_t^n be the value of x_t that solves the maximization problem. Also find $\tilde{V}_t^n(S_t^{n+})$.

Step 2c. Compute

$$\hat{v}_t^n = \tilde{V}_t^n(S_t^{n+}) - \tilde{V}_t^n(S_t^n).$$

Step 2d. If $t > 0$ update the value function:

$$\bar{V}_{t-1}^t = U^V(\bar{V}_{t-1}^{t-1}, S_{t-1}^{x,n}, \hat{v}_t^n).$$

Step 2e. Update the state variable

$$S_{t+1}^n = S^M(S_t^n, x_t^n, W_{t+1}(\omega^n)).$$

Step 3. Let $n = n+1$. If $n < N$, go to step 1.

Figure 14.1 Approximate dynamic programming algorithm for the asset acquisition problem.

higher rate of return, but this return fluctuates and there are transaction costs for moving money between cash and equities. The state variable has to include not just how much cash we have on hand but also information on the state of the stock market and interest rates. This information is part of the state variable, but the information evolves exogenously.

- *Gas storage.* Companies will store natural gas in coal mines, buying gas when the price is low and selling it when it is high. Gas can be purchased from multiple suppliers at prices that fluctuate over time. Gas may be sold at time t to be delivered at a time $t' > t$ (when demand is higher).
- *Managing vaccine inventories.* Imagine that we have a limited supply of vaccines that have to be made available for the highest risk segments of the

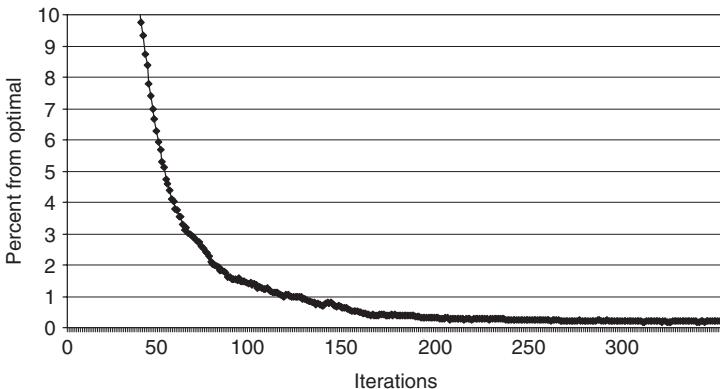


Figure 14.2 Percent from optimal for the simple asset acquisition problem using a piecewise-linear value function approximation.

population. The strategy has to balance the degree of risk now against potential future needs for the vaccine. Instead of satisfying a single demand, we face different types of demands with different levels of importance.

- *Maintaining spares.* There are many problems where we have to maintain spares (spare engines for business aircraft, spare transformers for electric power grids). We have to decide how many new spares to purchase, and whether to satisfy a demand against holding inventory for future (and potentially more important) demands.

Some of these problems can be solved using a minor variation of the algorithmic strategy that we described in this section. But this is not always the case. Consider a problem where we are managing the inventory of a single product (water, oil, money, spares) given by a scalar R_t , but where our decisions have to consider other variables (stock market, weather, technology parameters) that evolve exogenously. Let \tilde{S}_t be the vector of variables that influence the behavior of our system (which means they belong to the state variable) but that evolve exogenously. Our state variable is given by

$$S_t = (R_t, \tilde{S}_t).$$

For example, \tilde{S}_t might be temperature (if we have an energy problem), rainfall (if we are planning the storage of water in reservoirs), or even the S & P 500 stock index and interest rates. \tilde{S}_t might be a scalar, but it might consist of several variables.

We can still approximate the problem using a piecewise linear value function, but instead of one function, we have to estimate a family of functions that we would represent using $\bar{V}_t(R_t | \tilde{S}_t)$. It is just that instead of using $\bar{V}_t(R_t^x)$, we have to use $\bar{V}_t(R_t^x | \tilde{S}_t)$. The algorithm is still provably convergent, but computationally it can be much more difficult. When \tilde{S}_t is a vector, we encounter a curse-of-dimensionality problem if we estimate one function for each possible value of

\tilde{S}_t (a classic lookup table representation). Instead of estimating one piecewise-linear value function $\bar{V}_t(R_t^x)$, we might have to estimate hundreds or thousands (or more). If \tilde{S}_t is a vector, we might need to look into other types of functional representations.

14.2 THE BLOOD MANAGEMENT PROBLEM

The problem of managing blood inventories serves as a particularly elegant illustration of a resource allocation problem. We are going to start by assuming that we are managing inventories at a single hospital, where each week we have to decide which of our blood inventories should be used for the demands that need to be served in the upcoming week.

We have to start with a bit of background about blood. For the purposes of managing blood inventories, we care primarily about blood type and age. Although there is a vast range of differences in the blood of two individuals, for most purposes doctors focus on the eight major blood types: $A+$ (“A positive”), $A-$ (“A negative”), $B+$, $B-$, $AB+$, $AB-$, $O+$, and $O-$. While the ability to substitute different blood types can depend on the nature of the operation, for most purposes blood can be substituted according to Table 14.1.

A second important characteristic of blood is its age. The storage of blood is limited to six weeks, after which it has to be discarded. Hospitals need to anticipate if they think they can use blood before it hits this limit, as it can be transferred to blood centers that monitor inventories at different hospitals within a region. It helps if a hospital can identify blood it will not need as soon as possible so that the blood can be transferred to locations that are running short.

One mechanism for extending the shelf life of blood is to freeze it. Frozen blood can be stored up to 10 years, but it takes at least an hour to thaw, limiting its use in emergency situations or operations where the amount of blood needed is highly uncertain. In addition once frozen blood is thawed it must be used within 24 hours.

Table 14.1 Allowable blood substitutions for most operations

Donor	Recipient							
	$AB+$	$AB-$	$A+$	$A-$	$B+$	$B-$	$O+$	$O-$
$AB+$	X							
$AB-$	X	X						
$A+$	X		X					
$A-$	X	X	X	X				
$B+$	X				X			
$B-$	X	X			X	X		
$O+$	X		X		X		X	
$O-$	X	X	X	X	X	X	X	X

Note: “X” means a substitution is allowed

Source: From Cant (2006).

14.2.1 A Basic Model

We can model the blood problem as a heterogeneous resource allocation problem. We are going to start with a fairly basic model which can be easily extended with almost no notational changes. We begin by describing the attributes of a unit of stored blood using

$$b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} \text{Blood type}(A+, A-, \dots) \\ \text{Age(in weeks)} \end{pmatrix},$$

\mathcal{B} = set of all attribute types.

We will limit the age to the range $0 \leq a_2 \leq 6$. Blood with $a_2 = 6$ (which means blood that is already six weeks old) is no longer usable. We assume that decision epochs are made in one-week increments.

Blood inventories, and blood donations, are represented using

R_{tb} = Units of blood of type b available to be assigned or held at time t .

$R_t = (R_{tb})_{b \in \mathcal{B}}$.

\hat{R}_{tb} = Number of new units of blood of type b donated between $t - 1$ and t .

$\hat{R}_t = (\hat{R}_{tb})_{b \in \mathcal{B}}$.

The attributes of demand for blood are given by

$$d = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} \text{Blood type of patient} \\ \text{Surgery type: urgent or elective} \\ \text{Is substitution allowed?} \end{pmatrix},$$

d^ϕ = decision to hold blood in inventory (“do nothing”),

\mathcal{D} = set of all demand types d plus d^ϕ .

The attribute d_3 captures the fact that there are some operations where a doctor will not allow any substitution. One example is childbirth, since infants may not be able to handle a different blood type, even if it is an allowable substitute. For our basic model, we do not allow unserved demand in one week to be held to a later week. As a result we need only to model new demands, which we accomplish with

\hat{D}_{td} = Units of demand with attribute d that arose between $t - 1$ and t ,

$\hat{D}_t = (\hat{D}_{td})_{d \in \mathcal{D}}$.

We act on blood resources with decisions given by

x_{tbd} = Number of units of blood with attribute b that we assign to a demand of type d .

$x_t = (x_{tbd})_{b \in \mathcal{B}, d \in \mathcal{D}}$.

The feasible region \mathcal{X}_t is defined by the following constraints:

$$\sum_{d \in \mathcal{D}} x_{tbd} = R_{tb}, \quad (14.3)$$

$$\sum_{b \in \mathcal{B}} x_{tbd} \leq \hat{D}_{td}, \quad d \in \mathcal{D}, \quad (14.4)$$

$$x_{tbd} \geq 0. \quad (14.5)$$

Blood that is held simply ages one week, but we limit the age to six weeks. Blood that is assigned to satisfy a demand can be modeled as being moved to a blood-type sink, denoted, perhaps, using $b_{t,1} = \phi$ (the null blood type). The blood attribute transition function $r^M(b_t, d_t)$ is given by

$$b_{t+1} = \begin{pmatrix} b_{t+1,1} \\ b_{t+1,2} \end{pmatrix} = \begin{cases} \begin{pmatrix} b_{t,1} \\ \min\{6, b_{t,2} + 1\} \end{pmatrix}, & d_t = d^\phi, \\ \begin{pmatrix} \phi \\ - \end{pmatrix}, & d_t \in \mathcal{D}. \end{cases}$$

To represent the transition function, it is useful to define

$$\delta_{b'}(b, d) = \begin{cases} 1, & b_t^x = b' = r^M(b_t, d_t), \\ 0, & \text{otherwise,} \end{cases}$$

Δ = matrix with $\delta_{b'}(b, d)$ in row b' and column (b, d) .

We note that the attribute transition function is deterministic. A random element would arise, for example, if inspections of the blood resulted in blood that was less than six weeks old being judged to have expired. The resource transition function can now be written

$$R_{tb'}^x = \sum_{b \in \mathcal{B}} \sum_{d \in \mathcal{D}} \delta_{b'}(b, d) x_{tbd},$$

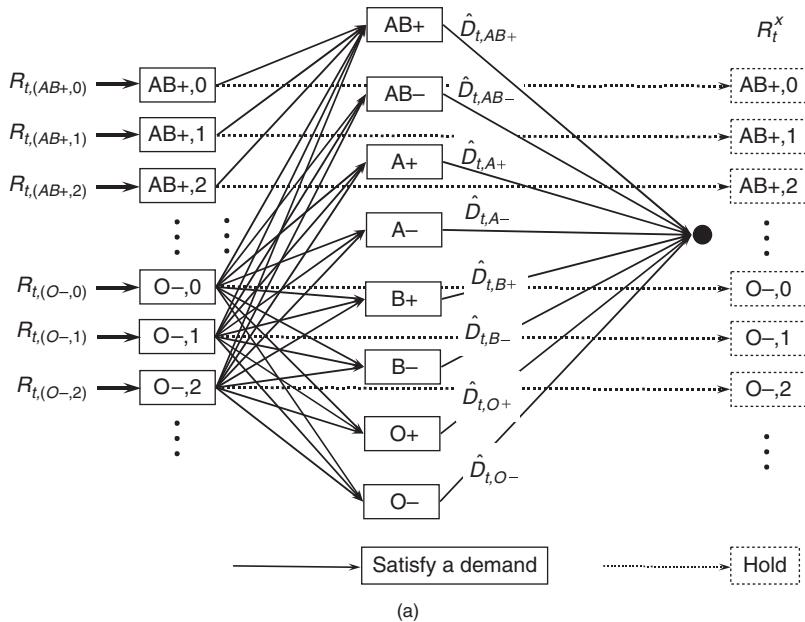
$$R_{t+1,b'} = R_t^x + \hat{R}_{t+1,b'}.$$

In matrix form, these would be written

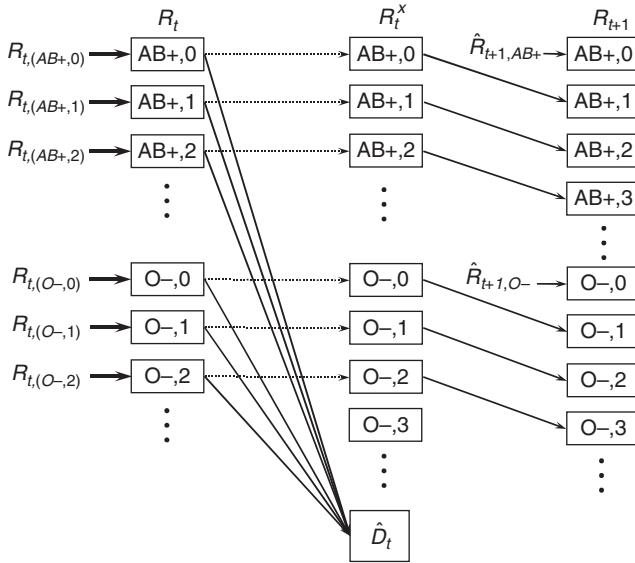
$$R_t^x = \Delta x_t, \quad (14.6)$$

$$R_{t+1} = R_t^x + \hat{R}_{t+1}. \quad (14.7)$$

Figure 14.3 illustrates the transitions that are occurring in week t . We have to decide either which type of blood to use to satisfy a demand (Figure 14.3a) or to hold the blood until the following week. If we use blood to satisfy a demand, it is assumed lost from the system. If we hold the blood until the following week, it is transformed into blood that is one week older. Blood that is six weeks old may



(a)



(b)

Figure 14.3 Assignment of different blood types (and ages) to known demands in week t (a), and holding blood until the following week (b). (Solid lines represent assigning blood to a demand, dotted lines represent holding blood.)

not be used to satisfy any demands, so we can view the bucket of blood that is six weeks old as a sink for unusable blood (the value of this blood would be zero). Note that blood donations are assumed to arrive with an age of 0. The pre- and post-decision state variables are given by

$$S_t = (R_t, \hat{D}_t), \\ S_t^x = (R_t^x).$$

There is no real “cost” to assigning blood of one type to demand of another type: we are not considering steps such as spending money to encourage additional donations, or transporting inventories from one hospital to another. We use instead the contribution function to capture the preferences of the doctor. We would like to capture the natural preference that it is generally better not to substitute, and that satisfying an urgent demand is more important than an elective demand. For example, we might use the contributions described in Table 14.2. Thus, if we use $O-$ blood to satisfy the needs for an elective patient with $A+$ blood, we would pick up a $-\$10$ contribution (penalty since it is negative) for substituting blood, a $+\$5$ for using $O-$ blood (something the hospitals like to encourage), and a $+\$20$ contribution for serving an elective demand, for a total contribution of $+\$15$.

The total contribution (at time t) is finally given by

$$C_t(S_t, x_t) = \sum_{b \in \mathcal{B}} \sum_{d \in \mathcal{D}} c_{tbd} x_{tbd}.$$

As before, let $X_t^\pi(S_t)$ be a policy (some sort of decision rule) that determines $x_t \in \mathcal{X}_t$ given S_t . We wish to find the best policy by solving

$$\max_{\pi \in \Pi} \mathbb{E} \sum_{t=0}^T \gamma^t C_t(S_t, X_t^\pi). \quad (14.8)$$

The most obvious way to solve this problem is with a simple myopic policy, where we maximize the contribution at each point in time without regard to the effect of our decisions on the future. We can obtain a family of myopic policies by adjusting the one-period contributions. For example, our bonus of $\$5$ for using $O-$ blood (in Table 14.2), is actually a type of myopic policy. We encourage using $O-$ blood

Table 14.2 Contributions for different types of blood and decisions

Condition	Description	Value
if $d = d^\phi$	Holding	0
if $b_1 = b_1$ when $d \in \mathcal{D}$	No substitution	0
if $b_1 \neq b_1$ when $d \in \mathcal{D}$	Substitution	-10
if $b_1 = O-$ when $d \in \mathcal{D}$	$O-$ substitution	5
if $d_2 = \text{Urgent}$	Filling urgent demand	40
if $d_2 = \text{Elective}$	Filling elective demand	20

since it is generally more available than other blood types. By changing this bonus, we obtain different types of myopic policies that we can represent by the set Π^M , where for $\pi \in \Pi^M$ our decision function would be given by

$$X_t^\pi(S_t) = \arg \max_{x_t \in \mathcal{X}_t} \sum_{b \in \mathcal{B}} \sum_{d \in \mathcal{D}} c_{tbd} x_{tbd}. \quad (14.9)$$

The optimization problem in (14.9) is a simple linear program (known as a “transportation problem”). Solving the optimization problem given by equation (14.8) for the set $\pi \in \Pi^M$ means searching over different values of the bonus for using $O-$ blood.

14.2.2 An ADP Algorithm

As a traditional dynamic program the optimization problem posed in equation (14.8) is quite daunting. The state variable S_t has $|\mathcal{B}| + |\mathcal{D}| = 8 \times 6 + 8 \times 2 \times 2 = 80$ dimensions. The random variables \hat{R} and \hat{D} also have a combined 80 dimensions. The decision vector x_t has $27 \times 8 = 216$ dimensions. Using the classical techniques of Chapter 3, this problem looks hopeless. Using the methods that we have presented up to now, obtaining effective solutions to this problem is fairly straightforward.

It is natural to use approximate value iteration to determine the allocation vector x_t using

$$x_t^n = \arg \max_{x_t \in \mathcal{X}_t^n} (C_t(S_t^n, x_t) + \overline{V}_t^{n-1}(R_t^x)), \quad (14.10)$$

where $R_t^x = R^M(R_t, x_t)$ is given by equation (14.6). The first (and most important) challenge we face is identifying an appropriate approximation strategy for $\overline{V}_t^{n-1}(R_t^x)$. A simple and effective approximation is to use separable piecewise-linear approximations, which is to say

$$\overline{V}_t(R_t^x) = \sum_{b \in \mathcal{B}} \overline{V}_{tb}(R_{tb}^x),$$

where $\overline{V}_{tb}(R_{tb}^x)$ is a scalar piecewise-linear function. It is easy to show that the value function is concave (as well as piecewise linear), so each $\overline{V}_{tb}(R_{tb}^x)$ should also be concave. Without loss of generality, we can assume that $\overline{V}_{tb}(R_{tb}^x) = 0$ for $R_{tb}^x = 0$, which means that the function is completely characterized by its set of slopes. We can write the function using

$$\overline{V}_{tb}^{n-1}(R_{tb}^x) = \left(\sum_{r=1}^{\lfloor R_{tb}^x \rfloor} \overline{V}_{tb}^{n-1}(r-1) + (R_{tb}^x - \lfloor R_{tb}^x \rfloor) \overline{V}_{tb}^{n-1}(\lfloor R_{tb}^x \rfloor) \right), \quad (14.11)$$

where $\lfloor R \rfloor$ is the largest integer less than or equal to R . As we can see, this function is determined by the set of slopes ($\overline{V}_{tb}^{n-1}(r)$) for $r = 0, 1, \dots, R^{\max}$, where R^{\max} is an upper bound on the number of resources of a particular type.

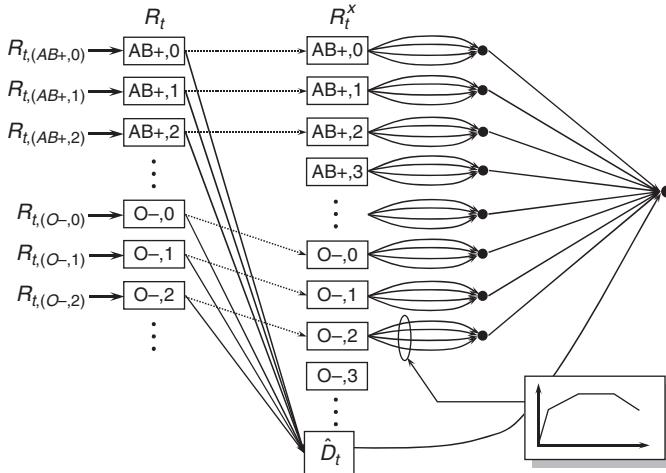


Figure 14.4 Network model for time t with separable, piecewise linear value function approximations.

Assuming that we can estimate this function, the optimization problem we have to solve (equation (14.10)) is the fairly modest linear program shown in Figure 14.4. As with Figure 14.3, we have to consider both the assignment of different types of blood to different types of demand, and the decision to hold blood. To simplify the figure, we have collapsed the network of different demand types into a single aggregate box with demand \hat{D}_t . This network would actually look just like the network in Figure 14.3a. The decision to hold blood has to consider the value of a type of blood (including its age) in the future, which we are approximating using separable piecewise-linear value functions. Here we use a standard modeling trick that converts the separable piecewise-linear value function approximations into a series of parallel links from each node representing an element of R_t^x into a super-sink. Piecewise-linear functions are not only easy to solve (we just need access to a linear programming solver), they are easy to estimate. For many problem classes (but not all) they have additionally been found to produce very fast convergence with high-quality solutions.

With this decision function, we would use approximate value iteration, following the trajectory determined by the policy. Aside from the customization of the value function approximation, the biggest difference is in how the value functions are updated, a problem that is handled in the next section. Of course, we could use other nonlinear approximation strategies, but this would mean that we would have to solve a nonlinear programming problem instead of a linear program. While this can, of course, be done, it complicates the strategy for updating the value function approximation.

For most operational applications this problem would be solved over a finite horizon (e.g., 10 weeks), giving us a recommendation of what to do right now. Alternatively, we could solve an infinite horizon version of the model by simply dropping the t index on the value function approximation.

14.2.3 Updating the Value Function Approximation

Since our value function depends on slopes, we want to use derivatives to update the value function approximation. Let

$$\tilde{V}_t(S_t) = \max_{x_t \in \mathcal{X}_t^n} (C_t(S_t^n, x_t) + \bar{V}_t^{n-1}(R_t^x))$$

be the value of our decision function at time t . But what we want is the derivative with respect to the previous post-decision state R_{t-1}^x . Recall that $S_t = (R_t, D_t)$ depends on $R_t = R_{t-1}^x + \hat{R}_t$. We want to find the gradient $\nabla \tilde{V}_t(S_t)$ with respect to R_{t-1}^x . We apply the chain rule to find

$$\frac{\partial \tilde{V}_t(S_t)}{\partial R_{t-1,b}^x} = \sum_{b' \in \mathcal{B}} \frac{\partial \tilde{V}_t(S_t)}{\partial R_{tb'}} \frac{\partial R_{tb'}}{\partial R_{t-1,b}^x}.$$

If we decide to hold an additional unit of blood of type b_{t-1} , then this produces an additional unit of blood of type $b_t = b^M(b_{t-1}, d)$, where $b^M(b, d)$ is our attribute transition function and d is our decision to hold the blood until the next week. For this application the only difference between b_{t-1} and b_t is that the blood has aged by one week. We also have to model the fact that there may be exogenous changes to our blood supply (donations, deliveries from other hospitals, as well as blood that has gone bad). This means that

$$R_{tb_t} = R_{t-1,b_{t-1}}^x + \hat{R}_{tb_t}.$$

Notice that we start with $R_{t-1,b_{t-1}}^x$ units of blood with attribute b_{t-1} , and then add the exogenous changes \hat{R}_{tb_t} to blood of type b_t . This allows us to compute the derivative of R_t with respect to R_{t-1}^x using

$$\frac{\partial R_{tb_t}}{\partial R_{t-1,b_{t-1}}^x} = \begin{cases} 1 & \text{if } b_t = b^M(b_{t-1}, d), \\ 0 & \text{otherwise.} \end{cases}$$

So we can write the derivative as

$$\frac{\partial \tilde{V}_t(S_t)}{\partial R_{t-1,b_{t-1}}^x} = \frac{\partial \tilde{V}_t(S_t)}{\partial R_{tb_t}}.$$

$\partial \tilde{V}_t(S_t)/\partial R_{tb}$ can be found quite easily. A natural byproduct from solving the linear program in equation (14.10) is that we obtain a dual variable \hat{v}_{tb}^n for each flow conservation constraint (14.3). This is a significant advantage over strategies where we visit state R_t (or S_t) and then update just the value of being in that single state. The dual variable is an estimate of the slope of the decision problem with respect to R_{tb} at the point R_{tb}^n , which gives us

$$\left. \frac{\partial \tilde{V}_t(S_t)}{\partial R_{t-1,b_{t-1}}^x} \right|_{R_{t-1,b_{t-1}}^x} = \hat{v}_{tb_t}^n.$$

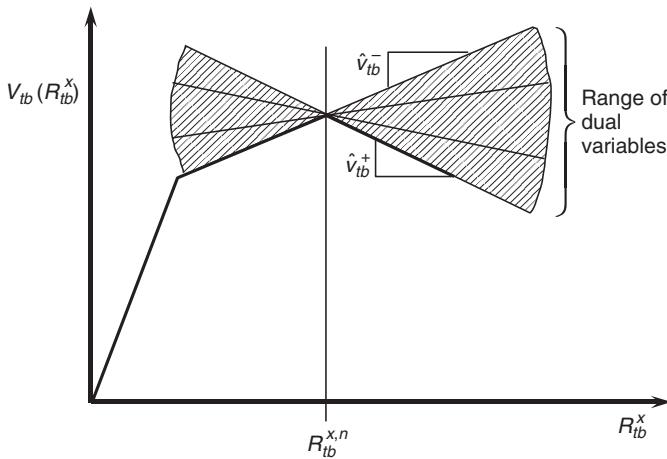


Figure 14.5 Range of dual variables for a nondifferentiable function.

Dual variables are incredibly convenient for estimating value function approximations in the context of resource allocation problems, partly because they approximate the derivatives with respect to R_{tb} (which is all we are interested in) but also because we obtain an entire vector of slopes, rather than a single estimate of the value of being in a state. However, it is important to understand exactly which slope the dual is (or is not) giving us.

Figure 14.5 illustrates a nondifferentiable function (any problem that can be modeled as a sequence of linear programs has this piecewise-linear shape). If we are estimating the slope of a piecewise-linear function $V_{tb}(R_{tb})$ at a point $R_{tb} = R_{tb}^n$ that corresponds to one of these kinks (which is quite common), then we have a left slope, $\hat{v}_{tb}^- = V_{tb}(R_{tb}) - V_{tb}(R_{tb} - 1)$, and a right slope, $\hat{v}_{tb}^+ = V_{tb}(R_{tb} + 1) - V_{tb}(R_{tb})$. If \hat{v}_{tb}^n is our dual variable, we have no way of specifying whether we want the left or right slope. In fact the dual variable can be anywhere in the range $\hat{v}_{tb}^+ \leq \hat{v}_{tb}^n \leq \hat{v}_{tb}^-$.

If we are comfortable with the approximation implicit in the dual variables, then we get the vector $(\hat{v}_{tb}^n)_{b \in \mathcal{B}}$ for free from our simplex algorithm. If we specifically want the left or right derivative, then we can perform a numerical derivative by perturbing each element R_{tb}^n by plus or minus 1 and reoptimizing. This sounds clumsy when we get dual variables for free, but it is surprisingly fast.

Once we have computed \hat{v}_{tb}^n for each $b \in \mathcal{B}$, we now have to update our value function approximation. Remember that we have to update the value function at time $t-1$ at the previous post-decision state. For this problem, attributes evolve deterministically (e.g., AB+ blood that is 3 weeks old becomes AB+ blood that is 4 weeks old). Let $b_t = b^M(b_{t-1}, d)$ describe this evolution (here, d is the decision to hold the blood). Let $\hat{v}_{tb_t}^n$ be the attribute corresponding to blood with attribute b_t at time t , and assume that $R_{t-1,b_{t-1}}^{x,n}$ was the amount of blood of type b_{t-1} at time $t-1$. Let $\bar{V}_{t-1,b_{t-1}}^{n-1}(r)$ be the slope corresponding to the interval $r \leq R_{t-1,b_{t-1}}^{x,n} \leq$

$r + 1$ (as we did in (14.11)). A simple update is to use

$$\overline{V}_{t-1,b_{t-1}}^n(r) = \begin{cases} (1 - \alpha_{n-1})\overline{V}_{t-1,b_{t-1}}^{n-1}(r) + \alpha_{n-1}\hat{v}_{tb_t}^n & \text{if } r = R_{t-1,b_{t-1}}^{x,n}, \\ \overline{V}_{t-1,b_{t-1}}^{n-1}(r) & \text{otherwise.} \end{cases} \quad (14.12)$$

Here we are only updating the element $\overline{V}_{t-1,b_{t-1}}^{n-1}(r)$ corresponding to $r = R_{t-1,b_{t-1}}^{x,n}$. We have found that it is much better to update a range of slopes, say in the interval $(r - \delta, r + \delta)$, where δ has been chosen to reflect the range of possible values of r . For example, it is best if δ is chosen initially so that we are updating the entire range, after which we periodically cut it in half until it shrinks to 1 (or an interval small enough that we are getting the precision we want in the decisions).

There is one problem with equation (14.12) that we have already seen before. We know that the value functions are concave, which means that we should have $\overline{V}_{tb}^n(r) \geq \overline{V}_{tb}^n(r+1)$ for all r . This might be true for $\overline{V}_{tb}^{n-1}(r)$, but it is entirely possible that after our update, it is no longer true for $\overline{V}_{tb}^n(r)$. Nothing new here. We just have to apply the fix-up techniques that were presented in Section 13.3.

The use of separable piecewise-linear approximations has proved effective in a number of applications, but there are open theoretical questions (how good is the approximation?) as well as unresolved computational issues (what is the best way to discretize functions?). What about the use of low-dimensional basis functions? If we use a continuously differentiable approximation (which requires the use of a nonlinear programming algorithm), we can use our regression techniques to fit the parameters of a statistical model that is continuous in the resource variable.

The best value function approximation does not just depend on the structure of the problem. It depends on the nature of the data.

14.2.4 Extensions

We can anticipate several ways in which we can make the problem richer:

- Instead of modeling the problem in single week increments, model daily decisions.
- Include the presence of blood that has been frozen, and the decision to freeze blood.
- A hospital might require weekly deliveries of blood from a community blood bank to make up for systematic shortages. Imagine that a fixed quantity (e.g., 100 units) of blood arrives each week, but the amount of blood of each type and age (the blood may have already been held in inventory for several weeks) might be random.
- We presented a model that focused only on blood inventories at a single hospital. We can handle multiple hospitals and distribution centers by simply adding a location attribute, and providing for a decision to move blood (at a cost) from one location to another.

This model can also be applied to any multiproduct inventory problem where there are different types of product and different types of demands, as long as we have

the ability to choose which type of product is assigned to each type of demand. We also assume that products are not reusable; once the product is assigned to a demand, it is lost from the system.

14.3 A PORTFOLIO OPTIMIZATION PROBLEM

A somewhat different type of resource allocation problem arises in the design of financial portfolios. We can start with some of the same notation we used to manage blood. Let r be an asset class (more generally, the attributes of an asset, such as the type of investment and how long it has been in the investment), where R_{tr} is how much we have invested in asset class r at time t . Let \mathcal{D} be our set of decisions, where each element of \mathcal{D} represents a decision to invest in a particular asset class (there is a one-to-one relationship between \mathcal{D} and the asset classes \mathcal{R}).

The resource transition functions are somewhat different. We let the indicator function $\delta_{r'}(r, d)$ be defined as before. However, now we have to account for transaction costs. Let

$$c_{rd} = \text{cost of acting on asset class } r \text{ with a decision of type } d, \\ \text{expressed as a fraction of the assets being transferred.}$$

The decision d might mean “hold the asset,” in which case $c_{rd} = 0$. However, if we are moving from one asset class to another, we have to pay $c_{rd}x_{rd}$ to cover the cost of the transaction, which is then deducted from the proceeds. This means that the post-decision resource vector is given by

$$R_{tr'}^x = \sum_{r \in \mathcal{R}} \sum_{d \in \mathcal{D}} \delta_{r'}(r, d) (x_{trd} - c_{rd}x_{trd}). \quad (14.13)$$

We now have to account for the market returns from changes in the value of the asset. This is handled by first defining

$$\hat{\rho}_{tr} = \text{relative return on assets of type } r \in \mathcal{R} \text{ between } t-1 \text{ and } t.$$

These returns are the only source of noise that we are considering at this point. Unlike the demands for the blood problem (where we tend to assume that demands are independent), returns tend to be correlated both across assets, and possibly across time. Since we work only with sample paths, we have no trouble handling complex interactions. With these data, our resource transition from R_t^x to R_{t+1} is given by

$$R_{t+1,r} = \hat{\rho}_{t+1,r} R_{tr}^x. \quad (14.14)$$

Normally there is one “riskless” asset class to which we can turn to find nominal, but safe, returns.

Finally, we have to measure how well we are doing. We could simply measure our total wealth, given by

$$\bar{R}_t = \sum_{r \in \mathcal{R}} R_{tr}.$$

We might then set up the problem to maximize our total wealth (actually the total expected wealth) at the end of our planning horizon. This objective ignores the fact that we tend to be risk averse, and we want to avoid significant losses in each time period. So we propose instead to use a utility function that measures the relative gain in our portfolio, such as

$$U(\bar{R}_t) = - \left(\frac{\bar{R}_t}{\bar{R}_{t-1}} \right)^{-\beta},$$

where β is a parameter that controls the degree of risk aversion.

Regardless of how the utility function is constructed, we approach its solution just as we did with the blood management problem. We again write our objective function as

$$\tilde{V}_t(R_t) = \max_{x_t \in \mathcal{X}_t^n} (U(\bar{R}_t) + \gamma \bar{V}_t^{n-1}(R_t^x)), \quad (14.15)$$

where \mathcal{X}_t is the set of x_t that satisfies

$$\sum_{d \in \mathcal{D}} x_{trd} = R_{tr}, \quad (14.16)$$

$$x_{trd} \geq 0. \quad (14.17)$$

R_t^x is given by equation (14.13) (which is a function of x_t), while \bar{R}_t is a constant at time t (it is determined by R_{t-1}^x and $\hat{\rho}_t$). This allows us to write (14.15) as

$$\tilde{V}_t(R_t) = U(\bar{R}_t) + F_t(R_t), \quad (14.18)$$

where

$$F_t(R_t) = \max_{x_t \in \mathcal{X}_t^n} (\gamma \bar{V}_t^{n-1}(R_t^x)).$$

Let \hat{v}_{tr}^n be the dual variable of the flow conservation constraint (14.16). We can use \hat{v}_{tr}^n as an estimate of the derivative of $F_t(R_t)$, giving us

$$\frac{\partial F_t(R_t)}{\partial R_{tr}} \Big|_{R_t^n = R_t} = \hat{v}_{tr}^n. \quad (14.19)$$

Differentiating $U(\bar{R}_t)$ gives

$$\frac{\partial U(\bar{R}_t)}{\partial R_{tr}} = \frac{\beta}{\bar{R}_{t-1}} \left(\frac{\bar{R}_t}{\bar{R}_{t-1}} \right)^{-\beta-1}, \quad (14.20)$$

where we used the fact that $\partial \bar{R}_t / \partial R_{tr} = 1$. We next apply the chain rule to obtain

$$\frac{\partial \tilde{V}_t(R_t)}{\partial R_{t-1,r}^x} = \sum_{r' \in \mathcal{R}} \frac{\partial \tilde{V}_t(R_t)}{\partial R_{tr'}} \frac{\partial R_{tr'}^x}{\partial R_{t-1,r}^x}. \quad (14.21)$$

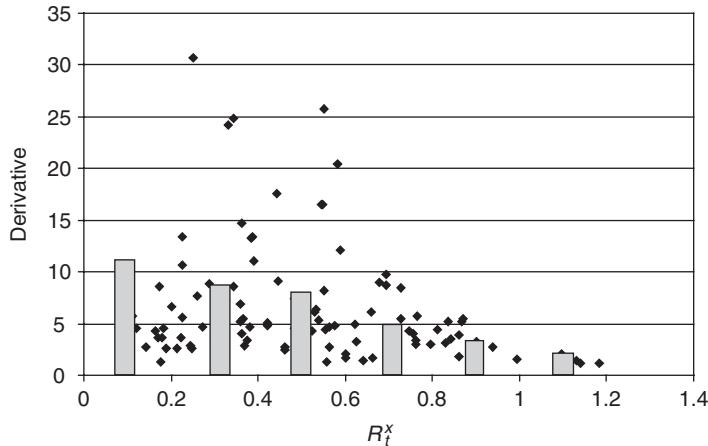


Figure 14.6 Sample realizations for derivatives of the objective function as a function of R_{tr}^x for a particular asset class a . Vertical bars show the average within a range.

Finally, we observe that

$$\frac{\partial R_{tr'}}{\partial R_{t-1,r}^x} = \begin{cases} \hat{\rho}_{tr} & \text{if } r' = r, \\ 0 & \text{otherwise.} \end{cases} \quad (14.22)$$

Combining (14.18) through (14.22) gives us

$$\frac{\partial \tilde{V}_t(R_t)}{\partial R_{t-1,r}^x} = \left(\frac{\partial U(\bar{R}_t)}{\partial R_{tr}} \Big|_{R_t=R_t^n} + \hat{v}_{tr}^n \right) \hat{\rho}_{tr}. \quad (14.23)$$

Figure 14.6 shows sample realizations of the derivatives computed in (14.23) for a problem using the utility function $U(\bar{R}_t) = -0.1(\bar{R}_t/\bar{R}_{t-1})^{-4}$. We would use these sample realizations to estimate a nonlinear function of $R_{t-1,r}^x$ for each resource class r . We expect the derivatives to decline, on average, with $R_{t-1,r}^x$, which is a pattern we see in the figure. However, it is apparent that there is quite a bit of noise in this relationship.

Now that we have our derivatives in hand, we can use it to fit a value function approximation. Again, we face a number of strategies. One is to use the same piecewise-linear approximations that we implemented for our blood distribution problem, which has the advantage of being independent of the underlying distributions. However, a low-order polynomial would also be expected to work quite well, and we might even be able to estimate a nonseparable one (e.g., we might have terms such as $R_{t-1,r_1}^x R_{t-1,r_2}^x$).

Identifying and testing different approximation strategies is one of the most significant challenges of approximate dynamic programming. In the case of the portfolio problem there is a considerable amount of research into different objective functions and the properties of these functions. For example, a popular class of

models minimizes a quadratic function of the variance, which suggests a particular quadratic form for the value function. Even if these functions are not perfect, they would represent a good starting point.

As with the blood management problem, we can anticipate several extensions we could make to our basic model:

- *Short-term bonds*. Our assets could have a fixed maturity, which means that we would have to capture how long we have held the bond.
- *Sales loads that depend on how long we have held an asset*. An asset might be a mutual fund, and some mutual funds impose different loads (sales charges) depending on how long you have held the asset before selling.
- *Transaction times*. Our model already handles transaction costs. Some equities (e.g., real estate) also incur significant transaction times. We can handle this by including an attribute indicating that the asset is in the process of being sold, as well as how much time has elapsed since the transaction was initiated (or the time until the transaction is expected to be completed).
- *Assets that might represent stocks in different industries*. We could design value functions that capture not only the amount we have invested in each stock, but also in each industry (we just aggregate the stocks into industry groupings).

The first three of these extensions can be handled by introducing an attribute that measures age in some way. In the case of fixed maturities, or the time until the asset can be sold without incurring a sales load, we can introduce an attribute that specifies the time at which the status of the asset changes.

14.4 A GENERAL RESOURCE ALLOCATION PROBLEM

The preceding sections have described different instances of resource allocation problems. While it is important to keep the context of an application in mind (every problem has unique features), it is useful to begin working with a general model for dynamic resource allocation so that we do not have to keep repeating the same notation. This problem class allows us to describe a basic algorithmic strategy that can be used as a starting point for more specialized problems.

14.4.1 A Basic Model

The simplest model of a resource allocation problem involves a “resource” that we are managing (people, equipment, blood, money) to serve “demands” (tasks, customers, jobs). We describe the resources and demands using

$R_t =$ Number of resources with attribute $r \in \mathcal{R}$ in the system at time t .

$$R_t = (R_{tr})_{r \in \mathcal{R}}.$$

$D_{tb} =$ Number of demands of type $b \in \mathcal{B}$ in the system at time t .

$$D_t = (D_{tb})_{b \in \mathcal{B}}.$$

Both r and b are vectors of attributes of resources and demands. In addition to the resources and demands, we sometimes have to model a set of parameters (the price of a stock, the probability of an equipment failure, the price of oil) that govern how the system evolves over time. We model these parameters using

ρ_t = a generic vector of parameters that affects the behavior of costs
and the transition function.

The state of our system is given by

$$S_t = (R_t, D_t, \rho_t).$$

New information is represented as exogenous changes to the resource and demand vectors, as well as to other parameters that govern the problem. These are modeled using

\hat{R}_{tr} = Exogenous changes to R_{tr} from information that arrives during time interval t (between $t - 1$ and t).

\hat{D}_{tb} = Exogenous changes to D_{tr} from information that arrives during time interval t (between $t - 1$ and t).

$\hat{\rho}_t$ = Exogenous changes to a vector of parameters (costs, parameters governing the transition).

Our information process then is given by

$$W_t = (\hat{R}_t, \hat{D}_t, \hat{\rho}_t).$$

In the blood management problem, \hat{R}_t included blood donations. In a model of complex equipment such as aircraft or locomotives, \hat{R}_t would also capture equipment failures or delays. In a product inventory setting, \hat{R}_t could represent theft of product. \hat{D}_t usually represents new customer demands, but can also represent changes to an existing demand or cancellations of orders. $\hat{\rho}_t$ could represent random costs, or the random returns in our stock portfolio.

Decisions are modeled using

\mathcal{D}^D = Decision to satisfy a demand with attribute b (each decision $d \in \mathcal{D}^D$ corresponds to a demand attribute $b_d \in \mathcal{B}$).

\mathcal{D}^M = Decision to modify a resource (each decision $d \in \mathcal{D}^M$ has the effect of modifying the attributes of the resource). which includes the decision to “do nothing.”

$$\mathcal{D} = \mathcal{D}^D \cup \mathcal{D}^M.$$

x_{trd} = Number of resources that initially have attribute r that we act on with a decision of type d .

$$x_t = (x_{trd})_{r \in \mathcal{R}, d \in \mathcal{D}}.$$

The decisions have to satisfy constraints such as

$$\sum_{d \in D} x_{trd} = R_{tr}, \quad (14.24)$$

$$\sum_{r \in \mathcal{R}} x_{trd} \leq D_{tbd}, \quad d \in \mathcal{D}^D, \quad (14.25)$$

$$x_{trd} \geq 0. \quad (14.26)$$

We let \mathcal{X}_t be the set of x_t that satisfy (14.24) through (14.26). As before, we assume that decisions are determined by a class of decision functions

$X_t^\pi(S_t) =$ Function that returns a decision vector $x_t \in \mathcal{X}_t$, where $\pi \in \Pi$ is an element of the set of functions (policies) Π .

The transition function is given generically by

$$S_{t+1} = S^M(S_t, x_t, W_{t+1}).$$

We now have to deal with each dimension of our state variable. The most difficult, not surprisingly, is the resource vector R_t . This is handled primarily through the attribute transition function

$$r_t^x = r^{M,x}(r_t, d_t),$$

where r_t^x is the post-decision attribute (the attribute produced by action of type d before any new information has become available). For algebraic purposes we define the indicator function

$$\delta_{r'}(r, d) = \begin{cases} 1 & \text{if } r' = r_t^x = r^{M,x}(r_t, d_t), \\ 0 & \text{otherwise.} \end{cases}$$

Using matrix notation, we can write the post-decision resource vector R_t^x as

$$R_t^x = \Delta R_t,$$

where Δ is a matrix in which $\delta_{r'}(r, d)$ is the element in row r' and column (r, d) . We emphasize that the function $\delta_{r'}(r, d)$ and matrix Δ are used purely for notational convenience; in a real implementation we work purely with the transition function $r^{M,x}(r_t, d_t)$. The pre-decision resource state vector is given by

$$R_{t+1} = R_t^x + \hat{R}_{t+1}.$$

We model demands in a simple way. If a resource is assigned to a demand, then it is “served” and vanishes from the system. Otherwise, it is held to the next time

period. Let

$$\begin{aligned}\delta D_{tb_d} &= \text{number of demands of type } b_d \text{ that are served at time } t, \\ &= \sum_{r \in \mathcal{R}} x_{trd} \quad d \in \mathcal{D}^D, \\ \delta D_t &= (\delta D_{tb})_{b \in \mathcal{B}}.\end{aligned}$$

The demand transition function can be written

$$\begin{aligned}D_t^x &= D_t - \delta D_t, \\ D_{t+1} &= D_t^x + \hat{D}_t.\end{aligned}$$

Finally, we are going to assume that our parameters evolve in a purely exogenous manner such as

$$\rho_{t+1} = \rho_t + \hat{\rho}_{t+1}.$$

We can assume any structure (additive, rule-based). The point is that ρ_t evolves purely exogenously.

The last dimension of our model is the objective function. For our resource allocation problem, we define a contribution for each decision given by

$$c_{rd} = \text{Contribution earned (negative if it is a cost) from using action } d \text{ acting on resources with attribute } r.$$

The contribution function for time period t is assumed to be linear, given by

$$C(S_t, x_t) = \sum_{r \in \mathcal{R}} \sum_{d \in \mathcal{D}} c_{rd} x_{trd}.$$

The objective function is now given by

$$\max_{\pi \in \Pi} \mathbb{E} \left\{ \sum_{t=0}^T C(S_t, X_t^\pi(S_t)) \right\}.$$

14.4.2 Approximation Strategies

The sections on blood management and portfolio optimization both follow our general strategy of solving problems of the form

$$X_t^\pi(S_t^n) = \arg \max_{x_t \in \mathcal{X}_t^n} (C_t(S_t^n, x_t) + \gamma \bar{V}_t^{n-1}(S_t^x)), \quad (14.27)$$

where S_t^n is our state at time t , iteration n , and $\bar{V}_t^{n-1}(S_t^x)$ is our value function approximation computed in iteration $n-1$, evaluated at the post-decision state $S_t^x = S^{M,x}(S_t^n, x_t)$. In the previous sections our state variable took the form

$S_t = (R_t, D_t, \rho_t)$ (for the portfolio problem, ρ_t captured market returns), and we used a value function approximation of the form

$$V_t(S_t) \approx \sum_{r \in \mathcal{R}} \overline{V}_{tr}(R_{tr}^x),$$

where R_{tr}^x is the (post-decision) number of assets with attribute r . A particularly powerful approximation strategy takes advantage of the fact that when we solve (14.27), we can easily obtain gradients of the objective function directly from the dual variables of the resource conservation constraints (contained in \mathcal{X}_t^n) which state that $\sum_{d \in \mathcal{D}} x_{trd} = R_{tr}$. If we do not feel that the duals are sufficiently accurate, we may compute numerical derivatives fairly easily. Section 14.2.3 shows how we can use these derivatives to estimate piecewise-linear approximations.

Linear Value Function Approximation

The simplest approximation strategy outside of a myopic policy (equivalent to $\overline{V}_t(R_t) = 0$), is one that is linear in the resource state, as in

$$\overline{V}_t(R_t) = \sum_{r' \in \mathcal{R}} \overline{V}_{tr'} R_{tr'}.$$

With a linear value function approximation the decision function becomes unusually simple. We start by writing R_{tr}^x using

$$R_{tr'}^x = \sum_{r \in \mathcal{R}} \sum_{d \in \mathcal{D}} \delta_{r'}(r, d) x_{trd}. \quad (14.28)$$

Here a decision $d \in \mathcal{D}$ represents acting on a resource, which might involve serving a demand or simply modifying the resource. Substituting this in our linear value function approximation gives us

$$\overline{V}_t(R_t^x) = \sum_{r' \in \mathcal{R}} \overline{V}_{tr'} \sum_{r \in \mathcal{R}} \sum_{d \in \mathcal{D}} \delta_{r'}(r, d) x_{trd}. \quad (14.29)$$

The decision function is now given by

$$\begin{aligned} X_t^\pi(S_t) &= \arg \max_{x_t \in \mathcal{X}_t} \left(\sum_{r \in \mathcal{R}} \sum_{d \in \mathcal{D}} c_{rd} x_{trd} + \sum_{r' \in \mathcal{R}} \overline{V}_{tr'} \sum_{r \in \mathcal{R}} \sum_{d \in \mathcal{D}} \delta_{r'}(r, d) x_{trd} \right) \\ &= \arg \max_{x_t \in \mathcal{X}_t(\omega)} \sum_{r \in \mathcal{R}} \sum_{d \in \mathcal{D}} \left(c_{rd} + \sum_{r' \in \mathcal{R}} \overline{V}_{tr'} \delta_{r'}(r, d) \right) x_{trd}. \end{aligned} \quad (14.30)$$

Recognizing that $\sum_{r' \in \mathcal{R}} \delta_{r'}(r, d) = \delta_{r^M(r_t, d_t)}(r, d) = 1$, we can write (14.30) as

$$X_t^\pi(S_t) = \arg \max_{x_t \in \mathcal{X}_t(\omega)} \sum_{r \in \mathcal{R}} \sum_{d \in \mathcal{D}} (c_{rd} + \gamma \overline{V}_{t,r^M(r,d)}^{n-1}) x_{trd}. \quad (14.31)$$

So this is nothing more than a network problem where we are assigning resources with attribute r to demands (for decisions $d \in \mathcal{D}^D$) or we are simply modifying resources ($d \in \mathcal{D}^M$). If we were using a myopic policy, we would just use the contribution c_{rd} to determine which decisions we should make now. When we use a linear value function approximation, we add in the value of the resource after the decision is completed, given by $\overline{V}_{t,r^M(r,d)}^{n-1}$.

Solving $X^\pi(S_t)$ is a linear program, and therefore returns a dual variable \hat{v}_{tr} for each flow conservation constraint. We can use these dual variables to update our linear approximation as

$$\overline{V}_{tr}^n = (1 - \alpha_{n-1})\overline{V}_{tr}^{n-1} + \alpha_{n-1}\hat{v}_{tr}^n.$$

Linear (in the resource state) approximations are especially easy to develop and use, but they do not always provide good results. They are particularly useful when managing complex assets such as people, locomotives, or aircraft. For such problems the attribute space is quite large, and as a result R_{tr} tends to be small (e.g., 0 or 1). In general, a linear approximation will work well if the size of the attribute space is much larger than the number of discrete resources being managed, although even these problems can have pockets where there are a lot of resources of the same type.

Linear value function approximations are particularly easy to solve. With a linear value function approximation we are solving network problems with the structure shown in Figure 14.7a, which can be easily transformed to the equivalent network shown in Figure 14.7b where all we have done is to take the slope of the linear value function for each downstream resource type (e.g., the slope v_1 for resources that have attribute r'_1 in the future) and add this value to the arc assigning the resource to the demand. So, if the cost of assigning resources with attribute r_1 to the demand has a cost of c_1 , then we would use a modified cost of $c_1 + v_1$.

Piecewise-Linear Separable Approximation

We solved both the blood management and the portfolio allocation problems using separable piecewise-linear value function approximations. This technique produces results that are more stable when the value function is truly nonlinear in the amount of resources that are provided.

The algorithm is depicted in Figure 14.8. At each time period we solve a linear program similar to what we used in the blood management problem (Figure 14.3), using separable piecewise-linear approximations of the value of resources in the future. The process steps forward in time, using value function approximations for assets in the future. The dual variables from each linear program can be used to update the value function for the previous time period, using the techniques described in Section 13.3.

A critical assumption in this algorithm is that we obtain \hat{v}_{tr} for all $r \in \mathcal{R}$. If we were managing a single asset, this would be equivalent to sampling the value of being in all possible states at every iteration. In Chapter 4, we referred to this strategy as *synchronous approximate dynamic programming* (see Section 4.9.2).

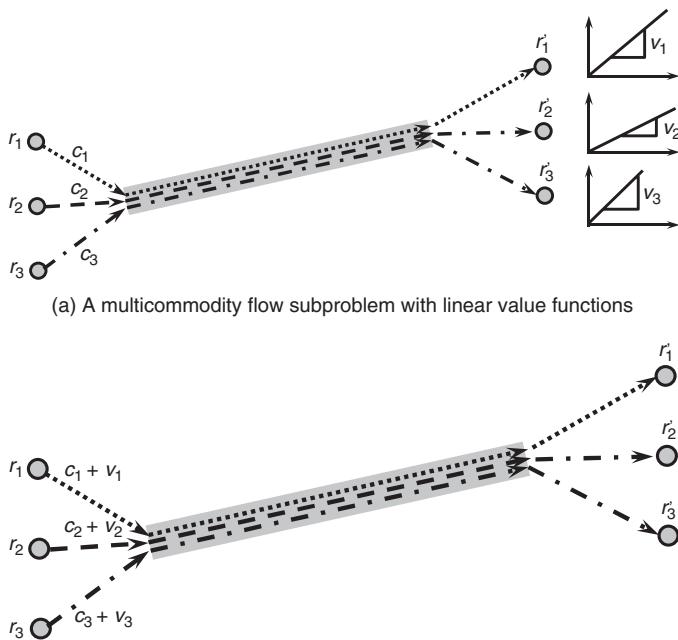


Figure 14.7 If linear value functions are used (a) the problem can be converted into an equivalent pure network (b).

Below (in Section 14.6) we describe an application where this is not possible, and we also describe strategies for overcoming this problem.

14.4.3 Extensions and Variations

Piecewise-linear separable approximations have worked quite well for a number of applications, but it is important to realize that this is just one of many potential approximations that could be used. The power of piecewise-linear functions is that they are extremely flexible and can be used to handle virtually any distribution for demands (or other parameters). But separable approximations will not work for every problem.

A popular approximation strategy is to use simple polynomials such as

$$\bar{V}_t(S_t) \approx \sum_{r \in \mathcal{R}} \sum_{r' \in \mathcal{R}} \theta_{rr'} R_{tr} R_{tr'}.$$

This is a linear-in-the-parameters regression problem that can be estimated using the techniques of Section 9.3 for recursive least squares. This approximation can be a challenge if the set \mathcal{R} is fairly large. Neural networks (Section 8.4.4) offer another avenue of investigation that, as of this writing, has not received any attention for this problem class.

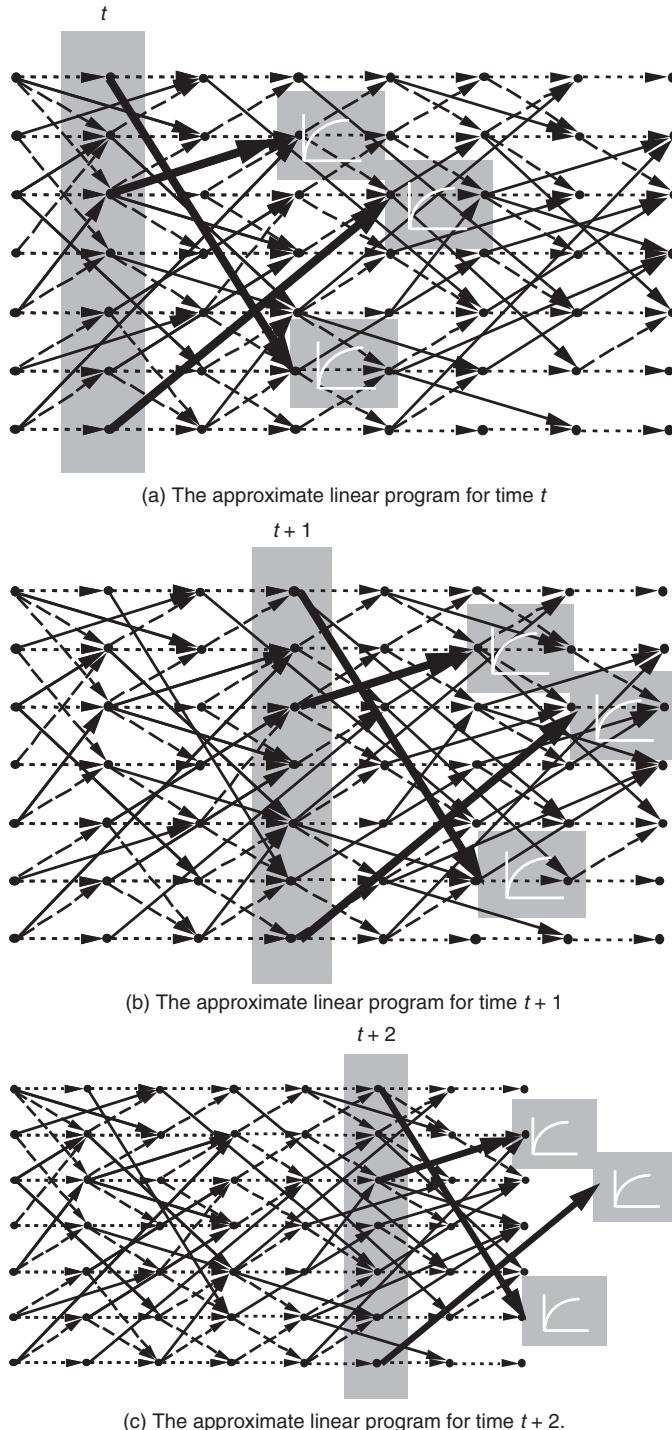


Figure 14.8 ADP for general multistage resource allocation problems using separable value function approximations.

We can retain the use of separable piecewise-linear approximations but partially overcome the separability approximation by writing the value function on an aggregated state space. For example, there may be subsets of resources which substitute fairly easily (e.g., people or equipment located nearby). We can aggregate the attribute space \mathcal{R} into an aggregated attribute space $\mathcal{R}^{(g)}$, where $r^g \in \mathcal{R}^{(g)}$ represents an aggregated attribute. We can then write

$$V_t(S_t) \approx \sum_{r^g \in \mathcal{R}^{(g)}} \bar{V}_{tr^g}(R_{tr^g}^x),$$

where $R_{tr^g}^x$ is the number of resources with aggregated attribute r^g . When we use aggregated value functions, we may have several gradients \hat{v}_a^n , giving the derivative with respect to R_{tr} , which need to be combined in some way to update a single aggregated value function $\bar{V}_{tr^g}(R_{tr^g}^x)$.

We may also consider using the techniques of Section 8.1.4 and multiple levels of aggregation, which would produce an approximation that looks like

$$V_t(S_t) \approx \sum_{g \in \mathcal{G}} \sum_{r^g \in \mathcal{R}^{(g)}} \theta_{r^g}^{(g)} \bar{V}_{tr^g}(R_{tr^g}^x),$$

where \mathcal{G} is a family of aggregations and $\theta_{r^g}^{(g)}$ is the weight we give to the function for attribute r^g for aggregation level g .

Finally, we should not forget that we can use a linear or piecewise-linear function of one variable such as R_{tr} that is indexed by one or more other variables. For example, we might feel that the value of resources of type r_1 are influenced by the number of resources of type r_2 . We can estimate a piecewise-linear function of R_{tr_1} that is indexed by R_{tr_2} , producing a family of functions as illustrated in Figure 14.9. This is a powerful strategy in that it does not assume any specific functional relationship between R_{tr_1} and R_{tr_2} , but the price is fairly steep if we want to have the value function indexed by more than just one additional element such as R_{tr_2} . We might decide that there are 5 or 10 elements that might affect the behavior

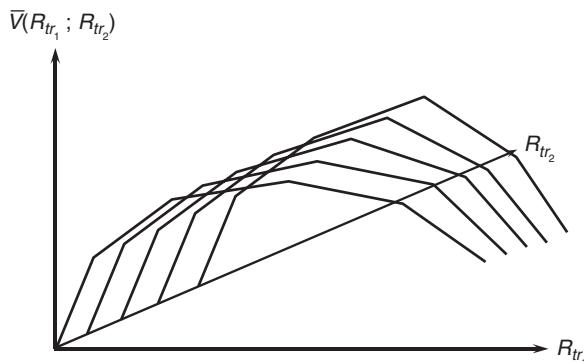


Figure 14.9 Family of piecewise-linear value function approximations.

of the function with respect to R_{tr_1} . Instead of estimating one piecewise-linear value function approximation for each R_{tr_1} , we might find ourselves estimating thousands, so there is a limit to this strategy.

A powerful idea which has been suggested (but we have not seen it tested in this setting) is the use of ridge regression. Create a scalar *feature resource variable* given by

$$\bar{R}_{tf} = \sum_{r \in \mathcal{R}} \theta_{fr} R_{tr}, \quad f \in \mathcal{F}.$$

Now estimate a value function approximation

$$V_t(S_t) \approx \sum_{f \in \mathcal{F}} \bar{V}_{tf}(\bar{R}_{tf}).$$

This approximation still consists of a series of scalar, piecewise-linear functions, but now we are using specially constructed variables \bar{R}_{tf} made up of linear combinations of R_{tr} .

14.4.4 Value Function Approximations and Problem Structure

Linear value functions have a number of nice features. They are easy to estimate (one parameter per attribute). They offer nice opportunities for decomposition. If city i and city j both send equipment to city k , a linear value function means that the decisions of city i and city j do not interact (at least not through the value function). Finally, it is often the case that the optimization problem is a special type of linear program known as a *pure network*, which means that it can be completely described in terms of nodes, links, and upper bounds on the flows. Pure networks have a nice property: if all the data (supplies, demands, upper bounds) are integer, then the optimal solution (in terms of flows) will also be integer. This is especially useful if our resources are discrete (people or equipment). It means that if we solve the problem as a linear program (ignoring the need for integer solutions), then the optimal solution is still integer.

Unfortunately, linear value function approximations often do not work very well.

Nonlinear value function approximations can improve the quality and stability of the solution considerably. We have already seen with our blood management and portfolio problems that separable piecewise-linear value function approximations can be particularly easy to estimate and use. However, if we are managing discrete resources, then integrality becomes an issue. Recall that our decision problem looks like

$$X^\pi(R_t) = \arg \max_{x_t \in \mathcal{X}_t} \left\{ \sum_{r \in \mathcal{R}} \sum_{d \in \mathcal{D}_r} c_{rd} x_{trd} + \sum_{r \in \mathcal{R}} \bar{V}_{tr}^{n-1}(R_{tr}^x(x_t)) \right\}. \quad (14.32)$$

If our only constraints are the flow conservation constraint ($\sum_d x_{trd} = R_{tr}$), nonnegativity, and possibly upper bounds on individual flows ($x_{trd} \leq u_{trd}$), then

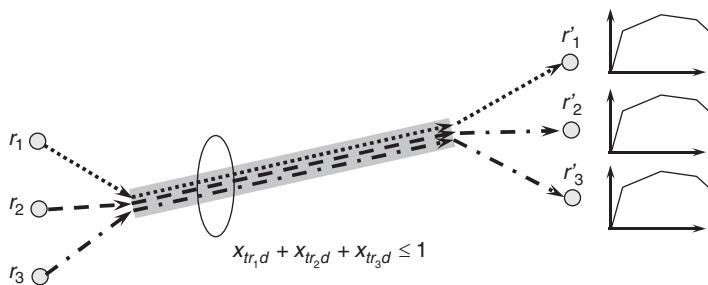


Figure 14.10 Different types of resources used to serve a demand and nonlinear value functions used to capture the value of each type of resource in the future, as bundled constraints used to capture the type of flow serving the demand.

equation (14.32) is a pure network, which means that we can ignore integrality (i.e., we can solve it as a continuous linear program) but still obtain integer solutions. What may destroy this structure are constraints of the form

$$\sum_{r \in R} x_{trd} \leq R_{tbd}^D, \quad d \in \mathcal{D}^D.$$

Constraints such as this are known as *bundle constraints*, since they bundle different decisions together (“we can use resources of type 1 or 2 or 3 to cover this demand”). The problem is illustrated in Figure 14.10. When we introduce bundle constraints, we are no longer able to solve the problem as a continuous linear program and count on getting integer solutions. We have to use instead an integer programming algorithm, which can be much slower (it depends on the structure of the problem).

We avoid this situation if our problem exhibits what we might refer to as the *Markov property*, which is defined as follows:

Definition 14.4.1 *We say that a resource allocation problem has the Markov property if*

$$r^M(r, d, W_{t+1}) = r^M(d, W_{t+1}),$$

which is to say, the attributes of a transition resulting from a decision \$d\$ acting on a resource with attribute \$r\$ is independent of \$r\$ (the transition is “memoryless”).

An example of a problem that exhibits the Markov property is the portfolio problem. If we decide to invest money in asset class \$k\$, we do not care if the money came from asset class \$i\$ or \$j\$. Another problem with the Markov property arises when we are managing a fleet of identical vehicles, where the attribute vector \$a\$ captures only the location of the vehicle. A decision to “move the vehicle to location \$j\$” determines the attribute of the vehicle after the decision is completed (the vehicle is now at location \$j\$).

If a problem exhibits the Markov property, then we obtain a network shown in Figure 14.11. If a linear program with this structure is solved using a linear

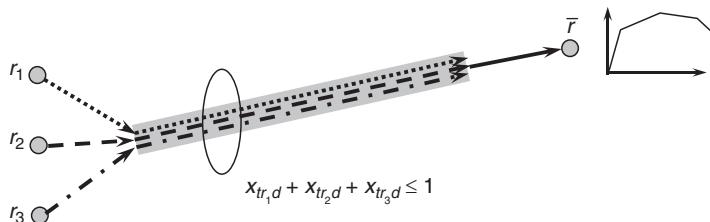


Figure 14.11 Flow, after serving a demand, with an attribute that depends only on the attributes of the demand. All the value ends in the same value function.

programming package without imposing integrality constraints, the solution will still be integer.

If we have a problem that does not exhibit the Markov property, we can restore this property by aggregating resources in the future. To illustrate, imagine that we are managing different types of medical technicians to service different types of medical equipment in a region. The technician might be characterized by his training, his current location, and how many hours he has been working that day. After servicing a machine at a hospital, he has changed his location and the number of hours he has been working, but not his training. Now assume that when we approximate the value of the technician in the future, we ignore his training and how many hours he has been working (leaving only the location). Now his attributes are purely a function of the decision (or equivalently, the job he has just completed).

The good news about this problem is that commercial solvers seem to easily produce optimal (or near optimal) integer solutions even when we combine piecewise-linear separable value function approximations in the presence of bundle constraints. Integer programs come in different flavors, and we have found that this problem class is one of the easy ones. However, it is important to allow the optimization solver to return “near optimal” solutions (all codes allow you to specify an epsilon tolerance in the solution). Obtaining provably optimal solutions for integer programs can be extremely expensive (the code spends most of the time verifying optimality), while allowing an epsilon tolerance can produce run times that are almost as fast as if we ignore integrality. The reason is that if we ignore integrality, the solution will typically be 99.99 percent integer.

14.4.5 Applications

There is a vast range of resource allocation problems that can be modeled with this framework. We began the chapter with three examples (asset acquisition, blood inventories, and portfolio optimization). Some examples of other applications include:

- *Inspecting passengers at airports.* The Transportation Safety Administration (TSA) assigns each arriving passenger to a risk category that determines the level of inspection for passenger (simple X-ray and magnetic detection, an inspector using the hand wand, more detailed interrogation). Each level

requires more time from selected resources (inspectors, machines). For each arriving passenger we have to determine the right level of inspection, taking into account future arrivals, the capacity of each inspection resource, and the probability of detecting a real problem.

- *Multiskill call centers*. People phoning in for technical support for their computer have to progress through a menu that provides some information about the nature of their problem. The challenge then is to match a phone call characterized by a vector of attributes a that captures the phone options, with a technician (each of whom has different skills). Assigning a fairly simple request to a technician with special skills might tie up that technician for what could be a more important call in the future.
- *Energy planning*. A national energy model might have to decide how much capacity for different types of energy we should have (coal, ethanol, wind mills, hydro, oil, natural gas) at each time period to meet future demands. Many forms of capacity have lifetimes of several decades, but these decisions have to be made under uncertainty about commodity prices and the evolution of different energy technologies.
- *Planning flu vaccines*. The Center for Disease Control (CDC) has to decide each year which flu vaccine (or vaccines) should be manufactured to respond to potential outbreaks in the upcoming flu season. It can take six months to manufacture a vaccine for a different strain, while flu viruses in the population evolve continuously.
- *Fleet planning for charter jet operators*. A company has to purchase different types of jets to serve a charter business. Customers might request a particular type of jet. If the company does not have jets of that type available, the customer might be persuaded to upgrade to a larger jet (possibly with the inducement of “no extra charge”). The company has to decide the proper mix of jets, taking into account not only the uncertainty in the customer demands but also their willingness to accept upgrades.
- *Work force planning*. Large companies, and the military, have to develop the skills in their employees to meet anticipated needs. The skills may be training in a specific activity (filling out types of forms, inspecting machinery, or working in a particular industry as a consultant) or could represent broader experiences (spending time abroad, working in accounting or marketing, working in the field). It may take time to develop a skill in an employee who previously did not have the skill. The company has to decide which employees to develop to meet uncertain demands in the future.
- *Hospital operations*. There are numerous problems requiring the management of resources for hospitals. How should different types of patients be assigned to beds? How should nurses be assigned to different shifts? How should doctors and interns be managed? Which patients should be scheduled for surgery and when?
- *Queueing networks*. Queueing networks arise when we are allocating a resource to serve customers who move through the system according to a

set of exogenous rules (i.e., we are not optimizing how the customer moves through the network). These applications often arise in manufacturing where the customer is a job (e.g., a computer circuit board) that has to go through a series of steps at different stations. At each station is a machine that might perform several tasks, but can only perform one task at a time. The resource allocation problem involves determining what type of task a machine should be working on at a point in time. Given a set of machines (and how they are configured), customers move from one machine to another according to fixed rules (that may depend on the state of the system).

Needless to say, the applications are nearly endless and cut across different dimensions of society (corporate applications, energy, medical, homeland security). All of these problems can be reasonably approximated using the types of techniques we describe in this chapter.

14.5 A FLEET MANAGEMENT PROBLEM

Consider a freight transportation company that moves loads of freight in trailers or containers. Assume that each load fills the container (we are not consolidating small shipments into larger containers). After the company moves a load of freight from one location to another, the container becomes empty and may have to be repositioned empty to a new location where it is needed. Applications such as these arise in railroads, truckload motor carriers, and intermodal container shipping.

These companies face two types of decisions (among others). (1) If there are too many customer demands (a fairly common problem), the company has to decide which customers to serve (i.e., which loads of freight to move) by balancing the contribution earned by moving the load, plus the value of the container when it becomes empty at the destination. (2) The company might have more containers in one location than are needed, making it necessary to move containers empty to a location where they may be needed (possibly to serve demands that are not yet known).

A model based on the algorithmic strategies described in this chapter was implemented at a major railroad to manage their freight cars. This section briefly describes the modeling and algorithmic strategy used for this problem.

14.5.1 Modeling and Algorithmic Strategy

This problem can be perfectly modeled using the notation in Section 14.4. The attribute vector r will include elements such as the location of the container, the type of container, and, because it usually takes multiple time periods for a container to move from one location to the next, the expected time of arrival (or, the number of time periods until it will arrive). The time to travel is usually modeled deterministically, but exercise 14.2 asks you to generalize this basic model to handle random travel times. An important characteristic of this problem is that while the

attribute space is not small, it is also not too large, and in fact we will depend on our ability to estimate values for each attribute in the entire attribute space.

The decision set \mathcal{D}^D represents decisions to assign a container to a particular type of load. The load attributes b typically include the origin and destination of the load, but may also include attributes such as the customer (this may determine which types of containers are allowed to be used), the type of freight, and the service requirements.

The problem is very similar to our blood management problem, with two major exceptions. First, in the blood problem, we could assign blood to a demand (after which it would vanish from the system) or hold it. With the fleet management problem, we also have the option of “modifying” a car, captured by the decision set \mathcal{D}^M . The most common “modification” is to move a car empty from one location to another (we are modifying the location of the car). Other forms of modification include cleaning a car (freight cars may be too dirty for certain types of freight), repair a car, or change its “ownership” (cars may be assigned to shipper pools, where they are dedicated to the needs of a particular shipper).

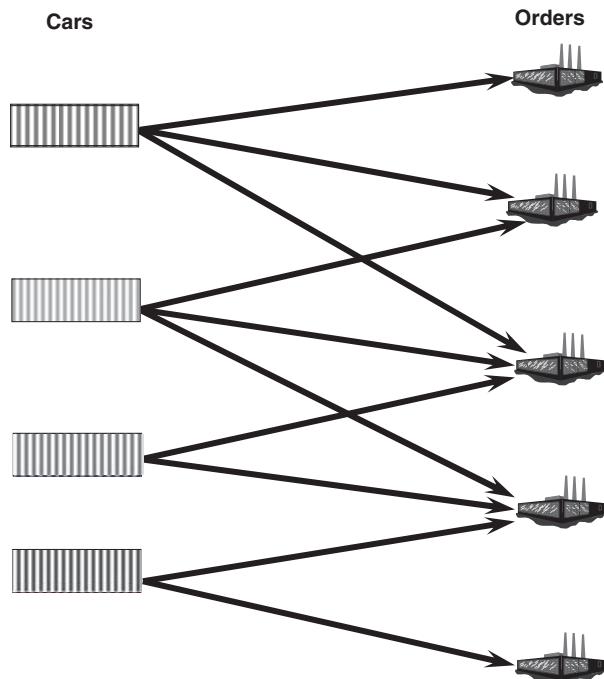
The second exception is that while used blood vanishes from the system, a car usually remains in the system after it is assigned to an order. We say usually because a railroad in the eastern United States may pick up a car that has to be taken to the West Coast. To complete this move, the car has to be transferred to a railroad that serves the West Coast. Although the car will eventually return to the first railroad, for planning purposes the car is considered lost. At a later time it will appear as if it is a random arrival (similar to our random blood donations).

The most widely used model in practice is an optimization model that assigns individual cars to individual orders at a point in time (Figure 14.12a). These models do not consider the downstream impact of a decision now on the future, although they are able to consider cars that are projected to become available in the future, as well as forecasted orders. However, they cannot handle a situation where an order has to be served 10 days from now which might be served with a car after it finishes an order that has to be served three days from now. Figure 14.12b shows the decision function when using value function approximations, where we not only assign known cars to known orders (similar to the myopic model in Figure 14.12a), but we also capture the value of cars in the future (through piecewise-linear value function approximations).

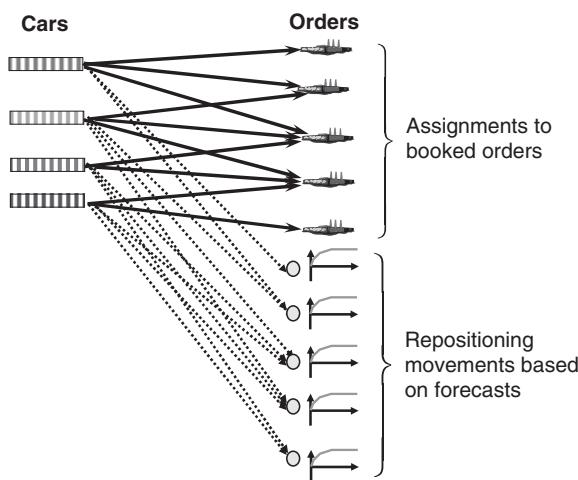
Railroads that use the basic assignment model (several major North American railroads do this) only solve them using known cars and orders. If we use approximate dynamic programming, we would first solve the problem at time $t = 0$ for known cars and orders, but we would then step forward in time and use simulated cars and orders. By simulating over a three-week horizon, ADP produces not only a better decision at time $t = 0$ but also a forecast of car activities in the future. This proved very valuable in practice.

In Section 14.4.2 we mentioned that we can use the dual variable \hat{v}_{tr}^n (or a numerical derivative) for the resource constraint

$$\sum_{d \in \mathcal{D}} x_{trd} = R_{tr}$$



(a) Basic car-to-order assignment problem



(b) Car assignment problem using ADP

Figure 14.12 (a) Basic assignment model for assigning cars to orders used in engineering practice. (b) Model that uses piecewise-linear separable value function approximations estimated by approximate dynamic programming.

to update the value function approximation $\overline{V}_{t-1,r}^{n-1}(R_{t-1,r}^x)$. When the attribute space is not too large (something that we assume for this problem class), we can assume that we are calculating \hat{v}_{tr}^n for each $r \in \mathcal{R}$. It turns out that this is an extremely powerful capability, and not one that we can do for more complex problems where the attribute space is too large to enumerate.

14.5.2 Laboratory Experiments

There are two types of laboratory experiments that help provide insights into the quality of an ADP solution: (1) those performed on a deterministic dataset and (2) those on a stochastic dataset. Deterministic experiments are useful because these are problems that can be solved to optimality, providing a tight bound to compare against. Stochastic datasets allow us to evaluate how well the method performs under uncertainty, but we have to compare against heuristic solutions since tight bounds are not available.

Deterministic Experiments

There are many variations of the fleet management problem. In the simplest, we have a single type of container, and we might even assume that the travel time from one location to another is a single time period (in this case the attribute vector r consists purely of location). We assume that there is no randomness in the resource transition function (which means that $\hat{R}_t = 0$), and that if a demand is not satisfied at time t , it is lost from the system (which means that $D_t = \hat{D}_t$). We may additionally assume that a demand originating from location i can only be served by containers that are already located at location i . In Section 14.6 below we relax a number of these assumptions.

If the problem is deterministic (which is to say, all the demands \hat{D}_t are known in advance), then the problem can be modeled as a network such as the one shown in Figure 14.13. This problem is easily solved as a linear program. Since the resulting linear program has the structure of what is known as a *pure network*, it is also the case that if the flows (and upper bounds) are integer, then the resulting solution is integer.

We can solve the problem using approximate dynamic programming for a deterministic application by pretending that \hat{D}_t is stochastic (i.e., we are not allowed to use \hat{D}_{t+1} when we solve the problem at time t), but where every time we “observe” \hat{D}_t we obtain the same realization. Using approximate dynamic programming, our decision function would be given by

$$X_t^\pi(S_t) = \arg \max_{x_t \in \mathcal{X}_t} (C(S_t, x_t) + \overline{V}_t^{n-1}(R_t^x)),$$

where we use a separable, piecewise linear value function approximation (as we did in the previous sections).

Experiments (reported in Powell and Godfrey, 2002) were run on problems with 20, 40, and 80 locations and with 15, 30, and 60 time periods. Since the problem is deterministic, we can optimize the entire problem (i.e., over all time periods) as

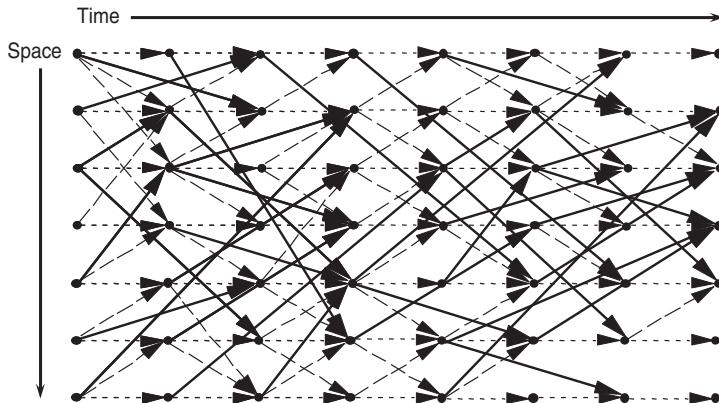


Figure 14.13 Pure network for time-staged, single commodity flow problems.

Table 14.3 Percentage of the optimal deterministic solution produced by separable piecewise-linear value function approximations

Locations	Simulation Horizon		
	15	30	60
20	100.00%	100.00%	100.00%
40	100.00%	99.99%	100.00%
80	99.99%	100.00%	99.99%

Source: From Powell and Godfrey (2002).

a single linear program to obtain the true optimal solution, allowing us to see how well our approximate solution compares to optimal (on a deterministic problem). The results are reported in Table 14.3 as percentages of the optimal solution produced by the linear programming algorithm. It is not hard to see that the results are very near optimal. We know that separable piecewise-linear approximations do not produce provably optimal solutions (even in the limit) for this problem class, but it appears that the error is extremely small.

Stochastic Experiments

We now consider what happens when the demands are truly stochastic, which is to say that we obtain different values for \hat{D}_t each time we sample information. For this problem we do not have an optimal solution. Although this problem is relatively small (compared to true industrial applications), if we formulate it as a Markov decision process we would produce a state space that is far larger than anything we could hope to solve using the techniques of Chapter 3. The standard approach used in engineering (which we have also found works quite well) is to use a rolling horizon procedure where at each time t we combine the demands that are known at time t with expectations of any random demands for future time

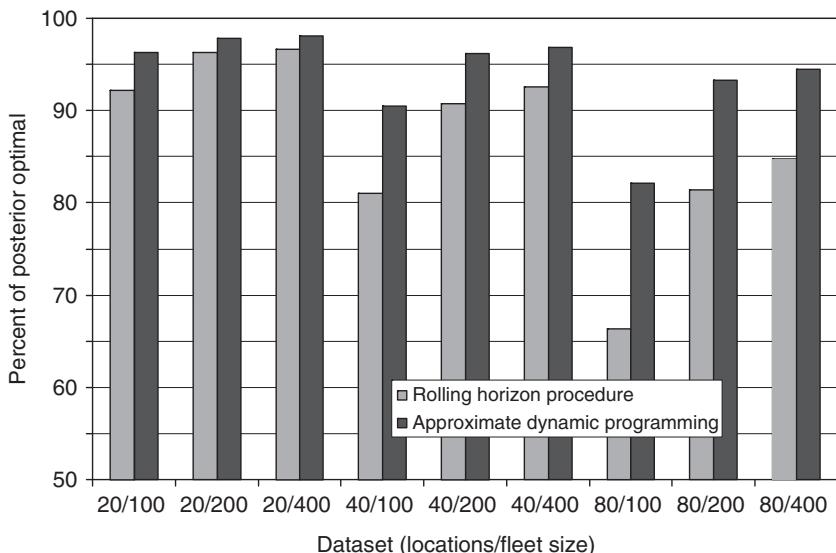


Figure 14.14 Percentage of posterior bound produced by a rolling horizon procedure using a point forecast of the future versus an approximate dynamic programming approximation.

periods. A deterministic problem is then solved over a *planning horizon* of length T^{ph} that typically has to be chosen experimentally.

For a specific sample realization of the demands we can still find an optimal solution using a linear programming solver, but this solution “cheats” by being able to use information about what is happening in the future. This solution is known as the *posterior bound* since it uses information that only becomes known after the fact. Although it cheats, it provides a nice benchmark against which we can compare our solutions from the ADP algorithm.

Figure 14.14 compares the results produced by a rolling horizon procedure to those produced using approximate dynamic programming (with separable piecewise-linear value function approximations). All results are shown as a percentage of the posterior bound. The experiments were run on problems with 20, 40, and 80 locations and with 100, 200, and 400 vehicles in our fleet. Problems with 100 vehicles were able to cover roughly half of the demands, while a fleet of 200 vehicles could cover over 90 percent. The fleet of 400 vehicles was much larger than would have been necessary. The ADP approximation produced better results across all the problems, although the difference was most noticeable for problems where the fleet size was not large enough to cover all the demands. Not surprisingly, this was also the problem class where the posterior solution was relatively the best.

These experiments demonstrate that piecewise-linear separable value function approximations work quite well for this problem class. However, as with all experimental evidence, care has to be used when generalizing these results to other datasets. This technique has been used in several industrial applications with

success, but without provable bounds it will be necessary to reevaluate the approximation for different problem classes. We have found, however, that the technique scales quite well to very large scale problems.

14.5.3 A Case Implementation

This strategy was implemented in an industrial application for managing freight cars for a major railroad. Two aspects of the problem proved to be much more complex than anticipated. First, in addition to the usual attributes of location and type of freight car, we also had to consider estimated time of arrival (for cars that are currently moving from one location to another), cleanliness (some shippers reject dirty cars), repair status, and ownership. The second aspect was the complexity of the information process. Random demands turned out to be only one of the sources of uncertainty. When a customer calls in an order, he will specify the origin of the order (this is where we send the car to), but not the destination (this is only revealed after the car is loaded). Also a customer might reject a car as being too dirty only after it is delivered to the customer. Finally, the time to load and unload the car, as well as transit times, are all uncertain. None of these issues presented a significant modeling or algorithmic challenge.

Figure 14.12 described two models: (1) a basic car-to-order assignment problem widely used in industry and (2) a model which uses approximate dynamic programming to measure the value of cars in the future. So this raises the question: How do these two methods compare? Figure 14.15 compares the performance of the

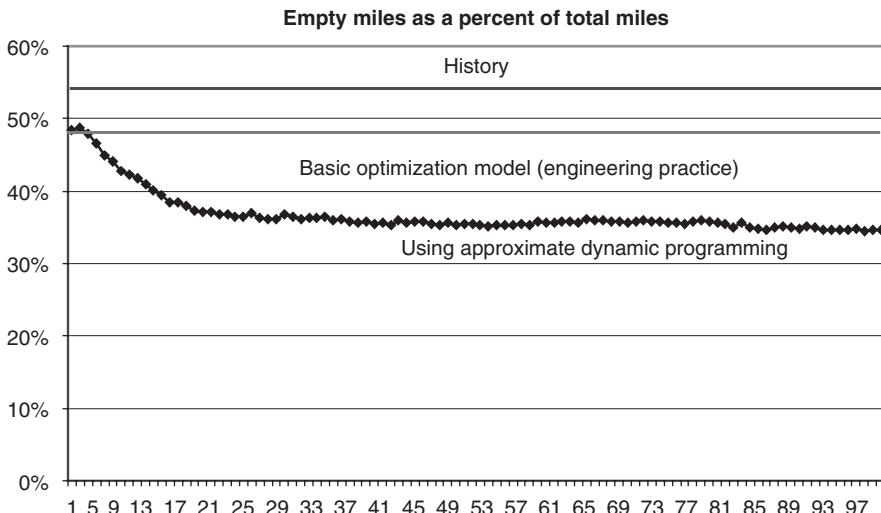


Figure 14.15 Empty miles as a percentage of total from an industrial application, showing historical performance (54 percent of total miles were empty), results using a myopic assignment model and when using approximate dynamic programming.

myopic model (Figure 14.12a) to the model using approximate dynamic programming (Figure 14.12b) against historical performance, with empty miles treated as a percentage of total as the performance measure. For this dataset, 54 percent of the total mileage generated by the cars were spent moving empty (a surprising statistic). In the simple assignment model (which we simulated forward in time) this number dropped to about 48 percent empty, which is a significant improvement. Using approximate dynamic programming (where we show the performance after each iteration of the algorithm), we were able to drop empty miles to approximately 35 percent of the total.

14.6 A DRIVER MANAGEMENT PROBLEM

In the previous section we managed a set of “containers” that were described by a fairly simple set of attributes. For this problem we assumed that we could compute gradients \hat{v}_{tr}^n for each attribute $r \in \mathcal{R}$. We lose this ability when we make the transition from managing simple resources such as a container to complex resources such as drivers.

We use as our motivating application the problem of managing a fleet of drivers for a truckload motor carrier. The mathematical model is identical to what we used for managing containers (which is the model we presented in Section 14.4). The only difference now is that the attribute vector r is large enough that we can no longer enumerate the attribute space \mathcal{R} . A second difference we are going to introduce is that if we do not handle a demand (to move a load of freight) at time t , then the demand is held over to the next time period. This means that if we cannot handle all the demands, we need to think about which demands we want to satisfy first.

This summary is based on an actual industrial project that required modeling drivers (and loads) at a very high level of detail. The company ran a fleet of over 10,000 drivers, and our challenge was to design a model that closely matched actual historical performance. It turns out that experienced dispatchers do a fairly good job of thinking about the downstream impact of decisions. For example, a dispatcher might want to put a team of two drivers (which are normally assigned to a long load, which takes advantage of the ability of a team to move constantly) on a load going to Boston, but it might be the case that most of the loads out of Boston are quite short. It might be preferable to put the team on a load going to Atlanta if there are more long loads going out of Atlanta.

14.6.1 Working with Complex Attributes

When we were managing simple resources (“containers”), we could expect that the resource state variable R_{tr} would take on integer values greater than 1 (in some applications in transportation, they could number in the hundreds). When the attribute vector becomes complex, then it is generally going to be the case (with rare exception) that R_{tr} is going to be zero (most of the time) or 1.

In an actual project with a truckload motor carrier the attribute vector was given by

$$r = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \\ r_8 \\ r_9 \\ r_{10} \end{pmatrix} = \begin{pmatrix} \text{Location} \\ \text{Domicile} \\ \text{Capacity type} \\ \text{Scheduled time at home} \\ \text{Days away from home} \\ \text{Available time} \\ \text{Geographical constraints} \\ \text{DOT road hours} \\ \text{DOT duty hours} \\ \text{Eight-day duty hours} \end{pmatrix}.$$

The presence of complex attributes introduces one nice simplification: we can use linear (in the resource state) value function approximations (rather than the piecewise-linear functions). This means that our value function approximation can be written

$$\overline{V}_t(R_t^x) = \sum_{r \in \mathcal{R}} \overline{V}_{tr} R_{tr}^x.$$

Instead of estimating a piecewise-linear function, we have only to estimate \overline{V}_{tr} for each r .

Now we have to update the value of a driver with attribute r . In theory, we would again compute the dual variable \hat{v}_{tr}^n for the resource constraint $\sum_{d \in \mathcal{D}} x_{trd} = R_{tr}$ for each r . This is where we run into problems. The attribute space can be huge, so we can only find \hat{v}_{tr}^n for some of the attributes. A common strategy is to find a dual only for the attributes where $R_{tr} > 0$. Even for a very large problem (the largest trucking companies have thousands of drivers), we have no problem solving linear programs with thousands of rows. But this means that we are not obtaining \hat{v}_{tr}^n when $R_{tr} = 0$.

This issue is very similar to the classical problem in ADP when we do not estimate the value of states we do not visit. Imagine if we were managing a single driver. In this case r_t would be the “state” of our driver at time t . Assuming that we get \hat{v}_{tr} for each r is like assuming that we visit every state at every iteration. This is fine for small state spaces (i.e., resources with simple attributes) but causes a problem when the attributes become complicated.

Fortunately, we already have the tools to solve this problem. In Section 8.1.4 we saw that we could estimate \overline{V}_{tr} at different levels of aggregation. Let $\mathcal{R}^{(g)}$ be the attribute space at the g th level of aggregation, and let $\overline{V}_{tr}^{(g)}$ be an estimate of a driver with attribute $r \in \mathcal{R}^{(g)}$. We can then estimate the value of a driver with (disaggregate) attribute r using

$$\overline{V}_{tr} = \sum_{g \in \mathcal{G}} w_r^{(g)} \overline{V}_{tr}^{(g)}.$$

Section 8.1.4 provides a simple way of estimating the weights $w_r^{(g)}$.

14.6.2 Backlogging Demands

We next address the problem of deciding which demand to serve when it is possible to hold unserved demands to future time periods. Without backlogging, the post-decision state variable was given by $S_t^x = (R_t^x)$. Equation (14.28) determines R_t^x as a function of R_t and x_t (the resource transition function). With backlogging, the post-decision state is given by $S_t^x = (R_t^x, D_t^x)$, where $D_t^x = (D_{tb}^x)_{b \in \mathcal{B}}$ is the vector of demands that were not served at time t . D_t^x is given simply using

$$\begin{aligned} D_{tb}^x &= \text{number of loads with attribute vector } b \text{ that have not been served} \\ &\quad \text{after decisions were made at time } t, \\ &= D_{tb} - \sum_{r \in \mathcal{R}} x_{trd}, \text{ where } b = b_d, d \in \mathcal{D}^D. \end{aligned} \tag{14.33}$$

In equation (14.33), for a decision $d \in \mathcal{D}^D$ to move a load, each element in \mathcal{D}^D references an element in the set of load attributes \mathcal{B} . So, for $d \in \mathcal{D}^D$, $\sum_{r \in \mathcal{R}} x_{trd}$ is the number of loads of type b_d that were moved at time t .

If we use a linear value function approximation, we would write

$$\overline{V}_t(R_t) = \sum_{r \in \mathcal{R}} \overline{V}_{tr}^R R_{tr}^x + \sum_{b \in \mathcal{B}} \overline{V}_{tb}^D D_{tb}^x.$$

As before, \overline{V}_{tr}^R is the value of a driver with attribute r , while \overline{V}_{tb}^D is the value of holding an order with attribute b . Just as \overline{V}_{tr}^R is estimated using the dual variable of the resource constraint

$$\sum_{d \in \mathcal{D}} x_{trd} = R_{tr},$$

we would estimate the value of a load in the future, \overline{V}_{tb}^D using the dual variable for the demand constraint

$$\sum_{r \in \mathcal{R}} x_{trd} \leq D_{tb_d}, \quad d \in \mathcal{D}^D.$$

Recall that each element in the set \mathcal{D}^D corresponds to a type of demand \mathcal{B} . We refer to \overline{V}_t^R as *resource gradients* and \overline{V}_t^D as *task gradients*.

Whenever we propose a value function approximation, we have to ask the following questions: (1) Can we solve the resulting decision function? (2) Can we design an effective updating strategy (to estimate \overline{V})? (3) Does the approximation work well (does it provide high-quality decisions)?

For our problem the decision function is still a linear program. The value functions can be updated using dual variables (or numerical derivatives) just as we did for the single-layer problem. So the only remaining question is whether it works well. Figure 14.16 shows the results of a simulation of a driver management problem where we make decisions using a purely myopic policy ($\overline{V}_t^R = \overline{V}_t^D = 0$), a

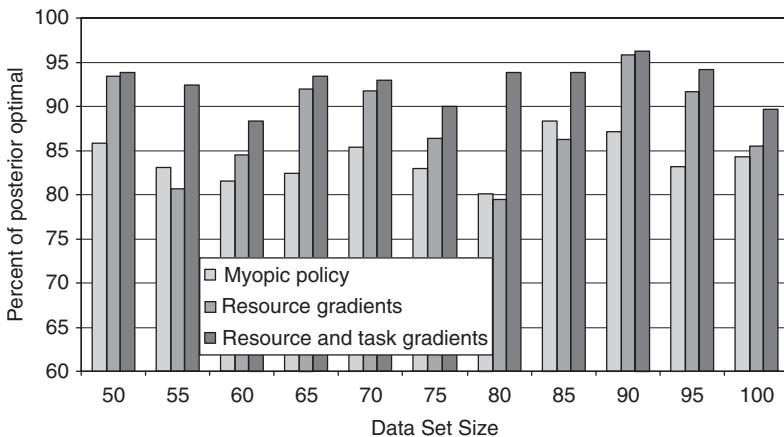


Figure 14.16 Value of including value functions for resources alone, and resource and tasks, compared to a myopic policy (from Spivey and Powell, 2004).

policy that ignored the value of demands (tasks) in the future ($\bar{V}^D = 0$), and a policy that used value functions for both resources and demands. Simulations were run for a number of datasets of increasing size. All of the runs are evaluated as a percentage of the posterior solution (which we can solve optimally by assuming we knew everything that happened in the future). For these datasets the myopic policy produced an average performance that was 84 percent of the posterior bound. Using value functions for just the first resource layer produced a performance of almost 88 percent, while including value functions for both resource layers produced solutions that were 92.6 percent of the posterior optimal.

14.6.3 Extensions

This basic model is capable of handling a number of extensions and generalizations with minor modifications.

Continuous Actionable Times versus Discrete Decision Epochs

Perhaps one of the most powerful features of approximate dynamic programming is its ability to model activities using a very detailed state variable while maintaining a computationally compact value function approximation. One dimension where this is particularly important is the modeling of time. In our transportation applications where we are often managing thousands of resources (drivers, equipment), it is typically necessary to solve decision subproblems in discrete time, possibly with a fairly coarse discretization (e.g., every 6 or 12 hours). However, we can still model the fact that real activities happen in continuous time.

A common mistake in modeling is to assume that if we make decisions in discrete time, then every activity has to be modeled using the same level of discretization. This could not be farther from the truth. We can introduce in our attribute vector r an attribute called the *actionable time*, which is the time at

which we can actually change the status of a resource (we can do the same for the demands, by adding an “actionable time” field to the demand attribute vector b). Suppose that we are making decisions at time $t = 0, 100, 200, \dots$ (times are in minutes). Assume that at time $t = 100$ we wish to consider assigning a resource (driver) with actionable time of $r_{\text{actionable}} = 124$ to a demand (load) with actionable time $b_{\text{actionable}} = 162$, and assume that it takes $\tau = 50$ minutes for the driver to move to the load. If we make this decision, then the earliest time that the load can actually be picked up is at time $t = 124 + 50 = 174$. We can model these activities as taking place down to the nearest minute, even though we are making decisions at 100-minute intervals.

Relaying Loads

The simplest resource allocation problem involves a single *resource layer* (blood, money, containers, people) satisfying demands. In our driver management problem we allowed, for the first time, unsatisfied demands to be held for the future, but we still assumed that if we assign a driver to a load, then the load vanishes from the system.

It is also possible to assume that if we act on a load with attribute b with decision d that we produce a load with attribute b' . Previously we described the effect of a decision d on a resource with attribute r using the attribute transition function

$$r_{t+1} = r^M(r_t, d, W_{t+1}).$$

We could use a similar transition function to describe how the attributes of a load evolve. However, we need some notational sleight of hand. When modeling the evolution of resources, we know that the attribute transition function has access to both the attribute of the driver, r_t , and the attribute of the load, which is captured implicitly in the decision d that tells us what type of load we are moving (the attributes are given by b_d). To know how the attributes of the load change, we can define a load transition function

$$b_{t+1} = b^M(r_t, d, W_{t+1}).$$

This looks a little unusual because we include r_t in the arguments but not b_t . In fact b_t is determined by the action d , and we need to know both the driver attributes r_t as well as the load attributes $b_t = b_d$. Algebraically we used the function $\delta_{r'}(r, d)$ to indicate the attribute produced by the decision d acting on the attribute r . When we have drivers and loads, we might use $\delta_{r'}^D(r, d)$ as the indicator function for drivers and use $\delta_b^L(b, d)$ as the indicator function for loads. All the algebra would carry through symmetrically.

Random Travel Times

We can handle random travel times as follows. Suppose that at time $t = 100$, we have a driver who is currently headed to Cleveland. We expect him to arrive at time $t = 115$, at which point he will be empty (status E) and available to be reassigned.

Although he has not arrived yet, we make the decision to assign him to pick up a load that will take him to Seattle, where we expect him to arrive at time $t = 218$. This decision needs to reflect the value of the driver in Seattle at time $t = 218$.

By time 200, the driver has not yet arrived but we have learned that he was delayed (our random travel time), and we now expect him to arrive at time $t = 233$. The sequence of pre- and post-decision states is given below.

$$\begin{array}{cccc} & t = 100 & t = 100 & t = 200 \\ \text{Label} & \text{pre-decision} & \text{post-decision} & \text{pre-decision} \\ \left(\begin{array}{c} \text{Location} \\ \text{Status} \\ \text{ETA} \end{array} \right) & \left(\begin{array}{c} \text{Cleveland} \\ E \\ 115 \end{array} \right) & \left(\begin{array}{c} \text{Seattle} \\ E \\ 218 \end{array} \right) & \left(\begin{array}{c} \text{Seattle} \\ E \\ 233 \end{array} \right). \end{array}$$

At time $t = 200$ we use his state (at the time) that he will be in Seattle at time 233 to make a new assignment. We use the value of the driver at this time to update the value of the driver in Seattle at time 218 (the previous post-decision state).

Random Destinations

There are numerous applications in transportation where we have to send a vehicle (container, taxi) to serve a customer before we know where the customer is going. People routinely get into taxi cabs and then tell the driver where they are going. Customers of railroads often ask for “50 box cars” without specifying where they are going.

We can handle random destinations much as we handled random travel times. Suppose we have the same driver headed to Cleveland, with an expected time of arrival of 115. Assume that at time 100 we decide to move the driver empty up to nearby Evanston to pick up a new load, but we do not know the destination of the load. The post-decision state would be “loaded in Evanston.” By time $t = 200$ we have learned that the load was headed to Seattle, with an expected arrival time (at time 200) of 233. The sequence of pre- and post-decision states is given by

$$\begin{array}{cccc} & t = 100 & t = 100 & t = 200 \\ \text{Label} & \text{pre-decision} & \text{post-decision} & \text{pre-decision} \\ \left(\begin{array}{c} \text{Location} \\ \text{Status} \\ \text{ETA} \end{array} \right) & \left(\begin{array}{c} \text{Cleveland} \\ E \\ 115 \end{array} \right) & \left(\begin{array}{c} \text{Evanston} \\ L \\ 132 \end{array} \right) & \left(\begin{array}{c} \text{Seattle} \\ E \\ 233 \end{array} \right). \end{array}$$

We again make a new decision to assign the driver, and use the value of the driver when we make this decision to update the value of the driver when he was loaded in Evanston (the previous post-decision state).

14.7 BIBLIOGRAPHIC NOTES

Section 14.1 Piecewise-linear functions are an especially convenient way of approximating concave functions. Powell and Godfrey (2001) proposed the CAVE algorithm, which has proved successful in a number of applications

(see Powell and Godfrey, 2002; Powell and Topaloglu, 2006). Powell et al. (2004) and Topaloglu and Powell (2003) provide convergence proofs for the variations called the leveling algorithm and the SPAR algorithm. Convergence proofs for scalar, multistage problems are provided by Nascimento and Powell (2009) and Nascimento and Powell (2010a).

Section 14.2 The material in this section is based on the undergraduate senior thesis of Lindsey Cant (Cant, 2006). The use of separable piecewise-linear approximations was developed in a series of papers arising in transportation and logistics (e.g., Powell and Godfrey 2001; Powell et al. 2004).

Section 14.3 The material in this section is based on Basler (2006).

Section 14.4 The notation in this section is based on a very general model for resource allocation given in Simao et al. (2001). The use of separable piecewise-linear value function approximations has been invested in Powell and Godfrey (2001), Powell and Godfrey (2002), and most thoroughly in Powell and Topaloglu (2006). The idea of using ridge regression to create value function approximations is due to Klabjan and Adelman (2007).

Section 14.5 This section is based on Powell and Godfrey (2002) and Powell and Topaloglu (2006). For a discussion of approximate dynamic programming for fleet management, using the vocabulary of stochastic programming, see Powell and Topaloglu (2003).

Section 14.6 Powell el al. (2002) and Spivey and Powell (2004) laid the foundations for this work in the truckload trucking industry. A thorough presentation of this project with a complete description of the algorithm is given in Simao et al. (2009). For a more accessible presentation, see Simao et al. (2010).

PROBLEMS

- 14.1** Revise the notation for the blood management problem to include the presence of testing that may show that inventories that have not yet reached the six week limit have gone bad. State any assumptions and introduce notation as needed. Be sure to model both pre- and post-decision states. Carefully describe the transition function.
- 14.2** Consider the problem of managing a fleet of vehicles as described in Section 14.5. There we assumed that the travel time required to move a container from one location to another required one time period. Now we are going to ask you to generalize this problem.
- (a) Assume that the time required to move from location i to location j is τ_{ij} , where τ_{ij} is deterministic and integer. Give two different ways for modeling this by modifying the attribute r in a suitable way (recall that if the travel time is one period, $r = \{location\}$).
- (b) How does your answer to (a) change if τ_{ij} is allowed to take on non-integer values?

- (c) Now assume that τ_{ij} is random, following a discrete uniform distribution (actually it can be any general discrete distribution). How do you have to define the attribute a ?

14.3 Orange futures project We have the opportunity to purchase contracts that allow us to buy frozen concentrated orange juice (FCOJ) at some point in the future. When we sign the contract in year t , we commit to purchasing FCOJ during year t' at a specified price. In this project $t' = 10$, while $1 \leq t \leq 10$. To model the problem, let

$x_{tt'} =$ quantity of FCOJ ordered at time t (more specifically, with the information up through time t), to be used during year t' ,

$$x_t = (x_{tt'})_{t' \geq t},$$

$R_{tt'} =$ FCOJ that we know about at time t that can be used during year t' ,

$$R_t = (R_{tt'})_{t' \geq t},$$

$p_{tt'} =$ price paid for FCOJ purchased at time t that can be used during year t' ,

$$p_t = (p_{tt'})_{t' \geq t},$$

$D_t =$ demand for FCOJ during year t .

For this problem, $t' = 10$. The demand for FCOJ (D_t) does not become known until year 10, and is uniformly distributed between 1000 and 2000 tons. If you have not ordered enough to cover the demand, you must make up the difference by purchasing on the spot market, with price given by $p_{10,10}$.

We are going to specify a stochastic model for prices. For this purpose, let

$$\begin{aligned} \rho_0^u &= \text{initial upper range for prices} \\ &= 2.0, \end{aligned}$$

$$\begin{aligned} \rho_0^\ell &= \text{initial lower range for prices} \\ &= 1.0, \end{aligned}$$

$$\begin{aligned} \rho^u &= \text{upper range for prices} \\ &= 1.2, \end{aligned}$$

$$\begin{aligned} \rho^\ell &= \text{lower range for prices} \\ &= 0.9, \end{aligned}$$

$$\begin{aligned} \beta &= \text{mean reversion parameter} \\ &= 0.5, \end{aligned}$$

$$\begin{aligned}\rho^{u,s} &= \text{upper range for spot prices} \\ &= 1.15, \\ \rho^{\ell,s} &= \text{lower range for spot prices} \\ &= 1.03.\end{aligned}$$

Let U represent a random variable that is uniformly distributed between 0 and 1 (in Excel, this is computed using RAND()). Let $\omega = 0$ be the initial sample, and let $\omega = 1, 2, \dots$ be the remaining samples. Let $p_{t,t'}(\omega)$ be a particular sample realization. Prices are randomly generated using

$$\begin{aligned}p_{0,10}(\omega) &= 1.7(\rho_0^\ell + (\rho_0^u - \rho_0^\ell)U), \omega \geq 0, \\ p_{t,10}(0) &= p_{t-1,10}(\rho^\ell + (\rho^u - \rho^\ell)U), t = 1, 2, \dots, 9, \\ p_{10,10}(\omega) &= p_{9,10}(\rho^{\ell,s} + (\rho^{u,s} - \rho^{\ell,s})U), \omega \geq 0, \\ \bar{p}_{t,10}(0) &= p_{t,10}(0), t \geq 0, \\ \bar{p}_{0,10}(\omega) &= 0.9\bar{p}_{0,10}(\omega - 1) + 0.05p_{0,10}(\omega), \omega \geq 1, \\ \bar{p}_{t,10}(\omega) &= 0.9\bar{p}_{t-1,10}(\omega - 1) + 0.05p_{t-1,10}(\omega), \omega \geq 1, \\ p_{t,10}(\omega) &= p_{t-1,10}(\rho^u - \rho^\ell)U + \beta(p_{t-1,10}(\omega) - \bar{p}_{t,10}(\omega)), t = 1, 2, \dots, 9, \omega \geq 1.\end{aligned}$$

This process should produce a random set of prices that tend to trend upward but that will move downward if they get too high (known as a mean reversion process).

- (a) Prepare a detailed model of the problem.
- (b) Design an approximate dynamic programming algorithm, including a value function approximation and an updating strategy.
- (c) Implement your algorithm and describe the algorithmic tuning steps you had to go through to obtain what appears to be a useful solution. Be careful that your stepsize is not too small.
- (d) Compare your results with different mean reversion parameters, such as $\beta = 0$ and 1.
- (e) How does your answer change if the upper range for spot prices is increased to 2.0?

- 14.4 Blood management project.** Implement the blood management model described in Section 14.2. Assume that total demand averages 200 units of blood each week, and total supply averages 150 units of blood per week. Assume that the actual supplies and demands are normally distributed with a standard deviation equal to 30 percent of the mean (set any negative realizations to zero). Use the table below for the distribution of supply and

demand (these are actuals for the United States). Use Table 14.2 for your costs and contributions.

Blood type	$AB+$	$AB-$	$A+$	$A-$	$B+$	$B-$	$O+$	$O-$
Percentage of supply	3.40	0.65	27.94	5.17	11.63	2.13	39.82	9.26
Percentage of demand	3.00	1.00	34.00	6.00	9.00	2.00	38.00	7.00

- (a) Develop a program that simulates blood inventories in steady state, using a discount factor of 0.80, where decisions are made myopically (the value function is equal to zero). Simulate 1000 weeks, and produce a plot showing your shortages each week. This is easiest to implement using a software environment such as Matlab. You will need access to a linear programming solver.
- (b) Next implement an ADP strategy using separable piecewise-linear value function approximations. Clearly state any algorithmic choices you have to make (e.g., the stepsize rule). Determine how many iterations appear to be needed in order for the solution to stabilize. Then run 1000 testing iterations and plot the shortages each week. Compare your results against the myopic policy. What types of behaviors do you notice compared to the myopic policy?
- (c) Repeat part (b) assuming first that the average supply is 200 units of blood each week, and then again with the average set to 250. Compare your results.

14.5 For the blood management problem, compare the results obtained using value functions trained for an infinite horizon problem against those trained for a 10-week horizon using a fixed set of starting inventories (make reasonable assumptions about these inventories). Compare your results for the finite horizon case using both sets of value functions (when using the infinite horizon value functions, you will be using the same value function for each time period). Experiment with different discount factors.

14.6 Transformer replacement project. There are many industries that have to acquire expensive equipment to meet needs over time. This problem is drawn from the electric power industry (a real project), but the issues are similar to airlines that have to purchase new aircraft or manufacturers that have to purchase special machinery.

Electric power is transported from power sources over high-voltage lines, which drop to successively lower voltages before entering your house by using devices (familiar to all of us) called transformers. In the 1960s the industry introduced transformers that could step power from 750,000 volts down to 500,000 volts. These high-capacity transformers have a lifetime that is expected to be somewhere in the 50- to 80-year range. As of this

writing, the oldest units are just reaching 40 years, which means there is no data on the actual lifetime. As a result there is uncertainty about the lifetime of the units.

The problem is that the transformers cost several million dollars each, can take a year or more to build, and weigh over 200 tons (so they are hard to transport). If a transformer fails, it creates bottlenecks in the grid. If enough fail, a blackout can occur. The more common problem, however, is that failures force the grid operators (these are distinct from the utilities themselves) to purchase power from more expensive plants.

To model the problem, define

$$x_t = \text{number of transformers ordered at the end of year } t \text{ that will arrive at the beginning of year } t+1$$

$$R_{tr} = \text{number of transformers at time } t \text{ with age } r,$$

$$R_t^A = \text{total number of active transformers at the end of year } t \\ = \sum_r R_{tr},$$

$$F_{tr} = \text{a random variable giving the number of failures that occur during year } t \text{ of transformers with age } r.$$

For \$2.5 million, the company can order a transformer in year t to arrive in year $t + 1$.

Table 14.4 gives the current age distribution (2 transformers are only 1 year old, while 10 are 39 years old). The probability $f(a)$ that a unit that is a years old will fail is given by

$$f(a) = \begin{cases} \rho^f & \text{if age } a \leq \tau, \\ \rho^f(a - \tau) & \text{if age } a > \tau. \end{cases}$$

ρ^f is the base failure rate for transformers with age $a \leq \tau$, where we assume $\rho^f = 0.01$ and $\tau = 40$. For transformers with age $a > \tau$ (a is an integer number of years), the failure rate rises according to the polynomial given above.

At an aggregate level the total network congestion costs (in millions of dollars per year) are given approximately by

$$C(R_t) = 5 (\max(145 - R_t^A, 0))^2. \quad (14.34)$$

So, if we have $R_t^A \geq 145$ active transformers, then congestion costs are zero.

Your challenge is to use approximate dynamic programming to determine how many transformers the company should purchase in each year. Your performance will be measured based on total purchase and congestion

Table 14.4 Initial age distribution of the equipment

Age	R_{0a}	Age	R_{0a}	Age	R_{0a}	Age	R_{0a}
1	2	11	2	21	1	31	12
2	1	12	2	22	1	32	0
3	0	13	1	23	2	33	8
4	9	14	0	24	7	34	10
5	2	15	1	25	2	35	16
6	4	16	1	26	3	36	6
7	2	17	0	27	1	37	2
8	7	18	4	28	1	38	6
9	1	19	2	29	2	39	10
10	6	20	1	30	7	40	0

costs over a 50-year period averaged over 1000 samples. For this project, do the following

- (a) Write up a complete model of the dynamic program.
- (b) Describe at least one value function approximation strategy.
- (c) Describe an updating method for your value function approximation. Specify your stepsize rule.
- (d) Implement your algorithm. Describe the results of steps to tune your algorithm. Compare different strategies (stepsize rules, approximation strategies) and present your conclusion of the strategy that works the best.
- (e) Now assume that you are not allowed to purchase more than 4 transformers in any given year. Compare the acquisition strategy to what you obtained without this constraint.

14.7 We now extend the previous project by introducing uncertainty in the lifetime. Repeat exercise 14.6, but now assume that the lifetime τ is a random variable that might be 40, 45, 50, 55, or 60 with equal probability. Describe how this changes the performance of the model.

14.8 We are going to revisit the transformer problem (holding the lifetime τ fixed at 40 years), but this time we are going to change the dynamics of the purchasing process. In exercise 14.6 we could purchase a transformer for \$2.5 million that would arrive the following year. Now, we are going to be able to purchase transformers for \$1.5 million, but they arrive in three years. That is, a transformer purchased at the end of year t is available to be used in year $t + 3$.

This is a significantly harder problem. For this version it will be necessary to keep track of transformers that have been ordered but that have not yet arrived. This can be handled by allowing the age a to be negative. So $R_{t,-2}$ would be the number of transformers we know about at time t that will first be available in 2 time periods.

In addition it will be necessary to choose between ordering a transformer quickly for a higher price, or more slowly at a lower price. You will need to design an approximation strategy that handles the different types of transformers (consider the use of separable, piecewise-linear value function approximations as we did for the blood management problem).

Repeat exercise 14.6, placing special care on the modeling of the state variable, the decision variable, and the transition function. Instead of transformers arriving to a single inventory, they now arrive to the system with different ages. Try starting with a separable piecewise-linear value function approximation that captures the value of transformers of different ages. However, try experimenting with other functional approximations. For example, you might have a single-value function that uses the total number of transformers that are not only available but that also have yet to arrive. Report your experience, and compare your results to the original problem when a transformer arrives in the next time period.

Implementation Challenges

So you are finally ready to solve your own problem with approximate dynamic programming. Of course, it is nice if you have a problem that closely matches one that we have already discussed in this book. But you don't. As is often the case, you have to fall back on general principles to solve a problem that does not look like any of the examples that we have presented.

In this chapter we describe a number of issues that tend to arise in real applications of approximate dynamic programming. These are intended to help as a rough guide to what can be expected in the path to a successful ADP application.

15.1 WILL ADP WORK FOR YOUR PROBLEM?

The first question you have to ask is: Do you need approximate dynamic programming to solve your problem? There are many problems where myopic policies are going to work quite well (in special cases they may be provably optimal). For example, the optimal solution to steady-state batch replenishment problems are known to have the structure where, if the amount of product on hand is less than s , then we should order an amount that brings our inventory up to S (these are known as (s, S) policies). If we are managing a group of identical repairmen serving jobs that pop up randomly (and uniformly) over a region, then it is also unlikely that a value function that helps us look into the future will noticeably improve the solution.

In short, we have to ask the question: Does a value function add value? If we estimate a value function, how do we think our solution will improve?

Problems where myopic policies work well (and in particular where they are provably optimal) tend to be quite simple (how much product to order, when to sell an asset). There are, of course, more complex problems where myopic policies are known to be optimal. For example, the problem of allocating funds among

Approximate Dynamic Programming: Solving the Curses of Dimensionality, Second Edition.

Warren B. Powell.

© 2011 John Wiley & Sons, Inc. Published 2011 by John Wiley & Sons, Inc.

a set of asset classes to balance risk can be solved with a myopic policy if we assume that there are no transaction costs (or transaction times). In fact it should not be surprising to find out that if it is possible to move from any state to any other state (instantly and with no cost), then a myopic policy will be optimal. Not surprisingly, ADP is not going to contribute very much to these problems.

In contrast, the hardest ADP problems tend to be applications where it takes numerous steps to obtain a reward. The easiest examples of such problems are games (chess, checkers, Connect-4) where many steps have to be made before we know if we won or lost. ADP works particularly well when we have access to a policy that works “pretty well” and we can use it to train value functions. A well-designed approximate value function can make a good policy even better. But where we have an intractably hard problem, we may have to start with naive rules that work poorly. We may avoid a decision to visit a state if we do not know how to properly behave once we reach the state. It is these problems where it is often necessary to turn to an outside supervisor who can guide the search algorithm during initial learning stages.

15.2 DESIGNING AN ADP ALGORITHM FOR COMPLEX PROBLEMS

There are numerous problems in discrete optimization that have to be solved over time, under uncertainty. For example, it may be necessary to decide which jobs should be assigned to different machines, and in which order, in the presence of random arrivals of jobs in the future. Service vehicles have to be assigned to customer demands (picking up packages, or providing household plumbing services) that arise randomly over time. Transportation companies have to dispatch trucks with shipments that arrive randomly to a terminal.

Deterministic versions of these problems (where all customer demands are known in advance) can be extremely difficult, and such problems are generally solved using heuristics. Stochastic dynamic versions of these problems are generally solved using simulation, where at time t we would solve an optimization problem using only what is known at time t . As new information becomes known, the problem would be re-optimized. Solving difficult integer programs in real time can be challenging, primarily because of the time constraints on solution times. But the solution quality can be relatively poor since the decisions do not reflect what might happen in the future.

The first step in developing an ADP strategy for these problems is to begin by implementing a myopic policy in the form of a simulation. Consider a problem faced by the military in the scheduling of unmanned aerial vehicles (UAVs) that have to be routed to collect information about various targets. A target is a request to visit an area to determine its status (Was a bridge destroyed? Is a missile battery operational? Are there people near a building?). Targets arise randomly, and sometimes it is necessary to plan the path of a UAV through a sequence of targets in order to determine which one should be visited first. The problem is illustrated in Figure 15.1, which shows UAVs both on the ground and in the air, with paths

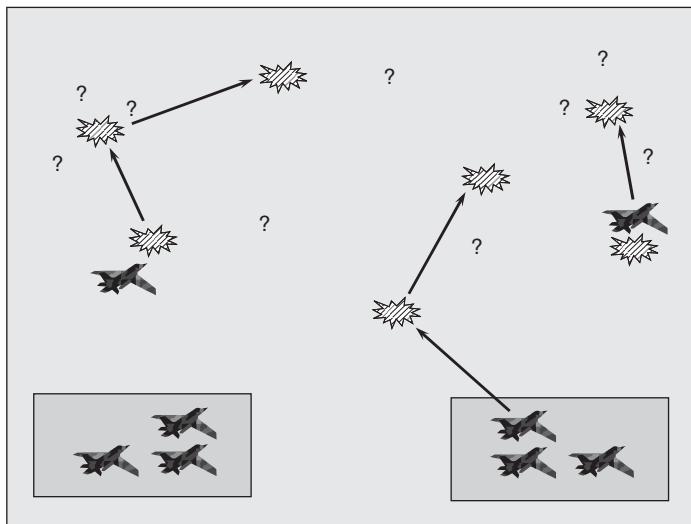


Figure 15.1 Dynamic routing of UAVs, showing known targets and potential targets.

planned through a series of known targets. In addition there is a series of potential targets (shown as question marks) that might arise.

One way to model this problem is to assume that we are going to optimally route a fleet of UAVs through a sequence of known targets. This problem can be formulated as an integer program and solved using commercial optimization packages as long as the problem is not too large. The biggest complication arises if the tours (the number of targets that a single UAV might cover in a single solution) become long. For example, it is possible to handle problems with thousands of UAVs if they are restricted to covering at most one target at a time. However, the problem becomes much more complex with tours of as few as three or four targets. If we have tours of five or ten targets, we might have to turn to heuristic algorithms that give good, but not necessarily optimal, solutions. These heuristics typically involve starting with one solution, and then testing alternative solutions to try to discover improvements. The simplest are local search algorithms (try making one or more changes; if the solution improves, keep the new solution and proceed from there). These algorithms fall under names such as local search, neighborhood search, and tabu search.

Suppose that we have devised a strategy for solving the problem myopically. It is quite likely that we will not be entirely satisfied with the behavior of the solution. For example, the system might assign two UAVs to serve two targets in an area that is not likely to produce additional targets in the future. Alternatively, the system might try to assign one UAV to cover two targets in an area that is likely to produce a number of additional targets in the future (too many to be covered by a single UAV). We might also want to consider attributes such as fuel level. We might prefer a UAV with a high fuel level if we expect other demands to arise. If

there is little chance of new demands, we might be satisfied assigning a UAV that is low on fuel with the expectation that it will finish the task and then return home.

It is very important, when designing an ADP strategy, to make sure that we have a clear idea of the behaviors we are trying to achieve that are not being produced with a myopic policy. With this in mind, the next step is to design a value function approximation. The choice of function depends on the type of algorithmic strategy we are using for the myopic problem. If we wish to solve the myopic problem using integer programming, then a nonlinear value function approximation is going to cause complications (now we have a nonlinear integer programming problem). If we use a local search procedure, then the value function approximation can take on almost any form.

Once we have designed a value function approximation, we have to make sure that we are going to be able to estimate it using information available from the solution procedure. For example, in Chapter 13 we showed how some value function approximations can be approximated using gradient information, which means that we need to be able to estimate the marginal value of, say, a UAV in a particular region. If we constrain ourselves to assigning a UAV to at most one target at a time, our myopic problem is the same as the dynamic assignment problem we considered in Section 14.6, where it is quite easy to obtain marginal values from dual variables. Local search heuristics, however, do not yield gradient information very easily. As an alternative, we might approximate the value of a UAV as the contribution earned over the tour to which it has been assigned. This can be thought of as the average value of a UAV instead of the marginal value. Not as useful, but perhaps better than nothing.

15.3 DEBUGGING AN ADP ALGORITHM

Imagine that you now have your algorithm up and running. There are several strategies for deciding whether it works well:

1. Plotting the objective function over the iterations.
2. Evaluating one or more performance statistics over the iterations.
3. Subjectively evaluating the behavior of your system after the algorithm has completed.

Ideally the objective function will show generally steady improvement (keep in mind the behavior in Figure 15.3). However, you may find that the objective function gets steadily worse, or wanders around with no apparent improvement. Alternatively, you may find that important performance measures that you are interested in are not improving as you hoped or expected. Explanations for this behavior can include:

1. If you are using a forward pass, you may be seeing evidence of a “bounce.” You may need to simply run more iterations.

2. You might have a problem where the value function is not actually improving the decisions. Your myopic policy may be doing the best that is possible. You might not have a problem that benefits from looking into the future.
3. Be careful that you are not using a stepsize that is going to zero too quickly.
4. Your value function may be a poor approximation of the true value function. You may not be properly capturing the structure of the problem.
5. You might have a reasonable value function, but the early iterations might have produced a poor estimate, and you are having difficulty recovering from this initial estimate.
6. You may not be updating the value function properly. Carefully verify that \hat{v}_t^n (whether it be the value of being in a state or a vector of derivatives) is being correctly calculated. This is a common error.

Diagnosing problems with an algorithm is often easiest when you can see specific decisions that you simply feel are wrong. If you are solving your decision problems properly (this is not always the case), then presumably the solution is due to an incorrect value function approximation. One problem is that $\bar{V}_t(S_t^x)$ may reflect a number of updates over many iterations. For this reason it is best to look at \hat{v}_t^n . If it takes the algorithm to a poor state, is \hat{v}_t^n small? If not, why do you think it is a good state? If it is clearly a bad state, why does it appear that \hat{v}_t^n is large? If \hat{v}_t^n is reasonable, perhaps the problem is in how the value function approximation is being updated (or the structure of the value function itself).

15.4 PRACTICAL ISSUES

Not surprisingly, there are a number of practical issues that arise when designing and testing approximate dynamic programming algorithms. This section provides a brief discussion of some of these.

15.4.1 Discount Factors

One way to improve the performance of the algorithm is to reduce the discount factor. Problems with discount factors close to 1 are simply much harder than a problem with a discount factor that is smaller. For this reason an effective strategy is to simply start by solving a problem with a smaller discount factor (e.g., $\gamma \leq 0.7$). If you feel that you are getting an effective solution with a smaller discount factor, then you may use the solution as an initial solution for the algorithm at a larger discount factor.

A similar issue arises with undiscounted problems with long horizons. You might either start with a relatively small horizon (e.g., 5 time periods) or introduce a discount factor that you can raise toward 1. Use the solution of the simpler problem as an initial solution to harder problems (longer horizons, higher discount factors).

15.4.2 Starting and Stopping

Two challenges we face in approximate dynamic programming is getting started, which means dealing with very poor initial approximations of the value function, and figuring out when we have converged. Solving these problems tends to be unique to each problem, but it is important to recognize that they need to be addressed.

15.4.3 Getting through the Early Iterations

One issue that always arises in approximate dynamic programming is that you have to get through the early iterations when the value function approximation is very inaccurate. While it may be possible to start with an initial value function approximation that is fairly good, it is often the case that we have no idea what the value function should be and we simply use zero (or some comparable default approximation). After a few iterations we might have updated the approximation with a few observations (allowing us to update the parameter vector θ), but the approximation may be quite poor.

As a rule, it is better to use a relatively simple approximation in the early iterations. The problem is that as the algorithm progresses, we may stall out at a poor solution. We would like to design a value function approximation that allows us to produce an accurate approximation, but this may require estimating a large number of parameters.

In Section 8.1 we introduced the idea of estimating the value of being in a state at different levels of aggregation. Rather than use any single level of aggregation, we showed that we could estimate the value of being in a state by using a weighted sum of estimates at different levels of aggregation. Figure 15.2 shows what happens when we use a purely aggregate estimate, a purely disaggregate estimate, and a weighted combination. The purely aggregate estimate produces much faster initial convergence, reflecting the fact that an aggregate estimate of the value function is much easier to obtain with a few observations. The problem is that this estimate does not work as well in the long run.

Using a purely disaggregate estimate produces slow initial convergence, but ultimately provides a much better objective function in the later iterations. However, using a weighted combination creates faster initial convergence and the best results over all. The lesson of this demonstration is that it can be better to use a simpler approximation in the early iteration, and transition to a more refined approximation as the algorithm progresses.

This idea can be applied in a variety of settings. Consider the problem of modeling the level of cash in a mutual fund that depends on the amount of cash we are currently holding R_t , the current market performance f_{t1} , and current interest rates f_{t2} . We can immediately produce value function approximations at three levels of aggregation: (1) $\bar{V}(R_t|\theta)$, (2) $\bar{V}(R_t|\theta(f_{t1}))$, and (3) $\bar{V}(R_t|\theta(f_{t1}, f_{t2}))$. The first value function approximation completely ignores the two financial statistics. The second computes a parameter vector θ for each value of f_{t1} . The third computes a parameter vector θ for each combination of f_{t1} and f_{t2} . Depending on how f_{t1}

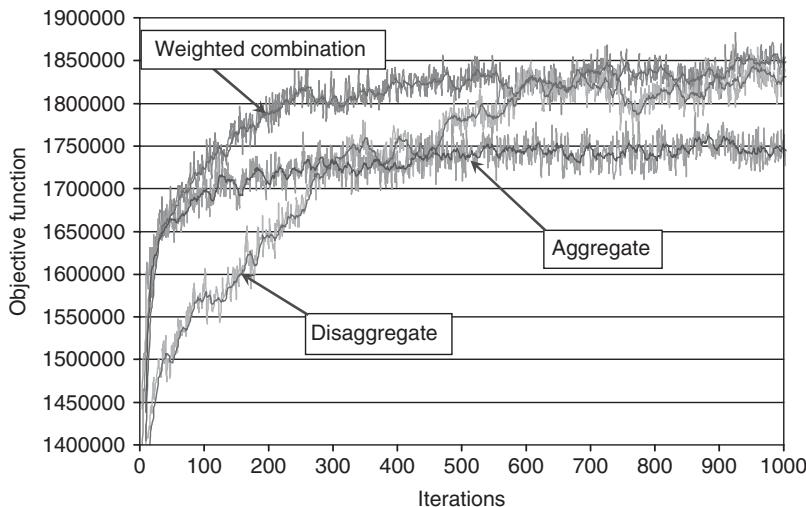


Figure 15.2 Objective function produced with a purely aggregate value function approximation, a purely disaggregate one, and one using a weighted combination of aggregate and disaggregate approximations.

and f_{t2} are discretized, we could easily be computing thousands of estimates of θ . As a result the quality of the estimate of $\theta(f_{t1}, f_{t2})$ for a particular combination of f_{t1} and f_{t2} could be very inaccurate due to statistical error. We handle this by using a weighted sum of approximations, such as

$$\bar{V}(S_t|\theta) = w_1 \bar{V}(R_t|\theta) + w_2 \bar{V}(R_t|\theta(f_{t1})) + w_3 \bar{V}(R_t|\theta(f_{t1}, f_{t2})).$$

The weights can be computed using the techniques described in Section 8.1.4. Computing the weights in this way produces a natural transition from simpler to more complex models with nominal computational effort.

It is easy to overlook the importance of using simpler functional approximations in the early iterations. A more complex function can be hard to estimate, producing large statistical errors in the beginning. It can be much more effective to do a better job (statistically speaking) of estimating a simpler function than to do a poor job of trying to estimate a more sophisticated function.

15.4.4 Convergence Issues

A significant challenge with approximate dynamic programming is determining when to stop. In Section 11.6 we encountered the problem of “apparent convergence” when the objective function appeared to have leveled off after 100 iterations, suggesting that we could stop the algorithm. In fact the algorithm made considerably more progress when we allowed it to go 1000 iterations. This is one reason that we have to be very careful not to allow the stepsizes to decline too quickly. Also before we decide that “100 iterations is enough,” we should do a few runs with substantially more iterations.

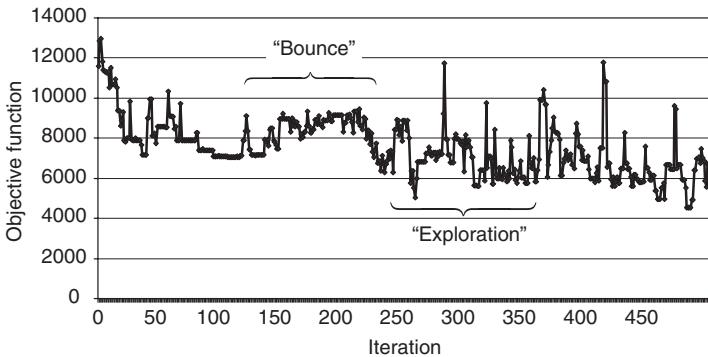


Figure 15.3 Objective function of an approximate dynamic programming algorithm.

Apparent convergence, however, is not our only headache. Figure 15.3 shows an example of the objective function (total costs, which we are minimizing) for an approximate value iteration algorithm. This graph shows a behavior that is fairly common when using a single-pass algorithm, which we refer to as the “bounce.” When using a single-pass algorithm, the value function for time t , iteration n reflects activities from time $t + s$ at iteration $n - s$, since values are passed backward one time period for each iteration. The result can be value function approximations that are simply incorrect, producing solutions that steadily get worse for a period of time.

Figure 15.3 also illustrates some other behaviors that are not uncommon. After the bounce, the algorithm goes through a period of exploration where little progress is made. While this does not always happen, it is not uncommon for algorithms to go through periods of trying different options and slowly learning the effectiveness of these options. This behavior can be particularly pronounced for problems where there is little structure to guide the solution.

Finally, the algorithm starts to settle down in the final 100 iterations, but even here there are short bursts where the solution becomes much worse. At this stage we might be comfortable concluding that the algorithm has probably progressed as far as it can, but we have to be careful not to stop on one of the poor solutions. This can be avoided in several ways. The most expensive option is to stop every K iterations and perform repeated samples. This strategy allows us to make strong statistical statements about the quality of the solution, but this can be expensive. Another approach is to simply smooth the objective function using exponential smoothing, and simultaneously compute an estimate of the standard deviation of the solution. Then simple rules can be designed to stop the algorithm when the solution is, for example, one standard deviation below the estimated mean (if we want to hope for a good solution).

15.4.5 Evaluating a Policy

If you are implementing an ADP algorithm, you might start asking for some way to evaluate the accuracy of your procedure. This question can be posed in two broad

ways: (1) How good is your policy? (2) How good is your value function approximation? Section 4.9.4 addressed the issue of evaluating a policy. In this section we turn to the question of evaluating the quality of the value function approximation. There are applications such as pricing an asset, estimating the probability of winning a game, or reaching a goal where the primary interest is in the value function itself.

Let $\bar{V}(S)$ be an approximation of the value of being in state S . The approximation could have a lookup table format, or it might be a regression of the form

$$\bar{V}(S) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(S).$$

Now imagine that we have access to the true value of being in state S , which we denote by $V(s)$. Typically this would happen when we have a problem that we can solve exactly using the techniques of Chapter 3, and we are now testing an ADP algorithm to evaluate the effectiveness of a particular approximation strategy. If we want an aggregate measure, we could use

$$v^{\text{avg}} = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} |V(s) - \bar{V}(s)|.$$

v^{avg} expresses the error as an average over all the states. The problem is that it puts equal weight on all the states, regardless of how often they are visited. Such an error is unlikely to be very interesting.

This measure highlights the problem of determining how to weight the states. Suppose that while determining $V(s)$, we also find $p^*(s)$, which is the steady-state value of being in state s (under an optimal policy). If we have access to this measure, we might use

$$v^{\pi^*} = \sum_{s \in \mathcal{S}} p^*(s) |V(s) - \bar{V}(s)|,$$

where π^* represents the fact that we are following an optimal policy. The advantage of this error measure is that $p^*(s)$ is independent of the approximation strategy used to determine $\bar{V}(s)$. If we want to compare different approximation strategies, where each produces different estimates for $\bar{V}(s)$, then we get a more stable estimate of the quality of the value function.

Such an error measure is not as perfect as it seems. Suppose that we have two approximation strategies that produce the approximations $\bar{V}^{(1)}(s)$ and $\bar{V}^{(2)}(s)$. Also let $\bar{p}^{(1)}(s)$ and $\bar{p}^{(2)}(s)$ be, respectively, estimates of the steady-state probabilities of being in state s produced by each policy. If a particular approximation has us visiting certain states more than others, then it is easy to argue that the value functions produced by this strategy should be evaluated primarily based on the frequency with which we visit states under that policy. For example, if we wish to obtain an estimate of the value of the dynamic program (using the first approximation), we

would use

$$\bar{V}^{(1)} = \sum_{s \in \mathcal{S}} p^{(1)}(s) \bar{V}^{(1)}(s).$$

For this reason it is fair to measure the error based on the frequency with which we actually visit states under the approximation, leading to

$$\nu^{(1)} = \sum_{s \in \mathcal{S}} p^{(1)} |V(s) - \bar{V}^{(1)}(s)|.$$

If we compute $\bar{V}^{(2)}$ and $\nu^{(2)}$, then provided that $\nu^{(1)} < \nu^{(2)}$, we could conclude that $\bar{V}^{(1)}$ is a better approximation than $\bar{V}^{(2)}$. But it would have to be understood that this result means that $\bar{V}^{(1)}$ does a better job of estimating the value of states that are visited under policy 1 than the job that $\bar{V}^{(2)}$ does in estimating the value of states that are visited under policy 2.

15.5 MODELING YOUR PROBLEM

There is a long history in the optimization community to first present a complete optimization model of a problem, after which different algorithms may be discussed. A mathematical programming problem is not considered properly stated unless we have defined all the variables, constraints and the objective function. The same cannot be said of the approximate dynamic programming community, where a more casual presentation style has evolved. It is not unusual for an author to describe the size of the state space without actually defining a state variable. Transition functions are often ignored, and there appear to be many even in the research community who are unable to write down an objective function properly.

The culture of approximate dynamic programming is closer to that of simulation where it is quite common to see papers without any notation at all. Such papers are often describing fairly complex problems of physical processes that are well understood by those working in the same area of application. Mathematical models of such processes can be quite cumbersome, and have not proved necessary for people who are familiar with the problem class. There are many papers in approximate dynamic programming which have evolved out of the simulation community, and have apparently adopted this minimalist culture for modeling.

ADP should be viewed as a potentially powerful tool for the simulation community. However, in a critical way it should be viewed quite differently. Most simulation papers are trying to mimic a physical process rather than optimize it. The most important part of simulation models are captured by the transition function, the details of which may not be particularly interesting. But when we use approximate dynamic programming, we are often trying to optimize (over time) complex problems that might otherwise be solved using simulation.

To properly describe a dynamic program (in preparation for using approximate dynamic programming), we need to specify the value function approximation and

the updating strategies used to estimate it. We need to evaluate the quality of a solution by comparing objective functions. These steps require a proper description of the state variable (by far the most important quantity in a dynamic program), the decision variables, exogenous information and the contribution function. But as with many simulation problems, it is not always clear that we need the transition function in painstaking detail. If we formulate a math programming problem, the transition function would be buried in the constraints. When modeling a dynamic system, it is well known that the transition function captures the potentially complex physics of a problem. For simple problems, there is no reason not to completely specify these equations, especially if we need to take advantage of the structure of the transition function. For more complex problems, modeling the transition function can be cumbersome, and the physics tends to be well known by people working with an application class.

If you wish to develop an approximate dynamic programming algorithm, it is important to learn to express your problem mathematically. There is a strong culture in the deterministic optimization community to write out the problem in its entirety using linear algebra. By contrast, there is a vast array of simulation problems that are expressed without any mathematics at all. Approximate dynamic programming sits between these two fields. It is simply inappropriate to solve a problem using ADP without expressing the problem mathematically. At the same time ADP allows us to address problems with far more complexity than are traditionally handled using tools such as linear programming. For this reason we suggest the following guidelines when modeling a problem:

- *State variable.* Clearly identify the dimensions of the state variable using explicit notation. This is the mechanism by which we can understand the complexity and structure of your problem.
- *Decision variable.* We need to know what dimensions of the problem we are controlling. It is often convenient to describe constraints on decisions that apply at a point in time (rather than over time).
- *Exogenous information process.* What information is arriving to our system from exogenous sources? What do we know (if anything) about the probability law describing what information might arrive?
- *Transition function.* If the problem is relatively simple, this should be stated mathematically in its entirety. But as the problem becomes more complicated, a more descriptive presentation may be appropriate. For many problems the details of the transition function are not central to the development of an approximate dynamic programming algorithm. Also, as problems become more complex, a complete mathematical model of the transition function can become tedious. Obviously the right level of detail depends on the context and what we are trying to accomplish.
- *Contribution function.* Also known as the cost function (minimizing), the reward function (maximizing), or the utility function (usually maximizing). We need to understand what goal we are trying to achieve.

- *Objective function.* This is where you specify how you are going to evaluate contributions over time (finite or infinite horizon? Discounted or average reward?).

Thus we feel that every component should be stated explicitly using mathematical notation with the single major exception of the transition function (which captures all the physics of the problem). Some problems are simply too complex, and for these problems the structure of the transition function may not be relevant to the design of the solution algorithm. A transition function might be a single simple equation, but it can also be thousands of lines of code. The details of the transition function are generally not that interesting if the focus is on the development of an ADP algorithm.

When writing up your model, pay particular attention to the modeling of time. Keep in mind that dynamic programs are solved at discrete points in time that we call decision epochs. Variables indexed at time t are being measured at time t , which determines the information that is available when we make a decision. It does *not* imply that physical activities are also taking place at these points in time. We can decide at time 100 to assign a technician to perform a task at time 117. It may be that we will not make the next decision until time 200 (or time 143, when the next phone call comes in). If you are modeling a deterministic problem, these issues are less critical. But for stochastic problems the modeling of information (what we know when we make a decision) needs to be separated from the modeling of physical processes (when something happens).

15.6 ONLINE VERSUS OFFLINE MODELS

When testing approximate dynamic programming algorithms, it is common to perform the testing in an “offline” setting. In this mode (which describes virtually every algorithm in this book) we start at time $t = 0$ and step forward in time, using information realizations (which we have denoted $W(\omega)$) that are either generated randomly, drawn from a file of sample paths (e.g., prices of stocks) or sampled from a known probability distribution.

In an online setting the model is running in real time alongside an actual physical process. The process might be a stock price, an inventory at a store, or a problem in engineering such as flying an aircraft or controlling the temperature of a chemical reaction. In such settings we may observe the exogenous information process (e.g., a stock price) or we may even depend on physical observations to observe state transitions. When we observe exogenous information, we do not need to estimate probability distributions for the random variables (e.g., the change in the stock price). These are sometimes referred to as distribution-free models. If we make a decision and then observe the state transition, then we do not need a mathematical transition function. This is referred to as model-free (or sometimes “direct”) approximate dynamic programming.

There are two important cases of online applications: steady state and transient. In a steady-state situation we do not have to forecast future information, we instead

need only to use live information to update the process. We can use live information to drive our ADP algorithm. For example, consider a steady-state version of the asset acquisition problem we solved in Section 14.1. Assume that our asset is a type of DVD player (a popular theft item). If we have R_t DVDs in inventory, we use some rule to decide to order an amount x_t . Suppose now that we observe that we sold \hat{D}_t units, allowing us to compute a profit

$$C_t(x_t) = p\hat{D}_t - cx_t,$$

where p is the price at which we sell the units, and c is the wholesale cost of new units purchased at time t . Note that \hat{D}_t is sales (rather than actual demand). \hat{D}_t is limited by the measured inventory R_t . We let $R_t^x = R_t - \hat{D}_t + x_t$ be our post-decision state, which is how much we think we have left over (but excluding loss due to theft). After this we measure the inventory R_{t+1} that differs from R_t^x because of potential theft.

We can use the methods in Chapter 13 to help estimate an approximate value function $\bar{V}(R_t^x)$. We would then make decisions using

$$x_t = \arg \max(C_t(x_t) + \gamma \bar{V}(R_t^x)).$$

Now assume that we have a nonstationary problem, where demands might be steadily increasing or decreasing, or might follow some sort of cyclic pattern (e.g., hourly, weekly or monthly cycles, depending on the application). We might use a time-series model with the form

$$F_{tt'} = \theta_{t0} + \sum_{c \in \mathcal{C}} \theta_{tc}^{cal} X_{t'c} + \sum_{\tau=1}^{\tau^M} \theta_{t\tau}^{time} p_{t-\tau}$$

where \mathcal{C} is a set of calendar effects (e.g., the seven days of the week), $X_{t'c}$ is an indicator variable (e.g., equal to 1 if t' falls on a Monday), θ^{cal} is the calendar parameters, $p_{t-\tau}$ is the price τ time periods ago, and $\theta_{t\tau}^{time}$ is the impact of the price τ time periods ago on the model. The precise specification of our forecast model is not important right now, but we observe that it is a function of a series of past prices, so we can define a price state variable $S^p = (p_{t-1}, \dots, p_{t-\tau^M})$ and a series of parameters $\theta_t = (\theta_{t0}, (\theta_{tc}^{cal})_{c \in \mathcal{C}}, (\theta_{t\tau}^{time})_\tau)$, which are updated after each time period. Suppose that we have devised a system for updating the parameter vector θ_t after we observe the demands \hat{D}_t .

In the presence of transient demands, to determine the decision x_t , we have to set up and solve a dynamic program over the time periods $t, t+1, \dots, t+\tau^{ph}$, where τ^{ph} is a specified planning horizon. This means that we have to use our forecast model $F_{tt'}$ over $t' = t, t+1, \dots, t+\tau^{ph}$. To run an ADP algorithm, we have to assume a distribution (e.g., normal with mean $F_{tt'}$ and variance $\bar{\sigma}_{t,t'-1}^2$) to generate random observations of demands. This means that we are solving an ADP model on a rolling horizon basis, reoptimizing over a horizon with each step forward in time.

15.7 IF IT WORKS, PATENT IT!

A successful ADP algorithm for an important problem class is, we believe, a patentable invention. While we continue to search for the holy grail of a general class of algorithms that work reliably on all problems, we suspect the future will involve fairly specific problem classes with a well-defined structure. For such problems we can, with some confidence, perform a set of experiments that can determine whether an ADP algorithm solves the problems that are likely to come up in that class. Of course, we encourage developers to keep these results in the public domain, but our point is that these represent specific technological breakthroughs that could form the basis of a patent application.

Bibliography

- Andreatta, G., and Romeo, L. (1988), “Stochastic shortest paths with recourse,” *Networks* **18**, 193–204.
- Ankenman, B., Nelson, B. L., and Staum, J. (2009), “Stochastic Kriging for simulation metamodeling,” *Operations Research* **58** (2), 371–382.
- Antos, A., Munos, R., and Szepesvari, C. (2008a), “Fitted Q-iteration in continuous action-space MDPs,” *Advances in Neural Information Processing Systems* **20**, 9–16.
- Antos, A., Szepesvari, C., and Munos, R. (2007), “Value-iteration based fitted policy iteration: Learning with a single trajectory,” *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pp. 330–337.
- Antos, A., Szepesvári, C., and Munos, R. (2008b), “Learning near-optimal policies with Bellman-residual minimization based fitted policy iteration and a single sample path,” *Machine Learning* **71** (1), 89–129.
- Auer, P., Cesa-bianchi, N., and Fischer, P. (2002), “Finite time analysis of the multiarmed bandit problem,” *Machine Learning* **47**(2–3), 235–256.
- Baird, L. C. (1995), “Residual algorithms: Reinforcement learning with function approximation,” *Proceedings of the Twelfth International Conference on Machine Learning* pp. 30–37.
- Banks, J., Nelson, B. L., and J. S. Carson, I. I. (1996), *Discrete-Event System Simulation*, Prentice-Hall, Englewood Cliffs, NJ.
- Barto, A. G., and Sutton, R. S. (1981), “Landmark learning: An illustration of associative search,” *Biological Cybernetics* **8**, 1–8.
- Barto, A. G., Bradtke, S. J., and Singh, S. (1995), “Learning to act using real-time dynamic programming,” *Artificial Intelligence* **72** (1–2), 81–138.
- Barto, A. G., Sutton, R. S., and Anderson, C. W. (1983), “Neuron-like elements that can solve difficult learning control problems,” *IEEE Transactions on Systems, Man and Cybernetics* **13**, 834–846.
- Barto, A. G., Sutton, R. S., and Brouwer, P. (1981), “Associative search network: A reinforcement learning associative memory,” *Biological Cybernetics* **40** (3), 201–211.

Approximate Dynamic Programming: Solving the Curses of Dimensionality, Second Edition.

Warren B. Powell.

© 2011 John Wiley & Sons, Inc. Published 2011 by John Wiley & Sons, Inc.

- Basler, J. T. (2006), "Optimal portfolio rebalancing: An approximate dynamic programming approach," Technical report, Department of Operations Research and Financial Engineering, Princeton University.
- Bean, J. C., Birge, J. R., and Smith, R. L. (1987), "Aggregation in dynamic programming," *Operations Research* **35**, 215–220.
- Bechhofer, R. E., Santner, T. J., and Goldsman, D. M. (1995), *Design and Analysis of Experiments for Statistical Selection, Screening, and Multiple Comparisons*, Wiley, New York.
- Bellman, R. (1971), *Introduction to the Mathematical Theory of Control Processes*, Vol. II, Academic Press, New York.
- Bellman, R., and Kalaba, R. (1959), "On adaptive control processes," *IRE Transactions on Automatic Control* **4** (2), 1–9.
- Bellman, R. E. (1957), *Dynamic Programming*, Princeton University Press, Princeton, NJ.
- Bellman, R. E., and Dreyfus, S. E. (1959), "Functional approximations and dynamic programming," *Mathematical Tables and Other Aids to Computation* **13**, 247–251.
- Benveniste, A., Metivier, M., and Priouret, P. (1990), *Adaptive Algorithms and Stochastic Approximations*, Springer, New York.
- Berry, D. A., and 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. (2005), *Dynamic Programming and Stochastic Control*, Vol. I, Athena Scientific, Belmont, MA.
- Bertsekas, D. P. (2009), *Approximate Dynamic Programming*, Vol. II, 3rd ed., Athena Scientific, Belmont, MA.
- Bertsekas, D. P., and Castanon, D. (1989), "Adaptive aggregation methods for infinite horizon dynamic programming," *IEEE Transactions on Automatic Control* **34**, 589–598.
- Bertsekas, D. P., and Castanon, D. A. (1999), "Rollout algorithms for stochastic scheduling problems," *Journal of Heuristics* **5**, 89–108.
- Bertsekas, D. P., and Nedić, A. (2003), "Least-squares policy evaluation algorithms with linear function approximation," *Journal of Discrete Event Systems* **13**, 79–110.
- Bertsekas, D. P., and Tsitsiklis, J. N. (1989), *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall, Englewood Cliffs, NJ.
- Bertsekas, D. P., and Tsitsiklis, J. N. (1996), *Neuro-dynamic Programming*, Athena Scientific, Belmont, MA.
- Bertsekas, D. P., Borkar, V. S., and Nedic, A. (2004), "Improved temporal difference methods with linear function approximation," in J. Si, A. G. Barto, W. B. Powell, and D. Wunsch, eds., *Handbook of Learning and Approximate Dynamic Programming*, IEEE Press, New York, pp. 233–257.
- Bertsekas, D. P., Tsitsiklis, J. N., and An (1991), "Analysis of stochastic shortest path problems," *Mathematics of Operations Research* **16** (3), 580–595.
- Bertsekas, D. P., Van Roy, B., Lee, Y., and Tsitsiklis, J. N. (1997), "A neuro-dynamic programming approach to retailer inventory management," *Proceedings of the IEEE Conference on Decision and Control*, Vol. 4, pp. 4052–4057.
- Bhatnagar, S., Sutton, R. S., Ghavamzadeh, M., and Lee, M. (2009), "Natural actor–critic algorithms," *Automatica* **45** (11), 2471–2482.

- Birge, J. R. (1985), "Decomposition and partitioning techniques for multistage stochastic linear programs," *Operations Research* **33**, 989–1007.
- Birge, J. R., and Louveaux, F. (1997), *Introduction to Stochastic Programming*, Springer, New York.
- Bishop, C. M. (2006), *Pattern Recognition and Machine Learning*, Springer, New York.
- Blum, J. (1954a), "Multidimensional stochastic approximation methods," *Annals of Mathematical Statistics* **25**, 737–744.
- Blum, J. R. (1954b), "Approximation methods which converge with probability one," *Annals of Mathematical Statistics* **25**, 382–386.
- Borkar, V., and Konda, V. (1997), "The actor–critic algorithm as multi-time-scale stochastic approximation," *Sadhana* **22** (4), 525–543.
- Boutilier, C., Dean, T., and Hanks, S. (1999), "Decision-theoretic planning: Structural assumptions and computational leverage," *Access* **11** (1), 1–94.
- Bradtko, S. J., and Barto, A. G. (1996), "Linear least-squares algorithms for temporal difference learning," *Machine Learning* **22** (1), 33–57.
- Bradtko, S. J., Barto, A. G., and Ydstie, B. (1994), "Adaptive linear quadratic control using policy iteration," *American Control Conference* **3**, 3475–3479.
- Brezzi, M., and Lai, T. (2002), "Optimal learning and experimentation in bandit problems," *Journal of Economic Dynamics and Control* **27** (1), 87–108.
- Brown, R. G. (1959), *Statistical Forecasting for Inventory Control*, McGraw-Hill, New York.
- Brown, R. G. (1963), *Smoothing, Forecasting and Prediction of Discrete Time Series*, Prentice-Hall, Englewood Cliffs, NJ.
- Busoniu, L., Babuska, R., De Schutter, B., and Ernst, D. (2010), *Reinforcement Learning and Dynamic Programming Using Function Approximators*, CRC Press, New York.
- Camacho, E. F., and Bordons, C. (2004), *Model Predictive Control*, Springer, New York.
- Cant, L. (2006), "Life saving decisions: A model for optimal blood inventory management." Senior thesis, Department of Operations Research and Financial Engineering, Princeton University.
- Cao, X.-R. (2007), *Stochastic Learning and Optimization*, Springer, New York.
- Chang, H. S., Fu, M. C., Hu, J., and Marcus, S. I. (2007), *Simulation-Based Algorithms for Markov Decision Processes*, Springer, New York.
- Chen, C. H. (1995), "An effective approach to smartly allocate computing budget for discrete event simulation," *34th IEEE Conference on Decision and Control*, Vol. 34, New Orleans, LA, pp. 2598–2603.
- Chen, H. C., Chen, C. H., and Yücesan, E. (2000), "Computing efforts allocation for ordinal optimization and discrete event simulation," *IEEE Transactions on Automatic Control* **45** (5), 960–964.
- Chen, H. C., Chen, C. H., Dai, L., and Yücesan, E. (1997), "New development of optimal computing budget allocation for discrete event simulation," *Proceedings of the Winter Simulation Conference*, IEEE, Piscataway, NJ, pp. 334–341.
- Chick, S. E., and Gans, N. (2009), "Economic analysis of simulation selection problems," *Management Science* **55** (3), 421–437.
- Chick, S. E., and Inoue, K. (2001), "New two-stage and sequential procedures for selecting the best simulated system," *Operations Research* **49** (5), 732–743.

- Chick, S. E., Chen, C. H., Lin, J., and Yücesan, E. (2000), "Simulation budget allocation for further enhancing the efficiency of ordinal optimization," *Discrete Event Dynamic Systems* **10**, 251–270.
- Choi, D. P., and Van Roy, B. (2006), "A generalized kalman filter for fixed point approximation and efficient temporal-difference learning," *Discrete Event Dynamic Systems* **16**, 207–239.
- Chong, E. K. P. (1991), "On-line stochastic optimization of queueing systems," PhD thesis, Department of Electrical Engineering, Princeton University.
- Chow, G. (1997), *Dynamic Economics*, Oxford University Press, New York.
- Chung, K. L. (1974), *A Course in Probability Theory*, Academic Press, New York.
- Clement, E., Lamberton, D., and Protter, P. (2002), "An analysis of a least squares regression method for american option pricing," *Finance and Stochastics* **17**, 448–471.
- Dantzig, G., and Ferguson, A. (1956), "The allocation of aircrafts to routes: An example of linear programming under uncertain demand," *Management Science* **3**, 45–73.
- Darken, C., and Moody, J. (1991), "Note on learning rate schedules for stochastic optimization," in R. P. Lippmann, J. Moody, and D. S. Touretzky, eds., *Advances in Neural Information Processing Systems 3*, Morgan Kaufmann, San Mateo, CA, pp. 832–838.
- Darken, C., and Moody, J. (1992), "Towards faster stochastic gradient search," in J. Moody, D. L. Hanson, and R. P. Lippmann, eds., *Advances in Neural Information Processing Systems 4*, Morgan Kaufmann, San Mateo, CA, pp. 1009–1016.
- Darken, C., Chang, J., and Moody, J. (1992), "Learning rate schedules for faster stochastic gradient search," *Neural Networks for Signal Processing 2—Proceedings of the 1992 IEEE Workshop*, IEEE Press, New York, pp. 3–12.
- de Farias, D. P., and Van Roy, B. (2000), "On the existence of fixed points for approximate value iteration and temporal-difference learning," *Journal of Optimization Theory and Applications*, Springer, New York **105** (3), 589–608.
- de Farias, D. P., and Van Roy, B. (2003), "The linear programming approach to approximate dynamic programming," *Operations Research* **51**, 850–865.
- Dearden, R., Friedman, N., and Andre, D. (1999), "Model-based bayesian exploration," *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, Stockholm, pp. 150–159.
- Dearden, R., Friedman, N., and Russell, S. (1998), "Bayesian Q-learning," *Proceedings of the National Conference on Artificial Intelligence*, John Wiley and Sons, Hoboken, NJ, pp. 761–768.
- DeGroot, M. H. (1970), *Optimal Statistical Decisions*, Wiley, New York.
- Deisenroth, M. P., Peters, J., Rasmussen, C. E., and Conference, A. C. (2008), "Approximate dynamic programming with gaussian processes," *Control. American Control Conference*, IEEE Press, New York, pp. 4480–4485.
- Denardo, E. V. (1982), *Dynamic Programming*, Prentice-Hall, Englewood Cliffs, NJ.
- Derman, C. (1962), "On sequential decisions and markov chains," *Management Science* **9**, 16–24.
- Derman, C. (1970), *Finite State Markovian Decision Processes*, Academic Press, New York.
- Doob, J. L. (1953), *Stochastic Processes*, Wiley, New York.
- Douglas, S. C., and Mathews, V. J. (1995), "Stochastic gradient adaptive step size algorithms for adaptive filtering," *Proceedings of the International Conference on Digital Signal Processing*, Limassol, Cyprus **1**, 142–147.

- Dreyfus, S., and Law, A. M. (1977), *The Art and Theory of Dynamic Programming*, Academic Press, New York.
- Duff, M. (2002), “Optimal learning: Computational procedures for Bayes-adaptive Markov decision processes,” PhD thesis. Department of Computer Science, University of Massachusetts at Amherst.
- Duff, M. O., and Barto, A. G. (1997), “Local bandit approximation for optimal learning problems,” *Advances in Neural Information Processing Systems* **9**, 1019.
- Duff, M. O., and Barto, A. G. (2003), “Local bandit approximation for optimal learning problems,” Technical report, Department of Computer Science, University of Massachusetts, Amherst, MA.
- Dvoretzky, A. (1956), “On stochastic approximation,” in J. Neyman, ed., *Proceedings 3rd Berkeley Symposium on Mathematical Statistics and Probability*, University of California Press, Berkeley, pp. 39–55.
- Dynkin, E. B., and Yushkevich, A. A. (1979), *Controlled Markov Processes*, Springer-Verlag, New York.
- Engel, Y., Mannor, S., and Meir, R. (2005), “Reinforcement learning with Gaussian processes,” *Proceedings of the 22nd International Conference on Machine Learning*, ACM Press, New York, pp. 201–208.
- Ermoliev, Y. (1988), “Stochastic quasigradient methods,” in Y. Ermoliev and R. Wets, eds., *Numerical Techniques for Stochastic Optimization*, Springer, Berlin.
- Even-dar, E., and Mansour, Y. (2003), “Learning rates for Q-learning,” *Journal of Machine Learning Research* **5**, 1–25.
- Fan, J., and Gijbels, I. (1996), *Local Polynomial Modelling and Its Applications*, Chapman and Hall, London.
- Farias, D., and Roy, B. (2001), “On constraint sampling for the linear programming approach to approximate dynamic,” *Mathematics of Operations Research* **29** (3), 462–478.
- Farias, D. P. D., and Van Roy, B. (2003), “The linear programming approach to approximate dynamic programming,” *Operations Research* **51** (6), 850–865.
- Ford, L. R., and Fulkerson, D. R. (1962), *Flows in Networks*, Princeton University Press, Princeton, NJ.
- Frank, H. (1969), “Shortest paths in probabilistic graphs,” *Operations Research* **17**, 583–599.
- Frazier, P. I., Powell, W. B., and Dayanik, S. (2008), “A knowledge gradient policy for sequential information collection,” *SIAM Journal on Control and Optimization* **47** (5), 2410–2439.
- Frazier, P. I., Powell, W. B., and Dayanik, S. (2009), “The knowledge-gradient policy for correlated normal beliefs,” *INFORMS Journal on Computing* **21** (4), 599–613.
- Frieze, A., and Grimmett, G. (1985), “The shortest path problem for graphs with random arc lengths,” *Discrete Applied Mathematics* **10**, 57–77.
- Fu, M. C. (2002), “Optimization for simulation: Theory vs. practice,” *INFORMS Journal on Computing* **14** (3), 192–215.
- Fu, M. C. (2008), “What you should know about simulation and derivatives,” *Naval Research Logistics* **55** (8), 723–736.
- Fu, M. C., Hu, J.-Q., Chen, C.-H., and Xiong, X. (2007), “Simulation allocation for determining the best design in the presence of correlated sampling,” *INFORMS Journal on Computing* **19**, 101–111.

- Gaivoronski, A. (1988), "Stochastic quasigradient methods and their implementation," in Y. Ermoliev and R. Wets, eds., *Numerical Techniques for Stochastic Optimization*, Springer, Berlin.
- Gardner, E. S. (1983), "Automatic monitoring of forecast errors," *Journal of Forecasting* **2**, 1–21.
- George, A. P., and Powell, W. B. (2006), "Adaptive stepsizes for recursive estimation with applications in approximate dynamic programming," *Journal of Machine Learning* **65** (1), 167–198.
- George, A., Powell, W. B., and Kulkarni, S. (2008), "Value function approximation using multiple aggregation for multiattribute resource management," *Journal of Machine Learning Research* **9**, 2079–2111.
- Giffin, W. C. (1971), *Introduction to Operations Engineering*, Irwin, Homewood, IL.
- Gittins, J. (1979), "Bandit processes and dynamic allocation indices," *Journal of the Royal Statistical Society, Series B (Methodological)* **41** (2), 148–177.
- Gittins, J., and Jones, D. (1974), *A Dynamic Allocation Index for the Sequential Design of Experiments*, North Holland, Amsterdam, pp. 241–266.
- Gittins, J. C. (1981), "Multiserver scheduling of jobs with increasing completion times," *Journal of Applied Probability* **16**, 321–324.
- Gittins, J. C. (1989), *Multi-armed Bandit Allocation Indices*, Wiley, New York.
- Gladyshov, E. G. (1965), "On stochastic approximation," *Theory of Probability and Its Applications* **10**, 275–278.
- Glasserman, P. (1991), *Gradient Estimation via Perturbation Analysis*, Kluwer Academic, Boston.
- Golub, G. H., and Loan, C. F. V. (1996), *Matrix Computations*, John Hopkins University Press, Baltimore, MD.
- Goodwin, G. C., and Sin, K. S. (1984), *Adaptive Filtering and Control*, Prentice-Hall, Englewood Cliffs, NJ.
- Gordon, G. J. (1995), "Stable function approximation in dynamic programming," *Proceedings of the 12th International Conference on Machine Learning, San Francisco*, Morgan Kaufmann, San Mateo, CA, pp. 261–268.
- Gordon, G. J. (2001), "Reinforcement learning with function approximation converges to a region," *Advances in Neural Information Processing Systems* **13**, 1040–1046.
- Gosavi, A. (2003), *Simulation-Based Optimization*, Kluwer Academic, Norwell, MA.
- Guestrin, C., Koller, D., and Parr, R. (2003), "Efficient solution algorithms for factored MDPs," *Journal of Artificial Intelligence Research* **19**, 399–468.
- Gupta, S. S., and Miescke, K. J. (1996), "Bayesian look ahead one-stage sampling allocations for selection of the best population," *Journal of Statistical Planning and Inference* **54**, 229–244.
- Hastie, T., Tibshirani, R., and Friedman, J. (2009), *The Elements of Statistical Learning: Data Mining, Inference and Prediction*, Springer, New York.
- Haykin, S. (1999), *Neural Networks: A Comprehensive Foundation*, Prentice Hall, Englewood Cliffs, NJ.
- He, D., Chick, S. E., and Chen, C.-h. (2007), "Opportunity cost and OCBA selection procedures in ordinal optimization for a fixed number of alternative systems," *IEEE Transactions on Systems Man and Cybernetics Part C-Applications and Reviews* **37** (5), 951–961.

- Heuberger, P. S. C., Van den Hov, P. M. J., and Wahlberg, B., eds. (2005), *Modeling and Identification with Rational Orthogonal Basis Functions*, Springer, New York.
- Heyman, D. P., and Sobel, M. (1984), *Stochastic Models in Operations Research*, Vol. II: *Stochastic Optimization*, McGraw-Hill, New York.
- Higle, J., and Sen, S. (1991), "Stochastic decomposition: An algorithm for two-stage linear programs with recourse," *Mathematics of Operations Research* **16** (3), 650–669.
- Ho, Y.-C. (1992), *Discrete Event Dynamic Systems: Analyzing Complexity and Performance in the Modern World*, IEEE Press, New York.
- Holt, C. C., Modigliani, F., Muth, J., and Simon, H. (1960), *Planning, Production, Inventories and Work Force*, Prentice-Hall, Englewood Cliffs, NJ.
- Hong, J., and Nelson, B. L. (2006), "Discrete optimization via simulation using COMPASS," *Operations Research* **54** (1), 115–129.
- Hong, L., and Nelson, B. L. (2007), "A framework for locally convergent random-search algorithms for discrete optimization via Simulation," *ACM Transactions on Modeling and Computer Simulation* **17** (4), 1–22.
- Howard, R. A. (1960), *Dynamic Programming and Markov Process*, MIT Press, Cambridge.
- Infanger, G. (1994), *Planning under Uncertainty: Solving Large-Scale Stochastic Linear Programs*, Boyd and Fraser, New York.
- Jaakkola, T., Jordan, M. I., and Singh, S. (1994), "On the convergence of stochastic iterative dynamic programming algorithms," *Neural Computation* **6** (6), 1185–1201.
- Judd, K. L. (1998), *Numerical Methods in Economics*, MIT Press, Cambridge.
- Kaelbling, L. P. (1993), *Learning in Embedded Systems*, MIT Press, Cambridge.
- Kaelbling, L. P., Littman, M., and Moore, A. W. (1996), "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research* **4**, 237–285.
- Kall, P., and Mayer, J. (2005), *Stochastic Linear Programming: Models, Theory, and Computation*, Springer, New York.
- Kall, P., and Wallace, S. W. (1994), *Stochastic Programming*, Wiley, New York.
- Kesten, H. (1958), "Accelerated stochastic approximation," *Annals of Mathematical Statistics* **29**, 41–59.
- Kiefer, J., and Wolfowitz, J. (1952), "Stochastic estimation of the maximum of a regression function," *Annals of Mathematical Statistics* **23**, 462–466.
- Kim, S.-H., and Nelson, B. L. (2001), "A fully sequential procedure for indifference-zone selection in simulation," *ACM Transactions on Modeling Computer Simulations*, **11**, 251–273.
- Kim, S. H., and Nelson, B. L. (2006), *Selecting the Best System*, Elsevier, Amsterdam.
- Kirk, D. E. (1998), *Optimal Control Theory: An Introduction*, Dover, New York.
- Klabjan, D., and Adelman, D. (2007), "An infinite-dimensional linear programming algorithm for deterministic semi-Markov decision processes on Borel spaces," *Mathematics of Operations Research* **32**, 528–550.
- Kmenta, J. (1997), *Elements of Econometrics*, University of Michigan Press, Ann Arbor.
- Konda, V. R., and Borkar, V. S. (1999), "Actor–critic-type learning algorithms for Markov decision processes," *SIAM Journal on Control and Optimization* **38**, 94.
- Konda, V. R., and Tsitsiklis, J. N. (2003), "On actor–critic algorithms," *SIAM Journal of Control and Optimization* **42** (4), 1143–1166.

- Kushner, H. J., and Clark, S. (1978), *Stochastic Approximation Methods for Constrained and Unconstrained Systems*, Springer, New York.
- Kushner, H. J., and Yin, G. G. (1997), *Stochastic Approximation Algorithms and Applications*, Springer, New York.
- Kushner, H. J., and Yin, G. G. (2003), *Stochastic Approximation and Recursive Algorithms and Applications*, Springer, New York.
- Lagoudakis, M., and Parr, R. (2003), “Least-squares policy iteration,” *Journal of Machine Learning Research* **4**, 1149.
- Lagoudakis, M., Parr, R., and Littman, M. (2002), “Least-squares methods in reinforcement learning for control,” *Methods and Applications of Artificial Intelligence* 752.
- Lai, T. L. (1987), “Adaptive treatment allocation and the multi-armed bandit problem,” *Annals of Statistics* **15**, 1091–1114.
- Lai, T. L., and Robbins, H. (1985), “Asymptotically efficient adaptive allocation rules,” *Advances in Applied Mathematics* **6**, 4–22.
- Lambert, T., Smith, R., and Epelman, M. (2004), “Aggregation in stochastic dynamic programming,” Technical report 04-07, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor.
- Landelius, T., and Knutsson, H. (1996), “Greedy adaptive critics for LQR problems: Convergence proofs,” Technical report, Department of Electrical Engineering, Linkoping University, Linkoping, Sweden.
- Law, A. M., and Kelton, E. D. (2000), *Simulation Modeling and Analysis*, McGraw-Hill, New York.
- LeBlanc, M., and Tibshirani, R. (1996), “Combining estimates in regression and classification,” *Journal of the American Statistical Association* **91**, 1641–1650.
- Ljung, L., and Soderstrom, T. (1983), *Theory and Practice of Recursive Identification*, MIT Press, Cambridge.
- Löhndorf, N., and Minner, S. (2010), “Optimal day-ahead trading and storage of renewable energies: An approximate dynamic programming approach,” *Energy Systems* **1** (1), 1–17.
- Longstaff, F., and Schwartz, E. S. (2001), “Valuing American options by simulation: A simple least squares approach,” *Review of Financial Studies* **14**, 113–147.
- Ma, J., and Powell, W. B. (2010a), “Convergence analysis of kernel-based on-policy approximate policy iteration algorithms for Markov decision processes with continuous multidimensional states and actions,” Technical report, Princeton University.
- Ma, J., and Powell, W. B. (2010b), “Convergence analysis of on-policy LSPI for multi-dimensional continuous state and action space MDPs and extension with orthogonal polynomial approximation,” Technical report, Princeton University.
- Manne, A. S. (1960), “Linear programming and sequential decisions,” *Management Science* **6** (3), 259–267.
- Marbach, P., and Tsitsiklis, J. N. (2001), “Simulation-based optimization of Markov reward processes,” *IEEE Transactions on Automatic Control*, **46** (2), 191–209.
- Mayer, J. (1998), *Stochastic Linear Programming Algorithms: A Comparison Based on a Model Management System*, Springer, New York.
- McClain, J. O. (1974), “Dynamics of exponential smoothing with trend and seasonal terms,” *Management Science* **20**, 1300–1304.

- Melo, F. S., and Ribeiro, M. I. (2007), "Q-Learning with linear function approximation," *Proceedings of the 20th Annual Conference on Learning Theory*, Berlin, Springer, pp. 308–322.
- Menache, I., Mannor, S., and Shimkin, N. (2005), "Basis function adaptation in temporal difference reinforcement learning," *Annals of Operations Research* **134** (1), 215–238.
- Mendelsohn, R. (1982), "An iterative aggregation procedure for Markov decision processes," *Operations Research* **30**, 62–73.
- Meyn, S. P. (1997), "The policy iteration algorithm for average reward Markov decision processes with general state space," *IEEE Transactions on Automatic Control*, **42** (12), 1663–1680.
- Mirozahmedov, F., and Uryasev, S. P. (1983), "Adaptive stepsize regulation for stochastic optimization algorithm," *Zurnal vichisl. mat. i. mat. fiz.* **6** (23), 1314–1325.
- Mulvey, J. M., and Ruszcynski, A. (1995), "A new scenario decomposition method for large scale stochastic optimization," *Operations Research* **43**, 477–490.
- Munos, R., and Szepesvari, C. (2008), "Finite-time bounds for fitted value iteration," *Journal of Machine Learning Research* **1**, 815–857.
- Nascimento, J. M., and Powell, W. B. (2009), "An optimal approximate dynamic programming algorithm for the lagged asset acquisition problem," *Mathematics of Operations Research* **34** (1), 210–237.
- Nascimento, J. M., and Powell, W. B. (2010a), "An optimal approximate dynamic programming algorithm for the energy dispatch problem with grid-level storage," Technical report, Princeton University.
- Nascimento, J. M., and Powell, W. B. (2010b), "Dynamic programming models and algorithms for the mutual fund cash balance problem," *Management Science* **56** (5), 801–815.
- Negoescu, D. M., Frazier, P. I., and Powell, W. B. (2010), "The knowledge-gradient algorithm for sequencing experiments in drug discovery," *Informs Journal on Computing*, (in press), doi 10.1287.
- Nelson, B. L., Swann, J., Goldsman, D., and Song, W. (2001), "Simple procedures for selecting the best simulated system when the number of alternatives is large," *Operations Research* **49**, 950–963.
- Nemhauser, G. L. (1966), *Introduction to Dynamic Programming*, Wiley, New York.
- Neveu, J. (1975), *Discrete Parameter Martingales*, North Holland, Amsterdam.
- Ormoneit, D., and Glynn, P. W. (2002), "Kernel-based reinforcement learning average-cost problems," *IEEE Transactions on Automatic Control*, 1624–1636.
- Ormoneit, D., and Sen, S. (2002), "Kernel-based reinforcement learning," *Machine Learning* **49**, 161–178.
- Papavassiliou, V. A., and Russell, S. (1999), "Convergence of reinforcement learning with general function approximators," *International Joint Conference on Artificial Intelligence*, pp. 748–757.
- Pearl, J. (1984), *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Upper Saddle River, NJ.
- Pereira, M. V. F., and Pinto, L. M. V. G. (1991), "Multistage stochastic optimization applied to energy planning," *Mathematical Programming* **52**, 359–375.

- Pflug, G. C. (1988), "Stepsize rules, stopping times and their implementation in stochastic quasi-gradient algorithms," in *Numerical Techniques for Stochastic Optimization*, Springer-Verlag, New York, pp. 353–372.
- Pflug, G. C. (1996), *Optimization of Stochastic Models: The Interface Between Simulation and Optimization*, Kluwer Academic, Boston.
- Pollard, D. (2002), *A User's Guide to Measure Theoretic Probability*, Cambridge University Press, Cambridge, UK.
- Porteus, E. L. (1990), *Handbooks in Operations Research and Management Science: Stochastic Models*, Vol. 2, North Holland, Amsterdam.
- Poupart, P., Vlassis, N., Hoey, J., and Regan, K. (2006), "An analytic solution to discrete Bayesian reinforcement learning," *Proceedings of the 23rd International Conference on Machine Learning*, ACM Press, New York, pp. 697–704.
- Powell, W. B., and Chen, Z.-L. (1999), "A convergent cutting-plane and partial-sampling algorithm for multistage linear programs with recourse," *Journal of Optimization Theory and Applications* **103**, 497–524.
- Powell, W. B., and Cheung, R. K.-M. (2000), "SHAPE: A stochastic hybrid approximation procedure for two-stage stochastic programs," *Operations Research* **48**, 73–79.
- Powell, W. B., and Godfrey, G. (2002), "An adaptive dynamic programming algorithm for dynamic fleet management, I: Single period travel times," *Transportation Science* **36** (1), 21–39.
- Powell, W. B., and Godfrey, G. A. (2001), "An adaptive, distribution-free approximation for the newsvendor problem with censored demands, with applications to inventory and distribution problems," *Management Science* **47** (8), 1101–1112.
- Powell, W. B., and Topaloglu, H. (2003), "Stochastic programming in transportation and logistics," *Handbooks in Operations Research and Management Science: Stochastic Programming*. Elsevier, New York, pp. 555–636.
- Powell, W. B., and Topaloglu, H. (2006), "Dynamic-programming approximations for stochastic time-staged integer multicommodity-flow problems," *Informs Journal on Computing* **18** (1), 31.
- Powell, W. B., and Van Roy, B. (2004), "Approximate dynamic programming for high dimensional resource allocation problems," in J. Si, A. G. Barto, W. B. Powell, and D. W. II, eds., *Handbook of Learning and Approximate Dynamic Programming*, IEEE Press, New York.
- Powell, W. B., Shapiro, J., and Simao, H. P. (2001), "A representational paradigm for dynamic resource transformation problems," in C. Couillard, R. Fourer, and J. H. Owen, eds., *Annals of Operations Research on Modeling*, **104**, 231–279.
- Powell, W. B., George, A., Lamont, A., and Stewart, J. (2011), "SMART: A stochastic multiscale model for the analysis of energy resources, technology and policy." *Informs Journal on Computing*. Available on <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.157.8434&rep=rep1&type=pdf>.
- Powell, W. B., Ruszcynski, A., and Topaloglu, H. (2004), "Learning algorithms for separable approximations of discrete stochastic optimization problems," *Mathematics of Operations Research* **29** (4), 814–836.
- Powell, W. B., Shapiro, J. A., and Simao, H. P. (2002), "An adaptive dynamic programming algorithm for the heterogeneous resource allocation problem," *Transportation Science* **36** (2), 231–249.

- Precup, D., and Perkins, T. (2003), "A convergent form of approximate policy iteration," in S. Becker, S. Thrun, and K. Obermayer, eds., *Advances in Neural Information Processing Systems*, The MIT Press, Cambridge, MA, pp. 1595–1602.
- Precup, D., Sutton, R. S., and Dasgupta, S. (2001), "Off-policy temporal-difference learning with function approximation," *Proceedings of the 18th International Conference on Machine Learning*, ACM Press, New York, pp. 417–424.
- Psaraftis, H. N., and Tsitsiklis, J. N. (1993), "Dynamic shortest paths in acyclic networks with Markovian arc costs," *Operations Research* **41**, 91–101.
- Puterman, M. L. (2005), *Markov Decision Processes*, 2nd Ed., Wiley, Hoboken, NJ.
- Robbins, H., and Monroe, S. (1951), "A stochastic approximation method," *Annals of Mathematical Statistics* **22** (3), 400–407.
- Rockafellar, R. T., and Wets, R. J.-B. (1991), "Scenarios and policy aggregation in optimization under uncertainty," *Mathematics of Operations Research* **16** (1), 119–147.
- Rogers, D., Plante, R., Wong, R., and Evans, J. (1991), "Aggregation and disaggregation techniques and methodology in optimization," *Operations Research* **39**, 553–582.
- Ross, S. (1983), *Introduction to Stochastic Dynamic Programming*, Academic Press, New York.
- Ross, S. R. (2002), *Simulation*, Academic Press, New York.
- Rust, J. (1997), "Using randomization to break the curse of dimensionality," *Econometrica* **65** (3), 487–516.
- Ruszczynski, A. (1980), "Feasible direction methods for stochastic programming problems," *Mathematical Programming* **19**, 220–229.
- Ruszczynski, A. (1987), "A linearization method for nonsmooth stochastic programming problems," *Mathematics of Operations Research* **12**, 32–49.
- Ruszczynski, A. (1993), "Parallel decomposition of multistage stochastic programming problems," *Mathematical Programming* **58**, 201–228.
- Ruszczynski, A. (2003), "Decomposition methods," in A. Ruszczynski and A. Shapiro, eds., *Handbook in Operations Research and Management Science: Stochastic Programming*, Elsevier, Amsterdam, pp. 141–212.
- Ruszczynski, A., and Shapiro, A. (2003), *Handbooks in Operations Research and Management Science: Stochastic Programming*, Vol. 10, Elsevier, Amsterdam.
- Ruszczynski, A., and Syski, W. (1986), "A method of aggregate stochastic subgradients with on-line stepsize rules for convex stochastic programming problems," *Mathematical Programming Study* **28**, 113–131.
- Ryzhov, I. O., and Powell, W. B. (2009), "A Monte Carlo knowledge gradient method for learning abatement potential of emissions reduction technologies," in M. Rossetti, R. R. Hill, A. Dunkin, and R. G. Ingals, eds., *Proceedings of the 2009 Winter Simulation Conference*, pp. 1492–1502.
- Ryzhov, I. O., and Powell, W. B. (2010a), "Approximate dynamic programming with correlated Bayesian beliefs," *Forty-Eighth Annual Allerton Conference on Communication, Control, and Computing, Monticello, IL*, IEEE Press, New York, pp. 1360–1367.
- Ryzhov, I. O., and Powell, W. B. (2010b), "Information collection on a graph," *Operations Research* 2012.

- Ryzhov, I. O., Frazier, P. I., and Powell, W. B. (2009), "Stepsize selection for approximate value iteration and a new optimal stepsize rule," Technical report, Department of Operations Research and Financial Engineering, Princeton University.
- Ryzhov, I. O., Powell, W. B., and Frazier, P. I. (2010), "The knowledge gradient algorithm for a general class of online learning problems," Technical report, Department of Operations Research and Financial Engineering, Princeton University.
- Samuel, A. L. (1959), "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development* **3**, 211–229.
- Schweitzer, P., and Seidmann, A. (1985), "Generalized polynomial approximations in Markovian decision processes," *Journal of Mathematical Analysis and Applications* **110**, 568–582.
- Secomandi, N. (2008), "An analysis of the control-algorithm resolving issue in inventory and revenue management," *Manufacturing and Service Operations Management* **10** (3), 468–483.
- Sen, S., and Higle, J. (1999), "An introductory tutorial on stochastic linear programming models," *Interfaces* **29** (2), 33–6152.
- Si, J., Barto, A. G., Powell, W. B., and Wunsch, D. (2004), *Handbook of Learning and Approximate Dynamic Programming*, Wiley-IEEE Press, Hoboken, NJ.
- Sigal, C. E., Pritsker, A. B., and Solberg, J. J. (1980), "The stochastic shortest route problem," *Operations Research* **28**, 1122–1129.
- Simao, H. P., Day, J., George, A. P., Gifford, T., Powell, W. B., and Nienow, J. (2009), "An approximate dynamic programming algorithm for large-scale fleet management: A case application," *Transportation Science* **43** (2), 178–197.
- Simao, H. P., George, A., Powell, W. B., Gifford, T., Nienow, J., and Day, J. (2010), "Approximate dynamic programming captures fleet operations for Schneider National," *Interfaces* **40** (5), 1–11.
- Singh, S., Jaakkola, T., Littman, M., and Szepesvari, C. (2000), "Convergence results for single-step on-policy reinforcement-learning algorithms," *Machine Learning* **38** (3), 287–308.
- Singh, S. P., Jaakkola, T., and Jordan, M. I. (1995), "Reinforcement learning with soft state aggregation," in *Advances in Neural Information Processing Systems*, Vol. 7, MIT Press, Cambridge, pp. 361–368.
- Smola, A. J., and Schölkopf, B. (2004), "A tutorial on support vector regression," *Statistics and Computing* **14** (3), 199–222.
- Soderstrom, T., Ljung, L., and Gustavsson, I. (1978), "A theoretical analysis of recursive identification methods," *Automatica* **78**, 231–244.
- Spall, J. C. (2003), *Introduction to Stochastic Search and Optimization: Estimation, Simulation and Control*, Wiley, Hoboken, NJ.
- Spivey, M. Z., and Powell, W. B. (2004), "The dynamic assignment problem," *Transportation Science* **38** (4), 399–419.
- Stengel, R. F. (1994), *Optimal Control and Estimation*, Dover, New York.
- Stokey, N. L., and R. E. Lucas, J. (1989), *Recursive Methods in Dynamic Economics*, Harvard University Press, Cambridge.
- Sutton, R. S., and Barto, A. G. (1981), "Toward a modern theory of adaptive networks," *Psychological Review* **88** (2), 135–170.

- Sutton, R. S., and Barto, A. G. (1998), *Reinforcement Learning*, Vol. 35, MIT Press, Cambridge.
- Sutton, R. S., and Singh, S. P. (1994), “On step-size and bias in temporal-difference learning,” *Proceedings of the Eighth Yale Workshop on Adaptive and Learning Systems*, Yale University, pp. 91–96.
- Sutton, R. S., Maei, H. R., Precup, D., Bhatnagar, S., Silver, D., Szepesvari, C., and Wiewiora, E. (2009a), “Fast gradient-descent methods for temporal-difference learning with linear function approximation,” in *Proceedings of the 26th Annual International Conference on Machine Learning*, ACM Press, New York, pp. 1–8.
- Sutton, R. S., Mcallester, D., Singh, S., Mansour, Y., Avenue, P., and Park, F. (2000), “Policy gradient methods for reinforcement learning with function approximation,” in A. Solla, T. K. Leen, and K. R. Muller, eds., *Advances in Neural Information Processing Systems*, MIT Press, Cambridge, pp. 1057–1063.
- Sutton, R. S., Szepesvari, C., and Maei, H. R. (2009), “A convergent $O(n)$ algorithm for off-policy temporal-difference learning with linear function approximation,” *Advances in Neural Information Processing Systems*, **21**, 1609–1616.
- Szepesvari, C. (2010), *Algorithms for Reinforcement Learning*, Morgan and Claypool, San Rafael, CA.
- Szita, I. (2007), “Rewarding excursions: Extending reinforcement learning to complex domains,” PhD thesis. Graduate School of Computer Science, Lorand University, Budapest.
- Taylor, H. (1967), “Evaluating a call option and optimal timing strategy in the stock market,” *Management Science* **12**, 111–120.
- Taylor, H. M. (1990), *Martingales and Random Walks*, Vol. 2, Elsevier Science, Amsterdam.
- Thrun, S. B. (1992), “The role of exploration in learning control,” in D. A. White, and D. A. Sofge, eds., *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, Van Nostrand Reinhold Company, New York.
- Topaloglu, H. (2001), “Dynamic programming approximations for dynamic resource allocation problems,” PhD thesis. Department of Operations Research and Financial Engineering, Princeton University.
- Topaloglu, H., and Powell, W. B. (2003), “An algorithm for approximating piecewise linear concave functions from sample gradients,” *Operations Research Letters* **31**, 66–76.
- Topkins, D. M. (1978), “Minimizing a submodular function on a lattice,” *Operations Research* **26**, 305–321.
- Trigg, D. W. (1964), “Monitoring a forecasting system,” *Operations Research Quarterly* **15**, 271–274.
- Trigg, D. W., and Leach, A. G. (1967), “Exponential smoothing with an adaptive response rate,” *Operations Research Quarterly* **18**, 53–59.
- Tsitsiklis, J. N. (1994), “Asynchronous stochastic approximation and Q-learning,” *Machine Learning* **16**, 185–202.
- Tsitsiklis, J. N., and Van Roy, B. (1996), “Feature-based methods for large scale dynamic programming,” *Machine Learning* **22** (1), 59–94.
- Tsitsiklis, J. N., and Van Roy, B. (1997), “An analysis of temporal-difference learning with function approximation,” *IEEE Transactions on Automatic Control* **42**, 674–690.

- Tsitsiklis, J. N., and Van Roy, B. (2001), "Regression methods for pricing complex american-style options," *IEEE Transactions on Neural Networks* **12**, 694–703.
- Tsitsiklis, J. N., Van Roy, B. (1997), "An analysis of temporal-difference learning with function approximation," *IEEE Transactions on Automatic Control* **42**, 674–690.
- Van Roy, B. (2001), "Neuro-dynamic programming: Overview and recent trends," in E. Feinberg and A. Shwartz, eds., *Handbook of Markov Decision Processes: Methods and Applications*, Kluwer Academic, Boston, pp. 431–460.
- Van Slyke, R., and Wets, R. (1969), "L-shaped linear programs with applications to optimal control and stochastic programming," *SIAM Journal of Applied Mathematics* **17**, 638–663.
- Venayagamoorthy, G. K., Harley, R. G., and Wunsch, D. G. (2002), "Comparison of heuristic dynamic programming and dual heuristic programming adaptive critics for neurocontrol of a turbogenerator," *IEEE Transactions in Neural Networks* **13**, 764–773.
- Wallace, S. W. (1986a), "Decomposing the requirement space of a transportation problem into polyhedral cones," *Mathematical Programming Study* **28**, 29–47.
- Wallace, S. W. (1986b), "Solving stochastic programs with network recourse," *Networks* **16**, 295–317.
- Wallace, S. W. (1987), "A piecewise linear upper bound on the network recourse function," *Mathematical Programming* **38**, 133–146.
- Wasan, M. T. (1969), *Stochastic Approximation*, Cambridge University Press, Cambridge, UK.
- Watkins, C. (1989), "Learning from delayed rewards," PhD thesis, King's College, Cambridge, UK.
- Watkins, C., and Dayan, P. (1992), "Q-Learning," *Machine Learning* **8**, 279–292.
- Weber, R. (1992), "On the Gittins index for multiarmed bandits," *Annals of Applied Probability* **2**, 1024–1033.
- Werbos, P. J. (1974), "Beyond regression: New tools for prediction and analysis in the behavioral sciences," PhD thesis, Harvard University.
- Werbos, P. J. (1989), "Backpropagation and neurocontrol: A review and prospectus," *International Joint Conference on Neural Networks*, pp. 209–216.
- Werbos, P. J. (1992a), "Approximate dynamic programming for real-time control and neural modelling," in D. A. White and D. A. Sofge, eds., *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, Van Nostrand Reinhold, New York, pp. 493–525.
- Werbos, P. J. (1992b), "Neurocontrol and supervised learning: An overview and evaluation," in D. A. White and D. A. Sofge, eds., *Handbook of Intelligent Control*, Von Nostrand Reinhold, New York, pp. 65–89.
- Werbos, P. J., Miller, W. T., and Sutton, R. S., eds. (1990), *Neural Networks for Control*, MIT Press, Cambridge.
- White, C. C. (1991), "A survey of solution techniques for the partially observable Markov decision process," *Annals of Operations Research* **32**, 215–230.
- White, D. A., and Sofge, D. A. (1992), *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, Van Nostrand Reinhold, New York.
- White, D. J. (1969), *Dynamic Programming*, Holden-Day, San Francisco.
- Whitt, W. (1978), "Approximations of dynamic programs I," *Mathematics of Operations Research*, **3** (3) 231–243.

- Whittle, P. (1982), *Optimization Over Time: Dynamic Programming and Stochastic Control*, Vols I and II, Wiley, New York.
- Williams, R. J. (1992), “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Machine Learning* **8**, 229–256.
- Williams, R. J., and Baird, L. C. (1990), “A mathematical analysis of actor–critic architectures for learning optimal controls through incremental dynamic programming,” in *Sixth Yale Workshop on Adaptive and Learning Systems*, New Haven, pp. 96–101.
- Wu, C. (1997), “Rollout algorithms for combinatorial optimization,” *Journal of Heuristics* **3**, 245–262.
- Wu, T. T., Powell, W. B., and Whisman, A. (2009), “The optimizing-simulator,” *ACM Transactions on Modeling and Computer Simulation* **19** (3), 1–31.
- Yang, Y. (2001), “Adaptive regression by mixing,” *Journal of the American Statistical Association*, **96**, 574–588.
- Yao, Y. (2006), “Some results on the Gittins index for a normal reward process,” in H. Ho, C. Ing, and T. Lai, eds., *Time Series and Related Topics: In Memory of Ching-Zong Wei*, Institute of Mathematical Statistics, Beachwood, OH, pp. 284–294.
- Young, P. (1984), *Recursive Estimation and Time-Series Analysis*, Springer, Berlin.
- Yücesan, E., Chen, C. H., Dai, L., and Chen, H. C. (1997), “A gradient approach for smartly allocating computing budget for discrete event simulation,” in J. Charnes, D. Morrice, D. Brunner, and J. Swain, eds., *Proceedings of the 1996 Winter Simulation Conference*, IEEE Press, Piscataway, NJ, pp. 398–405.
- Zipkin, P. (1980a), “Bounds for row-aggregation in linear programming,” *Operations Research* **28**, 903–916.
- Zipkin, P. (1980b), “Bounds on the effect of aggregating variables in linear programming,” *Operations Research* **28**, 403–418.

Index

- Actions, 187
- Actor–critic, 408
- Affine function, 64
- Aggregation, 290
 - modeling, 295
 - multiple levels, 299
- Algorithm
 - ADP for asset acquisition, 545
 - ADP with exact expectation, 120
 - ADP for policy iteration, 404, 405
 - ADP using post-decision state, 141
 - ADP with pre-decision state, 388
 - approximate expectation, 128
 - approximate policy iteration
 - kernel regression, 415
 - least squares temporal differencing, 407
 - approximate policy iteration with VFA, 409
 - asynchronous dynamic programming, 152
 - bias-adjusted Kalman filter stepsize, 447
 - CUPPS algorithm, 524
 - double-pass ADP, 392
 - finite horizon policy evaluation, 338
 - generic ADP, 147
 - infinite horizon generic ADP, 401
 - policy iteration, 74
 - policy search
 - indifference zone, 272
 - Q -learning
 - finite horizon, 390, 391
 - real-time dynamic programming, 127
 - relative value iteration, 70
 - roll-out policy, 225
 - SHAPE algorithm, 513
 - shortest path, 26, 27
- SPAR, 504
- stochastic decomposition, 523
- synchronous dynamic programming, 152
- temporal-difference learning for infinite horizon, 346
- temporal difference learning with linear model, 357
- tree-search, 238
- value iteration, 68, 69
- Aliasing, 304
- American option, 307
- Apparent convergence, 450
- Approximate policy iteration
 - kernel regression, 415
- Approximate value iteration, 341
 - multidimensional decision vectors, 395
 - post-decision state, 389
 - pre-decision state, 386
- Asset acquisition, 37, 38
 - ADP algorithm, 543
 - lagged, 39
 - variations, 544
- Asset valuation, 36
- Asynchronous dynamic programming, 150
- Attribute transition function, 203
- Backpropagation through time, 393
- Bandit problems, 47, 470
- Basis functions, 513
 - approximate linear programming, 411
 - geometric view, 312
 - Longstaff and Schwartz, 308
 - neural network, 320

Approximate Dynamic Programming: Solving the Curses of Dimensionality, Second Edition.

Warren B. Powell.

© 2011 John Wiley & Sons, Inc. Published 2011 by John Wiley & Sons, Inc.

- recursive time-series, 355
- tic-tac-toe, 311
- Basis functions, 165, 304
- Batch replenishment, 40, 79
- Behavior policy, 389, 125
- Belief state, 463
- Bellman
 - functional equation, 4
 - Hamilton–Jacobi, 4
 - optimality equation, 4
 - recurrence equation, 4
- Bellman error, 342
- Bellman’s equation, 4, 38, 58
 - deterministic, 59
 - expectation form, 60
 - operator form, 64
 - standard form, 60
 - vector form, 61
- Benders’ decomposition, 520
 - CUPPS algorithm, 524
 - stochastic decomposition, 523
- Bias, 293
 - statistical error in max operator, 397
- Blood management
 - ADP algorithm, 552
 - model, 548
- Boltzmann exploration, 467, 483
- Budgeting problem, continuous, 29
- Contribution function, 206
- Controls, 187
- Cost function, 206
- Cost-to-go function, 61
- CUPPS algorithm, 523
- Curses of dimensionality, 112
 - action space, 6
 - outcome space, 6
 - state space, 5
- Cutting planes, 516
- Decision node, 32
- Decision tree, 32
- Decisions, 187
- Discount factors, 597
- Double-pass algorithm, 392
- Dynamic assignment problem, 43
- Eligibility trace, 346
- Epsilon-greedy, 483
- Epsilon-greedy policy, 388
- Error measures, 600
- Exogenous information, 38, 51, 189
 - lagged, 192
- outcomes, 191
- scenarios, 191
- Expected opportunity cost, 271
- Experimental issues
 - discount factors, 597
 - starting, 598
- Exploitation, 464
- Exploration, 465
- Exploration policies
 - Boltzmann exploration, 262, 483
 - epsilon-greedy, 483
 - epsilon-greedy exploration, 262
 - interval estimation, 483
 - interval exploration, 262
 - local bandit approximation, 484
 - persistent excitation, 483
 - pure exploitation, 261
 - pure exploration, 262
- Exploration versus exploitation, 124, 152, 242, 257, 457
- Exponential smoothing, 128
- Factored representation of a state, 186
- Finite horizon, for infinite horizon models, 415
- Flat representation of a state, 186
- Forward dynamic programming, 114
- Gambling problem, 34
- Gittins exploration, 484
- Gittins indices, 470
 - basic theory, 472
 - foundations, 470
 - normally distributed rewards, 474
- Gradients, 498
- Greedy policy, 221
- Greedy strategy, 116
- Indifference zone selection, 271
- Infinite horizon, 66
 - finite-horizon approximations, 415
 - policy iteration, 403
 - temporal-difference learning, 345
- Information acquisition, 47
- Initialization, 150
- Interval estimation, 466, 483
- k*-nearest neighbor, 316
- Kernel regression, 317, 415
 - LSTD, 413
- Knowledge gradient, 477
- Knowledge state, 463
- L-shaped decomposition, 522
- Lagged information, 192

- Lasso regression, 314
Lattice, 81
Learning policies, 125
 Boltzmann exploration, 467
 exploitation, 464
 exploration, 465
 interval estimation, 466
 persistent excitation, 465
 upper confidence bound sampling algorithm, 467
Learning rate schedules, 425
Learning strategies
 epsilon-greedy exploration, 466
 Gittins exploration, 484
 Gittins indices, 470
 knowledge gradient, 477
 upper confidence bound, 467
Least squares temporal difference, 363
Leveling algorithm, 502
Linear filter, 128
Linear operator, 64
Linear programming method
 approximate, 411
 exact, 78
Linear regression, 305
 Longstaff and Schwartz, 307
 recursive estimation
 derivation, 377
 multiple observations, 353
 time-series, 354
 recursive least squares
 nonstationary data, 352
 stationary data, 350
 recursive methods, 349
 stochastic gradient algorithm, 347
Local bandit approximation, 484
Local polynomial regression, 319
Longstaff and Schwartz, 307
Lookahead policy, 221, 224
LSTD, 363, 413
- Markov decision processes, 57
max operator, 64
Measure-theoretic view of information, 211
min operator, 64
Model
 contribution function, 122, 206
 decisions, 187
 elements of a dynamic program, 168
 contribution function, 168
 decision variable, 168
 exogenous information, 168
 state variable, 168
 transition function, 168
exogenous information, 122
objective function, 206
policies, 221
resources, 174
 multiple, 175
 single discrete, 175
state, 178
time, 170
transition function, 122, 198
Modeling dynamic programs, 50
Monotone policies, 78, 79
 proof of optimality, 97
Monte Carlo sampling, 117
Multi-armed bandit problem, 462
Myopic policy, 221, 224
- Neural networks, 319
Nomadic trucker, 176
 learning, 457
Nonparametric models, 316
 k -nearest neighbor, 316
 kernel regression, 317
 local polynomial regression, 319
- Objective function, 51, 58, 209
Offline learning, 462
Online learning, 462
Optimal computing budget allocation, 273
Optimality equation, 58
 post-decision state, 138
 proof, 85
Optimistic policy iteration, 404
Ordinal optimization, 271
Outcome node, 32
Outcomes, 191
- Partial policy evaluation, 404
Partially observable states, 185
Persistent excitation, 465, 483
Physical state, 463
 myopic, 224
Policies, 197, 221
 behavior, 125, 389
 Boltzmann exploration, 244
 epsilon greedy, 388
 GLIE, 243
 greedy, 116, 221
 learning, 125
 lookahead, 221, 224
 myopic, 221
 optimistic, 404
 partial, 404
 policy function approximation, 221
 randomized, 242

- Policies (*Continued*)
 roll-out heuristic, 225
 rolling horizon procedure, 227
 deterministic, 227
 discounted, 231
 stochastic, 229
 sampling, 125, 389
 soft max, 244
 target, 125
 tree search, 225
 value function approximation, 221
 Policy function approximation, 221
 Policy iteration, 74
 infinite horizon, 403
 Post-decision state, 181
 optimality equations, 138
 perspective, 130
 Post-decision state variable, 129

Q-learning, 122, 389

 Randomized policies, 96, 242
 Ranking and selection, 462
 Real-time dynamic programming, 126
 Resource allocation
 asset acquisition, 541
 blood management, 547
 fleet management, 573
 general model, 560
 portfolio optimization, 557
 trucking application, 580
 Resources
 multiple, 175
 nomadic trucker, 176
 single discrete, 175
 Reward function, 206
 Ridge regression, 314
 Roll-out heuristic, 225
 Rolling horizon procedure, 227
 RTDP, 126

 Sample path, 116
 Sampling policy, 125, 389
 SARSA, 124
 Scenarios, 191
 SHAPE algorithm, 509
 proof of convergence, 528
 Sherman–Morrison, 379
 Shortest path
 deterministic, 2, 26
 information collecting, 50
 stochastic, 33
 SPAR
 projection, 504
 weighted projection, 504
 SPAR algorithm, projection operation, 534
 State
 alias, 304
 asset pricing, 36
 bandit problem, 48
 belief, 463
 budgeting problem, 28
 definition, 50, 178
 dynamic assignment problem, 46
 factored, 186
 flat, 186
 gambling problem, 34
 hyperstate, 464
 knowledge, 463
 multidimensional, 112, 134
 partially observable, 185
 physical, 463
 post-decision, 129, 181
 pre-decision, 129
 resource state, 38, 39
 sampling strategies, 150
 shortest path, 27
 state of knowledge, 48
 transformer replacement, 43
 State variable, definition, 179
 States of a system
 belief state, 180
 information state, 180
 physical state, 180
 Stepsize, 127
 apparent convergence, 451
 bias and variance, 293
 bias-adjusted, 447
 convergence conditions, 426
 deterministic, 425
 $1/n$, 430
 constant, 428
 harmonic, 430
 McClain, 431
 polynomial learning rate, 430
 search-then-converge, 431
 infinite horizon, bounds, 423
 optimal
 nonstationary I, 440
 nonstationary II, 442
 stationary, 438
 stochastic, 433
 convergence conditions, 434
 Kesten's rule, 436
 stochastic gradient adaptive stepsize, 436
 Trigg, 436

- Stepsize rule, 425
Stochastic approximation
 Martingale proof, 279
 older proof, 276
Stochastic approximation procedure, 254
Stochastic decomposition, 523
Stochastic gradient, 252
Stochastic gradient algorithm, 254
Stochastic programming, 516
 Benders, 520
Submodular, 82
Subset selection, 271
Superadditive, 83
Supermodular, 82
Supervisor, 311
Supervisory learning, 311
Supervisory processes, 196
Support vector regression, 314
Synchronous dynamic programming, 150
System model, 39
- Target policy, 125
Temporal difference (TD) learning, 341, 344
 infinite horizon, 345
 linear models
 off-policy, 358
 on-policy, 358
 nonlinear models
 on-policy, 358
Tic-tac-toe, 310
Time, 170
Trajectory following, 388
Transformer replacement, 42
Transition function, 3, 28, 39, 40, 51, 198
 attribute transition, 203
 batch, 41
 resource transition function, 201
 special cases, 204
Transition matrix, 57, 63
Tree search, 225
Two-stage stochastic program, 516
- Upper confidence bound, 476
Upper confidence bound sampling algorithm, 467
- Value function, 61, 62
Value function approximation, 114, 144, 221, 235
 aggregation, 290
 cutting planes, 516
 error measures, 600
 gradients, 498
 leveling, 502
 linear approximation, 499
 mixed strategies, 324
 neural networks, 319
 piecewise linear, 501
 recursive methods, 349
 regression, 304
 regression methods, 513
 SHAPE algorithm, 509
 SPAR, 504
 tic-tac-toe, 310
Value iteration, 68, 69
 bound, 72
 error bound, 95
 monotonic behavior, 71
 pre-decision state, 388
 proof of convergence, 89
 proof of monotonicity, 93
 relative value iteration, 70
 stopping rule, 69
Variance of estimates, 293

WILEY SERIES IN PROBABILITY AND STATISTICS
ESTABLISHED BY WALTER A. SHEWHART AND SAMUEL S. WILKS

Editors: *David J. Balding, Noel A. C. Cressie, Garrett M. Fitzmaurice, Harvey Goldstein, Iain M. Johnstone, Geert Molenberghs, David W. Scott, Adrian F. M. Smith, Ruey S. Tsay, Sanford Weisberg*
Editors Emeriti: *Vic Barnett, J. Stuart Hunter, Joseph B. Kadane, Jozef L. Teugels*

The *Wiley Series in Probability and Statistics* is well established and authoritative. It covers many topics of current research interest in both pure and applied statistics and probability theory. Written by leading statisticians and institutions, the titles span both state-of-the-art developments in the field and classical methods.

Reflecting the wide range of current research in statistics, the series encompasses applied, methodological and theoretical statistics, ranging from applications and new techniques made possible by advances in computerized practice to rigorous treatment of theoretical approaches.

This series provides essential and invaluable reading for all statisticians, whether in academia, industry, government, or research.

- † ABRAHAM and LEDOLTER · Statistical Methods for Forecasting
AGRESTI · Analysis of Ordinal Categorical Data, *Second Edition*
AGRESTI · An Introduction to Categorical Data Analysis, *Second Edition*
AGRESTI · Categorical Data Analysis, *Second Edition*
ALTMAN, GILL, and McDONALD · Numerical Issues in Statistical Computing for the Social Scientist
AMARATUNGA and CABRERA · Exploration and Analysis of DNA Microarray and Protein Array Data
ANDĚL · Mathematics of Chance
ANDERSON · An Introduction to Multivariate Statistical Analysis, *Third Edition*
* ANDERSON · The Statistical Analysis of Time Series
ANDERSON, AUQUIER, HAUCK, OAKES, VANDAELE, and WEISBERG · Statistical Methods for Comparative Studies
ANDERSON and LOYNES · The Teaching of Practical Statistics
ARMITAGE and DAVID (editors) · Advances in Biometry
ARNOLD, BALAKRISHNAN, and NAGARAJA · Records
* ARTHANARI and DODGE · Mathematical Programming in Statistics
* BAILEY · The Elements of Stochastic Processes with Applications to the Natural Sciences
BALAKRISHNAN and KOUTRAS · Runs and Scans with Applications
BALAKRISHNAN and NG · Precedence-Type Tests and Applications
BARNETT · Comparative Statistical Inference, *Third Edition*
BARNETT · Environmental Statistics
BARNETT and LEWIS · Outliers in Statistical Data, *Third Edition*
BARTOSZYNSKI and NIEWIADOMSKA-BUGAJ · Probability and Statistical Inference
BASILEVSKY · Statistical Factor Analysis and Related Methods: Theory and Applications
BASU and RIGDON · Statistical Methods for the Reliability of Repairable Systems
BATES and WATTS · Nonlinear Regression Analysis and Its Applications

*Now available in a lower priced paperback edition in the Wiley Classics Library.

†Now available in a lower priced paperback edition in the Wiley-Interscience Paperback Series.

- BECHHOFER, SANTNER, and GOLDSMAN · Design and Analysis of Experiments for Statistical Selection, Screening, and Multiple Comparisons
- BEIRLANT, GOEGEBEUR, SEGERS, TEUGELS, and DE WAAL · Statistics of Extremes: Theory and Applications
- BELSLY · Conditioning Diagnostics: Collinearity and Weak Data in Regression
- † BELSLY, KUH, and WELSCH · Regression Diagnostics: Identifying Influential Data and Sources of Collinearity
- BENDAT and PIERSOL · Random Data: Analysis and Measurement Procedures, *Fourth Edition*
- BERNARDO and SMITH · Bayesian Theory
- BERRY, CHALONER, and GEWEKE · Bayesian Analysis in Statistics and Econometrics: Essays in Honor of Arnold Zellner
- BHAT and MILLER · Elements of Applied Stochastic Processes, *Third Edition*
- BHATTACHARYA and WAYMIRE · Stochastic Processes with Applications
- BIEMER, GROVES, LYBERG, MATHIOWETZ, and SUDMAN · Measurement Errors in Surveys
- BILLINGSLEY · Convergence of Probability Measures, *Second Edition*
- BILLINGSLEY · Probability and Measure, *Third Edition*
- BIRKES and DODGE · Alternative Methods of Regression
- BISGAARD and KULAHCI · Time Series Analysis and Forecasting by Example
- BISWAS, DATTA, FINE, and SEGAL · Statistical Advances in the Biomedical Sciences: Clinical Trials, Epidemiology, Survival Analysis, and Bioinformatics
- BLISCHKE AND MURTHY (editors) · Case Studies in Reliability and Maintenance
- BLISCHKE AND MURTHY · Reliability: Modeling, Prediction, and Optimization
- BLOOMFIELD · Fourier Analysis of Time Series: An Introduction, *Second Edition*
- BOLLEN · Structural Equations with Latent Variables
- BOLLEN and CURRAN · Latent Curve Models: A Structural Equation Perspective
- BOROVKOV · Ergodicity and Stability of Stochastic Processes
- BOSQ and BLANKE · Inference and Prediction in Large Dimensions
- BOULEAU · Numerical Methods for Stochastic Processes
- BOX · Bayesian Inference in Statistical Analysis
- BOX · Improving Almost Anything, *Revised Edition*
- BOX · R. A. Fisher, the Life of a Scientist
- BOX and DRAPER · Empirical Model-Building and Response Surfaces
- * BOX and DRAPER · Evolutionary Operation: A Statistical Method for Process Improvement
- BOX and DRAPER · Response Surfaces, Mixtures, and Ridge Analyses, *Second Edition*
- BOX, HUNTER, and HUNTER · Statistics for Experimenters: Design, Innovation, and Discovery, *Second Edition*
- BOX, JENKINS, and REINSEL · Time Series Analysis: Forecasting and Control, *Fourth Edition*
- BOX, LUCEÑO, and PANIAGUA-QUIÑONES · Statistical Control by Monitoring and Adjustment, *Second Edition*
- BRANDIMARTE · Numerical Methods in Finance: A MATLAB-Based Introduction
- † BROWN and HOLLANDER · Statistics: A Biomedical Introduction
- BRUNNER, DOMHOF, and LANGER · Nonparametric Analysis of Longitudinal Data in Factorial Experiments
- BUCKLEW · Large Deviation Techniques in Decision, Simulation, and Estimation
- CAIROLI and DALANG · Sequential Stochastic Optimization
- CASTILLO, HADI, BALAKRISHNAN, and SARABIA · Extreme Value and Related Models with Applications in Engineering and Science
- CHAN · Time Series: Applications to Finance with R and S-Plus®, *Second Edition*
- CHARALAMBIDES · Combinatorial Methods in Discrete Distributions
- CHATTERJEE and HADI · Regression Analysis by Example, *Fourth Edition*

*Now available in a lower priced paperback edition in the Wiley Classics Library.

†Now available in a lower priced paperback edition in the Wiley-Interscience Paperback Series.

- CHATTERJEE and HADI · Sensitivity Analysis in Linear Regression
- CHERNICK · Bootstrap Methods: A Guide for Practitioners and Researchers,
Second Edition
- CHERNICK and FRIIS · Introductory Biostatistics for the Health Sciences
- CHILÈS and DELFINER · Geostatistics: Modeling Spatial Uncertainty
- CHOW and LIU · Design and Analysis of Clinical Trials: Concepts and Methodologies,
Second Edition
- CLARKE · Linear Models: The Theory and Application of Analysis of Variance
- CLARKE and DISNEY · Probability and Random Processes: A First Course with Applications, *Second Edition*
- * COCHRAN and COX · Experimental Designs, *Second Edition*
- COLLINS and LANZA · Latent Class and Latent Transition Analysis: With Applications in the Social, Behavioral, and Health Sciences
- CONGDON · Applied Bayesian Modelling
- CONGDON · Bayesian Models for Categorical Data
- CONGDON · Bayesian Statistical Modelling, *Second Edition*
- CONOVER · Practical Nonparametric Statistics, *Third Edition*
- COOK · Regression Graphics
- COOK and WEISBERG · An Introduction to Regression Graphics
- COOK and WEISBERG · Applied Regression Including Computing and Graphics
- CORNELL · A Primer on Experiments with Mixtures
- CORNELL · Experiments with Mixtures, Designs, Models, and the Analysis of Mixture Data, *Third Edition*
- COVER and THOMAS · Elements of Information Theory
- COX · A Handbook of Introductory Statistical Methods
- * COX · Planning of Experiments
- CRESSIE · Statistics for Spatial Data, *Revised Edition*
- CRESSIE and WIKLE · Statistics for Spatio-Temporal Data
- CSÖRGÖ and HORVÁTH · Limit Theorems in Change Point Analysis
- DANIEL · Applications of Statistics to Industrial Experimentation
- DANIEL · Biostatistics: A Foundation for Analysis in the Health Sciences, *Eighth Edition*
- * DANIEL · Fitting Equations to Data: Computer Analysis of Multifactor Data,
Second Edition
- DASU and JOHNSON · Exploratory Data Mining and Data Cleaning
- DAVID and NAGARAJA · Order Statistics, *Third Edition*
- * DEGROOT, FIENBERG, and KADANE · Statistics and the Law
- DEL CASTILLO · Statistical Process Adjustment for Quality Control
- DEMARIS · Regression with Social Data: Modeling Continuous and Limited Response Variables
- DEMIDENKO · Mixed Models: Theory and Applications
- DENISON, HOLMES, MALLICK and SMITH · Bayesian Methods for Nonlinear Classification and Regression
- DETTE and STUDDEN · The Theory of Canonical Moments with Applications in Statistics, Probability, and Analysis
- DEY and MUKERJEE · Fractional Factorial Plans
- DILLON and GOLDSTEIN · Multivariate Analysis: Methods and Applications
- DODGE · Alternative Methods of Regression
- * DODGE and ROMIG · Sampling Inspection Tables, *Second Edition*
 - * DOOB · Stochastic Processes
- DOWDY, WEARDEN, and CHILKO · Statistics for Research, *Third Edition*
- DRAPER and SMITH · Applied Regression Analysis, *Third Edition*
- DRYDEN and MARDIA · Statistical Shape Analysis
- DUDEWICZ and MISHRA · Modern Mathematical Statistics

*Now available in a lower priced paperback edition in the Wiley Classics Library.

†Now available in a lower priced paperback edition in the Wiley-Interscience Paperback Series.

- DUNN and CLARK · Basic Statistics: A Primer for the Biomedical Sciences,
Third Edition
- DUPUIS and ELLIS · A Weak Convergence Approach to the Theory of Large Deviations
- EDLER and KITSOS · Recent Advances in Quantitative Methods in Cancer and Human Health Risk Assessment
- * ELANDT-JOHNSON and JOHNSON · Survival Models and Data Analysis
- ENDERS · Applied Econometric Time Series
- † ETHIER and KURTZ · Markov Processes: Characterization and Convergence
- EVANS, HASTINGS, and PEACOCK · Statistical Distributions, *Third Edition*
- EVERITT · Cluster Analysis, *Fifth Edition*
- FELLER · An Introduction to Probability Theory and Its Applications, Volume I,
Third Edition, Revised; Volume II, *Second Edition*
- FISHER and VAN BELLE · Biostatistics: A Methodology for the Health Sciences
- FITZMAURICE, LAIRD, and WARE · Applied Longitudinal Analysis, *Second Edition*
- * FLEISS · The Design and Analysis of Clinical Experiments
- FLEISS · Statistical Methods for Rates and Proportions, *Third Edition*
- † FLEMING and HARRINGTON · Counting Processes and Survival Analysis
- FUJIKOSHI, ULYANOV, and SHIMIZU · Multivariate Statistics: High-Dimensional and Large-Sample Approximations
- FULLER · Introduction to Statistical Time Series, *Second Edition*
- † FULLER · Measurement Error Models
- GALLANT · Nonlinear Statistical Models
- GEISSER · Modes of Parametric Statistical Inference
- GELMAN and MENG · Applied Bayesian Modeling and Causal Inference from Incomplete-Data Perspectives
- GEWEKE · Contemporary Bayesian Econometrics and Statistics
- GHOSH, MUKHOPADHYAY, and SEN · Sequential Estimation
- GIESBRECHT and GUMPERTZ · Planning, Construction, and Statistical Analysis of Comparative Experiments
- GIFI · Nonlinear Multivariate Analysis
- GIVENS and HOETING · Computational Statistics
- GLASSERMAN and YAO · Monotone Structure in Discrete-Event Systems
- GNANADESIKAN · Methods for Statistical Data Analysis of Multivariate Observations, *Second Edition*
- GOLDSTEIN · Multilevel Statistical Models, *Fourth Edition*
- GOLDSTEIN and LEWIS · Assessment: Problems, Development, and Statistical Issues
- GOLDSTEIN and WOOFF · Bayes Linear Statistics
- GREENWOOD and NIKULIN · A Guide to Chi-Squared Testing
- GROSS, SHORTLE, THOMPSON, and HARRIS · Fundamentals of Queueing Theory, *Fourth Edition*
- GROSS, SHORTLE, THOMPSON, and HARRIS · Solutions Manual to Accompany Fundamentals of Queueing Theory, *Fourth Edition*
- * HAHN and SHAPIRO · Statistical Models in Engineering
- HAHN and MEEKER · Statistical Intervals: A Guide for Practitioners
- HALD · A History of Probability and Statistics and their Applications Before 1750
- HALD · A History of Mathematical Statistics from 1750 to 1930
- † HAMPEL · Robust Statistics: The Approach Based on Influence Functions
- HANNAN and DEISTLER · The Statistical Theory of Linear Systems
- HARMAN and KULKARNI · An Elementary Introduction to Statistical Learning Theory
- HARTUNG, KNAPP, and SINHA · Statistical Meta-Analysis with Applications
- HEIBERGER · Computation for the Analysis of Designed Experiments
- HEDAYAT and SINHA · Design and Inference in Finite Population Sampling
- HEDEKER and GIBBONS · Longitudinal Data Analysis

*Now available in a lower priced paperback edition in the Wiley Classics Library.

†Now available in a lower priced paperback edition in the Wiley-Interscience Paperback Series.

- HELLER · MACSYMA for Statisticians
- HERITIER, CANTONI, COPT, and VICTORIA-FESER · Robust Methods in Biostatistics
- HINKELMANN and KEMPTHORNE · Design and Analysis of Experiments, Volume 1: Introduction to Experimental Design, *Second Edition*
- HINKELMANN and KEMPTHORNE · Design and Analysis of Experiments, Volume 2: Advanced Experimental Design
- HOAGLIN, MOSTELLER, and TUKEY · Fundamentals of Exploratory Analysis of Variance
- * HOAGLIN, MOSTELLER, and TUKEY · Exploring Data Tables, Trends and Shapes
- * HOAGLIN, MOSTELLER, and TUKEY · Understanding Robust and Exploratory Data Analysis
- HOCHBERG and TAMHANE · Multiple Comparison Procedures
- HOCKING · Methods and Applications of Linear Models: Regression and the Analysis of Variance, *Second Edition*
- HOEL · Introduction to Mathematical Statistics, *Fifth Edition*
- HOGG and KLUGMAN · Loss Distributions
- HOLLANDER and WOLFE · Nonparametric Statistical Methods, *Second Edition*
- HOSMER and LEMESHOW · Applied Logistic Regression, *Second Edition*
- HOSMER, LEMESHOW, and MAY · Applied Survival Analysis: Regression Modeling of Time-to-Event Data, *Second Edition*
- HUBER · Data Analysis: What Can Be Learned From the Past 50 Years
- HUBER · Robust Statistics
- † HUBER and RONCHETTI · Robust Statistics, *Second Edition*
- HUBERTY · Applied Discriminant Analysis, *Second Edition*
- HUBERTY and OLEJNIK · Applied MANOVA and Discriminant Analysis, *Second Edition*
- HUNT and KENNEDY · Financial Derivatives in Theory and Practice, *Revised Edition*
- HURD and MIAMEE · Periodically Correlated Random Sequences: Spectral Theory and Practice
- HUSKOVA, BERAN, and DUPAC · Collected Works of Jaroslav Hajek—with Commentary
- HUZURBAZAR · Flowgraph Models for Multistate Time-to-Event Data
- IMAN and CONOVER · A Modern Approach to Statistics
- JACKMAN · Bayesian Analysis for the Social Sciences
- † JACKSON · A User's Guide to Principle Components
- JOHN · Statistical Methods in Engineering and Quality Assurance
- JOHNSON · Multivariate Statistical Simulation
- JOHNSON and BALAKRISHNAN · Advances in the Theory and Practice of Statistics: A Volume in Honor of Samuel Kotz
- JOHNSON and BHATTACHARYYA · Statistics: Principles and Methods, *Fifth Edition*
- JOHNSON, KEMP, and KOTZ · Univariate Discrete Distributions, *Third Edition*
- JOHNSON and KOTZ · Distributions in Statistics
- JOHNSON and KOTZ (editors) · Leading Personalities in Statistical Sciences: From the Seventeenth Century to the Present
- JOHNSON, KOTZ, and BALAKRISHNAN · Continuous Univariate Distributions, Volume 1, *Second Edition*
- JOHNSON, KOTZ, and BALAKRISHNAN · Continuous Univariate Distributions, Volume 2, *Second Edition*
- JOHNSON, KOTZ, and BALAKRISHNAN · Discrete Multivariate Distributions
- JUDGE, GRIFFITHS, HILL, LÜTKEPOHL, and LEE · The Theory and Practice of Econometrics, *Second Edition*
- JUREČKOVÁ and SEN · Robust Statistical Procedures: Asymptotics and Interrelations
- JUREK and MASON · Operator-Limit Distributions in Probability Theory

*Now available in a lower priced paperback edition in the Wiley Classics Library.

†Now available in a lower priced paperback edition in the Wiley-Interscience Paperback Series.

- KADANE · Bayesian Methods and Ethics in a Clinical Trial Design
- KADANE AND SCHUM · A Probabilistic Analysis of the Sacco and Vanzetti Evidence
- KALBFLEISCH and PRENTICE · The Statistical Analysis of Failure Time Data, *Second Edition*
- KARIYA and KURATA · Generalized Least Squares
- KASS and VOS · Geometrical Foundations of Asymptotic Inference
- † KAUFMAN and ROUSSEEUW · Finding Groups in Data: An Introduction to Cluster Analysis
- KEDEM and FOKIANOS · Regression Models for Time Series Analysis
- KENDALL, BARDEN, CARNE, and LE · Shape and Shape Theory
- KHURI · Advanced Calculus with Applications in Statistics, *Second Edition*
- KHURI, MATHEW, and SINHA · Statistical Tests for Mixed Linear Models
- * KISH · Statistical Design for Research
- KLEIBER and KOTZ · Statistical Size Distributions in Economics and Actuarial Sciences
- KLEMELÄ · Smoothing of Multivariate Data: Density Estimation and Visualization
- KLUGMAN, PANJER, and WILLMOT · Loss Models: From Data to Decisions, *Third Edition*
- KLUGMAN, PANJER, and WILLMOT · Solutions Manual to Accompany Loss Models: From Data to Decisions, *Third Edition*
- KOSKI and NOBLE · Bayesian Networks: An Introduction
- KOTZ, BALAKRISHNAN, and JOHNSON · Continuous Multivariate Distributions, Volume 1, *Second Edition*
- KOTZ and JOHNSON (editors) · Encyclopedia of Statistical Sciences: Volumes 1 to 9 with Index
- KOTZ and JOHNSON (editors) · Encyclopedia of Statistical Sciences: Supplement Volume
- KOTZ, READ, and BANKS (editors) · Encyclopedia of Statistical Sciences: Update Volume 1
- KOTZ, READ, and BANKS (editors) · Encyclopedia of Statistical Sciences: Update Volume 2
- KOVALENKO, KUZNETZOV, and PEGG · Mathematical Theory of Reliability of Time-Dependent Systems with Practical Applications
- KOWALSKI and TU · Modern Applied U-Statistics
- KRISHNAMOORTHY and MATHEW · Statistical Tolerance Regions: Theory, Applications, and Computation
- KROESE, TAIMRE, and BOTEV · Handbook of Monte Carlo Methods
- KROONENBERG · Applied Multiway Data Analysis
- KULINSKAYA, MORGENTHALER, and STAUDTE · Meta Analysis: A Guide to Calibrating and Combining Statistical Evidence
- KUROWICKA and COOKE · Uncertainty Analysis with High Dimensional Dependence Modelling
- KVAM and VIDAKOVIC · Nonparametric Statistics with Applications to Science and Engineering
- LACHIN · Biostatistical Methods: The Assessment of Relative Risks, *Second Edition*
- LAD · Operational Subjective Statistical Methods: A Mathematical, Philosophical, and Historical Introduction
- LAMPERTI · Probability: A Survey of the Mathematical Theory, *Second Edition*
- LANGE, RYAN, BILLARD, BRILLINGER, CONQUEST, and GREENHOUSE · Case Studies in Biometry
- LARSON · Introduction to Probability Theory and Statistical Inference, *Third Edition*
- LAWLESS · Statistical Models and Methods for Lifetime Data, *Second Edition*
- LAWSON · Statistical Methods in Spatial Epidemiology, *Second Edition*
- LE · Applied Categorical Data Analysis
- LE · Applied Survival Analysis

*Now available in a lower priced paperback edition in the Wiley Classics Library.

†Now available in a lower priced paperback edition in the Wiley-Interscience Paperback Series.

- LEE · Structural Equation Modeling: A Bayesian Approach
 LEE and WANG · Statistical Methods for Survival Data Analysis, *Third Edition*
 LEPAGE and BILLARD · Exploring the Limits of Bootstrap
 LEYLAND and GOLDSTEIN (editors) · Multilevel Modelling of Health Statistics
 LIAO · Statistical Group Comparison
 LINDVALL · Lectures on the Coupling Method
 LIN · Introductory Stochastic Analysis for Finance and Insurance
 LINHART and ZUCCHINI · Model Selection
 LITTLE and RUBIN · Statistical Analysis with Missing Data, *Second Edition*
 LLOYD · The Statistical Analysis of Categorical Data
 LOWEN and TEICH · Fractal-Based Point Processes
 MAGNUS and NEUDECKER · Matrix Differential Calculus with Applications in
 Statistics and Econometrics, *Revised Edition*
 MALLER and ZHOU · Survival Analysis with Long Term Survivors
 MALLOWS · Design, Data, and Analysis by Some Friends of Cuthbert Daniel
 MANN, SCHAFER, and SINGPURWALLA · Methods for Statistical Analysis of
 Reliability and Life Data
 MANTON, WOODBURY, and TOLLEY · Statistical Applications Using Fuzzy Sets
 MARCHETTE · Random Graphs for Statistical Pattern Recognition
 MARDIA and JUPP · Directional Statistics
 MARKOVICH · Nonparametric Analysis of Univariate Heavy-Tailed Data: Research and
 Practice
 MARONNA, MARTIN and YOHAI · Robust Statistics: Theory and Methods
 MASON, GUNST, and HESS · Statistical Design and Analysis of Experiments with
 Applications to Engineering and Science, *Second Edition*
 McCULLOCH, SEARLE, and NEUHAUS · Generalized, Linear, and Mixed Models,
 Second Edition
 McFADDEN · Management of Data in Clinical Trials, *Second Edition*
 * McLACHLAN · Discriminant Analysis and Statistical Pattern Recognition
 McLACHLAN, DO, and AMBROISE · Analyzing Microarray Gene Expression Data
 McLACHLAN and KRISHNAN · The EM Algorithm and Extensions, *Second Edition*
 McLACHLAN and PEEL · Finite Mixture Models
 McNEIL · Epidemiological Research Methods
 MEEKER and ESCOBAR · Statistical Methods for Reliability Data
 MEERSCHAERT and SCHEFFLER · Limit Distributions for Sums of Independent
 Random Vectors: Heavy Tails in Theory and Practice
 MENGERSEN, ROBERT, and TITTERINGTON · Mixtures: Estimation and
 Applications
 MICKEY, DUNN, and CLARK · Applied Statistics: Analysis of Variance and
 Regression, *Third Edition*
 * MILLER · Survival Analysis, *Second Edition*
 MONTGOMERY, JENNINGS, and KULAHCI · Introduction to Time Series Analysis
 and Forecasting
 MONTGOMERY, PECK, and VINING · Introduction to Linear Regression Analysis,
 Fourth Edition
 MORGENTHALER and TUKEY · Configural Polysampling: A Route to Practical
 Robustness
 MUIRHEAD · Aspects of Multivariate Statistical Theory
 MULLER and STOYAN · Comparison Methods for Stochastic Models and Risks
 MURRAY · X-STAT 2.0 Statistical Experimentation, Design Data Analysis, and
 Nonlinear Optimization
 MURTHY, XIE, and JIANG · Weibull Models
 MYERS, MONTGOMERY, and ANDERSON-COOK · Response Surface Methodology:
 Process and Product Optimization Using Designed Experiments, *Third Edition*

*Now available in a lower priced paperback edition in the Wiley Classics Library.

†Now available in a lower priced paperback edition in the Wiley-Interscience Paperback Series.

- MYERS, MONTGOMERY, VINING, and ROBINSON · Generalized Linear Models.
 With Applications in Engineering and the Sciences, *Second Edition*
- † NELSON · Accelerated Testing, Statistical Models, Test Plans, and Data Analyses
- † NELSON · Applied Life Data Analysis
- NEWMAN · Biostatistical Methods in Epidemiology
- OCHI · Applied Probability and Stochastic Processes in Engineering and Physical Sciences
- OKABE, BOOTS, SUGIHARA, and CHIU · Spatial Tesselations: Concepts and Applications of Voronoi Diagrams, *Second Edition*
- OLIVER and SMITH · Influence Diagrams, Belief Nets and Decision Analysis
- PALTA · Quantitative Methods in Population Health: Extensions of Ordinary Regressions
- PANJER · Operational Risk: Modeling and Analytics
- PANKRATZ · Forecasting with Dynamic Regression Models
- PANKRATZ · Forecasting with Univariate Box-Jenkins Models: Concepts and Cases
- PARDOUX · Markov Processes and Applications: Algorithms, Networks, Genome and Finance
- * PARZEN · Modern Probability Theory and Its Applications
- PEÑA, TIAO, and TSAY · A Course in Time Series Analysis
- PIANTADOSI · Clinical Trials: A Methodologic Perspective
- PORT · Theoretical Probability for Applications
- POURAHMADI · Foundations of Time Series Analysis and Prediction Theory
- POWELL · Approximate Dynamic Programming: Solving the Curses of Dimensionality, *Second Edition*
- PRESS · Bayesian Statistics: Principles, Models, and Applications
- PRESS · Subjective and Objective Bayesian Statistics, *Second Edition*
- PRESS and TANUR · The Subjectivity of Scientists and the Bayesian Approach
- PUKELSHEIM · Optimal Experimental Design
- PURI, VILAPLANA, and WERTZ · New Perspectives in Theoretical and Applied Statistics
- † PUTERMAN · Markov Decision Processes: Discrete Stochastic Dynamic Programming
- QIU · Image Processing and Jump Regression Analysis
- * RAO · Linear Statistical Inference and Its Applications, *Second Edition*
- RAO · Statistical Inference for Fractional Diffusion Processes
- RAUSAND and HØYLAND · System Reliability Theory: Models, Statistical Methods, and Applications, *Second Edition*
- RAYNER · Smooth Tests of Goodness of Fit: Using R, *Second Edition*
- RENCHER · Linear Models in Statistics
- RENCHER · Methods of Multivariate Analysis, *Second Edition*
- RENCHER · Multivariate Statistical Inference with Applications
- * RIPLEY · Spatial Statistics
- * RIPLEY · Stochastic Simulation
- ROBINSON · Practical Strategies for Experimenting
- ROHATGI and SALEH · An Introduction to Probability and Statistics, *Second Edition*
- ROLSKI, SCHMIDLI, SCHMIDT, and TEUGELS · Stochastic Processes for Insurance and Finance
- ROSENBERGER and LACHIN · Randomization in Clinical Trials: Theory and Practice
- ROSS · Introduction to Probability and Statistics for Engineers and Scientists
- ROSSI, ALLENBY, and McCULLOCH · Bayesian Statistics and Marketing
- † ROUSSEEUW and LEROY · Robust Regression and Outlier Detection
- ROYSTON and SAUERBREI · Multivariate Model Building: A Pragmatic Approach to Regression Analysis Based on Fractional Polynomials for Modeling Continuous Variables
- * RUBIN · Multiple Imputation for Nonresponse in Surveys
- RUBINSTEIN and KROESE · Simulation and the Monte Carlo Method, *Second Edition*

*Now available in a lower priced paperback edition in the Willey Classics Library.

†Now available in a lower priced paperback edition in the Willey-Interscience Paperback Series.

- RUBINSTEIN and MELAMED · Modern Simulation and Modeling
- RYAN · Modern Engineering Statistics
- RYAN · Modern Experimental Design
- RYAN · Modern Regression Methods, *Second Edition*
- RYAN · Statistical Methods for Quality Improvement, *Third Edition*
- SALEH · Theory of Preliminary Test and Stein-Type Estimation with Applications
- SALTELLI, CHAN, and SCOTT (editors) · Sensitivity Analysis
- * SCHEFFE · The Analysis of Variance
- SCHIMEK · Smoothing and Regression: Approaches, Computation, and Application
- SCHOTT · Matrix Analysis for Statistics, *Second Edition*
- SCHOUTENS · Levy Processes in Finance: Pricing Financial Derivatives
- SCHUSS · Theory and Applications of Stochastic Differential Equations
- SCOTT · Multivariate Density Estimation: Theory, Practice, and Visualization
- * SEARLE · Linear Models
- † SEARLE · Linear Models for Unbalanced Data
- † SEARLE · Matrix Algebra Useful for Statistics
- † SEARLE, CASELLA, and McCULLOCH · Variance Components
- SEARLE and WILLETT · Matrix Algebra for Applied Economics
- SEBER · A Matrix Handbook For Statisticians
- † SEBER · Multivariate Observations
- SEBER and LEE · Linear Regression Analysis, *Second Edition*
- † SEBER and WILD · Nonlinear Regression
- SENNOTT · Stochastic Dynamic Programming and the Control of Queueing Systems
- * SERFLING · Approximation Theorems of Mathematical Statistics
- SHAFER and VOVK · Probability and Finance: It's Only a Game!
- SHERMAN · Spatial Statistics and Spatio-Temporal Data: Covariance Functions and Directional Properties
- SILVAPULLE and SEN · Constrained Statistical Inference: Inequality, Order, and Shape Restrictions
- SINGPURWALLA · Reliability and Risk: A Bayesian Perspective
- SMALL and MCLEISH · Hilbert Space Methods in Probability and Statistical Inference
- SRIVASTAVA · Methods of Multivariate Statistics
- STAPLETON · Linear Statistical Models, *Second Edition*
- STAPLETON · Models for Probability and Statistical Inference: Theory and Applications
- STAUDTE and SHEATHER · Robust Estimation and Testing
- STOYAN, KENDALL, and MECKE · Stochastic Geometry and Its Applications, *Second Edition*
- STOYAN and STOYAN · Fractals, Random Shapes and Point Fields: Methods of Geometrical Statistics
- STREET and BURGESS · The Construction of Optimal Stated Choice Experiments: Theory and Methods
- STYAN · The Collected Papers of T. W. Anderson: 1943–1985
- SUTTON, ABRAMS, JONES, SHELDON, and SONG · Methods for Meta-Analysis in Medical Research
- TAKEZAWA · Introduction to Nonparametric Regression
- TAMHANE · Statistical Analysis of Designed Experiments: Theory and Applications
- TANAKA · Time Series Analysis: Nonstationary and Noninvertible Distribution Theory
- THOMPSON · Empirical Model Building, *Second Edition*
- THOMPSON · Sampling, *Second Edition*
- THOMPSON · Simulation: A Modeler's Approach
- THOMPSON and SEBER · Adaptive Sampling
- THOMPSON, WILLIAMS, and FINDLAY · Models for Investors in Real World Markets
- TIAO, BISGAARD, HILL, PEÑA, and STIGLER (editors) · Box on Quality and Discovery: with Design, Control, and Robustness

*Now available in a lower priced paperback edition in the Wiley Classics Library.

†Now available in a lower priced paperback edition in the Wiley-Interscience Paperback Series.

- TIERNEY · LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics
- TSAY · Analysis of Financial Time Series, *Third Edition*
- UPTON and FINGLETON · Spatial Data Analysis by Example, Volume II: Categorical and Directional Data
- † VAN BELLE · Statistical Rules of Thumb, *Second Edition*
- VAN BELLE, FISHER, HEAGERTY, and LUMLEY · Biostatistics: A Methodology for the Health Sciences, *Second Edition*
- VESTRUP · The Theory of Measures and Integration
- VIDAKOVIC · Statistical Modeling by Wavelets
- VINOD and REAGLE · Preparing for the Worst: Incorporating Downside Risk in Stock Market Investments
- WALLER and GOTWAY · Applied Spatial Statistics for Public Health Data
- WEERAHANDI · Generalized Inference in Repeated Measures: Exact Methods in MANOVA and Mixed Models
- WEISBERG · Applied Linear Regression, *Third Edition*
- WEISBERG · Bias and Causation: Models and Judgment for Valid Comparisons
- WELSH · Aspects of Statistical Inference
- WESTFALL and YOUNG · Resampling-Based Multiple Testing: Examples and Methods for *p*-Value Adjustment
- * WHITTAKER · Graphical Models in Applied Multivariate Statistics
- WINKER · Optimization Heuristics in Economics: Applications of Threshold Accepting
- WONNACOTT and WONNACOTT · Econometrics, *Second Edition*
- WOODING · Planning Pharmaceutical Clinical Trials: Basic Statistical Principles
- WOODWORTH · Biostatistics: A Bayesian Introduction
- WOOLSON and CLARKE · Statistical Methods for the Analysis of Biomedical Data, *Second Edition*
- WU and HAMADA · Experiments: Planning, Analysis, and Parameter Design Optimization, *Second Edition*
- WU and ZHANG · Nonparametric Regression Methods for Longitudinal Data Analysis
- YANG · The Construction Theory of Denumerable Markov Processes
- YOUNG, VALERO-MORA, and FRIENDLY · Visual Statistics: Seeing Data with Dynamic Interactive Graphics
- ZACKS · Stage-Wise Adaptive Designs
- * ZELLNER · An Introduction to Bayesian Inference in Econometrics
- ZELTERMAN · Discrete Distributions—Applications in the Health Sciences
- ZHOU, OBUCHOWSKI, and McCLISH · Statistical Methods in Diagnostic Medicine, *Second Edition*

*Now available in a lower priced paperback edition in the Wiley Classics Library.

†Now available in a lower priced paperback edition in the Wiley-Interscience Paperback Series.